# Enhancing SSR in Low-Thread Web Servers: A Comprehensive Approach for Progressive Server-Side Rendering with any Asynchronous API and Multiple Data Models

Fernando Miguel Carvalho[a] and Pedro Fialho
*ISEL, Polytechnic Institute of Lisbon, Portugal*

*https://cc.isel.pt/*

Keywords: Web Templates, Server-Side Rendering, Non-Blocking, Asynchronous, Concurrent.

Abstract: Naive server-side rendering (SSR) techniques require a dedicated server thread per HTTP request, thereby limiting the number of concurrent requests to the available server threads. Furthermore, this approach proves impractical for modern low-thread servers like WebFlux, VertX, and Express Node.js. To achieve progressive rendering, asynchronous data models provided by non-blocking APIs must be utilized. Nevertheless, this method can introduce undesirable interleaving between template view processing and data access, potentially resulting in malformed HTML documents. Some template engines offer partial remedies through specific templating dialects, but they encounter two limitations. Firstly, their compatibility is confined to specific types of asynchronous APIs, such as the reactive stream `Publisher` API. Secondly, they typically support only a single asynchronous data model at a time. In this research, we propose an alternative web templating approach that embraces any asynchronous API (e.g., `Publisher`, promises, suspend functions, flow, etc.) and allows for multiple asynchronous data sources. Our approach is implemented on top of HtmlFlow, a Java-based DSL for writing type-safe HTML. We evaluated against state-of-the-art reactive servers, specifically WebFlux, and compared it with popular templating idioms like Thymeleaf and KotlinX.html. Our proposal effectively overcomes the limitations of existing approaches.

SCIENCE AND TECHNOLOGY PUBLICATIONS

## 1 INTRODUCTION

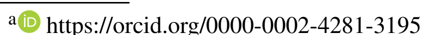
In traditional thread-per-request architectures, each incoming request is handled by a dedicated thread. The web server creates a new thread to handle each incoming request, which means that the number of threads in the system can grow quickly as the load increases. This can lead to performance issues and scalability problems, as the system can become bogged down by context switching and thread overhead (Kant and Mohapatra, 2000).

An alternative approach involves employing a user-level *M:N threading runtime*, which entails the execution of numerous user-level threads (*M*) by multiple system threads (*N*) in a seamless manner. User-level threads provide a lightweight solution for managing a higher volume of concurrent sessions due to their reduced per-thread overhead (Welsh et al., 2001; von Behren et al., 2003; Qin et al., 2018). Nevertheless, this approach still necessitates a user-level I/O subsystem capable of alleviating *system-level blocking* and vital for the performance of I/O-heavy applications (Karsten and Barghi, 2020).

Conversely, within contemporary *low-thread* architectures, also known as *event-driven* (Elmeleegy et al., 2004), the server can handle a large number of concurrent requests without blocking, which makes it more scalable and efficient (Schmidt, 1995). To that end, it uses *non-blocking* I/O to ensure that each thread can handle multiple requests at the same time.

The *non-blocking* I/O model used in low-thread servers is well suited for handling large volumes of data asynchronously (Meijer, 2012). The combination of low-thread servers and asynchronous data models has enabled the development of highly scalable, responsive, and resilient Web applications that can handle a high volume of data (Jin et al., 2015).

This idea gained significant attention with the rise of Node.js in 2009 (Chaniotis et al., 2014), and later several technologies embrace this approach in Java ecosystem namely, Netty (Maurer and Wolfthal, 2015), Akka HTTP (Davis, 2019), Vert.X (Fox, 2001)


[a] https://orcid.org/0000-0002-4281-3195



Carvalho, F. and 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.
DOI: 10.5220/0012165300003584
In *Proceedings of the 19th International Conference on Web Information Systems and Technologies (WEBIST 2023)*, pages 37-48
ISBN: 978-989-758-672-9; ISSN: 2184-3252
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

and Spring WebFlux (Johnson et al., 2004), being the latter the most widely used Web framework in Java according to several surveys, such as JetBrains' State of Java report (2021) and community activity, such as Github and Stackoverflow.

Migrating legacy Web applications to state-of-the-art *low-thread* Web servers, not only involves porting the Web handlers (e.g. *controllers* (Alur et al., 2001)), and *data repositories* (Fowler, 2002), but may also concern the Frontend, when it uses a Server-Side Rendering (SSR) approach (Alur et al., 2001), where the Web server is also responsible for generating the HTML for a Web page. Although, Single-Page Applications (SPAs) with Client-Side Rendering (CSR) (Klauzinski and Moore, 2016) have become a popular choice for building Frontend in modern Web applications, Server-Side Rendering (SSR) still has a significant installation base and continues to be used in many Multi-Page applications.

However, only a few Java template engines properly deal with asynchronous data models. In our work, we analyze a use-case of a Spring WebFlux application and some limitations, and harmful effects, that may emerge from the use of SSR Frontend with asynchronous data models, such as: 1) unresponsive blank page, 2) limited concurrency, 3) server runtime exceptions, 4) ill-formed HTML, 5) single model use, and 6) restricted asynchronous API.

Our work is built on top of HtmlFlow, a Java DSL library for HTML. We present a new proposal that addresses all the aforementioned issues through the combination of three ideas: 1) *functional templates*, which regards the capacity of implementing Web templates as *higher-order functions* (Carvalho. and Duarte., 2019); 2) *resume* callback in continuations (Haynes et al., 1984) to control transitions from asynchronous handlers, and 3) chain-of-responsibility design pattern (Gamma et al., 1995) to control the flow between a chain of template fragments.

In the next section, we highlight some issues that arise from the naive use of asynchronous techniques in web templates. Then, in Section 3, we describe state-of-the-art template engines for SSR. Following that, in Section 4, we compare these template engines in a case study of a Spring WebFlux application. Section 5 explains how our proposal mitigates the limitations of the competition. Next, in Section 6, we evaluate different web templates and their issues regarding asynchronous data models. We conclude and discuss future work in Section 7.

# 2 ASYNCHRONOUS APIs MATTER

The non-blocking IO model used in low-thread servers only functions properly if HTTP handlers do not block. This implies that an HTTP handler cannot wait for IO completion, such as reading a file, querying a database, or any other kind of IO operation. Therefore, handlers must be designed to initiate IO operations and return immediately, allowing other handlers to execute while the IO operation completes in the background. To ensure non-blocking I/O, handlers should use asynchronous APIs to access data. Asynchronous APIs allow handlers to submit requests and receive notifications when operations complete, all without blocking the thread.

However, while using a synchronous API can be straightforward with its direct style of producing results through the returned value of function calls, dealing with an asynchronous API can be more challenging. Asynchronous APIs do not have a standard approach, leading to several proposals such as *continuation-passing style* (CPS) (Sussman and Steele, 1975), async/await idiom (Syme et al., 2011), reactive streams (Reactive Streams, 2015), Kotlin Flow (Breslav, 2016), and others. The correct use of an asynchronous API can help ensure that code is efficient, reliable, and maintainable. On the other hand, dealing with asynchronous data models in a naive way, may produce several harmful effects. In the context of a web server we may observe the following problems, not limited to:

1. *unresponsive blank page*;
2. *limited concurrency*,
3. *server runtime exceptions*.

A naive way of dealing with an asynchronous API is blocking on completion. For example, in the Java `CompletableFuture`, which is an implementation of a *Promise* (Friedman and Wise, 1978), this may be achieved getting its result from its method `join(): T`. This blocking approach can be used in any template engine, including those analyzed in this work (i.e. Thymeleaf, KotlinX.html and HtmlFlow) with the same ill effect. These kind of handlers will produce an *unresponsive blank page* in the browser. Only when all data models are completed we may see a resulting HTML document. Rather than a progressive behavior where the page would be rendered smoothly as data becomes available, we will have an all or nothing effect starting with a blank page and finishing with the complete page.

Furthermore, while an handler is waiting for data completion it is blocking a thread and preventing it

from doing another task, such as processing another HTTP request, leading to *limited concurrency*. For single threaded environments it means that it will handle just one HTTP request at a time.

In some environments, blocking for completion may cause even worse issues, lead to malfunction and throw a server runtime exception. For instance, in Spring WebFlux blocking an HTTP handler may throw: `IllegalStateException` with the message: *block()/blockFirst()/blockLast() are blocking, which is not supported in thread reactor-http-nio-1*.

Another example is the Eclipse Vert.x for JVM (Fox, 2001), which includes a low-thread server based on Netty and in similar blocking situations may throw an exception, such as: *io.vertx.core.-VertxException: Thread [vert.x-eventloop-thread-1,5,main] has been blocked for 2997 ms, time limit is 2000 ms*

When a blocking operation occurs within such a server, it can lead to resource contention and potentially degrade the scalability of the server. The exceptions serve as warnings to developers that blocking operations are being performed in a context where they are not supported or recommended. They highlight the potential risk of blocking and encourage developers to refactor their code to use non-blocking alternatives or offload blocking operations to separate threads or asynchronous tasks.

While the web application might still appear to work fine despite these exceptions, it is important to address them to ensure the proper functioning and scalability of the application within the intended low-thread and non-blocking architecture. Ignoring or suppressing these exceptions can lead to potential performance issues, thread starvation, and decreased responsiveness of the web server under high load.

# 3 STATE OF THE ART

In this chapter, we will begin by introducing the latest advancements in Web Templates with SSR. Following that, we will address several issues that arise from the utilization of asynchronous data models.

## 3.1 Related Work

Thymeleaf (Fernández, 2011) is the default *template view* engine in Spring framework. Thymeleaf has a powerful expression language that allows developers to dynamically manipulate data in their HTML templates. Also, Thymeleaf is the only Spring built-in view engine that supports asynchronous data models with *progressive rendering* (Deinum and Cosmina, 2021), emitting the resulting HTML in a series of incremental updates as the template is being resolved, rather than waiting for the entire template to be resolved before emitting any HTML. All the others built-in view engines, such as Freemarker, Handlebars, Jade or JTwig produce an unresponsive blank page while awaiting for data completion. On the other hand, JSP (Bergsten, 2003) is not compliant and not even supported in WebFlux.

Web templates may also be classified according to the *domain-specific language* (DSL) they use, which is a programming language specialized to a particular application domain (e.g. HTML) (Landin, 1966). j2Html (Ase, 2015), KotlinX.html (Mashkov, 2015) and HtmlFlow (Carvalho, 2017), are examples of Java libraries DSLs for HTML. In common these DSLs use *functions* to define their languages. According to (Fowler, 2010) we can distinguish their APIs between: 1) *nested function* and 2) *method chaining*.

*Nested function* combines functions by making function calls arguments in higher-level function calls. This approach can be used to organize code and create reusable code blocks, allowing an efficient and modular programming style. In Listing 1 we present an example of a DSL for HTML using a *nested function* approach. Yet, a simple sequence of nested functions ends up being evaluated backwards to the order they are written. This means that arguments are first evaluated before the function being invoked. In Listing 1, `p()` is first evaluated and its resulting paragraph will be the argument of the call to `div()`, which in turn will be the argument of `body()` and henceforward.

```
html(
  head(
    title("My title")),
  body(
    div(
      p("Some text"))))
```

Listing 1: Nested function.

The backward evaluation behavior may incur in several issues. Since arguments are evaluated before the high-level function is called, this technique may not support *progressive rendering* to emit HTML on demand according to functions calls order. Otherwise the HTML would be generated backwards. Thus, it may require some sort of auxiliary data structure to manage elements processing and control HTML emission, which in turn may lead to additional performance overhead compared to other alternatives that do not require such a data structure.

*Method chaining* pattern avoids the backward evaluation behavior, since it is based on methods calls

```
Html()
  .head()
    .title("My title")
  .body()
    .div()
     .p("Some text")
```

Listing 2: Method chaining.

with *receiver*, which is the object the method is being called on (Evans et al., 2022). The *receiver* object is passed as an implicit argument to each method call, allowing for each subsequent method to be called on the result of the previous method. In the example of Listing 2, the result of `Html()` call is an HTML element that will be the receiver for the next `head()` call, which in turn produces an Head element that will be the receiver for the next `title()` call, and henceforward.

J2html uses a *nested function* approach where templates has a similar layout to that one presented in Listing 1. The result of the execution of a j2Html template is a tree structure composed of `Tag` objects(Gamma et al., 1995). The `render()` method is then used to traverse the tree and produce an HTML document. Also, j2Html does not have built-in support for asynchronous data models.

KotlinX.html also uses a *nested function* approach to generate HTML, but instead of using objects as arguments, it uses *function literals* (i.e. *lambdas*) to represent HTML tags and attributes. By using function literals as arguments, KotlinX.html is able to delay the evaluation of the HTML tags until the render stage, which solves the problem of backward evaluation that could occur with j2html's object-based approach.

HtmlFlow was designed to be a lightweight and efficient Java DSL library for generating HTML, and one of its key features is its fluent API with a *method chaining* approach, similar to the sample presented in Listing 2. When using HtmlFlow, developers can chain together a series of method calls to define the structure and content of their HTML templates. As each method is called, it emits HTML code directly, rather than instantiating and storing intermediate objects. This approach allows HtmlFlow to generate HTML more efficiently and with lower memory overhead than some other HTML generation libraries that may instantiate and store a large number of objects representing HTML nodes or elements.

In Table 1 we present a brief comparison between the web templates and the properties we have discussed in this section, regarding non-asynchronous data models. We also included a performance metric regarding the throughput of each web template in Spring templates benchmark (Reijn, 2015). These results are relatively to HtmlFlow throughput, which is the most performant engine among the evaluated web templates.

Table 1: Comparing template views in terms of DSL approach, host language, and ability of providing functional templates, progressive rendering and their relative performance to HtmlFlow in Spring templates benchmark.

| Library | DSL | Language | Functional | Prog. | Bench |
|---|---|---|---|---|---|
| **Thymeleaf** | External | Thymeleaf | × | ✓ | 32% |
| **j2html** | Nested | Java | ✓ | × | 26% |
| **KotlinX** | Nested | Kotlin | ✓ | ✓ | 58% |
| **HtmlFlow** | Chain | Java | ✓ | ✓ | 100% |

## 3.2 Asynchronous Data Models Issues

However, the use of asynchronous data models in web templates can introduce new issues, namely:

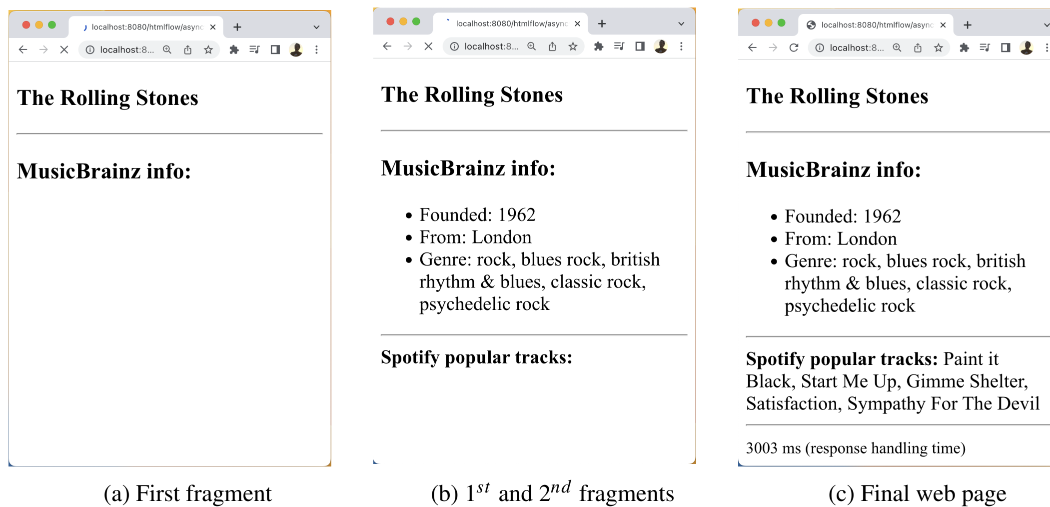
1. limited asynchronous API
2. single data model
3. ill-formed HTML
4. nested callbacks

One of the main challenges is the lack of a standard API for asynchronous calls, as there are multiple different APIs and idioms available, as described in Section 2. Additionally, web templates may only support a *limited asynchronous API*, which can pose a limitation on application development. This is especially true when the API provided by the non-blocking drivers is incompatible with the web template's asynchronous support.

Additionally, while most web templates can bind with multiple synchronous data models, they may not be able to do the same for asynchronous data models, limiting the progressive rendering to a *single data model*.

Despite this, even for web templates that do not face the *single data model* issue, another problem may arise regarding the correct emission of HTML. The asynchronicity between the dispatch and completion of the IO operation may lead to undesired interleaving between data access and HTML definition, potentially resulting in an *ill-formed HTML* document. An example of such an ill-formed HTML document is presented in Listing 6 of Section 4.3.

Even though a web page may still be readable and function correctly despite containing invalid HTML, it is still important to strive for valid and well-formed code. Valid HTML code ensures that the web page is accessible to a wider range of users and devices, and it also helps search engines to understand the content of the page better, which can improve search engine rankings.




(a) First fragment (b) $1^{st}$ and $2^{nd}$ fragments (c) Final web page

Figure 1: Expected output from *Disco* web app for The Rolling Stones band.

Finally for web templates dealing with data models through the *continuation-passing style* (CPS) (Sussman and Steele, 1975) we can observe an idiomatic pattern emerging on source code from the use of *nested callbacks*. The level of nesting will be proportional to the number of asynchronous models used in the web template which can lead to code that is difficult to read and maintain. This scenario is commonly referred to as ”*callback hell*” (Kambona et al., 2013).

## 4 CASE STUDY

In this chapter, we will present how Thymeleaf and KotlinX.html deal with asynchronous data models in a web application running in a state of the art non-blocking server, namely Spring WebFlux (Deinum and Cosmina, 2021).

In the next subsection we start describing our case study of a WebFlux application named *Disco*. After that, we analyze how Thymeleaf and KotlinX.html behave in this application.

### 4.1 *Disco* Non-Blocking Web Application

The *Disco* WebFlux application consumes data from two Web APIs, namely MusicBrainz (an open music encyclopedia that collects music metadata) and Spotify (an audio streaming). The Disco application will fetch information from MusicBrainz about the foundation, origin and genres of a band, and from Spotify it will get its popular tracks.

In Figure 1 we present the expected output of an HTML document from the Disco web application for The Rolling Stones band, in three accumulated fragments. The first fragment of Figure 1a is immediately rendered since it does not depend of any asynchronous data. Then, the rest two fragments of Figures 1b and 1c are emitted incrementally as their data models become available. We deliberately made, Spotify data model with a higher latency than MusicBrainz to observe a progressive render between these two steps. In Listing 3 we present the corresponding HTML source to the final web page of Figure 1c.

SCIENCE AND TECHNOLOGY PUBLICATIONS

```
<html>
  <body>
    <div>
      <h3>The Rolling Stones</h3>
      <hr>
      <h3>MusicBrainz info:</h3>
      <ul>
       <li>Founded: 1962</li>
       <li>From: London</li>
       <li>Genre: rock, blues rock, ...</
           li>
      </ul>
      <hr>
      <b>Spotify popular tracks:</b>
      <span>Paint it Black, Start ...</
          span>
      <hr>
    </div>
    <footer>
      <small>
        3003 ms (response handling time)
      </small>
    </footer>
  </body>
</html>
```

Listing 3: Expected HTML source code for web page of Figure 1c.

Notice, we are including a `footer` with the total processing time, since the handler receives the request, until the template finishes processing the `footer` element. To turn visible the effects of progressive rendering we inserted an explicit delay of 2 seconds on the first data source (i.e. MusicBrainz) and 3 seconds on the second one (i.e. Spotify). Since we are fetching data sources concurrently, then the handler processing duration is at least equal to the largest delay of the data sources, in this case 3 seconds.

Since the analyzed web templates are compatible with both Java and Kotlin, and Spring WebFlux is as well, we choose Kotlin as the programming language for the Disco web application due to its conciseness and expressiveness.

The Disco domain model has two main entities defined by `MusicBrainz` and `SpotifyArtist` classes. In this case we have a *repository* for each domain entity to access its data source (Fowler, 2002). Because we are dealing with asynchronous data sources our repositories produce asynchronous effects. In this case, we choose a `CompletableFuture` return type to represent an asynchronous computation that may complete at some point and produce a result.

## 4.2 Thymeleaf

Web templates (such as JSP, Handlebars or Thymeleaf) are based in HTML documents, which are augmented with template specific markers (e.g. `<%`, `{{}}` or `${}`) representing *dynamic* information that can be replaced at runtime by the results of corresponding computations to produce a *view*. In addition to specific Thymeleaf templating *marks* (i.e. `${}`) we may find thymeleaf *attributes* that define the execution of predefined logic, such as, for each loop (i.e. `th:each`).

In Listing 4 we present the Thymeleaf template that produces the view of Figure 1c. For simplicity we are just including the dynamic parts, since the static blocks are equal to those presented in Listing 3. This template deals with 4 model attributes: `artistName` (line 1), `musicBrainz` (lines 3), `spotify` (line 13) and `start` (line 18).

Except for attributes `artistName` and `start` that are a `String` and a `long`, both `musicBrainz` and `spotify` are instances of `CompletableFuture`. In this case, passing a `CompletableFuture` or any other kind of promise to a Thymeleaf template will postpone processing releasing the thread to other tasks. When data becomes available and the promise is fulfilled, the template processing resumes making the `CompletableFuture`'s content available as model attributes.

```
1  <h3 th:text="${artistName}"></h3>
2  ...
3  <ul th:each="item : ${musicBrainz}">
4   <li th:text="*{'Founded: ' + item.year
        }">
5   </li>
6   <li th:text="*{'From: ' + item.from}">
7   </li>
8   <li th:text="*{'Genre: ' + item.genres
        }">
9   </li>
10 </ul>
11 ...
12  <th:block
13   th:each="track: ${spotify.popularSongs
        }"
14     th:text="${track + ', '}">
15  </th:block>
16 ...
17 <small
18   th:text="${#dates.createNow().getTime
        () - start} + 'ms (response
        handling time)'">
19 </small>
```

Listing 4: Example of Thymeleaf template of Disco web application.

Yet, until model attributes `musicBrainz` and `spotify` become available the browser is displaying an unresponsive blank page waiting for server response. To address this issue on Spring WebFlux, the asynchronous objects must be wrapped into a `ReactiveDataDriverContextVariable`. When a model attribute of this type is present, the template engine enters data-driven mode, where data is streamed to the client as it becomes available, rather than waiting for the entire response to be generated before sending it. This allows the static parts of the web template to be immediately emitted, displaying an initial web page as presented in Figure 1a before the asynchronous data is completed and the web page is finished.

But, there are two notable limitations of `ReactiveDataDriverContextVariable`, regarding the issues 1) *limited asynchronous idiom* and 2) *single data model*, described in of Section 3. First, the model attribute must be a *reactive stream* `Publisher` (Reactive Streams, 2015), meaning that dealing with any other kind of asynchronous model always requires a conversion to `Publisher`. Second, the web template can only have a single model attribute of this type.

Regarding 1) *limited asynchronous idiom*, actually, converting from `CompletableFuture` to `Publisher` can be achieved through the `Mono.fromFuture()` method of the Reactor library, which returns a new instance of `Mono` that implements `Publisher`. A `Publisher` represents multi-valued data and it is expected that Thymeleaf web template contains some kind iteration on

the data-driver variable, normally by means of a `th:each` attribute. This means that it requires a `th:each` block to access `musicBrainz` even it just contains a single value (i.e. line 3 of Listing 4). However, this requirement may introduce a semantic mismatch when dealing with a single value, such as the `CompletableFuture<MusicBrainz>`. The use of the `th:each` block implies an iteration, whereas in this case, a continuation-based idiom would be more appropriate to convey the intended meaning.

On the other hand, regarding 2) *single data model* use, when we have multiple asynchronous objects, we can compose them into a single object using operators such as `zip`, `combineLatest`, or `merge`. However, when we do this, we won't have the benefit of progressive rendering that we get with `ReactiveDataDriverContextVariable`. Instead, we will have to wait for all the asynchronous operations to complete before the final result can be rendered.

## 4.3 KotlinX.html

Since KotlinX.html is a Kotlin DSL library for HTML we can take advantage of its host programming language features to deal with data models, without any special constructs like `ReactiveDataDriverContextVariable` to handle data-driven rendering. The Java `CompletableFuture` API provides completion handlers that can be used to chain dynamic blocks of a template to be processed only when data become available.

Here, we are using the `thenAccept` handler to register a callback that is called when the `CompletableFuture` completes. In the next example, the callback function `mb -> ...` is passed as an argument to the `thenAccept` method of the `musicBrainz CompletableFuture`:

```
musicBrainz.thenAccept { mb -> ... }
```

This callback function will be executed when the `musicBrainz` future completes, and it will receive the result of the future, which is represented by the `mb` parameter.

KotlinX.html can emit HTML to any output compatible with the `Appendable` interface. The extension function `appendHTML()` takes a receiver of a type that implements the `Appendable` interface, and returns a `TagConsumer` object that we can use to write HTML tags and attributes. On the other hand, WebFlux allows building reactive responses from a `Publisher<String>`. Thus, to use KotlinX.html in WebFlux, we use an auxiliary type `AppendableSink` that is able to produce a `Publisher<String>` from an `Appendable`.

```
1  fun artistView(
2      start: Long,
3      artisName: String,
4      musicBrainz: CF<MusicBrainz>,
5      spotify: CF<SpotifyArtist>
6  ): Publisher<String> =
       AppendableSink {
7   this
8   .appendHTML()
9   .html {
10   body {
11    div {
12     h3 { text("$artisName") }
13     hr()
14     h3 { text("MusicBrainz info:")
       }
15     ul { musicBrainz.thenAccept {
          mb ->
16      li { text("Founded:${mb.year}"
          ) }
17      li { text("From:${mb.from}") }
18      li { text("Genres:${mb.genres}
          ") }
19      hr()
20      b { text("Spotify ...: ") }
21      span { spotify.thenAccept {
          spt ->
22       text(spt.popularSongs.
          joinToString(", "))
23       hr()
24       footer {
25        small {
26         text("${currentTimeMillis()
             -start} ms (response
             handling time)")
27        }
28       }
29       this.close()
30      } // thenAccept
31     } // span
32    } // thenAccept
33   } // ul
34  } // div
35  } // body
36 } // html
37 .asFlux()
```

Listing 5: Example of KotlinX.html template of Disco web application. For readability we replaced type name `CompletableFuture` by `CF`.

In Listing 5 we present the definition in KotlinX.html for the web page of Figure 1c. 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*. For example, when we write `body{...}`, we are invoking the `body` function with a lambda as its argument. This lambda can be used to define the contents of the HTML tag represented by the `body` function.

The constructor of `AppendableSink` used in List-

ing 5 receives a lambda that is called with that `AppendableSink` object as the receiver (i.e. `this`). The last statment of the web template definition (line 29) should close the `AppendableSink` to make the resulting `Publisher` be completed. This is mandatory to let WebFlux know that the HTML emit is completed and the HTTP connection can be terminated. Finally, notice how `AppendableSink` is converted into a `Publisher` in line 37 through its method `asFlux`, where `Flux` is an implementation of `Publisher`.

Notice we make plain use of `thenAccept` completion handler to deal with the resulting data model (i.e. `mb` in line 15 and `spt` in line 21) without the need of a special construct like `th:each` in Thymeleaf. Using the non-blocking API of `CompletableFuture` we achieve a progressive rendering, which first emits the resulting web page of Figure 1a and then proceeds to the next partial page of Figure 1b resulting from completion of MusicBrainz data model and before completion of Spotify data. The web page will finish with the expected output of Figure 1c when the last data model completes.

Yet, there are still some issues to deal. First, the call to `thenAccept` returns immediately and the enclosing HTML builder (i.e. `ul` on line 15 and `span` on line 22) will emit the end tag before the continuation has been performed, leading to the issue 3) *ill-formed HTML* of Section 3.2. The resulting HTML will be out of order as presented in Listing 6. Notice, elements `ul`, `div`, `body` and `html` are closed (lines 7, 8, 9, and 10) before their inner elements have been emitted with data from MusicBrainz and Spotify web APIs.

Modern browsers have become quite forgiving when it comes to rendering HTML that does not comply with the specifications. They try to interpret and render the content as best as they can, even if the HTML source is invalid or contains errors.

Secondly, we can observe an emerging idiomatic anti-pattern in the source code, specifically in the chaining of callbacks (e.g., between lines 16 and 23). This anti-pattern often leads to a pyramid-like structure, particularly when dealing with nested callbacks, regarding the issue 4) *nested callbacks* of Section 3.2. As the template becomes dependent on more asynchronous data models, the callbacks become nested deeper within each other, exacerbating the issue of callback nesting.

```
1  <html>
2    <body>
3      <div>
4        <h3>The Rolling Stones</h3>
5        <hr>
6        <h3>MusicBrainz info:</h3>
7        <ul></ul>
8      </div>
9    </body>
10 </html>
11 <li>Founded: 1962</li>
12 <li>From: London</li>
13 <li>Genres: rock, blues rock, ...</li>
14 <hr>
15 <b>Spotify popular tracks: </b>
16 <span></span>Paint it Black, Start Me
       ...
17 <hr>
18 <footer>
19   <small>
20     3011 ms (response handling time)
21   </small>
22 </footer>
```

Listing 6: Example of ill-formed HTML source resulting from undesirable interleavings between template processing and asynchronous data access.

## 5 HtmlFlow ASYNCHRONOUS SUPPORT

HtmlFlow uses a *method chaining* approach as described in Section 3.1, and one of its key features that solves the problem of the ill-formed HTML with asynchronous models is its method *builder* `__()` to close HTML tags. Element tags should be explicitly closed through the invocation of this method.

Views in HtmlFlow are built from a function of type `HtmlTemplate`, specified by the following Java functional interface:

```
interface HtmlTemplate {
  void resolve(HtmlPage page);
}
```

The `HtmlPage` instance provides the HTML *builder* methods, including next special methods, where `P` is the type of the parent HTML element and `M` is the type of the data model:

```
__()
dynamic(BiConsumer<P,M> cont)
await(AwaitConsumer<P,M> cont)
```

For example in Listing 7 the HTML fragments delimited by a continuous blue line are *static* and resolved once, on first rendering. On the other hand, the HTML fragments delimited by a dashed red line are *dynamic* and resolved on every template processing.

Former implementation of HtmlFlow used an imperative flag-based approach to control the global

```
val artistView = HtmlFlow.viewAsync { page -> page
.html().body()
 .div()
  .h3()
   .dynamic<> { h3, model ->
    h3.text(model.artistName)
   }
  .__() // h3
  .hr().__()
  .h3().text("MusicBrainz info:").__()
  .ul()
   .await<> { ul, model, cb -> model
    .musicBrainz
     .thenAccept { mb -> ul
      .li().text("Founded: ${mb.year}").__()
      .li().text("From: ${mb.from}").__()
      .li().text("Genre: ${mb.genres}").__()
      cb.finish()
     }
   }
  .__() // ul
  .hr().__()
  .b().text("Spotify popular tracks:").__()
  .span()
   .await<> { span, model, cb -> model
    .spotify
     .thenAccept { spt -> span
      .text(spt.popularSongs.joinToString(", "))
      cb.finish()
     }
   }
  .__() // span
 .__() // div
 .hr().__()
 .footer().small()
  .dynamic<> { small, model -> small
   .text("${currentTimeMillis() - m.startTime} ms
       (response handling time)")
  }
 .__().__() // small footer
.__().__() // body html
}
```

Listing 7: HtmlFlow template of Disco web application. For readability we cleared the generic argument in calls to `await<ArtistAsyncModel>` method.

state of HTML emission, and determine whether it is inside a *static*, or a *dynamic* HTML fragment.

During the initial rendering, HtmlFlow retains an internal data structure containing the HTML outcomes from each static block. In the case of the template shown in Listing 7, HtmlFlow would store the designated static blocks depicted in Figure 8. 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 provided to `dynamic` and `await` builders.

Calling the `BiConsumer` of a `dynamic` fragment uses a direct style and when it returns it proceeds emitting HTML of the following static HTML block and henceforward.

Yet, HtmlFlow cannot follow a direct style when invoking the `AwaitConsumer` of an `await` builder. The `AwaitConsumer` denotes an asynchronous function that may return before completion. Emitting the subsequent static HTML fragment upon the return of the `AwaitConsumer` call could result in the *asynchronous fragment* appearing in an incorrect order.

Listing 8: Resulting static HTML blocks data structure resuting from HtmlFlow processing.

For that reason, the `AwaitConsumer<T,M>` receives a third parameter, which is a completion callback, used to notify HtmlFlow that it can proceed rendering the next HTML fragment. This is based on the same idea of the *resume* function in continuations and coroutines (Haynes et al., 1984), to transfer control from one coroutine to another.

Given that, the new HtmlFlow rendering process is based on a chain of continuations specified by the interface `HtmlContinuation`, where each implementation is responsible for emitting a *static*, a *dynamic* or an *asynchronous* fragment, and call the next node. This technique has similarities with the way how the chain-of-responsibility design pattern (Gamma et al., 1995) assembles the logic of a series of processing objects having the responsibility, to either handle a request or forward it to the next object on the chain. In our case, each continuation has both roles handling an HTML fragment and forwarding it to the next one.

To use the template of Listing 7 we must first create an `HtmlViewAsync`, which is a container of an `HtmlTemplate` that wil make a preprocessing of the static HTML fragments and dynamic ones. Later, we may write the resulting HTML of the `HtmlViewAsync` to any `Appendable` object through its method `writeAsync(Appendable out, Object model)`. This `writeAsync` method returns a `CompletableFuture` that completes when all HTML has been emitted.

To use HtmlFlow in WebFlux we take advantage of the same `AppendableSink` described in Section 4.3. to emit the resulting HTML to a `Publisher<String>`. This *sink* should be closed when the `CompletableFuture` from the `writeAsync` completes. Given an `HtmlViewAsync` for the template of the Listing 7 named `artistView`, then the resulting `Publisher<String>` with the HTML is produced with:

```
val html =  AppendableSink { artistView
    .writeAsync(this, model)
    .thenAccept { this.close() }
  }.asFlux()
```

# 6 EVALUATION

In the initial subsection, we provide a summary of the feature comparison among the assessed template engines for Server-Side Rendering (SSR). After that we describe the testing environment, followed by our analysis of the performance results in the next section.

## 6.1 Feature Comparison

In Table 2 we highlight the characteristics of each template engine dealing with asynchronous models through a non-blocking API, according to the issues pointed in Section 3.2: 1) *limited asynchronous API*, 2) *single data model*, 3) *ill-formed HTML* and 4) *nested callbacks.*

Table 2: Comparing template engines dealing with asynchronous models.

| Library | Asynchronous Model | | 3. Valid HTML | 4. Avoid nested callbacks |
|---|---|---|---|---|
| | 1. API | 2. Multiplicity | | |
| **Thymeleaf** | `Publisher` | 1 | - | ✓ |
| **KotlinX** | Any | * | × | × |
| **HtmlFlow** | Any | * | ✓ | ✓ |

Unlike Thymeleaf, building a web template with KotlinX.html is not limited either to: 1) a specific kind of asynchronous API (i.e. `Publisher`), nor to 2) a single asynchronous data model.

However, employing KotlinX.html to construct templates with asynchronous data models can potentially result in improperly structured HTML. Furthermore, when utilized with multiple asynchronous data models, it might give rise to nested callbacks and lead to a pyramid-like source code structure.

As a DSL for HTML, HtmlFlow has the same advantages of KotlinX.html, but it also solves the issues 3) and 4). It does not suffer either from the idiomatic callback hell, nor producing ill-formed HTML.

## 6.2 Environment

To perform an unbiased comparison we used one of the most popular benchmarks for template engines, the *Comparing Template engines for Spring MVC*, simply denoted as Spring templates benchmark(Reijn, 2015). This benchmark uses the Spring MVC middleware to host a web application that provides a route for each template engine. Each route deals with the same information to fill the template (i.e. `List<Presentation>`), which makes it possible to flood all the routes with a high number of requests and asserts which route responds faster. The `Presentation` domain entity is according to the Java definition of Listing 9

```
record Presentation(
        long id,
        String title,
        String speakerName,
        String summary
) { }
```

Listing 9: `Presentation` domain entity.

The repository has 10 instances of `Presentation` and each template produces an HTML document with a table of 10 rows. The generated HTML code is approximately 80 lines long and has a size of around 9 KB.

We have implemented three modifications to evaluate template engines in a non-blocking context:

1. changed repository to return a reactive `Publisher<T>` rather than a `List<T>`.
2. replaced Spring MVC by Spring WebFlux and their controllers with functional routes (Deinum and Cosmina, 2021).
3. integrated Java Microbenchmark Harness (JMH) (Shipilev, 2013) to conduct performance tests and gather precise results.

To execute requests to the functional routes during the JMH benchmark, we utilized the Spring `WebTestClient`. This approach differed from the original Spring templates benchmark, which employed an external Apache HTTP server benchmarking tool to stress test and perform HTTP requests. By directly testing the functional routes within the same process, we eliminated the need for the external tool and excluded the HTTP communication from the performance results.

Our tests were done on a local machine running Ventura 13.3.1 on a MacBook Pro with Apple M1 Pro, 8 cores (6 performance and 2 efficiency), and OpenJDK 64-Bit Server VM Corretto-17.0.5.8.1.

## 6.3 Results

We conducted our tests in JMH by varying the number of worker threads, specifically 1, 2, 4, and 8. As the number of worker threads increased, the level of concurrent requests also escalated, allowing us to observe the scalability of each template view.

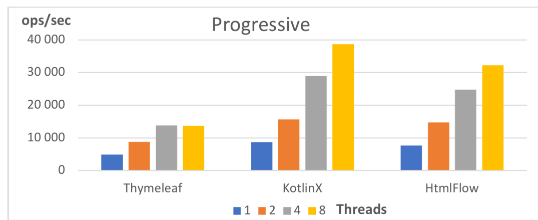



Figure 2: Throughput of Thymeleaf, KotlinX.html and HtmlFlow in Spring templates benchmark with WebFlux and JMH.

According to the results presented in Figure 2, when running these tests with progressive rendering, Thymeleaf demonstrates limitations in scalability under these conditions.

On the flip side, both KotlinX.html and HtmlFlow exhibit scalability as the number of handler threads increases.

KotlinX.html exhibits slightly better throughput than HtmlFlow. This is because KotlinX.html does not ensure well-formed HTML with an asynchronous data model, and it does not require any special care to guarantee the correct chaining of HTML elements, as HtmlFlow does. This alleviates the rendering process in KotlinX.html, resulting in slightly better performance compared to HtmlFlow.

## 7 CONCLUSIONS

Our proposal for server-side rendering Web templates on top of HtmlFlow is the only non-blocking solution that is able to deal with any number, and any kind of asynchronous data models and still produce well-formed HTML. Also, our use of CPS in asynchronous views avoids nesting callbacks, among different asynchronous models.

Now we plan to take a step further and take advantage of the async/await idiom (Syme et al., 2011) in asynchronous fragments. The async/await idiom enables writing asynchronous code that looks like synchronous code, without the need for callbacks. In Kotlin, the async/await idiom is implemented using coroutines and suspending functions. Yet, KotlinX.html builders use non-suspending functions as parameters, which means that they cannot be used directly with the async/await idiom.

Since HtmlFlow uses a method chaining approach, it is possible to extend its API through Kotlin extension functions that provide the ability to extend a class or an interface with new functionality without having to inherit from the class. Thus, can we simplify the asynchronous idiom of HtmlFlow through Kotlin extensions and async/await? Is there any intrinsic overhead on that approach? How do we compare it with present proposal? Those are some of the research questions for future work.

## REFERENCES

Alur, D., Malks, D., and Crupi, J. (2001). *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Ase, D. (2015). Kotlin dsl for html. Technical report.

Bergsten, H. (2003). *JavaServer Pages*. O'Reilly Media, Inc.

Breslav, A. (2016). *Kotlin Language Documentation*.

Carvalho, F. M. (2017). Htmlflow java dsl to write typesafe html. Technical report, https://htmlflow.org/.

Carvalho., F. M. and Duarte., L. (2019). Hot: Unleash web views with higher-order templates. In *Proceedings of the 15th International Conference on Web Information Systems and Technologies*, WEBIST '19, pages 118–129.

Chaniotis, I., Kyriakou, K.-I., and Tselikas, N. (2014). Is node.js a viable option for building modern web applications? a performance evaluation study. *Computing*, 97.

Davis, A. L. (2019). *Akka HTTP and Streams*, pages 105–128. Apress, Berkeley, CA.

Deinum, M. and Cosmina, I. (2021). *Building Reactive Applications with Spring WebFlux*, pages 369–420. Apress, Berkeley, CA.

Elmeleegy, K., Chanda, A., Cox, A. L., and Zwaenepoel, W. (2004). Lazy asynchronous i/o for event-driven servers. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, page 21, USA. USENIX Association.

Evans, B., Verburg, M., and Clark, J. (2022). *The Well-Grounded Java Developer, Second Edition*. Manning.

Fernández, D. (2011). Thymeleaf. Technical report, https://www.thymeleaf.org/.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Fox, T. (2001). Eclipse vert.x tool-kit for building reactive applications on the jvm. Technical report, https://vertx.io/.

Friedman and Wise (1978). Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289–296.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Co., Inc., Boston, MA, USA.

Haynes, C. T., Friedman, D. P., and Wand, M. (1984). Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, page 293–298.

Jin, X., Wah, B. W., Cheng, X., and Wang, Y. (2015). Significance and challenges of big data research. *Big Data Res.*, 2(2):59–64.

Johnson, R., Hoeller, J., Donald, K., Sampaleanu, C., Harrop, R., Risberg, T., Arendsen, A., Davison, D., Kopylenko, D., Pollack, M., et al. (2004). The spring framework–reference documentation. *interface*, 21:27.

Kambona, K., Boix, E. G., and De Meuter, W. (2013). 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.

Kant, K. and Mohapatra, P. (2000). Scalable internet servers: Issues and challenges. *ACM SIGMETRICS Performance Evaluation Review*, 28(2):5–8.

Karsten, M. and Barghi, S. (2020). User-level threading: Have your cake and eat it too. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1).

Klauzinski, P. and Moore, J. (2016). *Mastering JavaScript Single Page Application Development*. Packt Publishing.

Landin, P. J. (1966). The next 700 programming languages. *Communications of the ACM*, 9(3):157–166.

Mashkov, S. (2015). Kotlin dsl for html. Technical report.

Maurer, N. and Wolfthal, M. (2015). *Netty in Action*. Manning.

Meijer, E. (2012). Your mouse is a database. *Queue*, 10(3):20:20–20:33.

Qin, H., Li, Q., Speiser, J., Kraft, P., and Ousterhout, J. (2018). Arachne: Core-aware thread management. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 145–160, USA. USENIX Association.

Reactive Streams (2015). Reactive streams specification.

Reijn, J. (2015). Comparing template engines for spring mvc. Technical report, https://github.com/jreijn/spring-comparing-template-engines.

Schmidt, D. (1995). Reactor. an object behavioral pattern for concurrent event demultiplexing and event handler dispatching.

Shipilev, A. (2013). Java microbenchmark harness (the lesser of two evils).

Sussman, G. and Steele, G. (1975). *Scheme: an interpreter for extended lambda calculus*. AI Memo No. MIT, Artificial Intelligence Laboratory.

Syme, D., Petricek, T., and Lomov, D. (2011). The f# asynchronous programming model. In Rocha, R. and Launchbury, J., editors, *Practical Aspects of Declarative Languages*, pages 175–189, Berlin, Heidelberg. Springer Berlin Heidelberg.

von Behren, R., Condit, J., Zhou, F., Necula, G. C., and Brewer, E. (2003). Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 268–281, New York, NY, USA. Association for Computing Machinery.

Welsh, M., Culler, D., and Brewer, E. (2001). Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 230–243, New York, NY, USA. Association for Computing Machinery.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS