INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Reactive Web Templates

Pedro Filipe Gonçalves Fialho

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Júri:

Presidente: Doutor Nuno Miguel Soares Datia

Vogais: Doutor Fernando Miguel Gamboa de Carvalho
Doutor António Paulo Teles de Menezes Correia Leitão

Setembro, 2023

Acknowledgments

Gostaria de expressar a minha mais profunda gratidão às pessoas e instituição cuja contribuição e apoio tornaram possível a conclusão desta tese.

Em primeiro lugar, desejo manifestar a minha profunda apreciação ao orientador de tese, Fernando Miguel Gamboa de Caervalho, pela orientação, compromisso e dedicação.

Agradecer também à minha família, por todo o apoio e paciência no desenvolvimento desta tese.

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.

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

Resumo

Técnicas de renderização otimistas no lado do servidor (SSR) requerem uma thread dedicada por pedido HTTP, limitando assim o número de pedidos concurrentes às threads do servidor disponíveis. Além disso, essa abordagem mostra-se impraticável para servidores modernos com um baixo número de threads, como WebFlux, VertX e Express Node.js.

Para alcançar renderização progressiva, modelos de dados assíncronos fornecidos por APIs não bloqueantes devem ser utilizados. No entanto, este método pode introduzir uma sobreposição indesejável entre o processamento de visualização do modelo e o acesso a dados, potencialmente resultando em documentos HTML malformados.

Alguns *template engines* oferecem remédios parciais tirando partido de dialetos específicos, mas enfrentam duas limitações. Em primeiro lugar, a sua compatibilidade é restrita a tipos específicos de APIs assíncronas, como a API reativa `Publisher`. Em segundo lugar, geralmente suportam apenas um modelo de dados assíncrono.

Nesta pesquisa, propomos uma abordagem alternativa de templates web que abrange qualquer API assíncrona (por exemplo, `Publisher`, promessas, funções de suspensão, *flux*, etc.) e permite várias fontes de dados assíncronos. A Nossa abordagem é implementada usando como base o HtmlFlow, uma DSL (Linguagem de Domínio Específica) baseada em Java para escrever HTML usando forte tipificação de tipos.

Foi avaliado em servidores reativos de última geração, especificamente o WebFlux, e comparamos com idiomas populares de *templating*, como Thymeleaf e KotlinX.html. Nossa proposta supera as limitações das abordagens existentes.

Palavras-chave: Templates Web, renderização no servidor, IO não-bloquente, API assíncrona programação concorrente.

Contents

List of Figures xi

List of Tables xiii

List of Listings xv

1 Introduction 1

1.1 Motivation 2

1.2 Asynchronous APIs Matter 4

1.3 Outline of the Dissertation 6

2 State of the Art 7

2.1 **Related Work** 7

2.2 Spring SSR Basics 12

2.3 Web Templates 16

2.3.1 Thymeleaf 16

2.3.2 KotlinX SSR with Reactive Streams 19

2.3.3 Handlebars for SSR with Reactive Streams 24

2.3.4 HtmlFlow for SSR with Reactive Streams 27

2.4 Resolvers 30

2.4.1 Thymeleaf Resolvers 31

2.4.2 KotlinX Resolvers 40

2.4.3 Handlebars Resolvers 42

2.4.4 HtmlFlow Resolvers 44
2.5 Summary 46

3 Functional Reactive Templates **47**
3.1 Former HtmlFlow internal processing 47
3.1.1 Kinds of Templates 49
3.1.2 Visitors 50
3.2 DSL Proposal for asynchronous fragments 52
3.2.1 Why a new version was needed 52
3.2.2 An event-driven way of working 53
3.3 HtmlFlow asynchronous internal processing 55
3.4 Summary 60

4 Validation **61**
4.1 Disco Non-blocking web application 61
4.1.1 Thymeleaf 62
4.1.2 KotlinX.html 65
4.2 Evaluation 68
4.2.1 Environment 68
4.2.2 Results 69
4.2.3 Performance Evaluation 69
4.2.4 Memory Allocation Evaluation 70
4.3 Summary 71

5 Conclusions **73**
5.1 Main contributions 73
5.2 Future Work 74

References **75**

List of Figures

2.1 Diagram how Spring handles SSR requests from the ViewResolver to the View . 30

3.1 HtmlFlow `async then` architecture 54

4.1 Expected output from Disco web app for The Rolling Stones band. 62
4.2 Throughput of Thymeleaf, KotlinX.html and HtmlFlow in Spring templates benchmark with WebFlux and JMH. 69
4.3 Memory allocation in Kb/op for each template engine in Spring templates benchmark for single HTML emission and progressive rendering. 70

List of Tables

2.1 Comparing template views in terms of DSL approach, host language, and the ability to provide functional templates, progressive rendering and their relative performance to HtmlFlow in Spring templates benchmark. 11

List of Listings

2.1 Example of Thyemelaf tempalate 8
2.2 Nested function DSL. 9
2.3 Method chaining DSL. 10
2.4 Spring Controller annotation 13
2.5 Spring RestController annotation 13
2.6 Example of Controller to SSR 14
2.7 Example implementation of ViewResolver. numbers 14
2.8 Example view class responsible to process and render the template 15
2.9 Thymeleaf template for the vets 16
2.10 Vet and dependant object used for the templates 17
2.11 Reactive version of a Controller to use the vets template 17
2.12 Result of Thymeleaf html with reactive streams 18
2.13 KotlinX vets template 19
2.14 KotlinX Vet helper template creation 20
2.15 KotlinX helper to render fragments 20
2.16 KotlinX controller 21
2.17 KotlinX result from running the request 22
2.18 Handlerbars vets template 24
2.19 Handlerbars vets template 24
2.20 Handlebars helper to render fragments 25
2.21 Controller for Handlebars 26

2.22 HtmlFlow DSL template . . . 27
2.23 HtmlFlow dynamic consumer implementation, which is the equivalent of the Handlebars helper, as the helper from KotlinX was . . . 28
2.24 HtmlFlow partial template per vet . . . 28
2.25 HtmlFlow controller implementation . . . 28
2.26 Reactive version of the Thymeleaf ViewResolver . . . 31
2.27 Reactive version of a Controller to use the vets template . . . 31
2.28 Reactive version of how the template is rendered . . . 32
2.29 ReactiveDataDriverContextVariable is passed to the context for Thymeleaf to start the reactive processing . . . 32
2.30 Internal render of reactive fragment inside template . . . 33
2.31 Context initialization for fragment rendering . . . 34
2.32 Creation of stream based on output mode . . . 35
2.33 Creation of Flux containing each part of the HTML fragment . . . 36
2.34 Initialization of throttled template engine . . . 36
2.35 Delay of throttled processor initialization . . . 36
2.37 Merging of all the results from the many buffers . . . 36
2.36 Each phase of the reactive fragment rendering . . . 37
2.38 Create a never-ending Flux to continuations emmit data . . . 37
2.39 Control ending of Flux output . . . 37
2.40 Starting of emit HTML . . . 38
2.41 Feeds the values from the buffer to the output . . . 38
2.42 Starts emitting the end of the reactive HTML fragment . . . 38
2.43 Determines if it should finish the step or continue emmiting . . . 38
2.44 Signals the finish of the phase . . . 38
2.45 Ends the emission of data from the parent Flux . . . 39
2.46 BasicView, which is responsible for unwrapping the needed parameters and passing them to the received processor . . . 40
2.47 Extraction of response stream, buffer for the data and the write for said buffer . . 41
2.48 Creation of buffer to continuously write to the server output . . . 41
2.49 TemplateProcessor declaration . . . 41

2.50 KotlinX Processor implementation 41
2.51 KotlinX template which passes the model for template processing 42
2.52 Handlebars view resolver to fetch the template and compile it 42
2.53 Handlebars implementation of the Processor 43
2.54 Creation of the Handlebars context 43
2.55 HtmlFlow resolver implementation 44
2.56 HtmlFlow resolver implementation 44
2.57 HtmlFlow resolver implementation 45
3.1 Creation of simple HtmlFlow Template 47
3.2 HtmlTemplate interface responsible for specifying a function 48
3.3 Creation of a DynamicHtml that accepts models 49
3.4 Creation of template that is completed by a partial 49
3.5 Usage of partials inside a template 50
3.6 Thenable interface, definition 53
3.7 Await operation unwrapped 54
3.8 HtmlContinuation 56
3.9 HtmlContinuationSync execute 56
3.10 HtmlContinuationAsync execute 56
3.11 Call to preprocessing when viewAsync is called 57
3.12 Asynchronous implementation of chaining of continuations 57
3.13 Html template writeAsync 58
3.14 HtmlFlow usage with new version 59
3.15 Result of the new HtmlFlow reactive approach 60
4.1 Expected HTML source code for the web page of Figure 4.1c 63
4.2 Disco domain entities definition in Kotlin for `MusicBrainz` and `SpotifyArtist`. 63
4.3 Example of Thymeleaf template of Disco web application. 64
4.4 Example of KotlinX.html template of Disco web application. 66
4.5 Example of ill-formed HTML source resulting from undesirable interleavings between template processing and asynchronous data access. 67
4.6 `Presentation` domain entity. 68

1

Introduction

The research work that I described in this dissertation is concerned with the challenge of dealing with asynchronous data models [3, 21] in web applications with server-side rendering (SSR) [32]. Thus, the main goal is to provide a solution to make SSR work with asynchronous data models and achieve progressive rendering. The web applications that already work with SSR, would be able to work with asynchronous data models of any type of asynchronous API (e.g. reactive streams, flows, async-await, suspend functions, etc.), without the need to completely rework their entire infrastructure. Accounting for this new way of working with models.

We will explore the driving factors behind this work, primarily by examining the existing challenges within the current state-of-the-art related to using SSR with asynchronous models. Following this, we will provide an initial overview of our proposed solution aimed at addressing the identified problems.

1.1 Motivation

At the moment, Web Templates are a huge part when developing Web Applications. Although we see more and more Single Page Applications (SPA's) putting together techniques like client-side rendering (CSR), a huge part of Web Applications still rely on SSR. SSR is known for its simplicity in usage, how fast it can be implemented and in some cases even reduce some client workload.

It's also true that when Web Apps are being developed, the usage of Asynchronous APIs are becoming predominant. There is no way to hold the generation of a table for later while compiling the rest of the template and guaranteeing the validity of HTML regarding the template definition. This causes a major issue, as we need to block the Asynchronous APIs which forces the client, in this case, the browser, to be blocked while waiting for the response. This behavior is not acceptable to a user of a website. Another option would be to continue to use SSR with a traditional server-side blocking thread-per-request architectures.

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, meaning that the number of threads in the system can grow quickly as the load increases. Usually, there is a pool of threads that deals with thread creation and assignment. This can lead to performance issues and scalability problems, as the system can become bogged down by context-switching and thread overhead [17].

Another thing to consider is that a server has limited amounts of memory and CPU.

If we assume that one thread costs 1 MB, in a case where we have 1000 concurrent requests being made to the server there can be 1000 threads running simultaneously. Which can potentially correspond, to 1 GB being used just on threads.

Although even if with the current memory costs, 1 GB of memory is not that expensive, we need to consider the implications of such implementation.

This amount of threads running, and the overhead that comes with it, makes it very difficult to scale the server to support a high load of requests. A good server-side implementation should be flexible enough to be able to scale at demand. As more threads are running more resources the server will consume, which makes it very expensive to scale an application to serve more requests.

Another downside of this implementation is how hard it is to predict the amount of resources used, if we have a server running under the thread-per-request model, it can very quickly (under a lot of load) take a huge amount of resources, as we already discussed it.

In a world where microservices are imperative, we try to take as little, in terms of resources, as possible for each service. The uncertainty of the amount of resources a service can take makes it difficult to create a scalable architecture all around the board.

On the other hand, in modern *low-thread* architectures, the server uses a few threads to handle many requests. The server can handle many concurrent requests without blocking or slowing down the system, which makes it more scalable and efficient [27]. To that end, it uses non-blocking I/O to ensure that each thread can handle multiple requests at the same time. Non-blocking I/O means that the main thread returns immediately, indicating that the background processing was successfully initiated. The application can then perform other processing while the background operation completes. When the read response arrives, a signal or a thread-based callback can be generated to complete the transaction.

The non-blocking I/O model used in low-thread servers is well-suited for handling large volumes of data asynchronously [21]. The combination of low-thread servers and asynchronous data, models have enabled the development of highly scalable, responsive, and resilient Web applications that can handle a high volume of data [14].

The combination of all the components above presented is what we call a Reactive Web Server. This idea of a Reactive Web Server was already brought up a long time ago [26] where this concept of a server that can handle multiple requests at the same time, by taking advantage of non-blocking I/O is explained.

This idea gained significant attention with the rise of Node.js in 2009 [23], and later, several technologies embraced this approach in Java ecosystem namely, Netty [22], Akka HTTP [1], Vert.X [36] and Spring WebFlux [15], being the latter the most widely used middleware in Java Web applications 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* [2]), and *data repositories* [9], but may also concern the Frontend when it uses an SSR approach [2], 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) [30] becoming a popular choice for building Frontend, in modern Web applications, 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 suppresses 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* [5]; 2) resume callback in continuations [6] to control transitions from asynchronous handlers, and 3) chain-of-responsibility design pattern [11] to control the flow between a chain of template fragments.

1.2 Asynchronous APIs Matter

In the context of a Reactive Web Server, most of the options for the server-side to support this approach, described above, take advantage of the reactive programming approach.

This is a paradigm in which declarative code is issued to construct asynchronous processing pipelines. In other words, it's a programming technique where asynchronous data streams send data to a consumer as it becomes available, which enables developers to write code that can react to these state changes quickly and asynchronously.

One way to describe such asynchronous processing pipelines is through Reactive Streams.

Streams perform exceptionally well in the world of Reactive Programming due to their behavior and usability.

A stream is a sequence of ongoing events (state changes) ordered in time. According to the Reactive Streams standard, a stream can emit three different things: a value (of some type), an error, or a `completed` signal. The events are captured asynchronously, by defining a function that will execute when a value is emitted, another function when an error is emitted, and another function when `completed` is emitted. `Listening` to the stream is called subscribing. The functions we are defining are observers.

Streams are also composable, being able to build functions one on top of the other gives the programmer the possibility to chain events to be executed on a certain stream of elements.

Reactive Streams are an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols.

The main goal of Reactive Streams is to govern the exchange of stream data across an asynchronous boundary – like passing elements on to another thread or thread-pool – while ensuring that the receiving side is not forced to buffer arbitrary amounts of data.

We can then conclude that Reactive Streams take full advantage of a non-blocking IO model, to provide the so-wanted behavior for a Reactive Web Server.

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 are 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) [28], async/await idiom [3], reactive streams [25], Kotlin Flow [19], and others.

The correct use of an asynchronous API generally requires following established patterns and idioms for the particular programming language or framework being used. This can help to ensure that code is efficient, reliable, and maintainable. On the other hand, naively dealing with asynchronous data models, 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, and 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 the concept of Promise [24], this may be achieved by getting its result from its method `join(): T`.

This blocking approach can be used by any template engine, including those analyzed in this work (i.e. Thymeleaf, KotlinX.html and HtmlFlow) with the same ill effect. These kinds 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.

The progressive behavior of rendering the static parts of the web template first and then asynchronously rendering the dynamic parts as they complete gives the end-user a feeling of responsiveness and progress. It helps to avoid a situation where the user is left with an empty page that appears to be stuck or unresponsive. By using this approach, the user can see that the page is being loaded and something is happening in the background, which can improve their overall experience.

Furthermore, in the blocking approach, while a 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 ....`

Another example is the Eclipse Vert.x for JVM [36], which includes a low-thread server based on Netty and in similar blocking situations may throw an exception.

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.

1.3 Outline of the Dissertation

For the remainder of this paper, we will highlight in the following sections some issues arising from the naive use of asynchronous techniques in web templates. Then in Chapter 2 we describe state-of-the-art template engines for SSR and compare different asynchronous template dialects. In Chapter 3 we discuss their behavior in a case study of a Spring WebFlux application, followed by Chapter 4 where we propose how to mitigate the found limitations. Next, Chapter 5 presents a performance evaluation benchmark comparing different template engines dealing with asynchronous data models in a WebFlux web application. We conclude and discuss some future work in Chapter 6.

2

State of the Art

In this chapter, we will describe some concepts of the Spring framework and how it was built to support Web Templates. We will then move on to how Thymeleaf can handle Web Templating with Reactive Streams. Then we will compare this approach with other Template Engines such as Handlebars and KotlinX, which have no internal support for reactive data models. For the Thymeleaf examples, we used the demo project created by the Spring team Petclinic. To further demonstrate how and where other technologies fail at Web Templating with Reactive Streams, a demo under the scope of a Petclinic was also developed.

2.1 Related Work

Templates are pieces of software that provide a generic way to create a certain resource, where certain parts can be generated dynamically by filling it with information that comes from outside sources, such as a database. A Web Page is a simple document that can be displayed by a browser. These documents are written in HTML.

A Web Template is when we combine the Web Page idea with the template approach in order to generate a Web Page, where HTML represents the static parts of the website, together with some dynamic tags representing information that is not available and will be filled by some external sources at runtime.

When we use a Web Template to render an HTML document on the server that is called SSR (Server side rendering), which is the approach followed by engines such as Thymeleaf, JSP and Handlebars. For example, SSR generates the full HTML for a page on the server in response to client navigation (e.g. HTTP request).

Thymeleaf [35] is the default *template view* engine in Spring WebFlux middleware. Thymeleaf has a powerful expression language that allows developers to manipulate data in their HTML templates. Also, Thymeleaf is the only Spring built-in view engine that supports asynchronous data models with progressive rendering [37], 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 other built-in view engines, such as Freemarker, Handlebars, Jade or JTwig produces an unresponsive blank page while awaiting data completion. On the other hand, JSP [16] is not supported by WebFlux.

We can use as an example of what is a Web Template this Thymeleaf file.

```
1  <tbody>
2    <tr th:each="item : ${items}">
3      <td th:text="${item.name}|">[name]</td>
4    </tr>
5  </tbody>
```

Listing 2.1: Example of Thyemelaf tempalate

We can see here multiple tags that Thymeleaf uses to represent certain actions. For example, `${...}` can access object information such as fields. Whereas `th:each` represents the beginning of an interaction of something that has multiple objects inside, like a list.

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) [20]. DSLs can be divided into two types: *external* or *internal* [10]. *External* DSLs are languages created without any affiliation to a concrete programming language. An example of an external DSL is the regular expression search pattern [34], since it defines its syntax without any dependency of programming languages. On the other hand, an internal DSL is defined within a host programming language, and it relies on the language's syntax and constructs to define the DSL. For that reason, internal DSLs can also be referred to as embedded DSLs, since they are embedded in the programming language where they are used.

DSLs for HTML allow defining templates for generating HTML directly within the host language, rather than using textual template files, which enables the use of functional templates, that are templates defined using higher-order functions [5]. Using a Java DSL can have several benefits over using textual templates:

1. Type safety: Because the templates are defined in Java code, the compiler can check the syntax and types of the templates at compile time, which can help to catch errors earlier in the development process.

2. IDE support: Many modern IDEs provide code completion, syntax highlighting, and other features for working with Java code, which can make it easier to write and maintain templates.

3. Flexibility: Use all the features of the host programming language to generate HTML can make it easier to write complex templates and reuse code.

4. Integration: Because the templates are defined in Java code, it can easily integrate them with other Java code in the application, such as controllers, services, repositories and models.

j2Html [13], KotlinX.html [18] and HtmlFlow, are example of Java libraries DSLs for HTML. In common these DSLs use functions to define their languages. According to to [10] 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 2.2 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 backward to the order they are written. This means that arguments are first evaluated before the function is invoked. In Listing 2.2, `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.

```
1  html(
2    head(title("My title"),
3      body(
4          div(p("A paragraph"))
5      )
6    )
7  )
```

Listing 2.2: Nested function DSL.

```
1 Html().head().title("My title").body().div().p("A paragraph")
```

Listing 2.3: Method chaining DSL.

The backward evaluation behavior may incur 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 function calls order. Otherwise, the HTML would be generated backward. Thus, it may require some sort of auxiliary data structure to manage elements processing and controlling HTML emissions, 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 with receiver, which is the object of the method being called on [7]. 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 2.3, the result of `Html()` call is an HTML element that will be the receiver for the next `head()` call, which in turn produces a Head element that will be the receiver for the next `title()` call, and henceforward.

J2html uses a *nested function* approach where templates have a similar layout to that one presented in Listing 2.2. The result of the execution of a j2Html template is a tree structure composed of `Tag` objects[11]. 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, uses function literals (i.e. lambdas) to represent HTML tags and attributes. By using function literals as arguments, KotlinX.html can 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 2.3. 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 numerous objects representing HTML nodes or elements.

In Table 2.1 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 [31]. These results are relative to HtmlFlow throughput, which is the most performant engine among the evaluated Web Templates.

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

Table 2.1: Comparing template views in terms of DSL approach, host language, and the ability to provide functional templates, progressive rendering and their relative performance to HtmlFlow in Spring templates benchmark.

DSLs that are designed to work with Java can also be used in Kotlin without any issues. Similarly, DSLs designed to work with Kotlin can also be used in Java, although some Kotlin-specific syntax may not be available in Java.

However, the use of asynchronous data models in Web Templates can introduce new issues, namely:

1. limited asynchronous idiom;
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 Chapter 1. Additionally, Web Templates may only support a limited asynchronous idiom, which can put a limitation on application development. This is especially true when provided API 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, resulting in an ill-formed HTML document.

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.

Finally for Web Templates dealing with data models through the continuation-passing style (CPS) [28] 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" [4].

In the next chapter, we will have a look at how Spring can handle multiple Web Templates, and how each one can be integrated into the Spring Framework.

2.2 Spring SSR Basics

In this section, we will discuss how Spring handles Web Templating within the world of an SSR approach.

Spring framework is one of the most widely used frameworks for web development by Java & Kotlin developers, and having such an easy way to get started and using some of it's potential, makes it a special target for the usage of SSR when using Java and/or Kotlin.

The Spring Framework is built upon the Model-View-Controller (MVC) [8] pattern. The same architectural approach is followed in the SSR implementation. As a result, developers are required to adhere closely to the principles of MVC and apply them diligently.

MVC is a pattern in software design commonly used to implement user interfaces, data, and controlling logic. It emphasizes a separation between the software's business logic and display.

The Model is the part described as the layer that is responsible for giving the Controller the necessary information to execute some action.

The Model is completely independent of the Controller or the View. There could be a data source such as SQL DB, NoSQL DB or fetching some information from a `Web Server` to serve the data that is going to be used to render the template.

The Controller does not need to know the implementation of the Model, all it needs it's the information. This usually means hiding the Model behind an Interface. Asks for the data to the Model, and controls the flow of the application for the user, who is waiting for user input in order to execute some action. After the information is gathered from the Model, it is sent to the View which has a similar role to a Web Template, which renders what the Controller wants to present to the caller of the Service.

In the context of SSR, the View is responsible for rendering a certain HTML page with the data fetched from the Controller through the Model.

For the controller to be recognized by Spring to be used for SSR we need to apply the annotation

```
1 @Controller
```

Listing 2.4: Spring Controller annotation

to the controller class created.

This annotation should be used instead of broadly used:

```
1 @RestController
```

Listing 2.5: Spring RestController annotation

Due to the `RestController` annotation returning the response of an endpoint directly, whether it's a simple String or a JSON, whilst using `Controller` Spring tries to resolve this to a View if there is one.

If there is no specified behavior for handling templates and SSR, the method will return the value like a `RestController` does.

We can then assert that the `RestController` annotation removes the behavior of SSR.

The expected behavior for a Spring Controller that wants to implement SSR is to return a string containing the view name, and extension if applicable, in order for it to be resolved by the View part.

As we can see in this example,

```
@Controller
public class ExampleController {
  ...
  @GetMapping(value = {"/template"})
  public String getAllItems(Model model) {
    List<Item> items = new Items(itemService.findAll());
    model.addAttribute("items", items);
    return "template";
  }
}
```

Listing 2.6: Example of Controller to SSR

the List of Items is being put inside the View Model for the View to be able to access it. And the return of the Controller is the location of the template view inside the resource's folder.

Then, this return will be given to the View part of the spring framework to be resolved.

Inside the Spring framework, the View part is split into two parts: ViewResolver Handed the task to resolve a View instance based on a view name. View interface responsible for processing the Web Template with the parameters from the View Model.

ViewResolver is the component responsible to resolve a View based on a viewName, which is the return from the Controller method.

The way Spring knows how to call the created ViewResolver is by going through all the registered ViewResolvers.

This ViewResolver **must** either return an instance of a View that can render the template that is given by the viewName, or return an absence of value to indicate that this ViewResolver cannot provide a View to handle such a viewName.

```
@Override
protected View createView(final String viewName, final Locale locale) {
  if (!canHandle(viewName, locale)) {
      return null;
  }
  return ExampleView();
}
```

Listing 2.7: Example implementation of ViewResolver. numbers

In the case of an absence of a value, Spring will move on to the next ViewResolver until either one returns an instance of a View or, they all return the equivalent of an absence of a value.

If no `ViewResolver` can solve the `viewName` provided, then Spring will interpret the returned String as REST return, having the controller return the `String` as it is.

In a case where the `ViewResolver` can return an instance of a View, the View is the component responsible for rendering a template given a Map of parameters, and an instance of the current ongoing ServerRequest.

The Map parameters of a view are the result of unwrapping the Map inside the View Model, which contains all the objects and variables added by the controller to be possible to render this specific view.

Here we can see an implementation of the view instance that can render the `template` used in the controller above.

```
public class ExampleView implements View {

  public void render(Map model, HttpServletRequest request, HttpServletResponse response) {
    var templateString = this.generateResult(model);
    response.getWriter().write(templateWriter.toString());
    response.getWriter().flush();
  }
}
```

Listing 2.8: Example view class responsible to process and render the template

In line 4, we can notice the immediate call to `generateResult`, to render the entire template as one.

To render the entire template, the following lines 5 and 6 will write the resulting template into the `response writer` and flush the `writer`, so the result appears in the browser.

This MVC framework, that Spring created to work with templates still holds when working with Spring WebFlux and Reactive Streams. Spring just updated the return values from `X` to `Mono<X>` to work with Reactive Programming and Reactive Streams, we will look at how Thymeleaf implementation looks like for the Reactive approach as well, to show how other frameworks like Handlebars, KotlinX and HtmlFlow would look like.

It was already shown what an architecture for working with templates inside Spring looks like. Now we will present the current most used Web Templates for SSR with Spring while taking a look at how they work with SSR with an asynchronous model.

2.3 Web Templates

In this section, we will take a look at how they can work with SSR using only one asynchronous model, using Spring WebFlux to get the reactive behavior. This exploration will help us to understand the available options when working with these templates within the context of an asynchronous model.

We'll begin by examining Thymeleaf, as it serves as the base template engine for a Spring application. As we will discover, Thymeleaf is currently the only one capable of achieving progressive rendering with asynchronous models.

Subsequently, we will proceed to explore KotlinX, Handlebars, and finally, HtmlFlow.

2.3.1 Thymeleaf

We will now, first and foremost, take a look at the template we will be using.

```
...
<body>
<table>
<thead>
<tr>
  <th>Name</th>
  <th>Specialties</th>
</tr>
</thead>
<tbody>
<tr th:each="vet : ${vets.vetList}">
  <td th:text="|${vet.firstName} |">[firstName]</td>
  <td>
    <span th:each="specialty : ${vet.specialties}"> </span>
    <span th:if="${vet.nrOfSpecialties == 0}">none</span>
  </td>
</tr>
</tbody>
</table>
...
</body>
...
```

Listing 2.9: Thymeleaf template for the vets

Notice in line 11 that we are just iterating a list of vets and adding the many properties of a *Vet* to the dynamic part of the template.

The `Vet` class is what contains all the information that will be used for all the templates.

```
1  public class Vet {
2    private UUID id;
3    private String firstName;
4    private String lastName;
5    private Set<VetSpecialty> specialties;
6
7    ...
8  }
9
10 public class VetSpecialty{
11   private String id;
12   private String name;
13
14   ...
15 }
```

Listing 2.10: Vet and dependant object used for the templates

Notice in line 5 that there is a dependent object `VetSpecialty` which are all the specialties that a `Vet` can have. This dependent class is shown in line 10.

This *Vet* object reaches the template because the controller inserted it into the context of the request.

If we take a look at the controller,

```
1  @Controller
2  public class VetReactiveController {
3    (...)
4    @GetMapping(value = {"/vets"})
5    public String getAllVets(Model model) {
6      Flux<Vet> currentVets = vetServices.findAllVets();
7      var rx = new ReactiveDataDriverContextVariable(currentVets, 1);
8      model.addAttribute("vets", rx);
9      return "vets/vetList";
10   }
11 }
```

Listing 2.11: Reactive version of a Controller to use the vets template

Notice in line 1 that a controller in the Spring WebFlux framework also has to have `@Controller` annotation, we can see that the method still returns a string representing the Thymeleaf template file location and the controller also pushes the needed parameters into the Model.

The noticeable difference is in the implementation of the classes that are being put inside the Model.

The addition of a `IReactiveDataDriverContextVariable` in line 7, is needed for two reasons:

The first reason is for the View implementation to know how to handle reactive streams, as Thymeleaf looks for the presence of this interface in the Model. So passing the interface to the Model is needed to signal the correct mode, reactive processing, to execute the template processing.

The other reason why we need to use this interface is when any asynchronous variable, is sent to the Model, to be added to the Map, the `Marshelling` process will end up having to block the asynchronous type from being present. Having the wrapper, we can then guarantee the serialization **does not** interfere with the asynchronous variable.

The internals on how the `IReactiveDataDriverContextVariable` works will be more explored in the following section 2.4.

We should now see how Thymeleaf behaves when we ask for the resulting template. So, if we make an HTTP request, the resulting HTML is the following.

```
<html>
<head>
   ...
</head>
<body>
<h2>Veterinarians</h2>
<table>
    <thead>
    <tr><th>Name</th><th>Specialties</th></tr>
    </thead>
    <tbody>
    <tr><td>Helen Leary</td><td>radiology</td></tr>
    <tr><td>Henry Stevens</td><td>radiology</td></tr>
    <tr><td>Linda Douglas</td><td>surgery,dentistry</td></tr>
    <tr><td>James Carter</td><td>none</td></tr>
    <tr><td>Sharon Jenkins</td><td>none</td></tr>
    <tr><td>Rafael Ortega</td><td>surgery</td></tr>
    </tbody>
</table>
    ...
</body>
</html>
```

Listing 2.12: Result of Thymeleaf html with reactive streams

We will see that the HTML is properly formed, as it should, although we are using outside asynchronous sources.

What we mean by well-formed is, that the tags of the table that are contained inside the body tags still come as such when the HTML comes in the response.

We can then conclude that Thymeleaf has support for reactive streams and asynchronous models.

Instead of using Thymeleaf to render the template, this version uses Handlebars, KotlinX and HtmlFlow to render the template, but it is still built upon the Spring WebFlux framework.

2.3.2 KotlinX SSR with Reactive Streams

We'll begin by looking at Thymeleaf. Afterward, we'll move on to examining KotlinX. Similar to Thymeleaf, our approach will involve starting with the template, followed by an exploration of the controller, and finally, observing the outcome when requesting the resulting HTML.

For KotlinX, we initiate our exploration with the template. This template is fashioned using the KotlinX DSL.

In the case of KotlinX, due to its template being created with DSL syntax, developers must define their helpers or other components as they see fit when creating their templates.

For this demonstration, the decision was made to include a helper. This helper is generated and registered during the creation of the template using the DSL syntax.

A Helper refers to a small portion of HTML responsible for managing the dynamic aspects of the HTML content.

Thus, when we examine the KotlinX template, we can see:

```
val vets = { model : Map<String,Any> -> createHTML()
  .html {
    head {title(content = "Petclinic")}
    body {
      div {
        div { p { this.title = "Petclinic" } }
        div { helpProcessAsyncVets(model) }
      }
      hr { }
      footer {
        div {
          span{ this.text("Powered by Petclinic") }
          img(src = "/img/dog_footer.png", alt = "dog footer") {
              this.height = "410" }
        } //div
      } //footer
    } //body
  } // html
}
```

Listing 2.13: KotlinX vets template

As can be seen in line 7, the registering of the helper, another function was created that contains a portion of HTML just for dynamic parts of this template.

Helper DSL for each item,

```
val VET_HTML: (vet: Vet) -> String = { vet -> createHTML()
  .div {
    div {
      div {
        h5{this.text(vet.firstName)}
        h6 {this.text(vet.lastName)}
        p{this.text("Specialities ${vet.specialties}")}
      }
    }
  }
}
```

Listing 2.14: KotlinX Vet helper template creation

Notice that the partial template is just adding each property of the `Vet` type into HTML tags, from line 4 to line 6.

Now we take a look at the method `helpProcessAsyncVets` which is called in the template.

```
fun helpProcessAsyncVets(model: Map<String, Any>) = run {
  val vets : Flux<Vet> = (model["vets"]).get()
  val writer : OutputStreamWriter = model["writer"]
  val subscriber : MonoSink<DataBuffer> = model["subscriber"]
  val buffer : DataBuffer = model["buffer"]

  vets
    .map { VET_HTML(it) }
    .doOnNext {writer.append(it)}
    .doOnComplete {
        writer.flush()
        subscriber.success(buffer)
    }.subscribe()
}
```

Listing 2.15: KotlinX helper to render fragments

On line 8, the partial template is called per each `vet` inside the Vets Flux, which will enforce the partial template to be created per item inside the Flux.

In line 1 by using the `run` block, we can guarantee that the DSL creates blocks at this point for the processing of the reactive stream is acknowledged and submitted.

The way this helper works and can handle reactive streams is by pushing each item to the browser, without the browser closing the connection. This can be achieved by having a reference to a `subscriber` and keeping it from reaching a success state until we finish the processing.

This `MonoSink<DataBuffer>` will serve only to signal when the reactive has been read and processed, so the `ServerWebExchange` can close the connection. That's why it's called on the `doOnComplete` after we flush the `writer`.

We need to pass the `buffer` as we signal the completion of the processing, because this buffer is what the `ServerWebExchange` uses directly to write into the browser, the writer is just a

wrapper for the `DataBuffer` where we are pushing the template processing for the helper, that's being applied to each item in the `ReactiveBox<Vet>` stream which was added back in the controller.

Going to the controller, where the information is grabbed:

```
@Controller
public class VetsController {
  (...)

  @GetMapping("/kotlinx")
  public String vetsKotlinx(Model model) {
    Flux<Vet> vets = vetsService.getAllVets();
    model.addAttribute("vets", new ReactiveBox<>(vets));
    return "vets.kotlinx";
  }
}
```

Listing 2.16: KotlinX controller

It can be immediately noticed that we also have to have a wrapper (`ReactiveBox`) for the `Flux` (in line 7), just like the Thymeleaf version. If we add the stream directly to the Model, the `Flux` would be blocked and transformed into a `List` when it came time to use it in the template. This would remove the reactive and asynchronous approach we are trying to test, so it's very important to keep the same behavior as Thymeleaf.

So if we request the resulting HTML,

```
<html>
...
<body>
<div><div><p>Petclinic</p></div>
<div></div>
</div><hr>
...
</body>
</html>
<div>
  <div>
      <div>
        <h5>Rafael</h5><h6>Ortega</h6><p>Specialties: [VetSpecialty(surgery)]</p>
      </div>
  </div>
</div>
<div>
  <div>
    <div>
      <h5>Helen</h5><h6>Leary</h6><p>Specialties: [VetSpecialty(radiology)]</p>
    </div>
  </div>
</div>
<div>
  <div>
    <div>
      <h5>James</h5><h6>Carter</h6><p>Specialties: []</p>
    </div>
  </div>
</div>
<div>
  <div>
    <div>
      <h5>Henry</h5><h6>Stevens</h6><p>Specialties: [VetSpecialty(radiology)]</p>
    </div>
  </div>
</div>
<div>
<div>
  <div>
    <h5>Linda</h5><h6>Douglas</h6><p>Specialties: [VetSpecialty(surgery), VetSpecialty(
    dentistry)]</p>
  </div>
</div>
</div>
<div>
<div>
  <div>
    <h5>Sharon</h5><h6>Jenkins</h6><p>Specialties: []</p>
  </div>
</div>
</div>
```

Listing 2.17: KotlinX result from running the request

As it can be seen, the *html* tag ends at line 11, but only after that the result from the `Flux` is emitted to the HTML resulting output. Also notice in lines from 5 to 7 the empty div, where the result should be located.

This is because the remaining template is static and, therefore can be output immediately. The same cannot be said for the `Flux` source, for which we need to wait for each element that comes to emit the corresponding HTML.

We can then conclude that KotlinX does not support SSR with reactive streams.

Next, we will take a look at the Handlebars implementation.

2.3.3 Handlebars for SSR with Reactive Streams

Handlebars implementation for SSR with reactive streams is fairly similar to what we have already seen in the 2.3.2 subsection.

Starting by looking at the template `vets.hbs`,

```
<html>
<head>
  <title>Petclinic</title>
</head>
<body>
<div>
  <div>
    <p>Petclinic</p>
  </div>
  <div>
    {{{ vetsAsync this}}}
  </div>
</div>
<hr>
<footer>
  <div>
    <span>Powered by Petclinic</span>
    <img src="/img/dog_footer.png" height="410" alt="dog footer">
  </div>
</footer>
</body>
</html>
```

Listing 2.18: Handlerbars vets template

Notice in line 12, the call to a `vetsAsync this`, this is calling a partial template which can be processed by a `Helper`. The `this` is the current `context` inside Handlebars.

Here we can see the definition of the helper template `vet.hbs`.

```
<div>
  <div>
    <div>
      <h5>{{firstName}}</h5>
      <h6>{{lastName}}</h6>
      <p>Specialties: {{specialties}}</p>
    </div>
  </div>
</div>
```

Listing 2.19: Handlerbars vets template

Again, just like KotlinX noticed from lines 4 to 6 it's just using the properties of the `Vet` object into HTML tags.

The main template processing is handled by `Handlebars` itself, but the helper processing needs to be provided by the developer.

`Handlebars` knows that we have registered a helper by adding it in the controller, but the helper must implement the interface `Helper`, as can be seen:

```java
public class AsyncVetsHelper implements Helper<Map<String, Object>> {
  (...)
  @Override
  public CharSequence apply(Map<String, Object> model, Options options) {
    Flux<Vet> vets = model.get("vets").get();
    OutputStreamWriter writer = model.get("writer");
    MonoSink<DataBuffer> subscriber = model.get("subscriber");
    DataBuffer buffer = model.get("buffer");
    vets
      .map(applyTemplate())
      .doOnNext(view -> writer.append(view))
      .doOnError(Throwable::printStackTrace)
      .doOnComplete(() -> {
        writer.flush();
        subscriber.success(buffer);
      })
      .subscribe();
    return null;
  }
  (...)
}
```

Listing 2.20: Handlebars helper to render fragments

The `Helper` interface is what defines a `Helper` for consideration when handlebars are processing the main template. Handlebar knows how to connect the class to the correct helper by using the string passed as key, in this case `vetsAsync` when registering the `Helper`.

This logic is the same as the KotlinX one, except for line 18 where we return null. This is because it will not return any `CharSequence` due to being written directly into the output using the `writer` in line 11.

The logic on how the helper works with reactive streams is the same as the one given in the 2.3.2. Given that we used the same base implementation and logic for both all the parameters above `writer, subscriber and buffer` have the same type and behavior.

Taking a look at the controller for the handlebars,

```
1  @Controller
2  public class VetsController {
3    (...)
4    @GetMapping("/")
5    public String vets(Model model) {
6      Flux<Vet> vets = vetsService.getAllVets();
7      getHandlebars().registerHelper("vetsAsync", new AsyncVetsHelper(...));
8      model.addAttribute("vets", new ReactiveBox<>(vets));
9      return "vets.hbs";
10   }
11 }
```

Listing 2.21: Controller for Handlebars

It can be immediately noticed that we also have to have a wrapper for the `Flux` (in line 8), just like the KotlinX and Thymeleaf versions.

For the handlebars' endpoint, it can be noticed that we have to register a `Helper` with the name `vetsAsync`, as it was noted in the helper showcase.

This helper function will force the connection, to the browser, to be kept open and will be processing a small template and push each item from the stream to the browser for each `onNext(X)` function.

The reason we need the helper function is because, currently, since Handlebars does not provide any support for handling reactive streams, while using an iterative block, the content would be pushed to the browser as the template is being read. So, there would not appear any item as the `Flux` would most likely still not be finished.

That's why there was a need to create a helper function that would push a separate template for each item in the stream, as each item in the `Flux` signals completion.

If we run the request to see the resulting HTML, we will see the same result as 2.17

Just like KotlinX, we can see that the HTML tag ends at line 11, but only after that the result from the `Flux` is emitted to the HTML resulting output.

We can then also conclude that Handlebars do not support SSR with reactive streams.

Finally, we will look at how HtmlFlow, in its current state, behaves for SRR with Reactive Streams.

2.3.4 HtmlFlow for SSR with Reactive Streams

As KotlinX is a Kotlin DSL, HtmlFlow is a Java DSL to write typesafe HTML documents in a fluent style. The HTML is generated through the creation of a template. Templates are expressed in an internal DSL, meaning Java code is written to produce the template. This implies that whilst creating the template we can use the full Java toolchain.

Here we will present how HtmlFlow works for SSR with Reactive Stream, given the present version.

Just like the previous templates, we will start showing the template followed by the controller.

Like KotlinX but unlike Handlebars, since this is a DSL creating the HTML template, the helpers (or partials) are declared while we are creating the template.

So now looking at the template created through HtmlFlow DSL,

```
1  public void template(DynamicHtml<Map<String, Object>> view, Map<String, Object> model) {
2    view
3      .html()
4        .head()
5          .title()
6            .text("Petclinic")
7          .__() //title
8        .__() //head
9        .body()
10         .div()
11           .dynamic(asyncTableProcessing.consumeAsyncStream(model))
12           .br().__()
13         .__() //div
14         .footer()
15           .div()
16             .span().text("Powered By Petclinic").__()
17               .img().attrSrc("/img/dog_footer.png").attrAlt("dog footer").__()
18           .__() //div
19         .__() //footer
20       .__() //body
21     .__(); //html
22 }
```

Listing 2.22: HtmlFlow DSL template

Although this is the same HTML that was used in previous templates, it's important to notice the `dynamic` method used in line 10.

The function *dynamic* allows us to specify a `Consumer` that would be called upon the rendering of the template.

This `Consumer` represents an action that is being taken upon the current HTML *tag* we currently are.

This `consumer` will then be consumed, once the *render* method is called.

This `asyncTableProcessing.consumeAsyncStream(model)` is what constructs the helper part of this template which will handle the processing of the `Flux` inside the `model` parameter.

```
public Consumer<Div> consumeAsyncStream(Map<String, Object> model) {
  return __ -> {
    Flux<Vet> vets = model.get("vets").get();
    OutputStreamWriter writer = model.get("writer");
    MonoSink<DataBuffer> subscriber = model.get("subscriber");
    DataBuffer buffer = model.get("buffer");
    vets
      .map(HtmlFlowAsyncProcessing::template)
      .doOnNext(writer::append)
      .doOnError(Throwable::printStackTrace)
      .doOnComplete(() -> {
        writer.flush();
        subscriber.success(buffer);
      })
      .subscribe();
  };
}
```

Listing 2.23: HtmlFlow dynamic consumer implementation, which is the equivalent of the Handlebars helper, as the helper from KotlinX was

Lines 3 to 6 serve to fetch the needed values from the model which is the map passed all the way from the controller (containing the `Flux<Vet>`).

The processing of the `Flux` is pretty straightforward as it was for the other templates. But we should notice in line 8 the calling to a template method. This method has the partial HTML that will be filled per `Vet` inside the `Flux`.

```
private static String template(Vet vet) {
  return StaticHtml
        .view()
          .div().div().div()
                .h5().text(vet.getFirstName()).__()
                .h6().text(vet.getLastName()).__()
                .p().text(format(vet.getSpecialties())).__()
          .__().__().__() //divs
        .render();
}
```

Listing 2.24: HtmlFlow partial template per vet

Notice the usage here of `StaticHtml` instead of a `DynamicHtml` (line 2). Although we are using a model here, we do not want to render this template entirely after it has been processed. And we should remember that this is being created inside a `DynamicHtml`, so this will be processed again per request in the controller (because, per request, there is a new `View` being generated).

The usage of the `vet` param is the same as the other templates through lines 3 to 5 we are just adding the properties to HTML tags.

We can now see the controller implementation,

```
@Controller
public class VetsController {
  ...
```

```
4   @GetMapping("/flow")
5   public String vetsHtmlFlow(Model model) {
6     Flux<Vet> vets = vetsService.getAllVets();
7     model.addAttribute("vets", new ReactiveBox<>(vets));
8     return "vets.flow";
9   }
10 }
```

Listing 2.25: HtmlFlow controller implementation

In line 6 we can see the controller getting the `Flux` of vets, and just like the previous templates will put it into a wrapper to make sure we will receive the `Flux` instance intact.

Notice in line 8, that we define the template we want to process which is the `vets.flow`. Just like KotlinX, this template does exist as a resource file as it does with Handlebars or Thymeleaf. This is just so that the HtmlFlow resolver can process this request, and the other resolvers discard it.

If we run the request to see the result, which just like Handlebars is the same as the KotlinX on 2.17.

After examining how each template is created, exploring how we can invoke them through a controller, and observing the results they produce, the subsequent section is dedicated to delving into the internal mechanisms of each template.

2.4 Resolvers

This chapter's purpose is to examine the internal workings of each template and explore how progressive rendering is either attempted or achieved, particularly in the case of Thymeleaf.

Each subsection is dedicated to one of the four templates, and within these sections, we will illustrate the relationships among the `ViewResolver`, `View`, and `Processor` components. We will highlight which components are provided by Spring, as well as what we need to create to apply a consistent approach across all templates.

The following image serves as a way to explain how all the components from the `ViewReso-lver` to the `Processor` work.

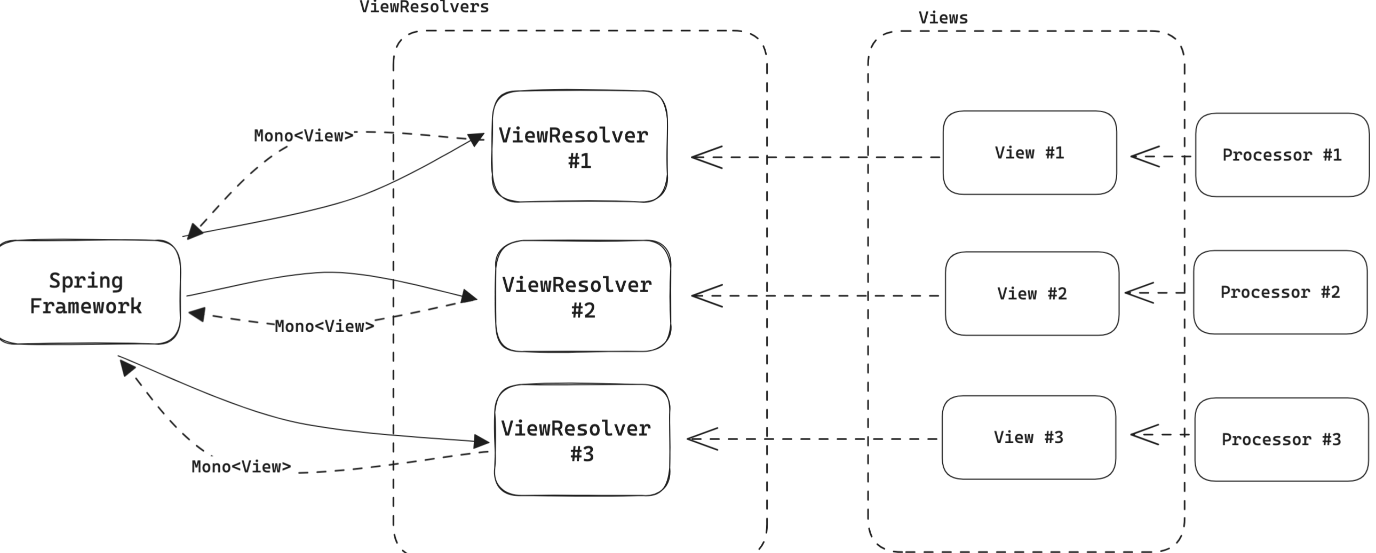


Figure 2.1: Diagram how Spring handles SSR requests from the ViewResolver to the View

As previously mentioned, the `ViewResolver` yields a `Mono<View>` instance. Internally, we incorporated the functionality of a `View` that encompasses a `Processor`, capable of handling the HTML template.

We proceed to see each template and how this architecture works in each.

2.4.1 Thymeleaf Resolvers

Just like in 2.3 we start by taking a look at Thymeleaf.

```
public class ThymeleafReactiveViewResolver extends ViewResolverSupport {

  @Override
  public Mono<View> resolveViewName(String viewName, final Locale locale) {
    if (!canHandle(viewName, locale)) {
      return Mono.empty();
    }
    ...
    return loadView(viewName, locale); //ThymeleafReactiveView
  }
}
```

Listing 2.26: Reactive version of the Thymeleaf ViewResolver

The first thing that should catch our eye is the change of the return type in the `resolveViewName` method in line 4. In the previous explanation, we pointed out that the return would be an instance of `View`, now we are looking at an instance of a `Mono<View>`.

As for the implementation itself, the important part is that this resolver keeps the responsibility of either returning an instance of a `View` or informing that it cannot process this template.

In line 5 we can see that if Thymeleaf cannot handle this template it returns an empty `Mono`, this will let Spring know that it should search for another available bean of a `Resolver`. Line 9 will load the bean for the instance of the wanted view, which in this case is `ThymeleafReactiveView`, and wrap it into a `Mono`, we can then assert that `ThymeleafReactiveView` implements the `View` interface.

Now if we take a look into a snippet of the `ThymeleafReactiveView`.

```
public class ThymeleafReactiveView extends AbstractView {
  @Override
  public Mono<Void> render(Map<String, ?> model, MediaType contentType, ServerWebExchange
    exchange) {
    Map<String,Object> enrichedModel = null;
    for (String executionAttributeName : executionAttributes.keySet()) {
      if (enrichedModel == null) {
          enrichedModel = new LinkedHashMap<>(model);
      }
      enrichedModel.put(modelAttributeName, modelAttributeValue);
    }
    enrichedModel = (enrichedModel != null ? enrichedModel : model);
    return super.render(enrichedModel, contentType, exchange);
  }
}
```

Listing 2.27: Reactive version of a Controller to use the vets template

Although the `render` function should just process the template and produce an output, Thymeleaf has many external and optional parameters that we can pass to the template. And so, if we take a look from line 6 to line 13, this `render` function will first take notice of such parameters and add those to a `enrichedModel`.

At line 15, the `render` function's main goal is to render the template, which is done by calling the parent function `super.render(...)`, which will call a method called `renderInternal`.

This method `renderInternal` implementation is very similar to what we show on the synchronous version of Thymeleaf.

```
@Override
protected Mono<Void> renderInternal(Map<String, Object> renderAttributes, MediaType
    contentType ServerWebExchange exchange) {
  return renderFragmentInternal(this.markupSelectors, renderAttributes, contentType, exchange)
    ;
}
...
protected Mono<Void> renderFragmentInternal(Set<String> markupSelectorsToRender, Map<String,
    Object> renderAttributes, MediaType contentType, ServerWebExchange exchange) {
  ServerHttpResponse response = exchange.getResponse();
  ...
  Publisher<DataBuffer> stream = viewTemplateEngine.processStream(
    templateName, processMarkupSelectors, context, response.bufferFactory(), contentType,
    charset,templateResponseMaxChunkSizeBytes);
  if (templateResponseMaxChunkSizeBytes == Integer.MAX_VALUE && !dataDriven) {
      return response.writeWith(stream);
  }
  return response.writeAndFlushWith(Flux.from(stream).window(1));
}
```

Listing 2.28: Reactive version of how the template is rendered

In line 3 we directly call the `renderFragmentInternal` in order to decide whether to render the entire template or just a fragment of it.

In line 8 we extract the underlying `ServerHttpResponse` of the current exchange happening, this is what will allow us to produce an output for the browser.

Line 10 is where we do the processing of the template and return `Publisher<DataBuffer>` as the output that needs to go to the browser.

Now, we need to decide if this is an entire template output or just a partial template output. So, if we notice at line 13 this is done by inspecting the variable `templateResponseMaxChunkSizeBytes` and if we are in mode `dataDriven` which determines if we have a limit of bytes, we should emit at one time.

Then finally at both line 14 and line 18, we use the `response` variable and output the resulting stream to the browser.

All of this is part of the progressive rendering that Thymeleaf does in `data-driven` mode, which is enabled when a type of variable `ReactiveDataDriverContextVariable` is present in the context of the engine.

This type `ReactiveDataDriverContextVariable` is passed to the context of the template in the controller, as the snippet below shows,

```
@GetMapping(value = {"/vets"})
public String getAllVets(Model model) {
  Flux<Vet> currentVets = vetServices.findAllVets();
```

```
4   var rx = new ReactiveDataDriverContextVariable(currentVets, 1);
5   model.addAttribute("vets", rx);
6   return "vets/vetList";
7 }
```

Listing 2.29: ReactiveDataDriverContextVariable is passed to the context for Thymeleaf to start the reactive processing

Notice the passage of the variable `rx` of type `ReactiveDataDriverContextVariable` in line 5.

When the internal rendering finds this type, 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 before the asynchronous data is completed, and the web page is finished. Thymeleaf does this in multiple steps, inside the internal rendering of the template.

```
1  Mono<Void> renderFragmentInternal(...) {
2    (...)
3    // Step 1
4    var ctx = createRequestContext(exchange, mergedModel);
5    var thymeleafRequestContext = RequestContext(ctx, exchange);
6    // Step 2
7    var dataDriven = isDataDriven(mergedModel);
8    (...)
9    // Step 3
10   var stream = templateEngine.processStream(...,maxChunkSizeBytes);
11   // Step 4
12   if (maxChunkSizeBytes == Integer.MAX_VALUE && !dataDriven) {
13       return response.writeWith(stream);
14   }
15   return response.writeAndFlushWith(Flux.from(stream).window(1));
16 }
```

Listing 2.30: Internal render of reactive fragment inside template

If we follow the steps inside the code, the first thing done is to initialize the RequestContext (reactive version) and add it to the model as another attribute, so that it can be retrieved from elsewhere.

```
var ctx = createRequestContext(exchange, mergedModel);
var thymeleafRequestContext = new RequestContext(ctx, exchange);
```

Listing 2.31: Context initialization for fragment rendering

This elsewhere being wherever fragments have access to the `SpringWebFluxThymeleafRequestContext`.

As a second step, Thymeleaf will then determine if we have a data-driver variable, and therefore will need to configure the flushing of output chunks and not flushed immediately after each buffer.

As described, this variable will be used when Thymeleaf is deciding how to output the values inside the stream, which is done in the final, fourth step.

Thymeleaf has three possible modes to process the template and emit the HTML, those being:

1. FULL mode
2. CHUNKED mode
3. DATA-DRIVEN mode

If no size limit for output chunks has been set (FULL mode), Thymeleaf lets the server applies its standard behavior by using `writeWith`.

Else, either we are in DATA-DRIVEN mode or a limit for output chunks has been set (CHUNKED mode), Thymeleaf uses `writeAndFlushWith` in order to make sure that output is flushed after each buffer.

These mods here described are ways that Thymeleaf can deal with template processing for Reactive Streams, there are three possible processing modes, for each of which a `Publisher<DataBuffer>` will be created in a different way.

The `DataBuffer` interface is used because that's what the WebFlux framework receives in the `ServerWebExchange` interface, which on itself represents the current response/request exchange happening within the browser, to be able to push content for the browser response.

The FULL mode outputs chunks not limited in size (`templateResponseMaxChunkSizeBytes == Integer.MAX_VALUE`) and no data-driven execution (no context variable of type Publisher driving the template engine execution). In this case, Thymeleaf will be executed unthrottled, in full mode, writing output to a single `DataBuffer` chunk instanced before execution, and which will be passed to the output channels in a single `onNext(buffer)` call (immediately followed by `onComplete()`).

CHUNKED mode outputs chunks limited in size (`responseMaxChunkSizeBytes`) but no data-driven execution (no `Publisher<X>` driving engine execution). All model attributes are expected to be fully resolved (in a non-blocking fashion) by WebFlux before engine execution

and the The Thymeleaf engine will execute in throttled mode, performing a full-stop each time the chunk reaches the specified size, sending it to the output channels with onNext(chunk) and then waiting until these output channels make the engine resume its work with a new request(n) call. This execution mode will request an output flush from the server after producing each chunk.

The final mode is DATA-DRIVEN, where one of the model attributes is a `Publisher<X>` wrapped inside an implementation of the IReactiveDataDriverContextVariable<?> interface. In this case, the Thymeleaf engine will execute as a response to onNext(`List<X>`) events triggered by this Publisher. The `bufferSizeElements` specified at the model attribute, will define the amount of elements produced by this Publisher that will be buffered into a `List<X>` before triggering the template engine each time (which is why Thymeleaf will react on `onNext (List<X>)` and not `onNext(X)`). Thymeleaf will expect to find an `th:each` iteration on the data-driven variable inside the processed template, and will be executed in throttled mode for the published elements, sending the resulting DataBuffer output chunks to the output channels via `onNext(chunk)` and stopping until a new `onNext(List<X>)` event is triggered. When execution is data-driven, a limit in size can be optionally specified for the output chunks (`responseMaxChunkSizeBytes`), which will make Thymeleaf never send to the output channels a chunk bigger than that (thus splitting the output generated for a `List<X>` of published elements into several chunks if required). When executing in DATA-DRIVEN mode, Thymeleaf will always request flushing of the output channels after producing each chunk.

Step 3 is where Thymeleaf knows which mode to choose by the variable `templateResponseMaxChunkSizeBytes` that is passed to the method `viewTemplateEngine.processStream` where,

```
1 var stream = processStream(...,templateResponseMaxChunkSizeBytes);
```

Listing 2.32: Creation of stream based on output mode

If `templateResponseMaxChunkSizeBytes` is MAX_VALUE then FULL/DATA-DRIVEN mode is chosen, else CHUNKED/DATADRIVEN is the chosen one.

The processing mode that is used is Data-Driven because the controller when returning the template adds a IReactiveDataDriverContextVariable to the Model.

If we go inside the `processStream` method and follow the implementations for Data-Driver development we reach the `createDataDrivenStream` where Thymeleaf handles the outputting of the HTML whilst dealing with reactive streams.

This whole process can be broken down into five steps.

```
Flux<DataBuffer> createDataDrivenStream(...) {
  // STEP 1: Obtain the data-driver variable and its metadata
  (...)
  // STEP 2: Replace the data driver variable with a DataDrivenTemplateIterator
  (...)
  // STEP 3: Create the data stream buffers.
  var dataDrivenBufferedStream =
      Flux.from(getDataStream(reactiveAdapterRegistry))
      .buffer(bufferSizeElements);
  // STEP 4: Initialize the (throttled) template engine for each subscriber.
  var dataDrivenWithContextStream = Flux.using((...));
  // STEP 5: React to each buffer of published data by creating one or many (concatMap)
    DataBuffers containing the result of processing only that buffer.
  var stream = dataDrivenWithContextStream.concatMap((step) -> (...));
}
```

Listing 2.33: Creation of Flux containing each part of the HTML fragment

In step 4, which starts by initializing the (throttled) template engine for each subscriber where normally there will only be one.

```
var dataDrivenWithContextStream = Flux.using (...)
```

Listing 2.34: Initialization of throttled template engine

Using the throttledProcessor as the state in this `Flux.using` allows us to delay the initialization of the throttled processor until the last moment, when output generation is really requested.

```
() -> {...
    return new StreamThrottledTemplateProcessor(processThrottled(...));
}
```

Listing 2.35: Delay of throttled processor initialization

This flux will be made by concatenating a phase for the head (template before data-driven iteration), another phase composed of several steps for the data-driven iteration, and finally a tail phase (template after data-driven iteration). But this concatenation will be done from a Flux created with , so that there is an opportunity to check if the processor has already signaled that it has finished, and in such case, the subscription to the upstream data driver might be able to be avoided if its iteration is not needed at the template.

We then reach step 5 where a flux is built to react to each buffer of published data by creating one or many (concatMap) DataBuffers containing the result of processing only that buffer.

```
var stream = dataDrivenWithContextStream.concatMap((step) -> (...))
```

Listing 2.37: Merging of all the results from the many buffers

The initial state is set to `TRUE` so that the first step executed for this Flux performs the initialization of the `dataDrivenIterator` for the entire Flux. It is necessary that this initialization be performed when the first step of this Flux is executed, because initialization actually consists

```
throttledProcessor ->
Flux.concat(
  Flux.generate(() -> DATA_DRIVEN_PHASE_HEAD,(phase, emitter) -> {
    if (throttledProcessor.isFinished()) {
      emitter.complete();
      return null;
    }
    switch (phase) {
      case DATA_DRIVEN_PHASE_HEAD:
        emitter.next(just(forHead(throttledProcessor)));
        return DATA_DRIVEN_PHASE_BUFFER;
      case DATA_DRIVEN_PHASE_BUFFER:
        emitter.next(map(forBuffer(throttledProcessor, it)));
        return DATA_DRIVEN_PHASE_TAIL;
      case DATA_DRIVEN_PHASE_TAIL:
        emitter.next(Mono.just(forTail(throttledProcessor)));
        emitter.complete();
      }
      return null;
    }
  )
)
```

Listing 2.36: Each phase of the reactive fragment rendering

of a lateral effect on a mutable variable (the `dataDrivenIterator`). This way we are certain that it is executed in the right order, given `concatMap` guarantees to us that these Fluxes generated here will be consumed in the right order and executed one at a time (and the Reactor guarantees us that there will be no thread visibility issues between Flux steps).

From then on, it will be FALSE so it is the first execution of this that initializes the (mutable) `dataDrivenIterator`.

```
Flux.generate(() -> Boolean.TRUE, (...))
```

Listing 2.38: Create a never-ending Flux to continuations emmit data

The dataDrivenIterator is then initialized. This is a lateral effect, this variable is mutable, so it is important to do it here to be sure that it is executed in the right order.

```
(initialize, emitter) ->
{
  var throttled = step.getThrottledProcessor();
  var templateIterator = throttled.getDataDrivenTemplateIterator();
  if (throttled.isFinished()) {
      emitter.complete();
      return Boolean.FALSE;
  }
}
(...)
```

Listing 2.39: Control ending of Flux output

This is where the order of execution is guaranteed by evaluating where the `step` is currently.

If the step is on the head, then there is a feed with no elements - we just want to output the part of the template that goes before the iteration of the data driver.

When we get to the `dataBuffer` part, we get to the value-based execution, where we have values and want to iterate them.

```
1 if (step.isHead()) {templateIterator.startHead();}
```

Listing 2.40: Starting of emit HTML

```
1 else if(step.isDataBuffer()){ templateIterator.feedBuffer(step.getValues()); }
```

Listing 2.41: Feeds the values from the buffer to the output

Now we finally get to the tail. It's time to signal feeding complete, indicating this is just meant to output the rest of the template after the iteration of the data driver. Note there is a case when this phase will still provoke the output of an iteration, and this is when the number of iterations is exactly ONE. In this case, it won't be possible to determine the iteration type (ZERO, ONE, MULTIPLE) until we close it with this `feedingComplete()`.

```
1 else {
2     dataDrivenTemplateIterator.feedingComplete();
3     dataDrivenTemplateIterator.startTail();
4 }
```

Listing 2.42: Starts emitting the end of the reactive HTML fragment

It's time to determine if the template should be executed another time for the same data-driven step or rather should consider to have done everything possible for this step (e.g.produced all markup for a data stream buffer) and just emit `complete` and go for the next step.

```
1 boolean phaseFinished = false;
2 if (throttledProcessor.isFinished()) {
3   phaseFinished = true;
4   dataDrivenTemplateIterator.finishStep();
5 } else {
6   (...)
7 }
```

Listing 2.43: Determines if it should finish the step or continue emmiting

We know everything before the data-driven iteration has already been processed because the iterator has been used at least once (i.e. its 'hasNext()' or 'next()' method has been called at least once). This will mean the context switches to the buffer phase.

```
1 if (step.isHead() && templateIterator.hasBeenQueried()) {
2   phaseFinished = true;
3   templateIterator.finishStep();
4 } else if (step.isDataBuffer() && !templateIterator.continueBufferExecution()) {
5   phaseFinished = true;
6 }
```

Listing 2.44: Signals the finish of the phase

We get to the final part of step 5, where there is a computation if the output for this step has been already finished (i.e. not only the processing of the model's events, but also any existing overflows). This has to be queried *before* the buffer is emitted.

If the step has finished, `complete` has to be emitted now, giving the opportunity to execute again if processing has finished.

```
1 boolean stepOutputFinished = templateIterator.isStepOutputFinished();
2 emitter.next(buffer);
3 if (phaseFinished && stepOutputFinished) {
4   emitter.complete();
5 }
6 return Boolean.FALSE;
```

Listing 2.45: Ends the emission of data from the parent Flux

When we reached the end of all the steps, Flux signaled that the processing is complete and can output the entire rendered HTML to the browser.

This is how Thymeleaf, which is an internal implementation of Spring, handles the SSR for reactive models.

We will now take a look at how we implemented this logic in the other templates.

2.4.2 KotlinX Resolvers

Just like the Thymeleaf version, we will start by showing the `Resolver`, followed by the `View`.

The resolver implementation is pretty straightforward, as is the Thymeleaf one.

```
1  class KotlinxViewResolver(suffixes) : AbstractViewResolver(suffixes) {
2    (...)
3    override fun resolveViewName(viewName, locale): Mono<View> =
4      checkIfPrefixIsSupported(viewName, locale)
5        .map { BasicView(KotlinXTemplateProcessor(resolveTemplate(it))) }
6  }
```

Notice that in line 4, we first check if this `resolver` can support the `viewName` passed. If the `viewName` is not supported then the return will be `Mono<Void>`.

Assuming that this `viewName` is supported, we need to return an instance of a `View` that can handle the processing of the template. Hence the creation of a `BasicView` instance, in line 5, with a processor as a parameter.

The `resolveTemplate(it)` method is to fetch the KotlinX template built using the DSL.

The `BasicView` is a generic implementation, shared by all the templates, except Thymeleaf, which takes a processor that is capable of parsing the template.

```
1  public class BasicView extends AbstractView {
2    private final TemplateProcessor processor;
3    (...)
4    protected Mono renderInternal(renderAttributes, exchange) {
5      // Step 1, prepare writing fields
6      var response = exchange.getResponse();
7      var dataBuffer = response.bufferFactory().allocateBuffer();
8      var writer = new OutputStreamWriter(dataBuffer.asOutputStream());
9
10     // Step 2, create a Mono with a subscriber to write into the exchange
11     return response.writeWith(Mono.create((sub -> {
12         var templateAttrs = Map.of("writer", writer, "subscriber", subscriber, "buffer",
       dataBuffer, "response", response);
13         this.processor.processTemplate(this.mergeModels(renderAttributes, templateAttrs),
       writer);
14       }))
15     );
16   }
17   (...)
18 }
```

Listing 2.46: BasicView, which is responsible for unwrapping the needed parameters and passing them to the received processor

Notice that in the class declaration (line 1) `BasicView` extends `AbstractView` which is the base class to represent a `View` in WebFlux, like seen in `ThymeleafReactiveView`.

The method `renderInternal` is the one responsible for starting the template processing and can be broken down into 2 steps.

Step 1, we prepare the writing fields for the template processing operation about to happen,

```
var response = exchange.getResponse();
var dataBuffer = response.bufferFactory().allocateBuffer();
var writer = new OutputStreamWriter(dataBuffer.asOutputStream());
```

Listing 2.47: Extraction of response stream, buffer for the data and the write for said buffer

The `response` field represents the current connection open to the browser, for writing output to the browser. This response will then allocate a buffer to write the content.

We then create a `writer` in to be able to write `strings` into the buffer. This `strings` are in fact the processed `html` template.

In step 2, we create a Mono with a subscriber to write into the response object.

```
response.writeWith(Mono.create(subscriber -> {
  var templateAttrs = Map.of(...);
  processTemplate(mergeModels(renderAttributes, templateAttrs), writer);
}))
```

Listing 2.48: Creation of buffer to continuously write to the server output

This mono will control what gets written into the response object and when it also when the `response` can close the connection and give the template processing work as completed. To have that kind of control, `Mono.create` method provides a subscriber of type `(MonoSink<DataBuffer>)`. This subscriber will serve only to signal when the reactive has been read and processed, so ServerWebExchange can close the connection.

The `processor` field represents an interface created to represent some implementation that can deal with the processing of a template. For each technology we use, we have a dedicated `TemplateProcessor` that applies the model to the template and deals directly with the respective template engine, this is done in the method `processTemplate`.

```
public interface TemplateProcessor {
  void processTemplate(Map renderAttributes, OutputStreamWriter writer);
}
```

Listing 2.49: TemplateProcessor declaration

The KotlinX version,

```
class KotlinXTemplateProcessor(val template : KotlinxTemplate) : TemplateProcessor {
  override fun processTemplate(renderAttributes : Map, writer: OutputStreamWriter) {
    applyTemplate(renderAttributes, writer)
  }
  (...)
  private fun applyTemplate(renderAttributes : Map, writer: OutputStreamWriter) {
    renderAttributes?.let { this.template.apply(it, writer) }
  }
}
```

Listing 2.50: KotlinX Processor implementation

As can be seen in line 4, we call the method applyTemplate which will call upon the Kotlinx-Template which will grab the Map containing all the parameters and replace them into the template (notice in line 12 the call to the apply method in the KotlinxTemplate class).

If we look at the KotlinxTemplate implementation, it's just converting the template and writing the output into the writer.

```
1  class KotlinxTemplate(val templateConverter : (model) -> String) {
2    fun apply(model : Map<String, Any> = mapOf(), writer: Writer) {
3        val converter = templateConverter(model)
4        writer.append(converter)
5    }
6  }
```

Listing 2.51: KotlinX template which passes the model for template processing

All of this resolver, view logic is called by the Spring framework when we declare a controller and return the template to use (just like Thymeleaf).

2.4.3 Handlebars Resolvers

The implementation of Handlebars follows precisely the same logic as KotlinX. Therefore, we will solely highlight the key differences in implementation, while omitting aspects that are identical to KotlinX.

Starting by the resolver,

```
1  public class HandlebarsViewResolver extends AbstractViewResolver {
2    (...)
3
4    public Template resolveTemplate(String templateName) {
5      return this.handlebars.compile(templateName);
6    }
7
8    public Mono<View> resolveViewName(String viewName, Locale locale) {
9      return this.checkIfPrefixIsSupported(viewName, locale)
10       .map(view -> new BasicView(new Processor(resolveTemplate(view))));
11   }
12 }
```

Listing 2.52: Handlebars view resolver to fetch the template and compile it

Just like for the KotlinX version, each ViewResolver is called to provide a View based on a certain viewName, and if it's not supported, it returns an instance of Mono<Void>. On our version, we also added the ability for the ViewResolver to be able to call the template engine and give to HandlebarsTemplateProcessor the correct template instance to use.

As for the view implementation, it is the same as the BasicView seen in the KotlinX subsection 2.3.2.

The only difference is the implementation of the `processor` which in the Handlebars goes as follows,

```
public class HandlebarsTemplateProcessor implements TemplateProcessor {
  private final Template compiledTemplate;
  @Override
  public void processTemplate(Map renderAttributes, writer) {
    applyMainTemplate(renderAttributes, writer);
  }

  void applyMainTemplate(Map<String,Object> renderAttributes, writer) {
    compiledTemplate.apply(this.getContext(renderAttributes), writer);
  }
}
```

Listing 2.53: Handlebars implementation of the Processor

Notice that the method `processTemplate` in line 5 is very similar to the KotlinX version.

If we go to the method `applyMainTemplate` in line 13, we can sew in line 14, that we take the template that was already compiled in the `ViewResolver` and process it by passing the context and the writer as parameters.

This `context` is composed of two parameters, the current map containing all parameters passed to the `Model` the controller, plus some more that was added in the View class, and an `Options` object existent in the handlebars library.

In order for Handlebars to be able to give the map with all these parameters, we need to build this `Context`. This is done by creating an instance of such a class and passing the `Model`.

```
private Context getContext(Map<String, Object> model) {
  return Context.newBuilder(model).resolver(INSTANCE).build();
}
```

Listing 2.54: Creation of the Handlebars context

By passing this `Context` into the Handlebars' template instance, Handlebars internally will apply all the needed objects into the HTML template, and any helpers that might be registered will also receive this context object unwrapped by the Map containing the model and an `Options` instance.

We can see the model creation and the template associated with the Helper in the controller.

As we can see, the only difference for Handlebars is the usage of the `Context` object which represents the model. In the case of KotlinX, we use the map directly.

Finally, we will take a look at HtmlFlow.

2.4.4 HtmlFlow Resolvers

The implementation of HtmlFlow adheres to the exact same logic as KotlinX and Handlebars. As a result, we will exclusively focus on delineating the primary differences in implementation, while excluding elements that are shared by both KotlinX and Handlebars.

The implementation of the `Resolver` does not change much from the 2.3.2 and 2.3.3 implementation, as can be seen here.

```
1  public class HtmlFlowViewResolver extends AbstractViewResolver {
2
3    private HtmlFlowTemplate resolveTemplate(String template) {
4      return new HtmlFlowTemplate(HtmlFlowKeyTemplateMatcher.match(template));
5    }
6    @Override
7    public Mono<View> resolveViewName(String viewName, Locale locale) {
8      return this.checkIfPrefixIsSupported(viewName, locale)
9        .map(name -> new BasicView(new HtmlFlowProcessor(resolveTemplate(name))));
10   }
11 }
```

Listing 2.55: HtmlFlow resolver implementation

Notice in line 4 that, also here we create a template that contains the matching name of the name (`template` parameter (line 3)) using the static method from `HtmlFlowKeyTemplateMatcher` class and return that `HtmlFlowTemplate` instance.

The method `resolveViewName` from lines 7 to 10 also returns a `Mono<View>` containing the instance of `BasicView` with the `HtmlFlowProcessor` that will process the template returned from the `resolveTemplate` method.

If we look at the processor implementation for HtmlFlow,

```
1  public class HtmlFlowProcessor implements TemplateProcessor {
2    private final HtmlFlowTemplate template;
3
4    @Override
5    public void processTemplate(Map renderAttributes, OutputStreamWriter writer) {
6      applyTemplate(renderAttributes, writer);
7      writer.flush();
8    }
9
10   private void applyTemplate(Map renderAttributes, OutputStreamWriter writer) {
11     this.template.apply(renderAttributes, writer);
12   }
13 }
```

Listing 2.56: HtmlFlow resolver implementation

As KotlinX and Handlebars implementations of the processors do, this also implements the interface `TemplateProcessor`.

Line 7 calls upon the method `applyTemplate` which itself on line 17 will call the `HtmlFlowTemplate` to apply the received model to the template created through the DSL.

Notice the `writer.flush()` in line 8 which will effectively push what was added to the writer into the output.

`HtmlFlowTemplate` is what uses the correct template and returns the `String` which is the result of the processing of such a template.

```
1  public class HtmlFlowTemplate {
2    private final HtmlFlowTemplates match;
3    public void apply(Map<String, Object> renderAttributes,
4          OutputStreamWriter writer) {
5      String view = DynamicHtml.view(match::template).render(renderAttributes);
6      writer.append(view);
7    }
8  }
```

Listing 2.57: HtmlFlow resolver implementation

Notice the method `view` used in line 6 after the `DynamicHtml` class.

A View in HtmlFlow can be seen as a wrapper for the complete HTML document. Depending on the type of view that the programmer chooses to create, certain methods and advantages can be used, or not.

A View instance acts like a container that mediates the connection between a template function and a Visitor.

In terms of Views, HtmlFlow presents two options for representing a view.

1. *DynamicHtml*
2. *StaticHtml*

The `DynamicHtml` is used (line 6) to create this template. This is because `DynamicHtml` should be used when we want to write a template that has dynamic components to it. As can be seen, the method `render` (at the end of line 6) receives the `renderAttributes` parameter which is all the parameters needed for the dynamic parts.

This will effectively make the template that the `HtmlFlowTemplates` return can use this `Map<String,Object>` to populate the dynamic parts of the template.

On the other hand *StaticHtml* does not support the render with the `Model` as a parameter. Because the idea for *StaticHtml* is to be used when we have a template that is always the same and does not depend on external objects, this makes it so this type of view can always be cached as not render every time it is called.

Another difference between *StaticHtml* and DynamicView is the missing of the method *dynamic* that is present in the DynamicView. Since *StaticHtml* does not support render with the View-Model, we cannot also call the method `dynamic` when using this type of view.

Going to the view implementation, it is the same as the previous templates, so it will not be shown again here.

But as we can see, given the Spring logic on how to handle SSR with reactive streams, all three templates do not differ much from how they are implemented.

2.5 Summary

As we have seen, there is a need to implement SSR with reactive streams given the new paradigm of a Reactive Server. There is currently no support for SSR with Reactive Streams in most template engines, being `Thymeleaf` the exception. In the petclinic demo, we showed how we can incorporate Reactive Stream processing, with a single source of reactive data, into the Handlebars, KotlinX and HtmlFlow template engines, by doing a workaround of using helpers classes.

Even with that workaround we still could not see the desired results in the browser, with all the content inside the streams being processed out of order and the HTML showing out-of-order as well. But being Thymeleaf a heavy template engine, like can be seen in [33] and being mostly coupled with the Spring framework, there is a need to find a template engine that can do the job and not be stuck to an asynchronous model type.

3

Functional Reactive Templates

In this Chapter, we will discuss the proposal for handling reactive web templates with a functional approach, by enhancing the framework `HtmlFlow`, with the goals of providing progressive rendering and supporting multiple asynchronous sources.

We will start by describing `HtmlFlow` architecture and functionality, followed by the proposal for supporting reactive templates.

3.1 Former HtmlFlow internal processing

HtmlFlow is a Java DSL to write typesafe HTML documents in a fluent style. The HTML is generated through the creation of a template. Templates are expressed in an internal DSL, meaning Java code is written to produce the template. This implies that whilst creating the template we can use the full Java toolchain.

As can seen in the example,

```
StaticHtml.view()
.html()
  .head()
    .title().text("title").__()
      .link().attrRel(EnumRelType.STYLESHEET).attrHref("/css/bootstrap.css").__()
    .__()
    .body()(...)
      .footer()(...).__()
    .__();
```

Listing 3.1: Creation of simple HtmlFlow Template

Each view is built from a function of type `HtmlTemplate`, specified by the following Java functional interface:

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

Listing 3.2: HtmlTemplate interface responsible for specifying a function

As can be seen in listing 3.2, the `HtmlTemplate` interface specifies a function that we can use to define a template through the usage of a `HtmlPage`.

The `HtmlPage` instance provides the HTML builder methods which allow us to construct the template 3.1, just like the `head` or `title` methods.

The template 3.1 has a `StaticHtml` associated (line 1), which is a type of view. As stated in the previous Chapter, at the end of the subsection 2.4.4, a View in HtmlFlow is a wrapper for the complete HTML document. But a View is also what makes the association to a Visitor.

So, the `HtmlTemplate` provides us with a function to define the template and create a view. The view itself is what binds all the created methods to a Visitor, which we will present later on in the Chapter.

In a way, HtmlFlow templates are essentially plain Java functions.

We will now have a look at two components that make HtmlFlow.

1. Templates
2. Visitors

3.1.1 Kinds of Templates

The basic definition of a template is a function that builds the HTML document through the Element API.

```
static void template(DynamicHtml<LabelValueModel> view, LabelValueModel model) {
  view
    .div()
      .label()
      .dynamic(label -> label.text(model.label)).__() //label
      .input()
        .dynamic(input -> input
          .attrType(EnumTypeInputType.TEXT)
          .attrId(model.id)
          .attrName(model.id)
          .attrValue(model.value.toString())
        )
      .__() //input
    .__(); //div
}
```

Listing 3.3: Creation of a DynamicHtml that accepts models

In HtmlFlow, there are three kinds of templates. There is what we call *normal template* that can be either a template created out of the *StaticHtml*, like example 3.1, or with *DynamicHtml* in example 3.3. And then we have Partials and Layouts.

A Partial in HtmlFlow is constructed just like any template, where we create the type of view we want and continue to define the rest of the partial. The difference is that we don't have to specify the entire HTML document for it to make sense.

The advantage of using Partials is that we only have to define small parts of HTML that will be used in a bigger template.

```
static void template(DynamicHtml<Pet> view, Pet pet) {
  view
  .form().attrMethod(EnumMethodType.POST)
    .div().attrClass("form-group")
      .dynamic(div -> view.addPartial(view, of(LocalDate.now())))
    .__() //div
  .__() //form
}
```

Listing 3.4: Creation of template that is completed by a partial

Usually, a partial is created to fill a hole in some other template. As seen in 3.4, line 5, the partial is added to the main template to complete the HTML.

This way of invoking partials is particularly useful when you need to use a smaller part (component) together with an existing template, to produce a bigger one. This is the most common usage of partials.

Another way of using partials is to construct a layout, which is the latter kind of template.

The layout is a normal template, but with a hole to be filed with partials. As we saw earlier, a partial has nothing special by itself. What is interesting it is the layout. Consider the following template.

```
static void template(DynamicHtml<T> view, T model, HtmlView[] partials) {
  view
  .html().head()
  (...)
    .title().text("My awesome templating system").__().__() //head
    .body().nav().attrClass("navbar navbar").addAttr("role","navigation")
    .dynamic(__ -> view.addPartial(partials[0]) )
      .__() //nav
      .div().attrClass("container-fluid")
        .div().attrClass("container xd-container")
          .dynamic(__ ->  view.addPartial(partials[1], model) )
        .__() //div
      .__() //div
    .__() //body
  .__(); //html
}
```

Listing 3.5: Usage of partials inside a template

Notice the third argument to the function template, this array of partials is the place where we receive the partials to fill the holes with our layout. To use them, we called two distinct signatures of `view.addPartial`; one with only the partial, and one with a partial and a model. Depending on the type of templates hidden behind `partials[0]` we would use one signature or the other.

3.1.2 Visitors

One of the reasons HtmlFlow is so diverse and flexible is the usage of the well-known Design Pattern `Visitors`.

In object-oriented programming and software engineering, the visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying the structures. It is one way to follow the open/closed principle.

The way HtmlFlow follows the principle mentioned above is to have a singular abstract class ElementVisitor. Each component that wants to be considered a `Visitor` should extend and present the correct definitions for the methods.

The programmer can, of course, decide which one to use. This is done by the overload of the method view the programmer calls. Each overload has its option of an implementation of `Visitor` that best suits the choice made.

The responsibility of the `Visitor` is to specify how the HTML is being built. This is defined by how we close and open tags, add attributes, and comments and how the content ends up being written in the end.

We also have another level of abstraction, that all the top-level `Visitor` end up extending from directly.

`HtmlVisitorCache` extends from `ElementVisitor` and helps define some of the previously not-defined methods to deal with visitor-specific methods, like `visitParent` and `visitElement`.

As explained, HtmlFlow is built on a cache system and we can *"turn if off"* when we have a dynamic element. This level of abstraction has the implementation to handle caching but leaves a new method open to be implemented, *write*.

The method *write* is what each top-level Visitor should implement to define how the content is being pushed downstream.

A programmer looking to implement their Visitor, for their very specific needs, can take advantage of this design pattern by extending the *cache* definition or going to the lower abstraction level and extending `ElementVisitor`.

3.2 DSL Proposal for asynchronous fragments

Now that we understand how HtmlFlow works internally, we can begin to explain our proposal for solving the problem of SSR with asynchronous sources to achieve progressive rendering.

Our proposal for SSR Web templates on top of HtmlFlow is the only non-blocking solution that can 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.

So here we are taking advantage of the functional nature of HtmlFlow which proposes the idea of functional templates using HoT - higher-order templates. Thus, we will implement a new idea of Functional Reactive Templates on top of HoT.

By using the already existent architecture of HtmlFlow with the visitors' strategy, we need to add a new layer that would handle Reactive Streams without forcing the client to wait for the response and guarantee the correct order of the HTML output.

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.

So in this Chapter, we will present how HtmlFlow supports asynchronous sources, starting from an initial version of a data structure with multiple strings to a more event-driven architecture with continuations.

The goal for this new version is to avoid nested call chains and achieve a fluent style, without being compromised to a specific type of asynchronous source. Furthermore, it should be easy for the user to work with, so it should spare the end-user asynchronous specific logic.

To achieve this goal, we went from an approach of a data structure with multiple `String` to a more event-driven architecture to support asynchronous sources.

3.2.1 Why a new version was needed

In the first version of the HtmlFlow API, the HTML was built taking advantage of the visitor pattern, where a visit happens per element and the output of the element is written into a `StringBuilder`.

When a visit happened, it would immediately write the content without having the context of the previous or next element to be written. Although the HTML was well-formed, once it was written, each element would be processed immediately when called.

For a version where there was no support for asynchronous behavior, this always worked as expected, because at the time each element needed to be written, its content was already present and ready to be read.

When we enter the world of asynchronous behavior, there is no predicting when the data will be ready to be read. This logic itself is already incongruous with the version explained above. Two things would happen when we brought asynchronous sources to this version.

The very first is that the HTML would not be well-formed when the elements were being written. The second was that if the content took too long to be available, it might not even appear on the HTML if the result was already produced.

So it was evident, that we needed a way to delay the writing of an element for when it's needed. This is to say that we need to keep the order of how the elements should be written. Each element should only be processed when the previous one has terminated.

3.2.2 An event-driven way of working

As explained, we needed a way to enforce the order of writing the overall HTML elements.

Since we work with asynchronous sources, we cannot block the data and make it work that way.

In the first iteration, we could not miss the similarity between our solution, to the one of a `LinkedList`. In this data structure, each node knows the next one. Also, each node is only processed when the list cursor advances.

So a structure of nodes was created, where the operation `async` would be the main asynchronous operation to occur, and the `then` synchronous operations that would occur after.

The type `Thenable` would encapsulate this entire logic.

```
1 interface Thenable {
2   Thenable async(Observable, Consumer);
3   Thenable then(Function cont);
4 }
```

Listing 3.6: Thenable interface, definition

Each method of the interface returns itself so that the end user can chain more actions, either more `async` or synchronous, by using the method `then`.

Each action would then be kept inside a data structure of async nodes, with the order of insertions guaranteed. Only the `async` nodes were being kept inside this data structure since they were the ones that needed to have the execution halted until it was required.

The `then` actions were seen as children of an `async` action. By the end of each `async` action the children nodes would be invoked.

Once the children of an `async` action are complete, the next async node would receive an event to start running, as seen in the following diagram.

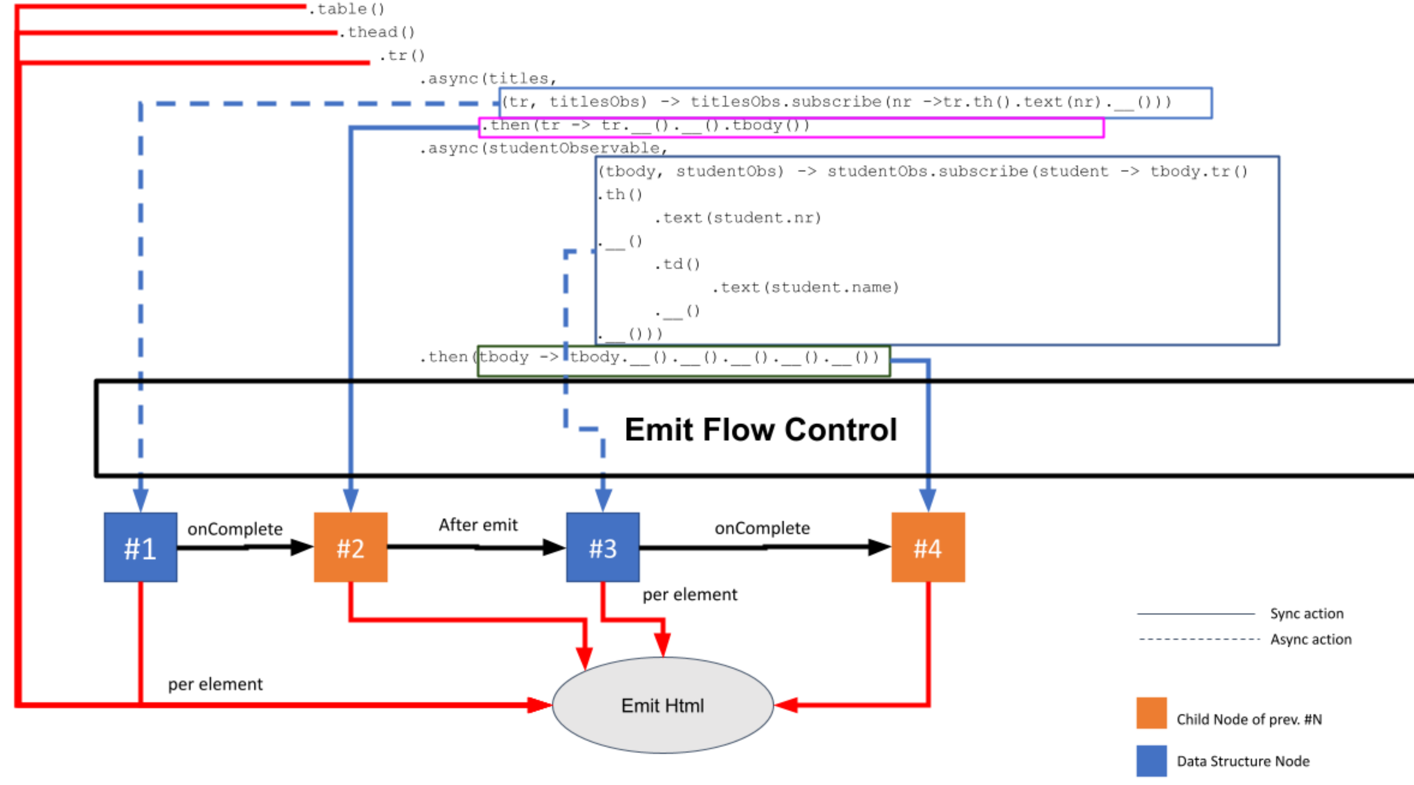


Figure 3.1: HtmlFlow `async` `then` architecture

Although this solution worked to reach the desired result, which is support for asynchronous sources, it didn't fully reach the desired goal. It was very verbose for the end-user as it introduced two new operations that contradict the goal of avoiding nested chain calling.

Having this first version as a base to start from, we focused on reaching our end goal. One of first things we did, was to try and trim the new operations to just one, making it easier for the end-user to work with and make a more seamless transition between previous HtmlFlow versions.

We added the `await` operation, which is a single operation that contains an asynchronous action. Where following HTML elements should wait for this action to finish before being processed.

This `await` operation can be unwrapped into three main parts, as can be seen in the following code.

```
1 <M> E await(AwaitConsumer<E,M> asyncAction);
2 interface AwaitConsumer<T,M> {
3   void accept(T first, M model, OnCompletion cb);
4 }
```

Listing 3.7: Await operation unwrapped

The `first` argument represents the current HTML element being processed, the `model` is an asynchronous model to be used, and the `cb` argument is the `onCompletion` signal for when the asynchronous source has ended.

To take full advantage of this new `await` operation, we need to alter the internal logic of HtmlFlow on how the elements are processed, to make every element knowledgeable of the element to be processed. Out of this necessity came the current version of the HtmlFlow API, where we have a preprocessing of the entire element tree and the construction of an internal chain of continuations.

3.3 HtmlFlow asynchronous internal processing

As it was previously acknowledged, to reach our goal we need every element of the entire HTML to have an added responsibility. Each HTML element should have a reference for the next element to be processed.

So, each element no longer holds the responsibility to know when it should be written into the output. But, instead, it holds the responsibility to let the next element in line know that it can start the processing.

As explained in 3.2.2, this way of traversing and processing elements it's very similar to a `LinkedList`.

In our case, each node represents a subset of HTML that needs a specific behavior. This behavior can vary, by the way of emitting HTML and when the next node should be called. Although each node contains a very specific logic, they all share the same responsibilities.

To group this responsibility, we created the term, *continuation*, which takes shape in our implementation `HtmlContinuation`.

```
abstract class HtmlContinuation {
  ...
  HtmlContinuation next;
  ...
  HtmlContinuation getNext() {return next;}
  abstract void execute(Object model);
}
```

Listing 3.8: HtmlContinuation

A `HtmlContinuation` is the base for a linked list of nodes, corresponding to `HtmlContinuation` objects, and it's responsible for emitting an HTML block and calling the next node.

There are two main continuations defined in this new architecture, `HtmlContinuationSync` and `HtmlContinuationAsync`.

`HtmlContinuationSync` is a continuation that defines a block of static HTML, so it's logic to emit and call the next node is fairly simple.

```
void execute(Object model) {
  ...
  emitHtml(model);
  if (next != null) {
    next.execute(model);
  }
}
```

Listing 3.9: HtmlContinuationSync execute

Notice that in line 3 of 3.9, we can directly emit the HTML and call upon the next HTML continuation, if it exists, since this is a block of static information.

As for the `HtmlContinuationAsync`, we need to consider that the next node can only be executed when the asynchronous operation has ended. Even, if the HTML can start to be emitted, we should have more control over the execution of the next continuation.

```
void execute(Object model) {
  ...
  consumer.accept(element, model, () -> {
    if (next != null) {
      next.execute(model);
    }
  });
}
```

Listing 3.10: HtmlContinuationAsync execute

Unlike the sync version, where we just emitted the HTML and called the next continuation, in the asynchronous version, we defined a consumer (line 3 in 3.7) which contains a function as a

parameter that is called when the asynchronous operations have ended. This is what triggers the next HTML continuation to be executed if it exists.

Now that we understand how the internal continuations communicate with each other, we need to take a look at how everything is put together.

For the continuations to work, as we described then, we need first and foremost to connect the nodes.

For that purpose, we created the step of `preprocessing`. Preprocessing is the first step of HTML creation. At the moment of template creation, all the continuations are created as we go along the HTML tree.

```
1 public static HtmlViewAsync viewAsync(HtmlTemplate template){
2   PreprocessingVisitorAsync pre = preprocessingAsync(template);
3   return new HtmlViewAsync(new HtmlViewVisitorAsync(true, pre.getFirst()));
4 }
```

Listing 3.11: Call to preprocessing when viewAsync is called

Notice in line 2, that what is called immediately is the `preprocessingAsync`, which effectively will go through all the HTML tree and chain the connections.

The first static HTML elements are grouped into a single continuation (line 3 in 3.11), without needing to be signaled to write to the output. However, it still requires the user to call the final function that triggers the HTML to start being written, maintaining the lazy behavior. And the `async` elements into a different one as it was explained.

When we get to the `async` node, all the nodes that proceed an `async` one, static or async, needs to be called to start writing to the output, as can be seen in the example below.

```
1 @Override
2 public void visitAwait(Element element, AwaitConsumer asyncHtmlBlock) {
3   HtmlContinuation asyncCont = new HtmlContinuationAsync<>(
4     (...),new HtmlContinuationSyncCloseAndIndent(this));
5   chainContinuationStatic(asyncCont);
6   indentAndAdvanceStaticBlockIndex();
7 }
```

Listing 3.12: Asynchronous implementation of chaining of continuations

Notice the passage of a `HtmlContinuationSyncCloseAndIndent` which will effectively deal with indentation problems between static and async blocks.

After the indentation is dealt with, the `asyncCont` is called to emit its HTML.

The `static` continuations hold their block of static HTML to be written, until it is required to do so, giving the guarantee that the order in which the template was created gets preserved.

As the name of this step suggests, all of this is preprocessing. Meaning that none of the elements are called and emit their HTML parts. They are just grouped into continuations that reflect the

needs we have per set of elements. This is important because one of the characteristics of HtmlFlow is that the writing of the template is *lazy*. Meaning it will only write into the output when it is required to.

The following example showcases how the chain of continuations is triggered to start running.

```
....
view.writeAsync(new PrintStream(mem), asyncModel); //user call
...
CompletableFuture<Void> writeAsync(out, model) {
  return visitor.clone(out).finishedAsync(model); // executes the visitor
}
...
CompletableFuture<Void> finishedAsync(model) {
  var cf = new CompletableFuture<Void>();
  var terminationNode = new HtmlContinuationAsyncTerminationNode(cf);
  setNext(last, terminationNode);
  this.first.execute(model); // Initializes render on first node.
  return cf;
}
```

Listing 3.13: Html template writeAsync

Listing 3.13 starts, in line 2, with an example of a user triggering the writing of the HTML using the method `writeAsync`.

Line 5 showcases how the visitor for the asynchronous operations is called upon to start the continuation chain by the method `finishedAsync`.

Notice the `terminationNode` at line 12. The `HtmlContinuationAsyncTerminationNode` is just a node to complete the `CompletableFuture` when all the processing has ended.

We guarantee that this node will be the last one because in line 14 we connect this one as a *next* node for *last* node.

Effectively, each part of the HTML tree is really just written when the user calls the `writeAsync` method. Which will trigger the first node to be executed (line 18) and the ones after that, as the preprocessing constructed them.

With this approach, we can guarantee that the HTML is well-formed and, by taking advantage of the *continuations*, we can have progressive rendering.

```
view
.html()
  .body()
    .div()
      .p().text("Students from a school board").__()
    .__() //div
    .div()
      .table()
        .thead()
          .tr()
          .<AsyncModel>await((tr, model, onCompletion) -> Flux
            .from(model.titles)
            .doOnComplete(onCompletion::finish)
            .doOnNext(nr -> tr.th().text(nr).__())
            .subscribe())
          .__() //tr
        .__() //thead
        .tbody()
          .<AsyncModel>await((tbody, model, onCompletion) -> Flux
            .from(model.items)
            .doOnComplete(onCompletion::finish)
            .doOnNext(student -> tbody.tr().th().text(student.getNr()).__()
                    .td().text(student.getName()).__().__())
            .subscribe())
        .__() //tbody
      .__() //table
    .__() //div
    .div().p().text("Best students in school").__() //p
    .__() //div
  .__() // body
.__(); //html
```

Listing 3.14: HtmlFlow usage with new version

In the template above, we see in line 4 the usage of `model.titles` and in line 10 `model.items`.

This involves the inclusion of two separate asynchronous sources within a single `model` object. In doing so, HtmlFlow is equipped to manage multiple asynchronous sources and facilitate progressive rendering.

By running the template, we get the following result.

```
<html>
  <body>
  <div><p>Students from a school board</p></div>
    <div>
      <table>
        <thead>
          <tr>
            <th>Nr</th>
            <th>Name</th>
          </tr>
        </thead>
        <tbody>
        <tr><th>1</th><td>Pedro</td></tr>
        <tr><th>2</th><td>Manuel</td></tr>
        <tr><th>3</th><td>Maria</td></tr>
        <tr><th>4</th><td>Clara</td></tr>
        <tr><th>5</th><td>Rafael</td></tr>
        </tbody>
      </table>
    </div>
    <div><p>Best students in school</p></div>
  </body>
</html>
```

Listing 3.15: Result of the new HtmlFlow reactive approach

3.4 Summary

As demonstrated in this Chapter, we have enhanced HtmlFlow by introducing *Continuations* objects, thus enabling seamless support for multiple asynchronous sources alongside synchronous sources. This advancement has enabled us to successfully achieve the primary objective of progressive rendering.

As evidenced in the final example, 3.14, the adoption of this new version of HtmlFlow has had a minimal impact on its usage. Despite the integration of two asynchronous sources, the resulting HTML remains well-structured and consistent.

In the following chapter, we will proceed to validate the presented approach with a use case.

4

Validation

In this chapter, we will present how Thymeleaf and KotlinX.html deal with multiple asynchronous data models in a web application running in a state-of-the-art non-blocking middleware, namely Spring WebFlux [37]. Followed by a benchmarking study to compare solutions.

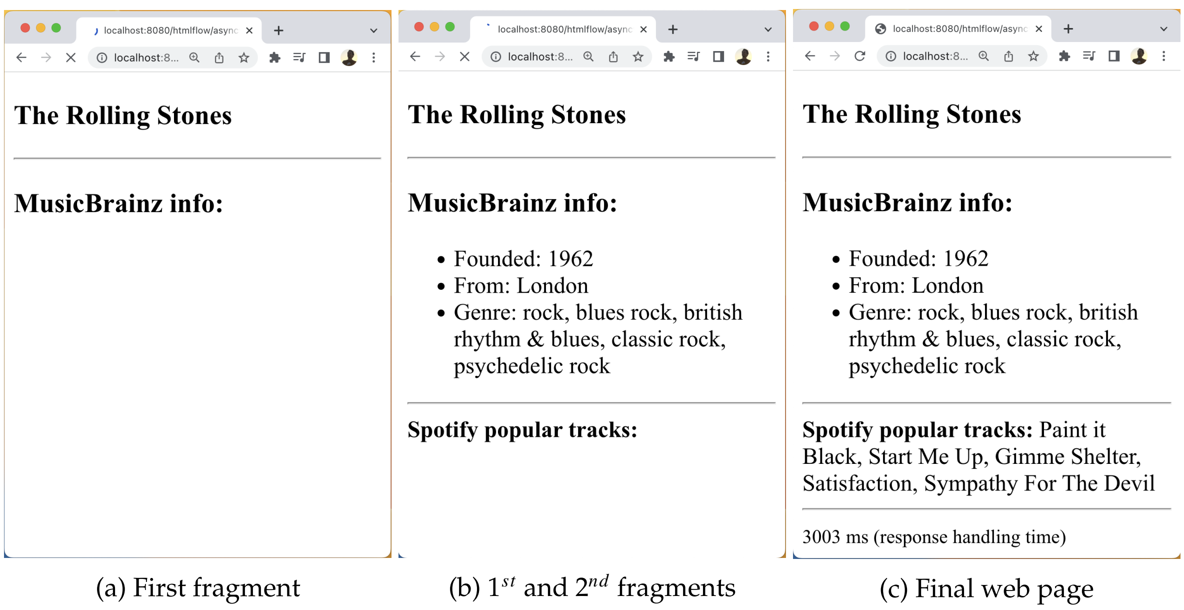
In the next subsection, we start describing our case study of a WebFlux application named Disco. After that, we analyze how Thymeleaf and KotlinX.html behave with this application.

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 4.1 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 4.1a is immediately rendered since it does not depend on any asynchronous data. Then, the remaining two fragments of Figures 4.1b and 4.1c are emitted incrementally as their data models become available. We deliberately made, a Spotify data model with a higher latency than MusicBrainz to observe a progressive render between these two steps. In Listing 4.1 we present the corresponding HTML source to the final web page.

Notice that 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 make visible the effects of progressive rendering, we inserted an explicit delay of 2 seconds on the first data



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

Figure 4.1: Expected output from Disco web app for The Rolling Stones band.

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, as well as Spring WebFlux, 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, according to the Kotlin definition of Listing 4.2. In this case we have a repository [9] for each domain entity to access its data source.

Repositories are a key component of the data access layer and provide a way to encapsulate and manage data access logic. Repositories are closely related to domain entities and typically represent a collection of entities of a particular type, such as `MusicBrainz` or `SpotifyArtist`. The main role of repositories is to provide a simplified and consistent API for working with data sources, such as databases, file systems, Web APIs, or others. 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.

4.1.1 Thymeleaf

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

```
<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 Me Up, ...
      </span>
      <hr>
    </div>
    <footer>
      <small>
        3003 ms (response handling time)
      </small>
    </footer>
  </body>
</html>
```

Listing 4.1: Expected HTML source code for the web page of Figure 4.1c

```
class MusicBrainz(
  val artist: String,
  val year: Int,
  val from: String,
  genresList: List<String>)
{
  val genres = genresList.joinToString(", ")
}

class SpotifyArtist(
  val artist: String,
  val popularSongs: List<String>
)
```

Listing 4.2: Disco domain entities definition in Kotlin for `MusicBrainz` and `SpotifyArtist`.

Thymeleaf attributes that define the execution of predefined logic, such as, `foreach` loop (i.e. `th:each`).

In Listing 4.3 we present the Thymeleaf template that produces the view of Figure 4.1c. For simplicity, we are just including the dynamic parts, since the static blocks are equal to those presented in Listing 4.1. This template deals with 4 model attributes: `artistName` (line 1), `musicBrainz` (lines 3), `spotify` (line 10) and `start` (line 15).

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 and 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.

```
<h3 th:text="${artistName}"></h3>
...
<ul th:each="item : ${musicBrainz}">
  <li th:text="*{'Founded: ' + item.year}"></li>
  <li th:text="*{'From: ' + item.from}"></li>
  <li th:text="*{'Genre: ' + item.genres}"></li>
</ul>
...
  <th:block
    th:each="track : ${spotify.popularSongs}"
      th:text="${track + ', '}">
  </th:block>
...
<small
  th:text="${#dates.createNow().getTime() - start} + 'ms (response handling time)'">
</small>
```

Listing 4.3: 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 the server response. To address this issue on Spring WebFlux, the asynchronous objects must be wrapped into a ReactiveDataDriver-ContextVariable, which we already discussed in chapter 2.

However, there are two notable limitations of ReactiveDataDriverContextVariable, regarding the issues 1) limited asynchronous idiom and 2) single data model, described in of Section 2.4.1. First, the model attribute must be a reactive stream Publisher [25], 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 The Thymeleaf web template contains some kind of iteration on the data-driver variable, normally by means of an th:each attribute. This means that it requires a th:each block to access musicBrainz even if it just contains a single value (i.e. line 3 of Listing 4.3). 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.

4.1.2 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`:

```
1 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`.

In Listing 4.4 we present the definition in KotlinX.html for the web page of Figure4.1c. 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 In Listing 4.4 receives a lambda that is called with that `AppendableSink` object as the receiver (i.e. `this`). The last statement of the web template definition (line 31) 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 39 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 16 and `spt` in line 23), without the need for 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 4.1a, and then proceeds to the next partial page of Figure 4.1b resulting from the completion of MusicBrainz data

```
fun artistView(
    start: Long,
    artisName: String,
    musicBrainz: CompletableFuture<MusicBrainz>,
    spotify: CompletableFuture<SpotifyArtist>
): Publisher<String> = AppendableSink {
 this
 .appendHTML()
 .html {
  body {
   div {
    h3 { text("$artisName") }
    hr()
    h3 { text("MusicBrainz info:") }
    ul { musicBrainz
     .thenAccept { mb ->
      li { text("Founded: ${mb.year}") }
      li { text("From: ${mb.from}") }
      li { text("Genres: ${mb.genres}") }
      hr()
      b { text("Spotify popular tracks: ") }
      span { spotify
       .thenAccept { spt ->
        text(spt.popularSongs.joinToString(", "))
        hr()
        footer {
         small {
          text("${currentTimeMillis()-start} ms (response handling time)")
         }
        }
        this.close()
       } // thenAccept
      } // span
     } // thenAccept
    } // ul
   } // div
  } // body
 } // html
 .asFlux()
}
```

Listing 4.4: Example of KotlinX.html template of Disco web application.

model and before completion of Spotify data. The web page will finish with the expected output of Figure 4.1c when the last data model completes.

Unlike Thymeleaf, building a web template with KotlinX.html is not limited either to any specific kind of asynchronous API nor to any number of asynchronous data models.

Yet, there are still some issues to deal with. 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. The resulting HTML will be out of order as presented in Listing 4.5. 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.

```
<html>
  <body>
    <div>
      <h3>The Rolling Stones</h3>
      <hr>
      <h3>MusicBrainz info:</h3>
      <ul></ul>
    </div>
  </body>
</html>
<li>Founded: 1962</li>
<li>From: London</li>
<li>Genres: rock, blues rock, ...</li>
<hr>
<b>Spotify popular tracks: </b>
<span></span>Paint it Black, Start Me...
<hr>
<footer>
  <small>
    3011 ms (response handling time)
  </small>
</footer>
```

Listing 4.5: Example of ill-formed HTML source resulting from undesirable interleavings between template processing and asynchronous data access.

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. As the template becomes dependent on more asynchronous data models, the callbacks become nested deeper within each other, exacerbating the issue of callback nesting.

4.2 Evaluation

In the first subsection, we describe the testing environment, followed by our analysis of the performance results in the next section.

4.2.1 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[31]. 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 4.6

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

Listing 4.6: `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 with Spring WebFlux and their controllers with functional routes [37], and 3) integrated Java Microbenchmark Harness (JMH) [29] 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.

4.2.2 Results

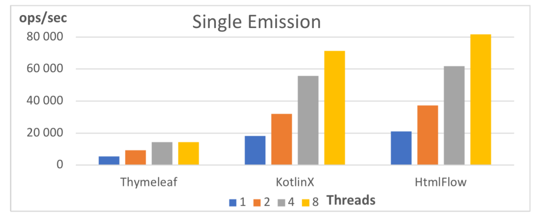
Spring WebFlux supports progressive rendering through a `ServerResponse` object constructed from a `Publisher<String>` that emits the generated HTML. While progressive rendering offers several user experience advantages, it incurs higher overhead compared to a single emission of the complete HTML document. So, first, we tested the three template views with a single emission of the resulting web page. To achieve this, we collected all instances of `Presentation` into an auxiliary `List<Presentation>`, without blocking, by mapping the `Publisher<Presentation>` to a `CompletableFuture<List<Presentations>>`, i.e. `cf` in next listing:

```
cf.thenAccept { l -> template.render(l) }
```

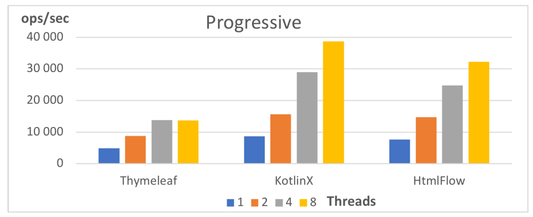
Upon `cf` completion, we can then proceed to render the template using the resulting `List`. In this case, the template handles a synchronous model and accumulates the generated HTML into an intermediate `StringBuilder`, sending the resulting documents at once.

4.2.3 Performance Evaluation

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. In Figure 4.2a we present the throughput of each template view.



(a) Single emission of the complete HTML document.



(b) Progressive rendering.

Figure 4.2: Throughput of Thymeleaf, KotlinX.html and HtmlFlow in Spring templates benchmark with WebFlux and JMH.

Based on the results shown in Figure 4.2a, it is evident that Thymeleaf does not exhibit scalability beyond 4 threads. When comparing these findings to the results obtained in Table 2.1, which were collected using Spring MVC, we can observe that Thymeleaf's performance is approximately 32% of that of HtmlFlow. However, in the context of WebFlux, Thymeleaf experiences a similar slowdown of around 25%.

Despite HtmlFlow having slightly better throughput than KotlinX.html, it does not match the performance as observed in Table 2.1. The new `HtmlContinuation` infrastructure presented

in Chapter 3, which supports asynchronous data models incurs additional overhead that did not exist in the former implementation of HtmlFlow.

According to the results presented in Figure 4.2b, when running these tests with progressive rendering, a generalized slowdown is observed, with throughput decreasing to approximately half of what was observed in Figure 4.2a. Thymeleaf also demonstrates limitations in scalability under these conditions. On the other hand, KotlinX.html now 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.

4.2.4 Memory Allocation Evaluation

JMH also offers the `-prof gc` profiler, which listens for Garbage Collector (GC) events, accumulate them, and normalize the allocation/churn rates based on the number of benchmark operations.

Our results demonstrate that, for all evaluated templates, the memory allocated in *Eden* space is equal to the amount that the GC removes from that space. Additionally, the amount of memory allocated in *Survivor* space [12] tends to be zero, indicating that most of the memory allocated in this benchmark is garbage that is completely reclaimed by the GC.

In our scenario, the benchmark code runs with a single thread, allowing the GC threads to run on separate threads and handle the garbage generated by the benchmark thread. Adding more benchmark threads creates competition between benchmark and GC threads for CPU resources, implicitly overwhelming the GC and generating more garbage.

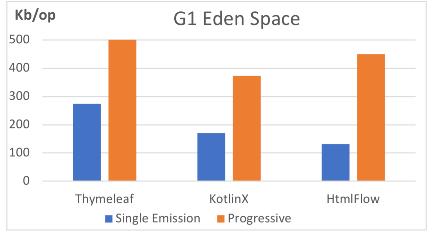


Figure 4.3: Memory allocation in Kb/op for each template engine in Spring templates benchmark for single HTML emission and progressive rendering.

Based on the results in Figure 4.3, it is evident that all templates undergo an increase in allocation pressure when supporting progressive rendering. In this scenario, HtmlFlow exhibits a higher value compared to KotlinX.html due to the use of continuations supported by instances of `AwaitConsumer` that require an additional resume function. All these functions are managed through lambdas, which correspond to the allocation of new anonymous objects, slightly increasing the allocation pressure in the JVM.

4.3 Summary

As demonstrated in this concise example, both `Thymeleaf` and `KotlinX` do not facilitate the utilization of multiple asynchronous sources.

We were already aware in advance that `KotlinX` lacks support for asynchronous sources in the context of well-formed HTML and progressive rendering. Therefore, the suboptimal performance of `KotlinX` and its absence of embedded support for multiple asynchronous types should not come as a surprise.

However, Thymeleaf, as previously indicated, possesses the capability to manage an asynchronous source for progressive rendering but encounters limitations in handling multiple asynchronous sources. It not only falls short of delivering progressive rendering but also, due to its reliance on `ReactiveDataDriverContextVariable`, whose constraints were previously outlined, prevents us from outright using multiple sources.

HtmlFlow is currently the only one that can support multiple asynchronous sources and maintain a well-formed HTML and progressive rendering. Even if the current solution of HtmlFlow dropped the score in the benchmarking test.

5

Conclusions

SSR and web templating continue to play crucial roles in the realm of web development. While SPAs have gained popularity for their dynamic capabilities, SSR remains a vital choice for rapidly crafting web pages. The inherent simplicity and efficiency of SSR in generating initial page content, make it a resilient choice that is unlikely to wane in importance.

Moreover, it's worth acknowledging that several of the present-day prominent web platforms, such as Facebook, still leverage SSR in various aspects of their operations. In light of this, our work assumes added significance as it strives to modernize web templating within the current landscape of server-side programming.

Our contribution is instrumental in enabling the development of HTML templates capable of seamlessly accommodating multiple asynchronous data sources.

5.1 Main contributions

This contribution can be already seen in the article accepted for `WebIst 2023` *Enhancing SSR in Low-Thread Web Servers: A Comprehensive Approach for Progressive Server-Side Rendering with Any Asynchronous API and Multiple Data Models*, where we take advantage of the proposed idiom in asynchronous fragments with calls to asynchronous APIs. In KotlinX the resulting HTML is out-of-order and thymeleaf has the already discussed limitations. HtmlFlow is the only one capable of dealing with any asynchronous API while keeping the HTML in the correct order.

In this article, through JMH benchmarking and a use case, we conclude that the HtmlFlow solution with the `async` idiom better suits the needs of the current state-of-the-art server-side programming paradigm.

This innovation not only streamlines the creation of dynamic web content but also facilitates the implementation of progressive rendering, a pivotal feature that enhances user experiences by incrementally displaying content as it becomes available. As a result, our work bridges the gap between traditional web templating and the evolving demands of modern, data-intensive web applications, ultimately bolstering the state-of-the-art in server-side programming.

5.2 Future Work

Now we plan to take a step further and take advantage of the async/await idiom [3] 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 the present proposal? Those are some of the research questions for future work.

References

[1] Adam L. Davis, “Akka http and streams”, in *Reactive Streams in Java: Concurrency with RxJava, Reactor, and Akka Streams*. Berkeley, CA: Apress, 2019, pages 105–128, ISBN: 978-1-4842-4176-9.

[2] Deepak Alur, Dan Malks, and John Crupi, *Core J2EE Patterns: Best Practices and Design Strategies*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001, ISBN: 0130648841.

[3] Don Syme, Tomas Petricek, and Dmitry Lomov, “The f# asynchronous programming model”, in *Practical Aspects of Declarative Languages*, Ricardo Rocha and John Launchbury, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pages 175–189, ISBN: 978-3-642-18378-2.

[4] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter, “An evaluation of reactive programming and promises for structuring collaborative web applications”, in *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, ser. DYLA ’13, Montpellier, France, 2013, ISBN: 9781450320412. DOI: 10.1145/2489798.2489802.

[5] Fernando Miguel Carvalho. and Luis Duarte., “Hot: Unleash web views with higher-order templates”, in *Proceedings of the 15th International Conference on Web Information Systems and Technologies*, ser. WEBIST ’19, Vienna, Austria, 2019, pages 118–129, ISBN: 978-989-758-386-5. DOI: 10.5220/0008167701180129.

[6] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand, “Continuations and coroutines”, in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ser. LFP ’84, Austin, Texas, USA, 1984, 293–298, ISBN: 0897911423. DOI: 10.1145/800055.802046.

[7] B. Evans, M. Verburg, and J. Clark, *The Well-Grounded Java Developer, Second Edition*. Manning, 2022, ISBN: 9781638355281.

[8] Martin Fowler, *Patterns of Enterprise Application Architecture*. 2002. [Online]. Available: https://martinfowler.com/books/eaa.html.

[9] ——, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN: 0321127420.

[10] ——, *Domain Specific Languages*, 1st. Addison-Wesley Professional, 2010, ISBN: 0321712943, 9780321712943.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Co., Inc., 1995, ISBN: 0-201-63361-2.

[12] P. Pufek, H. Grgić, and B. Mihaljević, "Analysis of garbage collection algorithms and memory management in java", in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pages 1677–1682. DOI: 10.23919/MIPRO.2019.8756844.

[13] David Ase, "Kotlin dsl for html", https://j2html.com/, Tech. Rep., 2015. [Online]. Available: https://j2html.com/.

[14] Xiaolong Jin, Benjamin W. Wah, Xueqi Cheng, and Yuanzhuo Wang, "Significance and challenges of big data research", *Big Data Res.*, vol. 2, no. 2, pages 59–64, Jun. 2015, ISSN: 2214-5796. DOI: 10.1016/j.bdr.2015.01.006.

[15] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, *et al.*, "The spring framework–reference documentation", *interface*, vol. 21, page 27, 2004.

[16] Hans Bergsten, *JavaServer Pages*. O'Reilly Media, Inc., 2003, ISBN: 9780596005634.

[17] Krishna Kant and Prasant Mohapatra, "Scalable internet servers: Issues and challenges", *ACM SIGMETRICS Performance Evaluation Review*, vol. 28, no. 2, pages 5–8, 2000.

[18] Sergey Mashkov, "Kotlin dsl for html", https://github.com/Kotlin/kotlinx.html, Tech. Rep., 2015. [Online]. Available: https://github.com/Kotlin/kotlinx.html.

[19] Andrey Breslav, *Kotlin Language Documentation*. 2016. [Online]. Available: https://kotlinlang.org/docs/kotlin-docs.pdf.

[20] Peter J Landin, "The next 700 programming languages", *Communications of the ACM*, vol. 9, no. 3, pages 157–166, 1966.

[21] Erik Meijer, "Your mouse is a database", *Queue*, vol. 10, no. 3, 20:20–20:33, Mar. 2012, ISSN: 1542-7730. DOI: 10.1145/2168796.2169076.

[22] N. Maurer and M. Wolfthal, *Netty in Action*. Manning, 2015, ISBN: 9781638353041.

[23] Ioannis Chaniotis, Kyriakos-Ioannis Kyriakou, and Nikolaos Tselikas, "Is node.js a viable option for building modern web applications? a performance evaluation study", *Computing*, vol. 97, Mar. 2014. DOI: 10.1007/s00607-014-0394-9.

[24] Friedman and Wise, "Aspects of applicative programming for parallel processing", *IEEE Transactions on Computers*, vol. C-27, no. 4, pages 289–296, 1978. DOI: 10.1109/TC.1978.1675100.

[25] Netflix, Pivotal, Red Hat, Oracle, Twitter, Lightbend, *Reactive streams specification*, 2015. [Online]. Available: https://www.reactive-streams.org/.

[26] Douglas C. Schmidt Irfan Pyarali Tim Harrison, "Significance and challenges of big data researchan object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events", 1997. [Online]. Available: https://www.dre.vanderbilt.edu/~schmidt/PDF/proactor.pdf.

[27] D. Schmidt, "Reactor. an object behavioral pattern for concurrent event demultiplexing and event handler dispatching", Jan. 1995.

[28] G.J. Sussman and G.L. Steele, *Scheme: an interpreter for extended lambda calculus*, ser. AI Memo No. MIT, Artificial Intelligence Laboratory, 1975.

[29] Aleksey Shipilev, *Java microbenchmark harness (the lesser of two evils)*, 2013. [Online]. Available: https://openjdk.java.net/projects/code-tools/jmh/.

[30] P. Klauzinski and J. Moore, *Mastering JavaScript Single Page Application Development*. Packt Publishing, 2016, ISBN: 9781785886447.

[31] Jeroen Reijn, "Comparing template engines for spring mvc", https://github.com/jreijn/spring-comparing-template-engines, Tech. Rep., 2015. [Online]. Available: https://github.com/jreijn/spring-comparing-template-engines.

[32] Harsha Kiran, *Php usage statistics: What you need to know in 2023)*, 2023. [Online]. Available: https://techjury.net/blog/php-usage-statistics/#:~:text=77%25%20of%20all%20live%20websites,today%20by%20around%2018%2C000%20websites.

[33] Fernando Miguel Carvalho, Luis Duarte, and Julien Gouesse, "Text web templates considered harmful", in *Web Information Systems and Technologies*, Alessandro Bozzon, Francisco José Domínguez Mayo, and Joaquim Filipe, Eds., Cham: Springer International Publishing, 2020, pages 69–95, ISBN: 978-3-030-61750-9.

[34] Ken Thompson, "Programming techniques: Regular expression search algorithm", *Commun. ACM*, vol. 11, no. 6, 419–422, Jun. 1968, ISSN: 0001-0782. DOI: 10.1145/363347.363387. [Online]. Available: https://doi.org/10.1145/363347.363387.

[35] Daniel Fernández, "Thymeleaf", https://www.thymeleaf.org/, Tech. Rep., 2011. [Online]. Available: https://www.thymeleaf.org/.

[36] Tim Fox, "Eclipse vert.x tool-kit for building reactive applications on the jvm", https://vertx.io/, Tech. Rep., 2001. [Online]. Available: https://vertx.io/.

[37] Marten Deinum and Iuliana Cosmina, "Building reactive applications with spring webflux", in *Pro Spring MVC with WebFlux: Web Development in Spring Framework 5 and Spring Boot 2*. Berkeley, CA: Apress, 2021, pages 369–420, ISBN: 978-1-4842-5666-4.