Text Web Templates considered Harmful

Fernando Miguel Carvalho[1][0000−0002−4281−3195], Luis Duarte[1][0000−0003−3967−6254], and Julien Gouesse[2]

[1] ADEETC, ISEL, Polytechnic Institute of Lisbon, Portugal
{mcarvalho,lcduarte}@cc.isel.ipl.pt
[2] Orange, France gouessej@orange.fr

Abstract. For the last decades text-based templates have been the primary option to build dynamic web pages. Until today, no other alternative has rebutted this approach. Yet, using text-based templates has several drawbacks including: 1. blocking resolution, 2. programming languages heterogeneity, 3. limited set of templating features and 4. opinionated idioms. In this paper we show how a domain specific language (DSL) for HTML (such as HtmlFlow, Kotlinx.html or React JSX) can suppress the text-based templates limitations and outperform state-of-the-art template engines (such as JSP, Handlebars, Thymeleaf, and others) in well known benchmarks. To that end, we use the Spring Framework and the sample application PetClinic to show how a DSL for HTML provides unopinionated web templates with boundless resolving features only ruled by the host programming language (such as Java, Kotlin or JavaScript).

Keywords: Web Templates · Dynamic Web Pages · Domain Specific Languages · Web Application · HTML.

1 Introduction

Web templates (such as JSP, Handlebars or Thymeleaf) are based in HTML documents, which are augmented with template specific markers (e.g. `<%`, {{}} or `${}`) representing *dynamic* information that can be replaced at runtime by the results of corresponding computations to produce a *view* [16,1]. The parsing and markers replacement process (i.e. *resolution*) is the main role of the *template engine* [31]. Even for those engines providing markers extensibility, these markers obey to a set of rules that restrict access to the host programming language features (e.g. Java) from the template. Thus, web templates development is dictated by the engine guidelines that force programmers to follow a set of given idioms and practices, i.e. *opinionated*[30].

In Listing 1.1 we show a sample Thymeleaf template[15] of a Spring web application [23] (part of the owner details view of the PetClinic web application[11]). This template builds a dynamic table containing a description list on each table row. In this case, the template receives a data model object `owner` with a `pets` property (line 2) and for each pet it generates a table row (`tr` in line 2) containing a description list (`dl` in line

Springer Nature Switzerland AG 2020, LNBIP 399,
*https://doi.org/*10.1007/978−3−030−61750−9_4

4) with the pet's name (line 6) and its birthdate (line 9). This Listing highlights the *heterogeneity* resulting from the technological mix of the HTML language with Thymeleaf template dialects (i.e. attributes beginning with `th`) and also the web framework programming language (i.e. Java) that is used, for example, on the auxiliary function call of line 9: `temporals.format(pet.birthDate,'yyyy-MM-dd')`.

Listing 1.1: Thymleaf template for owner details view of PetClinic web application.

```
1  <table class="table␣table-striped">
2    <tr th:each="pet␣:␣${owner.pets}">
3      <td valign="top">
4        <dl class="dl-horizontal">
5          <dt>Name</dt>
6          <dd th:text="${pet.name}"></dd>
7          <dt>Birth Date</dt>
8          <dd
9            th:text="${#temporals.format(pet.birth,␣'yyyy-MM-dd')}">
10         </dd>
11       </dl>
12     </td>
13   </tr>
14 </table>
```

Template engines distinguish themselves by the idiom and subset of available markers to control the dynamic content. But generally, all engines provide the same set of core templating features through specific dialects such as those used in the example of Listing 1.1 that we may group in the following categories:

1. *text replacement* – all expressions denoted with `${...}` markers in lines 2, 6 and 9.
2. *variables* – in line 2 we declare an auxiliary variable `pet`.
3. *control flow* such as conditional evaluation and loops – `th:each` *loop* in line 2.
4. *utility functions* – there is a set of built-in Thymeleaf utility objects available through the marker `#`, such as `#temporals` in line 9.
5. *partial views* or *fragments* inclusion (i.e. transclusion [35]) – in section 4.
6. *data binding* [29] – property `pets` of the `owner` *data model* [13], in line 2.

These dialects lead web frameworks to encompass at least 3 distinct programming languages on web development: 1) a high-level programming language used, for example, to gather data and build a *data model*, 2) HTML to build the skeleton of the web template and 3) template specific markers to manage the dynamic content of the web page. From these observations we argue that text-based templates, such as Thymeleaf templates incur in the following drawbacks:

1. no compile time validation;
2. require keen understanding of a diversity of technologies including the web framework host language (e.g. Java), HTML and the template engine dialects;
3. intricate definition from mixing idioms between HTML, template dialects and high-level programming language (e.g. line 9 of Listing 1.1);
4. restricted set of control flow features enforced by template dialects, such as `th:each`.

In this paper we state that an internal domain specific language (DSL) for HTML, such as HtmlFlow[8], KotlinX.html[27] or React JSX[37], can mitigate all pointed disadvantages and still provide the core templating features. These DSLs avoid specific dialects and give programmers all the freedom of the core programming language (Java, Kotlin or JavaScript). They let programmers interleave fluently HTML building blocks with any available construction of the host programming language. Thus, programmers may choose the most convenient idiom to write the template's control flow according to their preferences

In[10] we show how a *higher-order template* (HoT) does not block the template resolution and *pushes* the resulting HTML to the response stream as its content is being resolved [22,28]. Rather than *pulling* data from a source and fully complete markers of a template, HoT reacts to data and *push* the resulting HTML as it is being resolved. This approach provides better user experience keeping the browser responsive, even on presence of large data sets, and presents better rendering performance, in comparison to state of the art template engines, such as Handlebars and React, as shown in performance benchmarks[10,32]

To prove the effectiveness of HtmlFlow even for developing complex web applications we have replaced Thymeleaf templates of the Spring PetClinic[11] web application by HtmlFlow based templates. Spring PetClinic is an open-source Java application commonly used to demonstrate different design patterns and concepts, which was inspired by PetStore[34] that illustrates the use of J2EE to develop an eCommerce web application. Our work demonstrates how HtmlFlow simplifies several template idioms and still preserves the web templates role through HoT. Finally, we use the same approach with alternative DSLs for HTML namely j2Html[2], KotlinX.html and React, where HtmlFlow outperforms the competition in well-known benchmarks[10,32].

For the remainder of this paper we present in the next section state of the art solutions that deal with web templates and domain specific languages. Then in Section 3 we compare different alternative DSLs for HTML. In Section 4 we revisit the concept of *higher-order templates* (HoT) provided in HtmlFlow. After that in Section 5, we present the implementation of the most used template idioms with Thymeleaf and HtmlFlow in the PetClinic web application. In Section 6, we present a performance evaluation. Finally, in Section 7 we conclude and discuss some future work.

2 State of the Art

In this section we discuss related work in web templates and domain specific languages field. We first address in subsection 2.1 the main properties that dictates the design of web template engines. After that, in subsection 2.2, we present background work on domain specific languages and their main characteristics.

2.1 Web templates

Web template engines deal with data models as their *inputs* to produce HTML as *output* [16]. Martin Fowler distinguishes between two possible approaches followed by view engines: 1) *template view* and 2) *transform view*. The former is HTML centric and

thus oriented to the *output*. In this case, the view is written in the structure of the HTML document and embed markers to indicate where data model properties need to go. Since the seminal technologies JSP, ASP and PHP appeared with the *template view* pattern, many other alternatives emerged along the last two decades[3], turning this pattern into one of the most used approaches in web applications development.

On the other hand, the *transform view* is oriented to the *input* and how each part of the input is directly transformed into HTML. XSLT is maybe one of the most well-known programming languages to specify transformations around XML data. In this case the XML data takes the place of the *input* that is transformed by the XSLT to another format (e.g. HTML). Also, the functional nature of the transform view pattern enables its *composition* in a pipeline of transformations where each stage takes the result of the previous transformation as input and produces a new output that is passed to the next transformation. For example, the Cocoon Java library [6] provides a framework for building pipelines of XML transformations steps specified in XSLT.

The *transform view* pattern has similarities with the *higher-order templates*[10] approach of HtmlFlow where a view is a first-class function that takes an object model as parameter and applies transformations over its properties. The object model has the role of the *input* (e.g. XML data) and the HTML domain-specific language is the idiom used to transform the model into HTML.

2.2 Domain specific languages

The idea of domain-specific languages (DSL) was first approached by Landin [25], which introduces a framework that dictates the design of a specific language restricted by a domain. A DSL is a programming language specialized to a particular *application domain*[14]. This is in contrast to general-purpose languages, which are broadly applicable across domains.

DSLs can be divided in two types: *external* or *internal*[17]. *External* DSLs are languages created without any affiliation to a concrete programming language. An example of an external DSL is the regular expressions search pattern[36], since it defines its own syntax without any dependency of programming languages. Furthermore, regular expressions are easier to use and manipulate by experts than implementing the same set of verification rules through control flow instructions, such as `if/else` and `String` operations. Writing in Java programming language an equivalent validation to that one presented in Listing2.1 would require more than a single line of code to verify if a string is in 12-hour format with optional leading 0.

Listing 2.1: Regular expression for time in 12-hour format with optional leading 0.

```
(0?[0-9]|1[0-1]):([0-5][0-9])
```

On the other hand an *internal* DSL is defined within a host programming language as a library and tends to be limited to the syntax of the host language, such as Java. For that reason, internal DSLs can also be referred as *embedded* DSLs since they are embedded in the programming language where they are used.

[3] wikipedia.org/Comparison_of_web_template_engines

JQuery[33] is one of the most well-known examples of an internal DSL in Javascript, designed to simplify HTML DOM[21] tree traversal and manipulation. In the data structures field the Language Integrated Query (LINQ)[20] is an example of an internal DSL that enables querying of any kind of collection. An idea that is heavily inspired by the concept of *lazy lists*, also known as *streams*, first described in 1965 by Landin[24].

Another example of an internal DSL is jMock[18], which is a Java library that provides tools for test-driven development. In Listing 2.2 we can see that jMock uses a DSL to create expectations. In the concrete example it obtains the value of a property `Greeting` (`getGreeting()` in line 5), and asserts if it matches the expected value, which is `Good afternoon` in line 6. In this case the semantics of the methods used by jMock aim to simplify the programmer's understanding of the tests that are being performed.

Listing 2.2: jMock example.

```
1 final GreetingTime gt = context.mock(GreetingTime.class);
2 (new Greeting()).setGreetingTime(gt);
3
4 context.checking(new Expectations(){{
5   one(gt).getGreeting();
6   will(returnValue("Good␣afternoon"));
7 }});
```

In common all internal DSLs, such as JQuery, LINQ or jMock, use functions to define their languages. According to [17] we may identify three different patterns of combining functions to make a DSL: 1) *function sequence*; 2) *method chaining*, and 3) *nested function*.

A *function sequence* is about a combination of function calls as a sequence of statements. Consider for example the following set of functions: `html()`, `head()`, `body()`, `title()`, `div()` and `p()`, each one responsible for creating the corresponding HTML element with the same name of the function. So, the HTML document in Listing 2.4 could be the result of the execution of the corresponding Java program in Listing2.3.

Listing 2.3: Function sequence based DSL for HTML.

```
html();
  head();
    title();
      p("My␣title");
  body();
    div();
      p("A␣statement.");
```

Listing 2.4: Resulting HTML of 2.3.

```
<html>
  <head>
    <title>
      <p>My title</p>
    </title>
  </head>
  <body>
    <div>
      <p>A statement</p>
  ...
```

As we can see in Listing 2.3, if we try to lay out and organize a function sequence in an appropriate way, we can read it clearly as the resulting output in HTML. The major problem regarding this approach is whichever way we use to define a function sequence we will always need auxiliary context variables in order to know where we are in the building process. Considering the calls to `p()`, the builder needs to know which element will contain the resulting paragraph element. So, it does that by keeping track of the current HTML element in a variable. If these functions are global, then the state will end up being global too.

Method chaining pattern avoids this problem, since it is based on methods instances calls, where the target object may track any necessary context. In this case, it uses a sequence of method calls where each call acts on the result of the previous call. Thus, the methods are composed by calling one on top of the other. Yet, we still need some kind of bare function to begin the chain, such as `new Html()` in Listing 2.5 that instantiates the target root.

Listing 2.5: Method chaining based DSL for HTML.

```
new Html()
  .head()
    .title()
      .p("My␣title")
  .body()
    .div()
      .p("A␣statement.")
```

Listing 2.6: Nested function based DSL for HTML.

```
html(
  head(
    title(
      p("My␣title")),
    body(
      div(
        p("A␣statement."))))));
```

Nested function combines functions by making function calls arguments in higher-level function calls. This approach eliminates the need for a context variable, as the arguments are all evaluated before a function is called. A simple sequence of nested functions ends up being evaluated backwards to the order they are written. This means that arguments are first evaluated before the function being invoked. In Listing 2.6 `p()` is first evaluated and its resulting paragraph will be the argument of the call to `title()`, which in turn will be the argument of `head()` and henceforward.

Yet, the nested function pattern incurs into the same problems of functions globalness as function sequence pattern.

3 Domain-specific languages for HTML

Given the DSL design alternatives presented in section 2.1 we may describe DSLs for HTML according to the classification presented in the Table 1. For comparison we also include in Table 1 a non DSL-based engine, i.e. Thymeleaf, as the representative of most text-based templates such as JSP, Handlebars, Velocity and others. *Functional templates* regard the capacity of implementing Web templates as first-class functions, which is only possible with an internal DSL for HTML. The performance results in the Spring templates benchmark[32] are relatively to HtmlFlow, which is the most performant engine among these libraries. We are not considering React performance on these results, because this benchmark evaluates the performance of resolving HTML based

templates in the web server, whereas React only resolves HTML within the browser and just receives plain JSON from the web server. Nevertheless, in [10] we have already compared the performance including all the processing pipeline from the web server to the browser, which shows that React is almost 4 times slower than the competition rendering HTML for a data source with 10000 items.

Library	DSL	Template dialect	Functional templates	HTML safety	Data structureless	Spring templates benchmark
Thymeleaf	-	Thymeleaf	×	×	-	32%
j2html	Internal	Java	✓	×	×	26%
kotlinX.html	Internal	Kotlin	✓	✓	✓	58%
React	External	JavaScript	✓	×	×	-
HtmlFlow	Internal	Java	✓	✓	✓	100%

Table 1: Comparing Thymeleaf characteristics with DSLs for HTML.

In the following subsections we will describe each one of the DSLs for HTML and further properties: *HTML safety* and *data structureless*, which are an essential design key for the best performance presented by HtmlFlow.

3.1 j2Html

As other internal DSLs for HTML, j2html replaces the need of textual template files by templates defined within the Java language, which enables the use of all Java programming language features to control the flow of the dynamic parts. J2html uses a *nested function* approach where templates has a similar layout to that one presented in Listing 2.6. Thus, since arguments are first evaluated before the function being invoked, this technique does not allow the resulting HTML to be emitted on demand according to functions calls order. Otherwise the HTML would be generated backwards.

So, the result of the execution of a j2Html template is a tree structure that compose `Tag` objects[19]. This template must be further resolved through the `render()` method to produce an HTML document.

Furthermore, the major handicap of j2html is the lack of *validation* of the HTML language rules either at compile time or at runtime. Hence, it does not provide HTML safety because it does not ensure that the resulting HTML is conforming a valid HTML document.

3.2 KotlinX.html

The KotlinX.html [27] is another popular DSL for HTML and it has been written in Kotlin programming language. One of the Kotlin design principles was to create an inter-operative language with Java. Although its syntax is not compatible with the standard Java syntax, both languages can coexist within the same program source code.

Other main advantage of Kotlin is that it heavily reduces the amount of textual information needed to create code by using type inference and other techniques.

The KotlinX.html provides a fluent interface that mixes the three approaches of combining functions: function sequence, method chaining, and nested function. Yet, that mix is hidden when we are programming in Kotlin thanks to the mechanism of *function literals* (i.e. *lambdas*) with *receiver*[5].

Considering the use case of an internal DSL for HTML presented in section 2.2, the corresponding version for Kotlin using *function literals with receiver* would be written according to Listing 3.1. The equivalent use in Java of this Kotlin DSL is presented in Listing 3.2. In truth, whenever we write in Kotlin a block $\{\ldots\}$ following a function name (e.g. `head` $\{\ldots\}$), we are invoking that function with the block $\{\ldots\}$ as the the function's argument. To that end, the block $\{\ldots\}$ is passed as an anonymous function (or *lambda*) to the invoked function.

Method calls to `head`, `title`, `p`, etc in Listing 3.1 are equivalent to `this.head`, `this.title` and `this.p`. The `this` target is implicit and is the corresponding `self` lambda parameter of Listing 3.2. Declaring the `self` parameter as a *receiver* has the advantage of allowing the use of parameterless lambdas. In Kotlin, a parameterless lambda is denoted as $\{\ldots\}$ rather than $\{\ldots$ `->` $\ldots\}$, which leads to the idiom expressed in the example of Listing 3.1.

Listing 3.1: Kotlin type-safe builders for HTML.

```
html {
  head {
    title {
      p("My␣title")
    }
  }
  body {
    div {
      p("A␣statement.")
    }
  }
}
```

Listing 3.2: Kotlin type-safe builders for HTML invoked with Java code.

```
html(self -> {
  self.head(self ->
    self.title(self ->
      self.p("My␣title")
    )
  );
  self.body(self ->
    self.div(self ->
      self.p("A␣statement.")
    )
  );
});
```

As a result of using function arguments, it delays the evaluation of those functions and suppresses the backwards evaluation problem intrinsic to the *nested function* approach. So, the result of the execution of a KotlinX.html template does not require any auxiliary data structure and can be immediately emitted as functions are being invoked. Hence KotlinX.html templates resolution provides both modes: a DOM tree based and a *data structureless* mode that can be generated to an output stream.

Like HtmlFlow, the HTML fluent interface for KotlinX.Html is automatically built from the XSD definition of the HTML 5 language. Thus, the generated DSL ensures that each element only contains the elements and attributes stated in the HTML5 XSD document.

3.3 React

React introduces a syntax extension to JavaScript (JSX) that allows the use of plain HTML together with JavaScript. This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions.

As others internal DSLs for HTML, React allows programmers to use any valid JavaScript expression interleaved with the HTML template definition. JavaScript expressions should be wrapped in curly braces as denoted in line 6 of Listing 3.3.

Listing 3.3: Example of a React template defining an HTML divisory with current date.

```
1  class DateDiv extends React.Component {
2    render() {
3      return (
4        <div>
5          <h1>Date Time</h1>
6          <h2>It is {new Date().toLocaleTimeString()}.</h2>
7        </div>
8      }
9    }
10 }
```

Yet here, JSX behaves as an external DSL since HTML is a distinct idiom from JavaScript. After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

Moreover React templates result in a representation of the user-interface kept in memory to hold all information about the component tree. Although highly optimized this approach incurs in additional overheads of maintaining this auxiliary data structure.

3.4 HtmlFlow

The primary goal of HtmlFlow was to provide a DSL that helps developers to write safe HTML in Java programs. HtmlFlow distinguishes for providing:

1. *data structureless* – a key factor that contributes to achieve better performance than competition in several benchmarks;
2. *method chaining* that allows the calls to be chained together in a single statement (i.e. *method chaining*).
3. *HTML safety* ensuring that the resulting HTML is conforming to a valid HTML document.

Data Structureless A distinct aspect of HtmlFlow in comparison to other DSLs is that it follows a *data structureless* approach. This means that invoking the HtmlFlow API does not instantiate any objects representing nodes or elements, and it neither stores or maintains such useless objects in memory. Instead HtmlFlow API methods emit HTML on demand as they are being invoked. This is one of the key aspects that makes HtmlFlow one of the most performant engines in a diversity of state of the art benchmarks[10,32,9].

This core characteristic is the result of the fluent API nature of HtmlFlow, which enforces methods to be invoked by the same order they are chained. We will put it clear after giving some details about the HtmlFlow API design in next section.

Method chaining Regarding the *method chaining* requirement we stated that every HTML element (i.e. instance of `Element`) returned by a method call should have methods to create the next inner HTML element, such as the following pipeline:

`div().table().tr()....`, which should create a `div` element containing a `table` that in turn will have a table row element `tr`. On the other hand, we would like to have an auxiliary method that allows to navigate back in the elements tree. So, all HtmlFlow elements have the `__()` method that returns the corresponding element's parent. Moreover, this method should return the parent element with the correct type. Regarding the previous example, this means that calling `__()` after `tr()` should return an instance of `Table` whereas calling `__()` after `table()` should return an instance of `Div`. When we navigate back to the parent element we would like to get a consistent route.

We tackle this issue through *generics*[26], which allow us to keep track of the tree structure of the elements that are being created and keep adding elements, or moving up in the tree structure without losing the type information of the parent. In Listing 3.4 we can observe how we can take advantage of the type arguments.

Listing 3.4: Explicit use of type arguments in the subtypes of `Element`

```
Html<Element> html = new Html<>();
Body<Html<Element>> body = html.body();

P<Header<Body<Html<Element>>>> p1 = body.header().p();
P<Div<Body<Html<Element>>>> p2 =  body.div().p();

Header<Body<Html<Element>>> header =  p1.__();
Div<Body<Html<Element>>> div = p2.__();
```

When we create the `Html` element we should indicate that it has a parent, for consistency. Then, as we add elements, such as `Body`, we automatically return the recently created `Body` element, but with parent information that indicates that this `Body` instance is descendant of an `Html` element. After that, we create two distinct `P` elements, `p1`, which has an `Header` parent, and `p2`, which has a `Div` parent. This information is reflected in the type of both variables. Lastly, we can invoke the `__()` method,which returns the current element parent, and observe that each `P` instance returns its respective parent object, with the correct type.

In the example presented in Listing 3.4 the usage of the *fluent interface* might seem to be excessive verbose to define a simple HTML document. Yet, for most common purposes we can suppress the auxiliary variables and simplify its usage chaining method calls as we show in Listing 3.5.

HTML Safety Finally, the invocation chain should produce valid HTML. This means that HtmlFlow should not allow statements like `img().table();` since in HTML we may not include a table inside an image. To that end, every instance of `Element` returned

Listing 3.5: Example of the implicit use of type arguments in HtmlFlow API

```
Html<Element> html = new Html<>()
  .body()
    .header()
      .p().__()
    .__() // header
    .div()
      .p().__();
    .__() // div
  .__(); // body
```

by HtmlFlow only provides methods to create children elements conforming to HTML 5 rules. So, every concrete element is represented by a class extended from `Element` with a subset of methods that respects the kind of permitted children elements.

Following this idea we have developed the `xmlet` platform[12] that parses the XSD schema of HTML5 and automatically generates all elements classes and interfaces required by HtmlFlow. All the implementations of the `Element` interface corresponding to all kind of HTML elements available in HTML 5 are automatically built from the XSD definition of the HTML 5. These implementations are part of the **HtmlApi** auxiliary library used by the HtmlFlow as denoted in Figure 1.

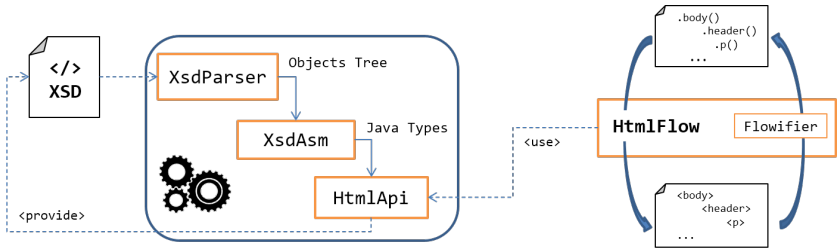

Fig. 1: xmlet framework build process and its organization in three main components: XsdParser, XsdAsm and HtmlApi.

In turn, the **HtmlApi** is built with the support of ASM [4] (bytecode instrumentation tool) implemented by the **XsdAsm** component, which consumes the information gathered and provided by the **XsdParser**.

The HtmlFlow also includes the **Flowifier** feature that allows to get an HtmlFlow template view definition from the corresponding HTML source. This feature was crucial to translate the PetClinic HTML views to the equivalent definition in HtmlFlow idiom.

4 HoT: Higher-Order Templates

A higher-order template (HoT) is an advanced technique for resolving a *template view* progressively as data from its *context object* is being available, rather than waiting for whole data and resolve the entire template. A higher-order template (HoT) defines a *template view* as a function and its *context object* as other function received by argument. Also, a higher-order template can receive other templates as parameters. In this case, these parameters play the role of *partial views*. Like a higher-order function may take one or more functions as arguments, a higher-order template may take one or more *templates views* as arguments. This compositional nature enables reusing template logic.

Templates in HtmlFlow are specified through Java functions, which can be defined as named or anonymous functions (i.e. *lambdas*) For example, the template of Listing 1.1 can be expressed in HtmlFlow with the `tracksTpl` function of Listing 4.1.

Listing 4.1: HtmlFlow template function for a division element with a dynamic unordered list

```
HtmlTemplate<Stream<Track>> tracksTpl =  (view, tracks) -> view
    .div()
      .ul()
      .of(ul -> tracks.forEach (item -> ul
        .li().text(item.getName()).__() // li
      ))
      .__ () // ul
    .__(); // div
```

The `tracksTpl` function receives two parameters: an HtmlFlow view and a context object (e.g. `tracks`). The `view` parameter provides the HTML *fluent interface* [17] with methods corresponding to the name of HTML elements and two additional methods: 1) `__()` to close an HTML element tag, and 2) `of(elem -> ...)` useful to chain a statement that starts with an expression different from the previous element (i.e. `elem` in the lambda). The `tracks` parameter acts like the *model* in the *model-view-controller* pattern [13]. Here the `tracks` is a Java `Stream`, which is an abstraction over a lazy sequence of objects (i.e. instances of `Track`). Some may argument that a Java `Stream` is not a first-class function, such as we have claimed that a *context object* would be for a HoT. Yet, that is only a design choice of the Java environment, since in truth any sequence (such as `Stream`) can be implemented through a higher-order function[3]. Nevertheless, in this case, the `tracks` object is traversed in the `forEach` method call (i.e. `tracks.forEach(...)`) chained within the template definition.

Regarding a template with *partial views* then the corresponding function should receive a further argument for each partial view. For example, considering that the `tracksTpl` function takes an additional `footer` argument, then it can include this partial view through the method `addPartial()` that is available in all elements objects. We can chain the call to `addPartial()` in the template definition as depicted in the example of the Listing 4.2, that adds the `footer` after the definition of the unordered list.

Listing 4.2: HtmlFlow template with a partial view `footer`

```
HtmlTemplate<Stream<Track>> tracksTpl = (v, trks, footer) -> v
    .div()
    ... // adds ul and an li for each track
    .of(div -> div.addPartial(footer));
    .__() // div
```

If a partial view has no context object and does not require model binding, then we can discard the template function and directly create that view from an expression, through the `view()` factory method of the class `StaticHtml`. For example, we may define a billboard division (i.e. `bbView`) as depicted in the following view definition:

```
HtmlView bbView = StaticHtml.view().div().text("Dummy billboard").__();
```

Another advantage of using views as first-class functions is to allow views composition. For example, if want to define a partial view (e.g. `footerView`) with a placeholder for another partial view (e.g. `banner`), then we may define a `footerView` method that takes an `HtmlView` as the `banner` parameter and returns a new `HtmlView` as depicted in Listing 4.3.

Listing 4.3: Partial view definition of a `footer` that takes another `banner` view as parameter

```
HtmlView footerView(HtmlView banner) {
  return StaticHtml.view()
    .div()
      .of(div -> div.addPartial(banner))
      .p().text("Created␣with␣HtmFlow").__() // p
    .__(); // div
}
```

Thus, we may finally compose the `tracksTpl` template of Listing 4.2 with the `footerView`, which in turn will be filled with the `bbView`. This creates the following pipeline: `tracksTpl <- footerView <- bbView`.

For the `tracksTpl` function we can create a corresponding view (i.e. `tracksView`) through the `view()` factory method of the class `DynamicHtml` as depicted in line 2 of Listing 4.4. Finally, we may compose all the parts of the `tracksView` through the `render` method of `HtmlView`. For example, given a `tracks` stream, the `footerView` and the `bbView` we may resolve the `tracksView` as depicted in line 3 of Listing 4.4. Here we can observe the pipeline: `tracksView <- footerView <- bbView` with `tracksView` taking the `footerView` as argument, which in turn receives the `bbView` as argument.

Listing 4.4: Composing and resolving the `tracksView`

```
Stream<Track> tracks = ...
HtmlView<Stream<Track>> tracksView = DynamicHtml.view(tracksTpl);
String html = tracksView.render(tracks, footerView(bbView));
```

Having all the compositional parts of a template view defined as first-class functions (the template itself, the context object and partial views) is a key feature to achieve the HtmlFlow compositional nature.

5 Templates Idioms

We will use the PetClinic Spring application[11] as a use case of web templates to compare several Thymeleaf dialects with the equivalent construction in a DSL for HTML. The templates are responsible for displaying a *data model* and performing any display logic that is particular to the type of view being rendered. In this, case we are going to analyze how the following templates patterns are solved through specific Thymeleaf dialects for each particular view:

1. variable assignment and conditional evaluation – `createOrUpdateOwnerForm.html` view using `th:with` and `th:text`.
2. switch statement – `inputField.html` view using `th:switch` and `th:case`.
3. if statement and iteration loop – `vetList.html` view using `th:if` and `th:each`.
4. binding inputs and preprocessing expression – `selectField.html` view using `th:field` and dialect `__${expression}__`.
5. fragments and layouts – `layout.html` view using `th:fragment` and `th:replace`.

Here we are only illustrating a small subset of Thymleaf dialects including 10 different building blocks. We also highlight how a DSL for HTML can suppress the need of this auxiliary features giving examples of the same views built with HtmlFlow, KotlinX.html or React that does not require any auxiliary mechanism beyond their host programming language, i.e. Java, Kotlin or JavaScript.

5.1 Variable assignment and conditional evaluation

Thymeleaf The PetClinic application uses the same view to update or create an owner object. This view presents an HTML form with the submit button presented in Listing 5.1, which contains the label *'Add Owner'* or *'Update Owner'* depending on whether the view is returned from the `/owners/new` or `/owners/ownerId/edit` path.

Listing 5.1: Thymleaf template button to create or update owner.

```
1 <button
2   th:with="text=${owner.isNew()} ? 'Add Owner' : 'Update Owner'"
3   class="btn btn-default"
4   type="submit"
5   th:text="${text}">
6 </button>
```

To that end, the corresponding HTML button defines its text content through the Thymeleaf attribute `th:text`, i.e. line 5 of Listing 5.1. The value of this attribute is

equals to result of the expression `${text}`, where `text` is a variable previously declared and assigned with attribute `th:with` in line 2 of Listing 5.1. In turn, the assignment statement of line 2 uses the Thymeleaf ternary operator to build an expression that results in *'Add Owner'* or *'Update Owner'* depending on whether the `owner` object has its property `isNew()` set to true, or not. Note, this statement uses a mix of Java to evaluate the property `isNew()` of the `owner` object and the Thymeleaf dialect to build the ternary expression. Thus, the resulting complexity comes from the use of the additional Thymeleaf dialect and two further attributes (`th:text` and `th:with`) to achieve the desired behavior.

HtmlFlow In opposition, HtmlFlow let programmers interleave any kind of Java statements to achieve this goal. We may choose between a Java `if` statement or a ternary operator according to the developer programming stylistic preferences. In Listing 5.2 we present the equivalent HtmlFlow definition to the button of Listing 5.1. Note that we can discard the auxiliary variable `text` used in the Thymeleaf example, since we are customizing the button through a pure Java statement.

Listing 5.2: HtmlFlow template button to create or update owner.

```
1 .button()
2     .attrClass("btn␣btn-default")
3     .attrType(EnumTypeButtonType.SUBMIT)
4     .text(owner.isNew() ? "Add␣Owner" : "Update␣Owner")
5 .__()
```

Kotlinx.html Kotlinx.html shares the same expressiveness advantages of HtmlFlow: fluent and HTML safety. Despite the idiomatic and syntax differences between Java and Kotlin, the definition of Listing 5.2 share the same layout of Listing 5.3.

Listing 5.3: KotlinX.html template button to create or update owner.

```
1 button {
2     classes = setOf("btn␣btn-default")
3     type = ButtonType.submit
4     text(if (owner.isNew) "Add␣Owner" else "Update␣Owner")
5 }
```

ReactJS Similar to HtmlFlow and Kotlinx.html the React also allows the use of the core framework programming language (i.e. JavaScript) in its entire plenitude. Yet, since JSX is an external DSL mixed with pure JavaScript we must delimit JavaScript blocks with a special character, i.e. $\{\ldots\}$. Due to the JSX dialect nature, some words are neither HTML nor JavaScript, such as `className` in line 2, of Listing 5.4. Nevertheless, the conditional evaluation stated with the JavaScript ternary operator in line 5 is analogous to the expression of the same line in Listing 5.2.

Listing 5.4: React template button to create or update owner.

```
<button
  className='btn btn-default'
  type='submit'
  onClick={this.onSubmit}>
    {owner.isNew ? 'Add Owner' : 'Update Owner'}
</button>
```

5.2 Switch statement

Another way to display content conditionally is using the equivalent switch statement. The PetClinic application takes advantage of this construction to provide a single input control implementation with different behaviors according to the type of data. To that end, it defines a Thymeleaf *fragment* that exports an HTML `div` element with a `label` and an `input` elements. In Listing 5.5 we present a simplified version of the `inputField.html` fragment that only includes the `input` definition.

Listing 5.5: Thymleaf fragments `inputField.html`.

```
1  <th:block th:fragment="input (label, value, type)">
2    <div class="form-group">
3    ...
4      <div th:switch="${type}">
5        <input th:case="'text'" type="text" value="{value}" />
6        <input th:case="'date'" type="text" value="{value}"
7          placeholder="YYYY-MM-DD"
8          title="Enter a date in this format: YYYY-MM-DD"
9          pattern="(?:19|20)[0-9]{2}-(?:(?:0[1-9]|1[0-2])-..."/>
10     </div>
11   </div>
12 </th:block>
```

This fragment is defined with the tag `th:block` and attribute `th:fragment` that we will describe ahead in section 5.5. Note in Listing 5.5 that depending on the parameter `type` (`th:switch` on line 4) this fragment will present a different kind of `input` element that can be according a free input text or a date format. Yet, part of the definition of this input is repeated in lines 5 and 6, because there are a couple of attributes definitions shared in both cases, namely the attributes `type` and `value`. On the other hand, the HtmlFlow template definition of Listing 5.6 does not need to repeat the assignment of the attributes `type` and `value` and thus, it only includes a single use of these attributes (line 4).

Thanks to the compositional nature of higher-order templates this fragment is defined with a first-class function as any other template view in HtmlFlow, avoiding specific markers such as Thymleaf `<th:block th:fragment>`. Also this HtmlFlow partial view does not require the additional parameter `type` to check the class of the instance `value`. Instead, we can simply use the Java `instanceof` operator (line 5) to verify the type of the instance `value`.

Listing 5.6: HtmlFlow fragment `InputField`.

```
HtmlView<LabelAndValue> view = DynamicHtml.view((v, model) -> v
  .div().attrClass("form-group")
  ...
    .input().attrType(TEXT).attrValue(model.value)
        .of(__ -> { if(model.value instanceof LocalDate) input
          .attrPlaceholder("YYYY-MM-DD")
          .attrTitle("Enter␣a␣date␣in␣this␣format:␣YYYY-MM-DD")
          .attrPattern("(?:19|20)[0-9]{2}-(?:(?:0[1-9]|...));
        })
    .__()
  .__()
)
```

5.3 If statement and iteration loop

The `VetList` view aims to show a list of all veterinaries, where each row presents the veterinarian's full name and all its specialties concatenated in a single String. Yet, if a veterinary does not have any specialty then it should present the String `none`. To that end, the corresponding Thymeleaf template of Listing 5.7 takes advantage of the `th:each` to traverse the `vets.vetList` (line 2) and to join all specialties (line 6 and 7). On the other hand, it uses the `th:if` (line 9) to check whether the list is empty and present the String `none` if it is.

Listing 5.7: `vetList.html` Thymeleaf sample for displaying veterinaries.

```
<tbody>
  <tr th:each="vet␣:␣${vets.vetList}">
    <td th:text="${vet.firstName␣+␣'␣'␣+␣vet.lastName}"></td>
    <td>
      <span
        th:each="specialty␣:␣${vet.specialties}"
        th:text="${specialty.name␣+␣'␣'}"
      />
      <span th:if="${vet.nrOfSpecialties␣==␣0}">none</span>
    </td>
  </tr>
</tbody>
```

Again a DSL for HTML suppresses the use of additional dialects and may achieve the desired layout only with Java statements, such as the HtmlFlow sample presented in Listing 5.8. Here we are only taking advantage of the `forEach()` traversal method provided by the Java Stream API (line1) and a Java ternary operator (line 5).

Listing 5.8: `VetList` HtmlFlow sample for displaying veterinaries.

```
.tbody().of(tbody -> vets.forEach(v -> tbody
  .tr()
```

```
3       .td().text(v.getFirstName() + "␣" + v.getLastName()).__()
4       .td().span()
5         .text(v.nrOfSpecs() == 0 ? "none" : join("␣", v.specs()))
6       .__().__()
7     .__() //tr
8   )).__() //tbody
```

5.4 Binding inputs and preprocessing expression

The `select` HTML element has similar behavior to the well known drop-down component, also known as *combo box*. The `SelectField` view aims to bind all elements of an `items` list to a `select` element. Yet, it must also assign the HTML attribute `selected` of the corresponding `option` element that matches the `selected` parameter passed to this view. This is exactly the algorithm implemented by the HtmlFlow `SelectField` view that receives a `src` parameter with the `items` list and the `selected` object, as presented in Listing 5.9.

Listing 5.9: `SelectField` HtmlFlow partial view sample.

```
1  .select()
2    .of(select -> src.items.forEach(item -> select
3      .option().attrValue(item.toString()).of(opt -> {
4          if(src.selected.equals(item.toString()))
5            opt.attrSelected(true);
6          })
7          .text(item)
8      .__() //option
9    ))
10 .__() //select
```

In line 2 of Listing 5.9 we are traversing the `items` list through the `forEach()` method and in line 3 we are adding an HTML `option` element for each of those items. If an item is equal to the `selected` parameter (line 4) then we will add the attribute `selected` to the related `option` element (line 5). Finally in line 7 we include the `item` as the text of the `option` element.

The Thymeleaf corresponding version mitigates all the details of the algorithm expressed in Listing 5.9. Yet, it also requires keen knowledge of the Thymeleaf dialects able for preprocessing expressions as depicted in Listing 5.10.

Listing 5.10: `selectField.html` Thymeleaf partial view sample.

```
1  <select th:field="*{__${name}__}">
2    <option th:each="item␣:␣${items}" th:value="${item}"
3      th:text="${item}">
4        dog
5    </option>
6  </select>
```

First the `th:field` attribute (line 1 of Listing 5.10) behaves differently depending on whether it is attached to an `input`, `select` or `textarea`. Then values for

the `th:field` must be selection expressions denoted as `*{...}` because they will be evaluated on the form-backing bean and not on the context variables. And finally the `__${...}__` is a *preprocessing expression*, which is an inner expression that is evaluated before the whole expression. This is the tricky part of the Thymeleaf dialect that will check the property identified by `name` in the `item` object (line 2) that may be selected, or not, in the corresponding option element.

5.5 Fragments and layouts

Layouts aim to reuse a common structure (usually composed by header, footer and navigation bar) among different template views. To that end the PetClinic application defines a `layout.html` fragment parametrized with a `template` view that is included dynamically based on the inclusion of template fragments. In the sample of Listing 5.11 the line 2 defines the `layout` fragment and in line 5 it includes the `template` view embedding its content within the layout through the `th:block th:insert` tag.

Listing 5.11: `layout.html` view sample.

```
1 <!doctype html>
2 <html th:fragment="layout␣(template)">
3   ...
4     <div class="container">
5       <th:block th:insert="${template}"/>
6   ...
7 </html>
```

On the other hand, each view inheriting the `layout.html` should specify the content that should be passed to the layout as the `template` parameter. In Listing 5.12 we present an example of how the PetClinic application includes a Thymeleaf template view within the `layout.html`. In line 2 it uses the `th:replace` to invoke the layout and pass the `body` element as argument. In turn, this `body` element includes all the content to be presented as the `VetList` web page.

Listing 5.12: `vetList.html` view inclusion in `layout.html`.

```
1 <html
2   th:replace="~{fragments/layout␣::␣layout␣(~{::body},'vets')}">
3 <body>
4   <h2>Veterinarians</h2>
5   ...
6 </body>
```

Again HtmlFlow takes advantage of functions composition to define a template layout. In this case we define a distinct class `Layout` with a static field `view` corresponding to the layout function, as depicted in Listing5.13. Then, it should include the partial view through the `addPartial` method (line 7).

Listing 5.13: `vetList.html` view inclusion in `layout.html`.

```
1 public class Layout {
2   static DynamicHtml view = view((v, model, partials) -> { v
```

```
3       .html()
4       ...
5         .div().attrClass("container")
6         ...
7           .of(__ -> v.addPartial(partials[0], model);)
8     }
9   }
```

In opposition to the inclusion approach followed by Thymeleaf templates, the HtmlFlow templates are agnostic with respect to the layout. On the other hand, the controller is responsible for composing the layout with the template view as presented in the following example that assembles the `VetList` view:

```
Layout.view.render(vets, VetList.view);
```

6 Performance Evaluation

To perform an unbiased comparison we used two of most popular benchmarks for template engines: 1) Comparing Template engines for Spring MVC, simply denoted as Spring templates benchmark[32] and 2) JMH benchmark for popular Java template engines[7]. We integrated the missing engines in our forks of these benchmarks available at Github repositories: xmlet/spring-comparing-template-engines and xmlet/template-benchmark. These tests were done on a local machine running Microsoft Windows 10, Education OS Version: 10.0.17134 with Java(TM) SE Runtime Environment 18.9 (build 11+28) Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11+28, mixed mode) and with Intel(R) Core(TM) i7-7700 HQ CPU @ 2.80GHz, 2808 Mhz, 4 Cores.

6.1 Spring templates benchmark

This benchmark uses the Spring MVC framework to host a web application that provides a route for each template engine to benchmark. Each template engine uses the same template (i.e. `Presentations`) and receives the same information to fill the template, which makes it possible to flood all the routes with a high number of requests and asserts which route responds faster, consequently asserting which template engine is faster. Following the benchmark recommendation we used the Apache HTTP server benchmarking tool as a stress tester running it with 25 concurrent requests and 25000 requests in total, which corresponds to the following settings:

```
ab -n 25000 -c 25 -k http://localhost:8080/<template engine>
```

The performance of each template engine was measured according to the guidelines specified in the Spring benchmark repository, which states at least two dry runs with the exact same settings, to make sure that initialization of the engines, warm up of the JVM and additional caches have taken place.

After that, we calculate the average time taken by 5 iterations of the same benchmark for each template engine. The results are presented in Figure 2 corresponding to

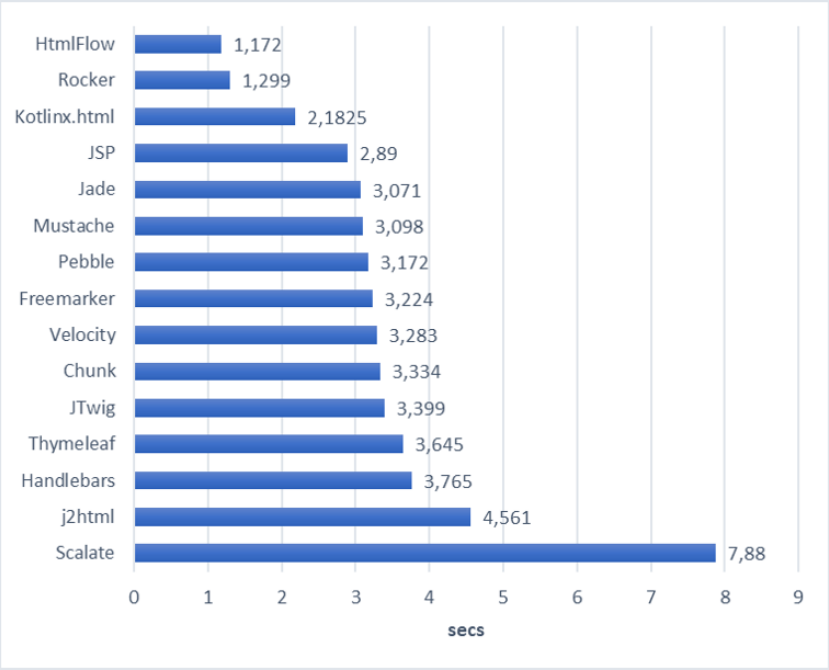

Fig. 2: Performance results in seconds for Spring templates benchmark.

the total time taken for processing 25000 requests with a concurrency level of 25 (lower is better).

This approach of measuring the template engines performance is misleading because the render time of the template engines is dismissible when compared to the overhead introduced by the Spring framework. Thus, the performance differences among some template engines are very tiny.

In this context, we mostly agree with the JMH templates benchmark proposal, which uses JMH to implement the benchmark. So we end up using the `Presentations` template from the Spring benchmark and integrated it in our fork of JMH templates benchmark to get more reliable measures.

6.2 JMH benchmark for Java template engines

The advantage of this benchmark is that it focuses exclusively on evaluating the rendering process of each template engine. In this case, it does not use any web servers to handle a request, which is a more consistent approach. The general idea of this benchmark is the same, it includes many template engine solutions that define the same template and use the same data as a context object to generate the resulting HTML document. But, in this case, it uses Java Microbenchmark Harness, which is a Java tool to benchmark code. With JMH we indicate which methods to benchmark with annotations and configure different benchmark options such as the number of warm-up iterations, the

number of measurement iterations, or the numbers of threads to run the benchmark method.

This benchmark contained eight different template engines: Freemarker, Handlebars, Mustache, Pebble, Thymeleaf, Trimou, Velocity, and Rocker. In addition, we integrated J2Html, KotlinX.html, and HtmlFlow.

The JMH templates benchmark used only one template, i.e. `Stocks` template. In addition we included the `Presentations` template from the Spring templates benchmark. By using two different templates the objective was to observe if the results were maintained throughout the different solutions. The main difference between both templates is that the `Stocks` template introduces much more binding operations: 1) it has more fields that will be accessed in the template, and 2) it has twenty objects in the default data set while Presentations only has ten objects in his data set. This means that the `Stocks` template will generate more string operations to the classic template engine solutions and more Java method calls for the solutions that have the template defined as a function in Java or Kotlin.

In Figure3 we present the results of the JMH templates benchmark corresponding to the mean value of five forked iterations, each one of the forks running eight different iterations, performed after five warm-up iterations. This approach intends to remove any outlier values from the benchmark. The benchmark values were obtained without any other programs running, nor background tasks, and only with the command line running the benchmark.

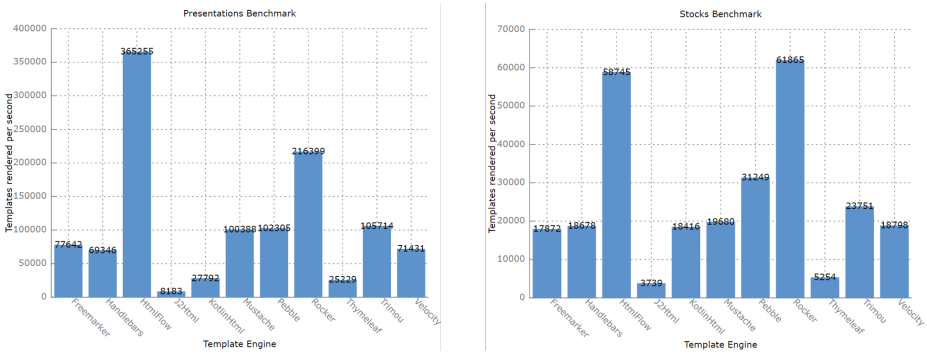

Fig. 3: Performance results in operations per second for JMH templates benchmark.

Regarding the classical template engines, i.e. Mustache, Pebble, Freemarker, Trimou, Velocity, Handlebars, and Thymeleaf, we can observe that most of them share the same level of performance, which should be expected since they all roughly have the same methodology. Regarding the remaining template engines, i.e. Rocker, J2Html, KotlinX.html, and HtmlFlow, the situation is diverse. On one hand, we have Rocker, which gives great performance when the number of placeholders increases, i.e. the `Stocks` benchmark, taking into consideration that it provides many compile-time verifications regarding the context objects, it presents a good improvement in comparison

to the classical template engines. On the other hand, we have J2Html and KotlinX.html. Regarding J2Html we observe that the tradeoff of moving the template to a function of the environment language (i.e. Java) had a significant performance cost since it is consistently one of the two worst solutions performance-wise. Regarding KotlinX.html, its approach was definitely a step towards in the right direction since it validates the HTML rules and introduces compile-time validations, but, either due to the Kotlin language performance issues or poorly optimized code, it did not achieve the level of acceptable performance.

Lastly, HtmlFlow proved to have the best performance with the `Presentation` template. It achieved performance gains that surpass the second best solution by twice the operations per second. Regarding the `Stocks` template, the HtmlFlow held the top place even though the number of placeholders for dynamic information increased significantly. If we compare HtmlFlow to a similar solution, KotlinX.html, we observe a huge gain of performance on the HtmlFlow part.

In conclusion, HtmlFlow introduces domain language rule verification, removes the requirements of text files and additional syntaxes, adds many compile-time verifications, and, while doing all of that this, is still the best solution performance wise.

7 Conclusions and Future Work

Electing a template engine may be a choice between the set of available templating features and the simplicity of resulting templates. And, it is not viable to stretch both axis. For example, Mustache is a logic-less template syntax, which provides minimal templating through double-braces tags (i.e. $\{\{\ldots\}\}$), but at same time it is too much restrictive on the limited set of available features. On the other hand, Thymeleaf provides an enlarged number of functionalities, but it turns templates very intricated.

As more features we take from the engine more complex become the templates. And, as more simple we require the template, less features the engine provides.

In truth, web frameworks are build on top of a high-level programming language and enriching templating dialects will enforce developers to deal with two different programming languages. Moreover, these two languages are heterogeneous and have different building natures. While the host programming language spreads from the functional to imperative paradigms, the templating dialects are usually embedded in markups, which results in a mix of declarative and imperative idioms. This anti-paradigm inception of templating dialects turn templates difficult to manage as we have shown with 10 of the most used building blocks of Thymeleaf.

These problems arises from two unavoidable facts: 1. HTML is the main user interface programming language, and 2. HTML is a markup language with distinct characteristics from the host programming language. Thus we argue that empowering templating dialects is a counterproductive approach that leads to harmful templates.

In this work we propose a different methodology to answer developers needs on web templating. Rather than augmenting HTML markups with additional templating constructions we propose to keep HTML plain and let programmers interleave HTML with the use of the host programming language.

To that end we take advantage of an internal DSL to HTML to bring the power of the host programming language (such as Java) into web templates. We have shown the use of a DSL to HTML in three different hosting languages, namely Java, Kotlin and Javascript through the related DSL libraries HtmlFlow, KotlinX.Html and JSX. Furthermore, we have presented how the most used template patterns (e.g. using Thymeleaf) can be easily achieved with pure Java statements and the support of HtmlFlow Java library without any further dialect beyond the Java programming language.

To prove the effectiveness of our proposal we have used a well-known sample web application - PetClinic - and migrated the exiting Thymeleaf templates to Java with the support of HtmlFlow library. The last release of HtmlFlow (3.5) includes the translation tool `Flowifier` that allows the conversion of an HTML document in its equivalent definition in HtmlFlow idiom. This tool was essential to enhance the translation process of existing templates into the related HtmlFlow definition. The wide variety of templating idioms used in PetClinic and their successful translation to Java and Htmlflow, give us confidence about future applications of HtmlFlow to other real web applications.

References

1. Alur, D., Malks, D., Crupi, J.: Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR, Upper Saddle River, NJ, USA (2001)
2. Ase, D.: Kotlin dsl for html. Tech. rep., https://j2html.com/ (2015), https://j2html.com/
3. Baker, H.G.: Iterators: Signs of weakness in object-oriented languages. SIGPLAN OOPS Mess. **4**(3), 18–25 (Jul 1993). https://doi.org/10.1145/165507.165514, https://doi.org/10.1145/165507.165514
4. Binder, W., Hulaas, J., Moret, P.: Advanced java bytecode instrumentation. In: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java. pp. 135–144. PPPJ '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1294325.1294344, http://doi.acm.org/10.1145/1294325.1294344
5. Breslav, A.: Kotlin Language Documentation (2016), https://kotlinlang.org/docs/kotlin-docs.pdf
6. Brogden, B., D'Cruz, C., Gaither, M.: Cocoon 2 programming: web publishing with XML and Java. John Wiley & Sons (2006)
7. Bösecke, M.: Jmh benchmark of the most popular java template engines. Tech. rep., https://github.com/mbosecke/template-benchmark (2015), https://github.com/mbosecke/template-benchmark
8. Carvalho, F.M.: Htmlflow java dsl to write typesafe html. Tech. rep., https://htmlflow.org/ (2017), https://htmlflow.org/
9. Carvalho., F.M., Duarte., L.: Modern type-safe template engines (2018), https://dzone.com/articles/modern-type-safe-template-engines
10. Carvalho., F.M., Duarte., L.: Hot: Unleash web views with higher-order templates. In: Proceedings of the 15th International Conference on Web Information Systems and Technologies - Volume 1: WEBIST,. pp. 118–129. WEBIST '19, INSTICC, SciTePress (2019). https://doi.org/10.5220/0008167701180129
11. Donald, K., Isvy, M., Leau, C.: Spring petclinic sample application. Tech. rep., https://projects.spring.io/spring-petclinic// (2013), https://projects.spring.io/spring-petclinic/

12. Duarte., L., Carvalho., F.M.: xmlet. Tech. rep., https://github.com/xmlet (2018), https://github.com/xmlet
13. E. Krasner, G., Pope, S.: A description of the model-view-controller user interface paradigm in the smalltalk80 system. Journal of Object-oriented Programming - JOOP **1**, 26–49 (1988)
14. Evans, E., Fowler, M.: Domain-driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2004), https://books.google.pt/books?id=7dlaMs0SECsC
15. Fernández, D.: Thymeleaf. Tech. rep., https://www.thymeleaf.org/ (2011), https://www.thymeleaf.org/
16. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
17. Fowler, M.: Domain Specific Languages. Addison-Wesley Professional, 1st edn. (2010)
18. Freeman, S., Mackinnon, T., Pryce, N., Talevi, M., Walnes, J.: Jmock library for test-driven development with mock objects. Tech. rep., http://jmock.org (2008), http://jmock.org
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
20. Heijlsberg, A., Torgersen, M.: The .net standard query operators (2007)
21. Hors, A.L., Hégaret, P.L., Wood, L., Nicol, G., Robie, J., Champion, M., Arbortext, Byrne, S.: Document object model (dom) level 3 core specification. Tech. rep., https://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/ (2004), https://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/
22. Jin, X., Wah, B.W., Cheng, X., Wang, Y.: Significance and challenges of big data research. Big Data Res. **2**(2), 59–64 (Jun 2015). https://doi.org/10.1016/j.bdr.2015.01.006, http://dx.doi.org/10.1016/j.bdr.2015.01.006
23. Johnson, R., Hoeller, J., Donald, K., Sampaleanu, C., Harrop, R., Risberg, T., Arendsen, A., Davison, D., Kopylenko, D., Pollack, M., et al.: The spring framework–reference documentation. interface **21**, 27 (2004)
24. Landin, P.J.: Correspondence between algol 60 and church's lambda-notation: part i. Communications of the ACM **8**(2), 89–101 (1965)
25. Landin, P.J.: The next 700 programming languages. Communications of the ACM **9**(3), 157–166 (1966)
26. Marx, S., Odersky, M., Buckley, A.: Jsr 14: Add generic types to the java. Tech. rep., Java Community Process (2004), https://jcp.org/en/jsr/detail?id=14
27. Mashkov, S.: Kotlin dsl for html. Tech. rep., https://github.com/Kotlin/kotlinx.html (2015), https://github.com/Kotlin/kotlinx.html
28. Meijer, E.: Your mouse is a database. Queue **10**(3), 20:20–20:33 (Mar 2012). https://doi.org/10.1145/2168796.2169076, http://doi.acm.org/10.1145/2168796.2169076
29. Mutschler III, E.O., Stefaniak, J.P.: Method for extending the hypertext markup language (html) to support enterprise application data binding (Aug 17 1999), uS Patent 5,940,075
30. Parker, H.: Opinionated analysis development. PeerJ Preprints **5**, e3210v1 (2017)
31. Parr, T.J.: Enforcing strict model-view separation in template engines. In: Proceedings of the 13th International Conference on World Wide Web. pp. 224–233. WWW '04, ACM, New York, NY, USA (2004). https://doi.org/10.1145/988672.988703, http://doi.acm.org/10.1145/988672.988703
32. Reijn, J.: Comparing template engines for spring mvc. Tech. rep., https://github.com/jreijn/spring-comparing-template-engines (2015), https://github.com/jreijn/spring-comparing-template-engines
33. Resig, J.: Pro JavaScript Techniques. Apress (2007)
34. Singh, I., Johnson, M., Stearns, B.: Designing enterprise applications with the J2EE platform. Addison-Wesley Professional (2002)

35. Ted, N.: Literary Machines. Mindful Press, Sausalito, California (1994)
36. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM **11**(6), 419–422 (Jun 1968). https://doi.org/10.1145/363347.363387, https://doi.org/10.1145/363347.363387
37. Walke, J.: React javascript library for building user interfaces. Tech. rep., https://reactjs.org/ (2013), https://reactjs.org/