

HoT: Unleash Web Views with Higher-order Templates

Fernando Miguel Carvalho^a and Luis Duarte

ADEETC, ISEL, Polytechnic Institute of Lisbon, Portugal
{mcarvalho, lduarte}@cc.isel.ipl.pt

Keywords: Template View, Big Data, Front-end, Web Application, HTML.

Abstract: Over the past decades, templates views have been the most used approach to build dynamic HTML pages. Simply put, a template engine (such as JSP, Handlebars, Thymleaf, and others) generates HTML by merging templates with given data models. Yet, this process may turn impractical for large data sets that postpone the HTML resolution until all data become available to the engine. This behavior results in poor user experience preventing the browser to render the end user-interface. In this paper we introduced the concept of higher-order templates (HoT) provided in Java implementation of HtmlFlow, which allows HTML to be resolved on demand as data becomes available. This lets the user-interface to be rendered incrementally by the browser in line with the availability of the data. Finally we also show some advantages of HtmlFlow over state of the art front-end frameworks such as ReactJS.

1 INTRODUCTION

Contrary to static HTML pages, which do not change from request to request, dynamic Web pages may take results from a data source, like database or web service, and embed them into the HTML. Dynamic Web pages share a similar structure to static HTML but they also include markers that can be resolved into calls to gather dynamic information. Since the static part of the page acts as a template, this pattern is known as *template view* (Fowler, 2002) or *template-based view strategy* (Alur et al., 2001). When the template view is used to service a request, the markers are replaced by the results of corresponding computations. The parsing and replacement process (i.e. *resolution*) is the main role of the *template engine* (Parr, 2004).

Template engines distinguish themselves by the idiom and subset of available markers to control the dynamic content. Generally, all engines provide a specialized tag for iteration, which can be used for example to generate a dynamic list such that one presented in Listing 1 expressed in the Handlebars idiom (an extension to the Mustache templating language (Lerner, 2011)).


In this example, the template receives a *context object* with a `tracks` property and generates a `li` element with the track's name for each item in the `tracks` list. The process of resolving the template

and producing HTML is *blocking* and it only finishes when all template markers have been replaced with data from the context object. In the case of an iteration tag (i.e. `#each` of Listing 1) it must traverse the entire list (i.e. `tracks`) to complete the template resolution. The larger the data (i.e. `tracks.size()`), the longer the engine takes to finish the resolution. While the engine is resolving the view, the browser is presenting a blank page to the end user giving him an unresponsive behavior.

Listing 1: Handlebars template view for a dynamic unordered list.

```
<ul>
  {{#each tracks}}
    <li>{{name}}</li>
  {{/each}}
</ul>
```

Visualization is one of the major concerns in big data (Agrawal et al., 2015; Storey and Song, 2017). Also, people tend to prefer web based applications as their favorite visualization support for data analysis (Yaqoob et al., 2016). Yet, even using data reduction strategies (Liu et al., 2013; ur Rehman et al., 2016) the resulting data sets are large enough to make unfeasible the use of template views. This problem arises in both, server-side or client-side (i.e. *front-end*) web application development. In the former case, the view resolution process prevents the server to send the response and in the latter, the number of DOM nodes (Hors et al., 2004) increases with the size

^a <https://orcid.org/0000-0002-4281-3195>

of dynamic data, incurring in additional performance overhead for the browser. For these reasons traditional ways of presenting data in web applications is inadequate to handle big data (Agrawal et al., 2015).

In this work, we propose a new approach based on *higher-order templates* (HoT), which aims to achieve a similar methodology to the server side template view pattern, but useful even on the presence of big data. Briefly, our goal is to establish data-centric push-style approach (Jin et al., 2015) and *push* the resulting HTML to the response stream as its content is being resolved. Rather than *pulling* data from a source and fully complete markers of a template view, we propose to react to data and *push* the resulting HTML as it is being resolved. This follows the idea of (Meijer, 2012) that states that the *Velocity* axis of big data ranges from pulling data from the source to pushing data to the clients.

This solution is implemented in HtmlFlow Java library, which implements a domain specific language (DSL (Mernik et al., 2005)) for HTML. Briefly, HtmlFlow implementation is based on the following key ideas:

1. do not block the template resolution until its completion (Meijer, 2012);
2. discard textual template files;
3. use HTTP chunked transfer encoding (Fielding and Reschke, 2014);
4. define views as first-class functions.

For the remainder of this paper we present in the next section state of the art solutions that deal with web presentation of large data sets. Then in Section 3 we present our proposal of higher-order templates. After that in Section 4, we present the Java library HtmlFlow, which provides a domain-specific language for HTML. In Section 5, we discuss related work in the HtmlFlow field. In Section 6, we present an experimental evaluation of developing a web application—*topgenius.eu*¹—that presents the top tracks ranking for a given country. This application uses data from Last.fm RESTful API and dynamically builds a list with hundreds of thousands of records. Here we compare three different technological approaches: 1) server-side template view, 2) ReactJS and 3) HtmlFlow. Finally, in Section 7 we conclude and discuss some future work.

2 STATE OF THE ART

Dealing with large data sets on web applications usually leads the browser to present an unresponsive behavior due to the template view resolution process. One way of dealing with this problem is to reduce big data into manageable size (Feldman et al., 2013). Yet, even using reduction techniques the resulting subset may still be large enough to turn impractical the use of a template engine.

Web applications usually mitigate this problem with two possible solutions (but not limited to): 1) numbered pagination, 2) infinite scrolling. Numbered pagination was one of the first solutions adopted to deal with the web presentation of big data, such as the results retrieved by web search engines. Later the MSN search’s image search was the first one to introduce the infinite scrolling technique with a single page of results, which you can scroll to automatically load more images (Farago et al., 2007). Both options reduce latency and turn the browser responsive, but both also have some drawbacks.

Numbered pagination forbids, for example, the end user to use the browser to find something among all resulting pages. On the other hand, infinite scrolling uses AJAX to fetch data (Kesteren et al., 2006) and DOM (Hors et al., 2004) to update the user-interface as data results become available. This approach is usually known as *front-end web development* (Smith, 2012). But, manipulating large collections of DOM nodes is slow and increases browser processing overhead. Moreover we lose the overall structure of the template view that now is mixed in with DOM manipulation logic.

Regarding the example of Listing 1 and its use with a front-end technology, then the template would be only the `ul` element with a unique identifier (e.g. `listOfTracks`) such as:

```
<ul id="listOfTracks"></ul>
```

In turn, the `li` elements would be manipulated programmatically through DOM.

Keeping a view definition with whole HTML structure similar to a template was one of the motivations for the appearance of front-end frameworks, such as Angular or ReactJs. This kind of frameworks allow the definition of a view based on a context object that is latter bound to the view by the framework through DOM.

Yet, these frameworks do not solve the performance issues and do not let end users to get page source code in an infinite scroll scenario. For example in ReactJs, the page source of resulting HTML will

¹Source code is available at github.com/xmllet/topgenius

only display `<div id="root"></div>` corresponding to the `div` element where the framework places HTML elements from DOM manipulation.

Briefly, web front-end development approaches circumscribe the server-side templates problems by moving the HTML resolution from the server to web agents (i.e. browser). This let the browser to update the user-interface as data becomes available. Yet, this approach also have the following drawbacks:

1. proliferation of the web development heterogeneity, which adds the use of DOM and AJAX;
2. loose the chance of getting the page source of the resulting HTML;
3. the number of DOM nodes are proportional to the size of dynamic data and big data sets may kill the browser performance.

3 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 can be expressed in `HtmlFlow` with the `tracksTpl` function of Listing 2.

Listing 2: `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* (Fowler, 2010) with methods corresponding to the name of HTML elements and an additional `__()` method to close an HTML element tag. The `tracks` parameter acts like the *model* in the *model-view-controller* pattern (E. Krasner and Pope, 1988). Here the `tracks` is a Java `Stream`, which is an abstraction over a lazy sequence of objects (i.e. instances of `Track`). In this case, the `tracks` object is traversed in the `forEach` method call (i.e. `tracks.forEach(...)`) chained within the template definition.

Also, an `HtmlFlow` template can receive other templates as parameters. In this case, these parameters play the role of *partial views*. Regarding a template view with partials 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`. We can chain the call to `addPartial` in the template definition as depicted in the example of the Listing 3, that adds the `footer` after the definition of the unordered list.

Listing 3: `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.

Listing 4: 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 3 with the `footerView`, which in turn will be filled with the `bbView`. This creates the following pipeline: `tracksTpl <- footerView <- bbView`.

Given the `tracksTpl` function we may create the corresponding view (i.e. `tracksView`) through the `view()` factory method of the class `DynamicHtml` as depicted in line 2 of Listing 5. 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 5. Here we can observe the pipeline: `tracksView <- footerView <- bbView`, where `tracksView` takes the `footerView` as argument, which in turn receives the `bbView` as argument.

Listing 5: Composing and resolving the `tracksView`.

```
Stream<Track> tracks = ...
var 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.

4 HtmlFlow

`HtmlFlow` is a Java library DSL for HTML. `HtmlFlow` is type-safe and conforms with HTML 5 schema. In the following subsection we present the `HtmlFlow` design and its core types. After that, we distinguish between the *pull* and *push* output approaches provided by `HtmlFlow`. Finally in subsection 4.3 we show some implementation details of `HtmlFlow`.

4.1 Design

In addition to the HTML fluent interface, the `HtmlView` class encapsulates the logic of what to do on each HTML element visit. To that

end, the `HtmlView` delegates to an implementation of the `ElementVisitor` interface, the responsibility of defining what to do on each invocation to the `HtmlFlow` fluent interface (e.g. `.div()`, `.ul()`, etc). This feature enables the integration with different kind of visitors such as collecting the resulting HTML into a *string buffer* (i.e. `HtmlVisitorStringBuilder`) or emitting the HTML to an output stream (i.e. `HtmlVisitorPrintStream`), as depicted in the class diagram of Figure 1. Note that every HTML element is an implementation of the `Element` interface and also the `HtmlView` itself.

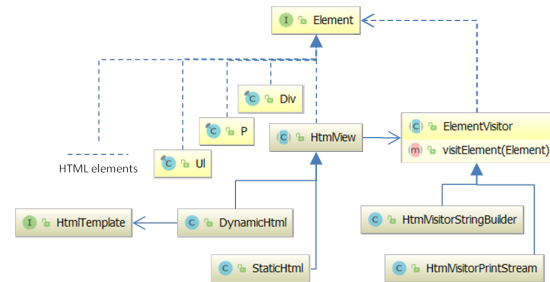


Figure 1: Class diagram of `HtmlFlow` core types.

In turn, the type `HtmlView` has two different subtypes: `StaticHtml` and `DynamicHtml`. The former does not support model binding and can be useful to define and reuse partial views in different kinds of template views (e.g. `footerView` and `bbView`). The latter is intended for templates with a *context object* that require model binding on their resolution.

Thus a `DynamicHtml` view depends of an implementation of the `HtmlTemplate`, such as the lambda: `(view, tracks, footer) -> ...` of the Listing 3. This lambda is conforming the following definition of the interface `HtmlTemplate`:

```
public interface HtmlTemplate<T> {
    void resolve(
        HtmlView<T> view,
        T model,
        HtmlView<T>...partials);
}
```

Note that both `view` and `partials` are of type `HtmlView` allowing the composition between any kind of views.

4.2 Output Approaches

`HtmlFlow` provides two ways of resolving a view: *pull* or *push* approach. In the *pull* approach we get the resulting HTML from the `render()` method call, as shown in line 3 of the example of Listing 5 and then we may proceed to do whatever we want with that

HTML. On the other hand, with the push approach, we set the `HtmlFlow` with a `PrintStream` and let the resolution process push the resulting HTML to that stream. The equivalent push resolution of the `tracksView` of Listing 5 would be:

```
tracksView
    .setPrintStream(...)
    .write(tracks, footerView(bbView));
```

According to the requirements stated in Section 1 of a web application, we want to push the resulting HTML to the output stream as the template view is being resolved. To that end we should use a visitor that writes the resulting HTML to the HTTP response stream. This allows the HTML to be sent to the end-user agent (e.g. browser) as tracks are being iterated through the `forEach` loop. `HtmlFlow` does not need to wait for `tracksTpl` completion to start sending the HTML. As HTML elements are visited the HTML is generated and emitted immediately to the output channel allowing the browser to progressively render the user-interface.

For a web application purpose we should wrap the HTTP response stream into a `PrintStream` object and assign it to the `HtmlView`. In Listing 6 we show the usage of a `HttpResponsePrinter` object that wraps an instance of `HttpServletResponse` representing an HTTP response in a VertX web application (Fox, 2001). First we take the `HttpServletResponse` object from the `RoutingContext` and set the content-type as `text/html` (lines 2 and 3). After that, we set the output stream of the `view` (line 5) and then we pass to it the `model` object (line 6) that will be internally bound to the template. Finally, we close the HTTP response stream (line 7).

Listing 6: Setting an `HtmlView` to the HTTP response stream.

```
<T> void handler(
    RoutingContext ctx,
    HtmlView<T> view,
    T model)
{
    HttpServletResponse resp = ctx.response();
    resp.putHeader("content-type", "text/html");
    view
        .setPrintStream(new HttpResponsePrinter(resp))
        .write(model);
    resp.end();
}
```

The handler method of Listing 6 can be used to send the result of `tracksView` resolution to the HTTP response stream. Every time the `tracksView` resolution emits HTML to its `PrintStream`, the

Listing 7: A naive implementation of `PrintStream` that forwards data to a `HttpServletResponse`.

```
class HttpResponsePrinter
    extends PrintStream
{
    final HttpServletResponse resp;
    public HttpResponsePrinter(... resp) {
        super(new OutputStream() {
            @Override
            public void write(int b) {
                char c = (char) b;
                resp.write(String.valueOf(c));
            }
        });
    }
    this.resp = resp;
}
```

`HttpResponsePrinter` may push that data to the HTTP response stream. In Listing 7 we show a naive implementation of the `HttpResponsePrinter` that writes every received byte immediately to the HTTP response stream.

A more effective approach may buffer those bytes into an internal buffer and flush that buffer to the output stream only when a certain threshold is achieved.

4.3 HtmlFlow Internals

There is a primary goal in the `HtmlFlow` design: the HTML *fluent interface* should be easily navigable. This is crucial to provide a good user experience while creating templates through the `HtmlFlow` API. There are two main aspects, the *fluent interface* should be easily navigable and always implement the concrete language restrictions. We tackle this issue through the use of *type parameters*, 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 8 we can observe how we can take advantage of the type arguments.

Listing 8: 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

Listing 9: Example of the implicit use of type arguments in HtmlFlow API.

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

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 8 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 9, which is an equivalent definition to the previous template of Listing 8. The type arguments are still important and inferred by the methods chain usage to validate and provide the correct available API to the next element. Therefore, typing for example `p().div()` is disallowed because it goes against the content allowed by a `P` element according to HTML5.

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`. The `HtmlApi` is built with the support of ASM (Binder et al., 2007) a bytecode instrumentation tool. Thus `HtmlFlow` depends of a build process organized in three different components that are part of `xmlet`² framework depicted in Figure 2.

Each component of Figure 2 has the following role:

1. `XsdParser` – parses all the external rules defined by the XSD document into a tree of objects.
2. `XsdAsm` – deals with the generation of *bytecodes* that make up the types of the Java *fluent interface*. This project should translate as many rules of the

²<https://github.com/xmlet/>

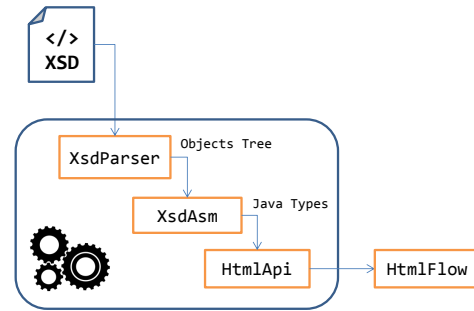


Figure 2: xmlet framework build process and its organization in three main components: `XsdParser`, `XsdAsm` and `HtmlApi`.

parsed language definition, its XSD file, into the Java language in order to make the resulting *fluent interface* as similar as possible to the language definition.

3. `HtmlApi` – it is a concrete client of the `XsdAsm` project, it will use the HTML5 language definition file in order to request a strongly typed *fluent interface*, named `HtmlApi`. This is used by the `HtmlFlow` library to manipulate the HTML language and write well formed documents.

To support the foundations of the XSD language there is a common infrastructure in every *fluent interface* generated by this project. This infrastructure is composed by the following main interfaces: `Element` and `Attribute`, which are implemented by every generated class for each `xsd:element` or `xsd:attribute` defined in the XSD.

This solution focus on how the code is organized rather than making complex code. All the methods present in the generated classes have very low complexity, mainly adding information to the element children or to the attribute list. To reduce code repetition we created many interfaces with default methods implementations so that different classes can extend them and reuse the code. The complexity of the generated code is mostly present in the `AbstractElement` class, which implements most of the `Element` interface methods. Another very important aspect of the generated classes is the extensive use of *type arguments*, also known as generics, which allows the navigation in the element tree while keeping type information, which is essential to ensure the specific language restrictions.

The client of the `HtmlApi` (i.e. `HtmlFlow`) is responsible for establishing what to do on `HtmlApi` methods calls. To that end the `HtmlApi` follows the Visitor design pattern. Thus, on every element or attribute instantiation we directly invoke the `ElementVisitor` visit method. Since the `ElementVisitor` instance is shared by all elements

then we can invoke the visit method in the constructor of the classes generated based on a XSD `<xsd:element>`. In Listing 10 we show a simplified example of the generated `Html` class.

Listing 10: `Html` Class Generated by `XsdAsm`.

```
final class Html<Z extends Element> {
    protected final Z parent;
    protected final ElementVisitor visitor;

    public Html(Z parent) {
        this.parent = parent;
        this.visitor = parent.getVisitor();
        this.visitor.visitElementHtml(this);
    }
    ...

    public Body<T> body() {
        return new Body(this);
    }
    public Head<T> head() {
        return new Head(this);
    }
}
```

Since adding elements results in the creation of new objects, such as `Body` and `Head`, it results in the invocation of their respective visit method due to the visit method being called on each class constructor. The attributes have a very similar behavior, although they do not create instances on their methods invocation. On the other hand, their restrictions are validated through the invocation of static validate methods present on each attribute class.

5 RELATED WORK

`HtmlFlow` distinguish from competition in three main features: 1) *composability* provided by *higher-order templates* (HoT), 2) *HTML validation* and 3) *performance*. In this section we analyze these characteristics and related work for each of these fields.

HTML view engines deal with data models as their *inputs* to produce HTML as *output* (Fowler, 2002). 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 decades³, turning this pattern into

³wikipedia.org/Comparison_of_web_template_engines

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 (?) provides a framework for building pipelines of XML transformations steps specified in XSLT.

The *transform view* pattern has similarities with the *higher-order templates* 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.

The `j2html` (Ase, 2015) is an alternative Java DSL for HTML. The major handicap of `j2html` in comparison to `HtmlFlow` is the lack of *validation* of the HTML language rules either at compile time or at runtime. Hence, it does not ensure that the resulting HTML is conforming a valid HTML document.

The `KotlinX.Html` (Mashkov, 2015) is another popular DSL for HTML and it has been written in Kotlin programming language. Kotlin may run on top of the Java Virtual Machine (JVM) and provide interoperability between Java, Android, and browser environments. 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. This is achieved through type inference and the Kotlin compiler. Yet, unlike `HtmlFlow`, the `KotlinX.Html` does not validate attributes and accepts any kind of values.

`HtmlFlow` has two main advantages over competition: 1) shows better performance in several benchmarks including the spring benchmark (Reijn, 2015), and 2) flexible output approach with support to the visitor design pattern integration (Gamma et al., 1995). This latter feature allows the integration of an output stream that *pushes* the resulting HTML to the HTTP response stream as the view is being resolved. The *push-based* style is proposed by

Erik Meijer (Meijer, 2012) as an alternative to the usual *pull-based* collections through the iterator pattern (Gamma et al., 1995) in the context of the velocity dimension of big data.

6 EXPERIMENTAL EVALUATION

To evaluate our proposal of HtmlFlow *higher-order templates*, we compare it with two different alternative approaches: a state of the art server side template engine (*Handlebars*) and a front-end framework (*ReactJS*). We used these three different technologies to build a VertX web application (Fox, 2001) that presents a listing resulting from a large data set provided by Last.fm API. This API provides a social music playground that allows anyone to build their own programs including web applications.

In our experimental evaluation we built a Java web application deployed in Heroku with the domain `topgenius.eu`. The `topgenius.eu` provides the most popular tracks on Last.fm last week for a given country. To that end we consume data from the *geo.getTopTracks* service of Last.fm API.

In the following subsection we present the high-level architecture of the `topgenius.eu` web application. Next in the subsection 4.2 we explain the domain model. Finally in subsection 4.3 we describe the testing approach and the results of the experimental evaluation.

6.1 TopGenius Architecture

The `topgenius.eu` provides 3 different pages with exact same functionality, but built with 3 different technologies: *Handlebars*, *HtmlFlow* and *ReactJS*. At the index we can find the corresponding links to these pages, each one presenting a similar user-interface to that one of Figure 3.

The web page of Figure 3 also includes an input for the limit, which specifies the maximum number of tracks that should be returned by `topgenius` web server. To avoid useless roundtrips to Last.fm we keep an internal per user cache in `topgenius` web server with data gathered from the *geo.getTopTracks* service. On the first request for a given country the `topgenius` will dispatch a sequence of fetch operations to gather that country's data from the Last.fm. These fetch operations are performed asynchronously and `topgenius` does not need to wait for their completion. For example, if the end user asks for the top hundred tracks from Australia, then the `topgenius` will fetch all available tracks from Australia on Last.fm, but it will conclude the response as the first 100 tracks are available.

Later, the same request to `topgenius` may already get all available tracks from its cache with a low latency.

Also, as presented in Figure 3 we provide a clear cache feature that allows users to recycle the cache for a given country and let them take the user experience of fetching data from the Last.fm data source with a high latency.

The `topgenius` web server provides three routes for each of the view engines: `/handlebars`, `/htmlflow` and `/react`, as depicted in Figure 4. In addition, it also includes a fourth route `/api/toptracks` used by the *ReactJS* application to get a country top tracks.

While the *geo.getTopTracks* service of Last.fm API provides paginated results with 50 records per page, the `/api/toptracks` route serves those same pages in a single data stream. This allows the *ReactJS* application to get whole required data with a single fetch operation, similar to what happens for *Handlebars* and *HtmlFlow* approaches, where the browser gets the resulting HTML document in a single HTTP GET request. To that end the `/api/toptracks` route uses the media type `application/stream+json` (Snell, 2012) to transmit JSON objects in newline-delimited JSON format (NDJSON). In Figure 4 we use a distinct dashed arrow to represent data emitted by `/htmlflow` and `/api/toptracks` routes regarding the chunked transfer encoding (Fielding and Reschke, 2014) used in both cases to stream data. On the other hand, the `/handlebars` route compromises a well-known `Content-Length`, which shows that `topgenius` server starts to send the resulting response only when the entire HTML document is completed. This prevents the browser from presenting the response of the `/handlebars` route while the server has not conclude the template view resolution. Rather than using this all-or-nothing basis, the `/htmlflow` route starts streaming the end HTML as the resolution proceeds, sending the final data in a series of chunks with unknown size. This same streaming behavior applies to the `/api/toptracks` route. Thus in both cases the browser will render content progressively: 1) as HTML is received from the `/htmlflow` route, or 2) as JSON is received from the `/api/toptracks` route and the *ReactJS* component dynamically updates HTML through DOM manipulation.

6.2 TopGenius Domain Model

In the `topgenius` Java application the class `LastfmWebApi` provides methods to access the Last.fm API. In this case, the access to the *geo.getTopTracks* service is provided by the method `countryTopTracks` with the following signature:

Country: <input type="text" value="australia"/> Limit: <input type="text" value="10000"/> <input type="button" value="Top Tracks"/> <input type="button" value="Clear Cache for australia"/>		
Server processing time: 0.252 ms for australia		
Rank	Track	Listeners
1	The Less I Know the Better	448928
2	Can't Feel My Face	471050

Figure 3: Topgenius.eu user-interface to present the most popular tracks on Last.fm last week for a given country.

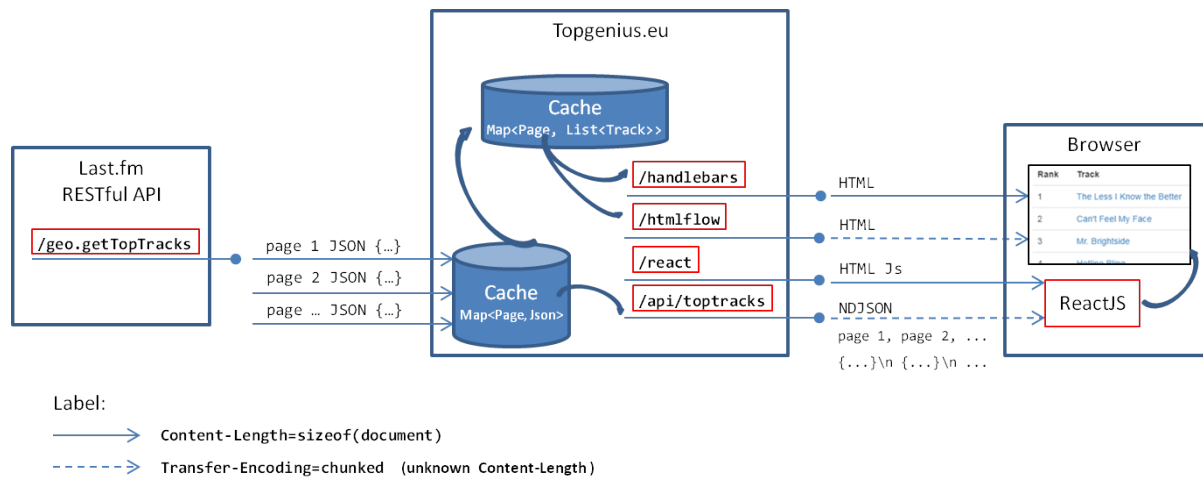


Figure 4: Topgenius architecture and interactions with Last.fm RESTful API and the browser.

```
Track[] countryTopTracks(String country,
int page)
```

To gather the first 100 thousand tracks of Australia Last.fm top tracks into a single `Stream` of `Track` objects we may perform the following query:

Listing 11: Stream query pipeline to gather the first 100 thousand tracks from Australia.

```
Stream<Track> tracks = IntStream
    .rangeClosed(1, 100000)
    .mapToObj(p ->
        countryTopTracks("Australia", p))
    .flatMap(Stream::of)
```

The resulting `Stream<Track>` is the model (i.e. *context object*) for the templates built with `Handlebars` and `HtmlFlow`. But notice that `tracks` stream is a lazy sequence, meaning that its elements are processed only on demand, when they are fetched by a terminal operation such as a *for each* loop. In the case of `HtmlFlow` we can provide this `tracks` stream as it is to the template which consumes its elements through the chained operation of `(_ -> tracks.forEach(...))` as denoted in the example

of Listing 2. Yet, for the `Handlebars` template, we have to first collect the entire `tracks` stream into a `List<Stream>` before passing it to the template. This prefetch of the stream incurs in useless memory overhead and higher latency in HTTP server response. Thus for limits over 10 thousands of tracks the `Handlebars` HTTP handler may violate the Heroku platform requirements and responds with a internal server error status code.

In `ReactJS` a `React.Component` represents a *template view* and its state property is equivalent to the model in MVC or the *context object* in a template. The main characteristic of `ReactJS` is that whenever the state changes, the component automatically re-renders. The `React` component `setState()` method schedules an update to a component's state object that results in its rendering.

Thus, in the `ReactJS` approach rather than gathering all the top tracks pages in a single stream we concatenate the resulting arrays in the component state array and let `React` do the job and re-render the component. The `geographicTopTracks` method of Listing 12 uses the native Javascript `fetch` function to perform an HTTP request to `/api/toptracks` and a

ndjson parser to deliver pages through the Javascript async iterable protocol. Note that the `concat()` operation does not change the existing arrays, but instead returns a new array that is set as the new state of the React component.

Listing 12: ReactJS `geographicTopTracks` method that recursively parses each page retrieved from `/api/toptracks` route.

```
geographicTopTracks(url) {
  fetch(url)
    .then(resp => {
      const reader =
        ndjson(resp.body.getReader())
      reader.next().then(
        function cons({value, done}) {
          if(done) return
          const tracks = this
            .state
            .tracks
            .concat(value.tracks.track)
          this
            .setState({
              'tracks': tracks
            })
          reader
            .next()
            .then(cons)
        })
    })
}
```

6.3 Performance Evaluation

To measure and compare the performance of different approaches we used JMeter with Selenium WebDriver. To that end we built a Selenium script for each approach that opens the topgenius corresponding url, fetches the top tracks of Australia and when the browser finishes receiving and rendering all content, then it scrolls to the end of the page. In Listing 13 we depict the script used to test Handlebars. For HtmlFlow it only changes the url, but for ReactJS we need an additional wait condition to detect the rendering completion. Note that for ReactJS the page body is empty and all content is loaded dynamically through DOM.

Listing 13: Selenium script to evaluate the performance in JMeter.

```
WDS
  .browser
  .get("/handlebars?country=...")
WDS
  .browser
  .executeScript(scrollTo(
    0,
    document.body.scrollHeight)
  )
```

All tests were executed on a single machine with the JVM version 11, SE Runtime Environment 18.9 (build 11+28). During the measurements there were no other workloads running. To discard the network overheads we run both topgenius web server and the browser in the same machine. And, before starting the measurement we ensured that all required data was already collected and available in the topgenius internal cache.

The results of Figure 5 were taken with 3 different workloads for a limit of 1000, 5000 and 10000 tracks. These results show that for small data sets there are no significant differences, yet for larger data sets the HtmlFlow gets an improvement of up to 4-fold in performance in comparison to ReactJS.

Despite we are including here the Handlebars results, they only have meaning within this benchmark ecosystem. On presence of a realistic data source such as Last.fm with high latency, Handlebars it is not able to show any preliminary results until all records were fetched, which is not an acceptable user experience. We can observe this behavior in topgenius.eu if we do not use cache.

7 CONCLUSIONS AND FUTURE WORK

In this paper we propose a new approach of *higher-order templates* implemented in HtmlFlow solution. HtmlFlow assembles all the homogeneity and simplicity of server-side Web development, enabling its usage even with large data sets. HtmlFlow is a Java DSL library for HTML that replaces textual template files by templates defined as first-class functions. This library not only competes in performance with state of the art alternatives(Reijn, 2015), but it also provides a full set of safety features not met together in any other library individually, such as:

- Well-formed documents;
- Fulfillment of all HTML rules regarding elements and attributes;
- Fully support of the HTML5 specification.

HtmlFlow started as an academic use case of a fluent interface for HTML, which was not released nor disseminated, but still attracted the attention of some developers. This community was looking for a Java library that helps them to dynamically produce papers, reports, emails and other kind of HTML documents in complex Java applications where classic templates engines do not fit all the requirements. For example, textual templates are not properly suited for complex programming tasks involving the dynamic build

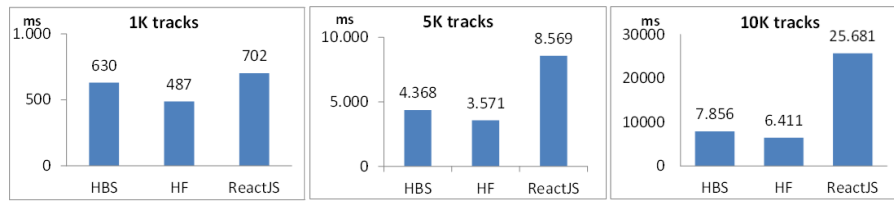


Figure 5: Performance results in milliseconds taken with JMeter for three view engines: Handlebars, HtmlFlow and ReactJS.

of user-interface components, which may depend on runtime introspection data (e.g. reflection).

The increasing attention around HtmlFlow raised the idea of developing a mechanism that automatically generates a fluent interface based on the HTML language specification, specified in a XSD file, which gave rise to the `xmlet` platform.

Having templates views defined as first-class functions suppress the limitations of textual templates for server-side views specially in the presence of large data sets. Thus HtmlFlow keeps template views simple with focus on domain-driven design and avoiding auxiliary decorations such as numbered pagination or infinite scrolling via DOM manipulation techniques.

One of our goals for a future release of HtmlFlow is to include an automatic translation tool that allows to convert an HTML document in its equivalent definition in HtmlFlow idiom. This is an essential module to spread the HtmlFlow in mainstream use allowing web designers to easily convert their web themes to integrate a main HtmlFlow project.

More interesting and the next evolution of HtmlFlow will be the support of asynchronous models with particular focus in Java reactive streams. Our goal is to deal with the `Observable` or `Publisher` as a result of the transformation provided by a *higher-order template* that can be asynchronously processed to emit the resulting HTML to the HTTP response stream.

ACKNOWLEDGEMENTS

We express our gratitude to Antonio Rito Silva from INESC-ID, Professor at University of Lisbon for many suggestions and improvements of our work. We would like to thank Pedro Felix for all feedback and help to enhance our work.

REFERENCES

Agrawal, R., Kadadi, A., Dai, X., and Andres, F. (2015). Challenges and opportunities with big data visualization. In *Proceedings of the 7th International Confer-*

ence on Management of Computational and Collective Intelligence in Digital EcoSystems, MEDES '15, pages 169–173, New York, NY, USA. ACM.

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

Ase, D. (2015). Kotlin dsl for html. Technical report, <https://j2html.com/>.

Binder, W., Hulaas, J., and Moret, P. (2007). Advanced java bytecode instrumentation. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 135–144, New York, NY, USA. ACM.

E. Krasner, G. and Pope, S. (1988). A description of the model-view-controller user interface paradigm in the smalltalk80 system. *Journal of Object-oriented Programming - JOOP*, 1:26–49.

Farago, J. H., Williams, H. E., Walsh, J. E., Whyte, N. A., Goel, K. J., Fung, P., Lazier, A. J., Moss, K. A., and Ray, E. N. (2007). Object search ui and dragging object results.

Feldman, D., Schmidt, M., and Sohler, C. (2013). Turning big data into tiny data: Constant-size coresets for k-means, pca and projective clustering. In *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 1434–1453, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Fielding, R. and Reschke, J. (2014). Chunked transfer coding. Rfc, <https://tools.ietf.org/html/rfc7230#section-4.1>.

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

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

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

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

Hors, A. L., Hégaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., Arbortext, and Byrne, S. (2004). Document object model (dom) level 3 core specification. Technical report, <https://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/>.

Jin, X., Wah, B. W., Cheng, X., and Wang, Y. (2015). Sig-

- nificance and challenges of big data research. *Big Data Res.*, 2(2):59–64.
- Kesteren, A. V., Aubourg, J., Song, J., and Steen, H. R. M. (2006). Xmlhttprequest specification. Technical report, <https://www.w3.org/TR/XMLHttpRequest/>.
- Lerner, R. M. (2011). At the forge: Mustache.js. *Linux J.*, 2011(210).
- Liu, Z., Jiang, B., and Heer, J. (2013). immens: Real-time visual querying of big data. In *Proceedings of the 15th Eurographics Conference on Visualization*, EuroVis '13, pages 421–430, Chichester, UK. The Eurographics Association & John Wiley & Sons, Ltd.
- Mashkov, S. (2015). Kotlin dsl for html. Technical report, <https://github.com/Kotlin/kotlinx.html>.
- Meijer, E. (2012). Your mouse is a database. *Queue*, 10(3):20:20–20:33.
- Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344.
- Parr, T. J. (2004). Enforcing strict model-view separation in template engines. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 224–233, New York, NY, USA. ACM.
- Reijn, J. (2015). Demo project to show different java templating engines in combination with spring mvc. Technical report, <https://github.com/jreijn/spring-comparing-template-engines>.
- Smith, P. (2012). *Professional Website Performance: Optimizing the Front-End and Back-End*. Wrox Press Ltd., Birmingham, UK, UK, 1st edition.
- Snell, J. M. (2012). The application/stream+json media type. Internet-draft.
- Storey, V. C. and Song, I.-Y. (2017). Big data technologies and management. *Data Knowl. Eng.*, 108(C):50–67.
- ur Rehman, M. H., Liew, C. S., Abbas, A., Jayaraman, P. P., Wah, T. Y., and Khan, S. U. (2016). Big data reduction methods: A survey. *Data Science and Engineering*, 1(4):265–284.
- Yaqoob, I., Hashem, I. A. T., Gani, A., Mokhtar, S., Ahmed, E., Anuar, N. B., and Vasilakos, A. V. (2016). Big data: From beginning to future. *International Journal of Information Management*, 36(6, Part B):1231 – 1247.