Modern Type-Safe Template Engines (Part 1)

Template engines are a great way to build dynamic HTML pages, but are you using the best template engine for your purposes ?

by Luis Duarte · by Miguel Gamboa · Nov. 19, '19 · Tutorial

Over the past two decades, textual templates have been the most used approach to build dynamic HTML documents. Textual templates engines fit the main web development requirements, however, we leave here two considerations:

- They are slow.
- Most of them are not safe.

In this article (part 1), we present four recent alternatives (Rocker, J2Html, KotlinX.html, and HtmlFlow) that embrace *disruptive* and innovative techniques to suppress some of the common handicaps in traditional template engines. The next installment (part 2) includes a performance comparison between these engines and other state of the art technologies such as Velocity, Handlebars, Thymeleaf, etc. In some of the most challenging benchmarks, Rocker and HtmlFlow are at least two-times faster than the competition.

Disclaimer: we are the authors of HtmlFlow project.

Introduction

Since the web was invented, there has been a wide consensus around the use of *textual template engines* to build dynamic HTML documents. From the vast list of existing *web template engines* all of them share no less basic *textual template files*. From a simple point of view, these engines provide two main features: 1) dynamic binding, which enables the reuse of a template with different domain objects (aka *context object*), and 2) macro instructions to control flow.

Despite all advantages, *template engines* also have some drawbacks, namely:

- Unsafe and type checkless - Lack of validation (static or dynamic) of the HTML language rules, which may result in illegal HTML documents. Moreover, many engines do not provide static validation of the context object used, resulting in invalid binding at runtime.
- Lack of performance - There is an intrinsic overhead regarding text files load, which slows the overall performance. On the other hand, the heavy use of `String` operations, which are inherently slow, also contributes to performance degradation.
- Complexity - It introduces one more idiom in addition to the HTML and the environment programming language (e.g. Java). For example, a Java application using the Mustache *template engine* forces the programmer to use at least three distinct languages: Java, HTML, and the Mustache idiom to build the dynamic parts of the template.
- Limited flexibility - The macros syntax provided by *template engines* is limited and mostly restricted to a few control flow instructions such as `if/else` operations and `for` each loops. Although this is enough to build simple HTML documents, it turns out to be much harder to deal with complex dynamic binding tasks. In this way, a DSL for HTML such as J2html, KotlinX.html, or HtmlFlow enable the full use of all Java or Kotlin features, which makes it easier to codify complex binding assignments.

In this article, we are going to present some recent innovations introduced by modern template engines like Rocker, J2Html, KotlinX.html, and HtmlFlow, in order to solve or minimize some of the problems listed above. We are going to compare their features and create a general landscape implementing the same template in each idiom. To that end, we will build a simple dynamic document binding the properties `Name` and `Nr` of a `Student` context object. In the following listing we present the basis for this template in Mustache idiom:

```
1  <html>
2    <body>
3      <ul>
4      {{#student}}
5        <li>{{name}}</li>
6        <li>{{number}}</li>
7      {{/student}}
8      </ul>
9    </body>
10 </html>
```

We will address three different issues in our comparison analysis:

- Issue 1 - Guarantees of well-formed documents.
- Issue 2 - Validation of the HTML language rules.
- Issue 3 - Validation of context objects.

Finally, we will make a performance comparison using the most popular benchmarks for template engines: template-benchmark and spring-comparing-template-engines. In this comparison, we include state of the art template engines such as Velocity, Handlebars, Thymeleaf, and others, which fall far short from the performance shown by Rocker and HtmlFlow.

Disclaimer: Since the above benchmarks do not include all these template engines, we integrated them in our forks of these benchmarks available at xmlet/template-benchmark and xmlet/spring-comparing-template-engines.

Rocker

The Rocker library is very similar to the classic template engine solution since it still uses a textual file to define the template. But, in opposition, Rocker just uses the textual file at compile time rather than at run-time. Rocker uses the textual template file only to automatically generate a Java class that replicates the specific template in Java language. In the following example, we show a first listing of the `Student` template defined in Rocker and a simplified view of the corresponding Java class (i.e. `studentTemplate`) resulting from the compilation of the Rocker template:

```
1  @import com.mitchellbosecke.benchmark.model.Student
2  @args (Student student)
3  <html>
4    <body>
5      <ul>
6        <li>@student.getName()</li>
7        <li>@student.getNumber()</li>
8      </ul>
9    </body>
10 </html>
```

```
1  public class studentTemplate extends DefaultRockerModel {
2
3    private Student student;
4
5    ...
6    @Override
7    protected void __doRender() throws IOException,RenderingException{
8      __internal.writeValue(PLAIN_TEXT_0_0);
9      __internal.renderValue(student.getName(), false);
10     __internal.writeValue(PLAIN_TEXT_1_0);
11     __internal.renderValue(student.getNumber(), false);
12     __internal.writeValue(PLAIN_TEXT_2_0);
13   }
14   ...
15   private static class PlainText {
16     static final String PLAIN_TEXT_0_0 = "\n<html>\n    <body>\n      <ul>\n        <li>\n          ";
17     static final String PLAIN_TEXT_1_0 = "\n        </li>\n        <li>\n          ";
18     static final String PLAIN_TEXT_2_0 = "\n        </li>\n      </ul>\n    </body>\n</html>";
19   }
20 }
```

Note that Rocker stores three Strings in static variables of class `PlainText`: the `String` before the use of `@student.getName()` (i.e. field `PLAIN_TEXT_0_0`), the `String` between the two bindings (i.e. field `PLAIN_TEXT_1_0`), and lastly the `String` after the `@student.getNumber()` (i.e. field `PLAIN_TEXT_2_0`. The `doRender()` method joins the different parts of the template.

Thus, the resulting Java class (i.e. `studentTemplate`) combines the static information (i.e. class `PlainText`) with data of the context object (i.e. `student`). This approach has two main advantages: (1) it can validate the type of the context objects used to create the template at compile time; (2) it shows very good performance due to all the static parts of the template being hardcoded into a Java class. This was by far one of the most performant template engines.

The biggest downside of Rocker is that it does not verify the HTML language rules or even well-formed HTML documents. Regarding its use, Rocker is a bit more complex than other alternatives since we have to deal with three distinct aspects: the template, the generated Java class, and the Java code needed to render it. In the following listing we show an example rendering the `studentTemplate`:

```
1  String document = templates
2    .studentTemplate
3    .template(new Student(39378, "Luis Duarte"))
4    .render()
5    .toString();
```

J2HTML

J2html is a Java DSL 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.

The major handicap of j2html is the lack of verification of the HTML language rules either at compile time or at runtime, which is a major downside in comparison to KotlinX Html and HtmlFlow, which include both of these capabilities.

In the following listing we show an example of the `Student` template defined with j2Html:

```
1  static final String studentTemplate(Student student) {
2    return
3      html(
4        body(
5          ul(
6            li(student.getName()),
7            li(student.getNumber())
8          )
9        )
10     ).render();
11 }
```

Regarding j2html use, it is simple because it provides an API similar to HTML syntax, which makes it easily understandable. To reuse the same template function (i.e. `studentTemplate`) with different context objects, you just need to invoke that function with a given `Student` instance (e.g. `studentTemplate(new Student(39378, "Luis Duarte"))`).

KotlinX Html

Kotlin is a programming language that runs on the Java Virtual Machine (JVM). Kotlin provides an inter-operative language between Java, Android, and browser applications. Its syntax is not compatible with the Java syntax but both languages are inter-operable. One of Kotlin's main advantages is that it heavily reduces the amount of textual information needed to create code by using type inference and other techniques.

One of his children projects, KotlinX Html, defines a DSL for the HTML language. Like in J2html and HtmlFlow, the template is embedded within the Kotlin language, suppressing the need for textual template files and allowing the use of Kotlin syntax to define templates, which is richer than the regular template engine syntax.

The API for KotlinX Html is automatically built from the XSD definition of the HTML 5 language. Thus, the generated Kotlin 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, there is no validation of attributes which accept any kind of values.

In the following listing, we show an example of the `Student` template defined with KotlinX HTML:

```
1  fun studentTemplate(student: Student): String {
2    return createHTMLDocument()
3      .html {
4        body {
5          ul {
6            li { student.name }
7            li { student.number }
8          }
9        }
10     }.serialize(false)
11 }
```

Just like in j2html, to reuse the same template function (i.e. `studentTemplate`) with different context objects, you just need to invoke it with a given `Student` instance. To use it in Java you may define the `studentTemplate` function inside a companion object, just like:

```
1  class KotlinTemplates {
2    companion object {
3      fun studentTemplate(student: Student): String { ... }
4    }
5  }
```

And then you can bind and render the `studentTemplate` with a `Student` object in the following way: `KotlinTemplates.Companion.KotlinTemplates(new Student(39378, "Luis Duarte"))`

HtmlFlow

The motivation behind HtmlFlow is to provide a library that allows for the writing of well-formed, type-safe HTML documents. HtmlFlow is a similar solution to KotlinX HTML. The main differences are the better performance shown by HtmlFlow and the attributes validations checked by HtmlFlow API use. HtmlFlow takes advantaged of attribute restrictions defined in the HTML XSD definition in order to increase the verifications. Both solutions also use the Visitor pattern in order to abstract themselves from the concrete usage of the DSL. In the following listing, we show an example of the `Student` template defined in HtmlFlow. The HtmlFlow API is pretty straightforward and quite similar to the example shown with J2html. Yet, HtmlFlow has the advantage of obeying to the HTML rules. For example, the instruction `h1().div()` gives a compilation error because it goes against the content allowed by `h1` according to HTML5. So, whenever you type `.` after an element, the IntelliSense will just suggest the set of allowed elements in that context.

```
1  HtmlView<Student> studentView = DynamicHtml.view(CurrentClass::studentTemplate);
2
3  static void studentTemplate (DynamicHtml<Student> view, Student student){
4    view
5      .html()
6        .body()
7          .ul()
8            .li().dynamic(li -> li.text(student.getName()))._()
9            .li().dynamic(li -> li.text(student.getNumber()))._()
10         ._()
11       ._()
12     ._();
13 }
```

Finally, to reuse the `studentView` with different context objects, you just need to render it with a given `Student` instance in the following way: `studentTemplate.render(new Student(39378, "Luis Duarte"))`

Conclusion to part 1

Here we started to present some of the common handicaps in traditional template engines and how 4 modern engines(Rocker, J2Html, KotlinX.html, and HtmlFlow) tackle the template role from a different perspective to reach performance and safety.

The next installment (part 2) presents a feature and performance comparison of these solutions.

Template | Engine | Java (Programming Language) | HTML

Popular on DZone

- Real-Time Analytics for IoT
- How We Solved an OOM Issue in TiDB with GOMEMLIMIT
- How To Build a Spring Boot GraalVM Image
- Master Spring Boot 3 With GraalVM Native Image