HtmlFlow is a Java DSL to write typesafe HTML documents in a fluent style. You may use the utility `Flowifier.fromHtml(String html)` if you need the HtmlFlow definition for an existing HTML source:

output ↴

```
HtmlFlow
  .doc(System.out)
    .html() // HtmlPage
      .head()
        .title().text("HtmlFlow").__()
      .__() // head
      .body()
        .div().attrClass("container")
          .h1().text("My first page with HtmlFlow").__()
          .img().attrSrc("http://bit.ly/2MoHwrU").__()
          .p().text("Typesafe is awesome! :-)").__()
        .__() // div
      .__() // body
    .__(); // html
```

```
<html>
    <head>
        <title>HtmlFlow</title>
    </head>
    <body>
        <div class="container">
            <h1>My first page with HtmlFlow</h1>
            <img src="http://bit.ly/2MoHwrU">
            <p>Typesafe is awesome! :-)</p>
        </div>
    </body>
</html>
```

↸ `Flowifier.fromHtml("...")`

Beyond `doc()`, the `HtmlFlow` API also provides `view()` and `viewAsync()`, which build an `HtmlPage` with a `render(model)` or `renderAsync(model)` methods depending of a model (asynchronous for the latter).
Finally HtmlFlow is the **most performant** engine among state of the art template engines like Velocity, Thymleaf, Mustache, etc and other DSL libraries for HTML such as j2Html and KotlinX Html. Check out the performance results in the most popular benchmarks at spring-comparing-template-engines and our fork of xmlet/template-benchmark.

Check the implementation of the sample Spring-based petclinic with HtmlFlow views xmlet/spring-petclinic. You can find different kinds of dynamic views there.

Getting started

All builders (such as `body()`, `div()`, `p()`, etc) return the created element, except `text()` which returns its parent element (e.g. `.h1().text("...")` returns the `H1` parent object). The same applies to attribute methods - `attr<attribute name>()` - that also return their parent (e.g. `.img().attrSrc("...")` returns the `Img` parent object).
There are also a couple of special builders:

- `__()` - returns the parent element. This method is responsible for emitting the end tag of an element.
- `of(Consumer<E> cons)` - It returns the same element `E`, where `E` is the parent HTML element. This is useful whenever you need to chain any Java statement fluently. For instance, when we need to chain a conditional expression, such as the following example where we may add, or not, the class `minus` to the element `td` depending on the value of a stock change:

```
….td().of(td -> { if (stock.getChange() < 0) td.attrClass("minus"); }).…
```

- `dynamic(BiConsumer<E, M> cons)` - similar to `.of()` but the consumer receives an additional argument `M` corresponding to the model (i.e. context object) of `render(model)` or `write(model)`. Read more in dynamic views.

These builders avoid special templating dialects and allow the use of any Java statement interleaved in the web template.

The HTML resulting from HtmlFlow respects all HTML 5.2 rules (e.g. `h1().div()` gives a compilation error because it goes against the content allowed by `h1` according to HTML5.2). So, whenever you type `.` after an element the intelissense will just suggest the set of allowed elements and attributes.
The HtmlFlow API is according to HTML5.2 and is generated with the support of an automated framework (xmlet) based on an XSD definition of the HTML5.2 syntax. Thus, all attributes are strongly typed with enumerated types which restrict the set of accepted values.

Finally, HtmlFlow also supports *dynamic views* with *data binders* that enable the same HTML view to be bound with different object models.

Output approaches

When you build an `HtmlPage` with `HtmlFlow.doc(Appendable out)` you may use any kind of output compatible with `Appendable`, such as `Writer`, `PrintStream`, `StringBuilder`, or other (notice some streams, such as `PrintStream`, are not buffered and may degrade performance).
HTML is emitted as builder methods are invoked (e.g. `.body()`, `.div()`, `.p()`, etc).
However, if you build an `HtmlView` with `HtmlFlow.view(view -> view.html().head()...)` the HTML is only emitted when you call `render(model)` or `write(model)` on the resulting `HtmlView`. Then, you can get the resulting HTML in two different ways:

```
HtmlView view = HtmlFlow.view(view -> view
    .html()
        .head()
            ....
);
String html = view.render();        // 1) get a string with the HTML
view
    .setOut(System.out)
    .write();                       // 2) print to the standard output
```

Regardless the output approach you will get the same formatted HTML document.

`HtmlView` does a preprocessing of the provided function (e.g. `view -> ...`) computing and storing all static HTML blocks for future render calls, avoiding useless concatenation of text and HTML tags and improving performance.

Dynamic Views

`HtmlView` is a subclass of `HtmlPage`, built from a template function specified by the functional interface:
`interface HtmlTemplate { void resolve(HtmlPage page); }`
Next we present an example of a view with a template (e.g. `taskDetailsTemplate`) that will be later bound to a domain object `Task`. Notice the use of the method `dynamic()` inside the `taskDetailsTemplate` whenever we need to access the domain object `Task` (i.e. the *model*). This model will be passed later to the view through its method `render(model)` or `write(model)`.
`HtmlView view = HtmlFlow.view(HtmlLists::taskDetailsTemplate);`

```
public static void taskDetailsTemplate(HtmlPage view) {
    view
        .html()
            .head()
                .title().text("Task Details").__()
            .__() //head
            .body()
                .<Task>dynamic((body, task) -> body.text("Title:").text(task.getTitle()))
                .br().__()
                .<Task>dynamic((body, task) ->
                     body.text("Description:").text(task.getDescription()))
                .br().__()
                .<Task>dynamic((body, task) -> body.text("Priority:").text(task.getPriority()))
            .__() //body
        .__(); // html
}
```

Next we present an example binding this same view with 3 different domain objects, producing 3 different HTML documents.

```
List<Task> tasks = Arrays.asList(
    new Task(3, "ISEL MPD project", "A Java library for serializing objects in HTML.",
Priority.High),
    new Task(4, "Special dinner", "Moonlight dinner!", Priority.Normal),
    new Task(5, "US Open Final 2018", "Juan Martin del Potro VS  Novak Djokovic", Priority.High)
);
for (Task task: tasks) {
    Path path = Paths.get("task" + task.getId() + ".html");
    Files.write(path, view.render(task).getBytes());
    Desktop.getDesktop().browse(path.toUri());
}
```

Finally, an example of a dynamic HTML table binding to a stream of tasks. Notice, we do not need any special templating feature to traverse the `Stream<Task>` and we simply take advantage of Java Stream API.

```
static HtmlView tasksTableView = HtmlFlow.view(HtmlForReadme::tasksTableTemplate);

static void tasksTableTemplate(HtmlPage page) {
    page
        .html()
            .head()
                .title().text("Tasks Table").__()
            .__()
            .body()
                .table()
                    .attrClass("table")
                    .tr()
                        .th().text("Title").__()
                        .th().text("Description").__()
                        .th().text("Priority").__()
                    .__()
                    .tbody()
                        .<Stream<Task>>dynamic((tbody, tasks) ->
                            tasks.forEach(task -> tbody
                                .tr()
                                    .td().text(task.getTitle()).__()
                                    .td().text(task.getDescription()).__()
                                    .td().text(task.getPriority().toString()).__()
                                .__() // tr
                            ) // forEach
                        ) // dynamic
                    .__() // tbody
                .__() // table
            .__() // body
        .__(); // html
}
```

Asynchronous HTML Views

`HtmlViewAsync` is another subclass of `HtmPage` also depending of an `HtmlTemplate` function, which can be bind with both synchronous, or asynchronous models.
Notice that calling `renderAsync()` returns immediately, without blocking, while the `HtmlTemplate` function is still processing, maybe awaiting for the asynchronous model completion.
Thus, `renderAsync()` and `writeAsync()` return `CompletableFuture<String>` and `CompletableFuture<Void>` allowing to follow up processing and completion.
To ensure well-formed HTML, the HtmlFlow needs to observe the asynchronous models completion. Otherwise, the text or HTML elements following an asynchronous model binding maybe emitted before the HTML resulting from the asynchronous model.

Thus, to bind an asynchronous model we should use the builder `.await(parent, model, onCompletion) -> ...)` where the `onCompletion` callback is used to signal HtmFlow that can proceed to the next continuation, as presented in next sample:
`static HtmlViewAsync tasksTableViewAsync = HtmlFlow.viewAsync(HtmlForReadme::tasksTableTemplateAsync);`

```
static void tasksTableTemplateAsync(HtmlPage page) {
    page
        .html()
            .head()
                .title() .text("Tasks Table") .__()
            .__()
            .body()
                .table().attrClass("table")
                    .tr()
                        .th().text("Title").__()
                        .th().text("Description").__()
                        .th().text("Priority").__()
                    .__()
                    .tbody()
                    .<Flux<Task>>await((tbody, tasks, onCompletion) -> tasks
                        .doOnNext(task -> tbody
                            .tr()
                                .td().text(task.getTitle()).__()
                                .td().text(task.getDescription()).__()
                                .td().text(task.getPriority().toString()).__()
                            .__() // tr
                        )
                        .doOnComplete(onCompletion::finish)
                        .subscribe()
                    )
                    .__() // tbody
                .__() // table
            .__() // body
        .__(); // html
}
```

In previous example, the model is a Flux, which is a Reactive Streams `Publisher` with rx operators that emits 0 to N elements.
HtmlFlow *await* feature works regardless the type of asynchronous model and can be used with any kind of asynchronous API.

License

MIT

About

HtmlFlow was created by Miguel Gamboa (aka fmcarvalho), an assistant professor of Computer Science and Engineering of ISEL, Polytechnic Institute of Lisbon.