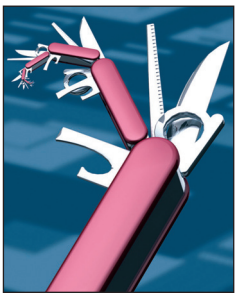
Clojure Templating Libraries: Fleet and Enlive



Glenn Vanderburg • *InfoEther*

Last issue, Aaron Bedra wrote about Compojure, a Clojure-based Web framework.[1] Here, I follow up on his article, delving into one particular aspect of Web development using Clojure and Compojure: templating.

On a recent project, I had the pleasure of working with Bedra, using Compojure. I had done Clojure programming before, but that project was the first time I had used it for Web development. One of the earliest decisions we had to make was which templating system to use.

Compojure Out of the Box

Typically, Compojure applications use the clj-html library, which is a builder-like HTML-generation library. The examples Bedra provided in his article used that system.[1] Figure 1 shows another example, slightly refactored from the version he supplied: the sample application's main (or index) page.

The *render* method renders a page body within a standard layout, which always includes a login box (this box automatically becomes a logout link if the user is logged in). You can see that clj-html's HTML macro reads a data structure that represents an HTML document or fragment and generates HTML from that structure.

Every Web framework should include a library for programmatically generating HTML, and clj-html is a good one. However, many situations arise in which you'd prefer to use a more traditional templating system. Particularly if you're working with HTML-savvy designers, you'd like for the HTML in your system to look like HTML, rather than Lisp code, as in clj-html's case.

StringTemplate and Model/View Separation

For our project, Bedra and I chose the StringTemplate templating library for Java. But, of course, Clojure runs on the Java Virtual Machine (JVM) and provides wonderful, convenient access to the underlying Java libraries, so it's easy to wrap StringTemplate to use in Compojure.

Initially, StringTemplate seemed like a very good fit for Clojure. The templating language itself is built on functional principles — for instance, calling another template is modeled as a function application. Additionally, StringTemplate was designed to enforce a strong separation between the domain model and the view layer; it attempts to eliminate business logic from templates. Only four template constructs are provided:

- using the value of an object's attribute,
- conditionally calling a template on the basis of whether an attribute is present,
- calling a template with arguments (including recursive calls), and
- mapping a template over a collection (that is, simple iteration).

In the end, StringTemplate proved to be a poor fit for Compojure. Furthermore, the mismatch provides some insight into the general templating problem and what we would like in a templating system for functional languages.

StringTemplate was designed for an object-oriented system — that is, the parameters passed to a template are assumed to be objects, and the template can refer to attributes of those objects. The usual JavaBean-style objects are supported (with attributes represented by accessor methods), as are maps (Java's term for hashes or associative arrays). As we were working in Clojure, we chose to pass maps into our templates.

However, we found ourselves wanting richer constructs in the templates. Generating HTML can be a complex task all by itself, and StringTemplate's simple conditionals (which can depend on the presence or absence of an attribute, but not its value) weren't enough. Even very simple things proved difficult, like choosing a singular or plural noun based on the value of a number.

1089-7801/10/$26.00 © 2010 IEEE

```
(defn login-box
  []
  (if (is-logged-in)
    (do [:span {:class "login-text"}
      (get-user) " - "
          [:a {:href (get-logout-url "/")}
              "sign out"]])
    [:span {:class "login-text"}
      [:a {:href (get-login-url "/")} "sign in"]]))

(defn render
  "The base layout for all pages"
  [body]
  (html
    (doctype :html4)
    [:head (include-css "/stylesheets/style.css")]
    [:body
      [:div {:class "container"}
        [:div {:id "login"} (login-box)]
        [:div {:id "content"} body]]]))

(defn index
  [request]
  (render "Hello App Engine"))
```

Figure 1. HTML generation. This code generates a simple webpage using the clj-html library. No traditional template exists; the page is modeled as a Clojure data structure.

We found ourselves building a layer to preprocess the data before handing it off to the view.

All this led us to wonder what all those happy StringTemplate users are thinking. Well, it turns out that StringTemplate's separation of model and view has a big loophole in the object-oriented world, where it's widely used. Because attribute access is most often mediated by an accessor method, useful behavior and transformations can occur in those methods. However, because we were using simple maps as input to our templates, rather than behavior-rich domain objects, that option wasn't open to us.

Of course, Clojure's Java interoperability is very good, and it would have been easy for us to build true Java-style objects, attaching methods to take care of those template-time chores. However, that didn't seem like the right approach for Compojure. Further reflection taught us that we really wanted something very different from a templating library.

The oft-repeated dictum "no logic in views" is too simplistic. HTML generation (or, more generally, data display in any format) has its own complexities, and sometimes a lot of logic is required to do it correctly. The real rule is "only view logic in views." A good templating system will acknowledge that view logic is necessary and will provide mechanisms (and encouragement) to the programmer to keep that view logic just as well factored as the rest of the system.

I'm aware of two templating systems for Clojure that I think provide better solutions. They're very different in approach, but both make room for presentation-oriented logic, and both make it easy to keep that code well factored, with very little logic inside the templates themselves.

Fleet

Fleet is a Clojure templating library written by Ilia Ablamonov (see http://github.com/Flamefork/fleet). It's a fairly traditional templating system, in that it allows embedding of properly delimited programming language code within the template itself. Obviously, that opens the door for the programmer to insert too much code (or inappropriate, misplaced code) into the template. Fleet relies on the programmer's discipline to avoid that problem but provides mechanisms that make it easy to factor presentation logic out of the template and keep it well separated from the application's core domain logic. Figure 2 shows what the main page of Bedra's system would look like in Fleet.

You can initialize the Fleet system by a call to `fleet-ns`. The first argument is a namespace, and the second is the path to a directory in the file system that contains all of the template files. Each template is turned into a function within that supplied namespace. So, for example, the template file "templates/index.html.fleet" becomes the function `index` in the `view` namespace.

Within the template, embedded Clojure code occurs within `<( … )>` brackets, which Fleet interprets as function calls. Fleet applies the `str` function to the result of those calls and inserts that string representation into the template at that point. If a string within such embedded Clojure code begins with `>` and ends with `<`, the system also processes that string as a Fleet template and removes those beginning and trailing angle brackets.

With those templates and declarations in place, rendering an index page with some body text just takes this call: `(view/index "Hello, World!")`.

Numerous other features and details are available, but the core question is, how does Fleet stand

up to the criteria I mentioned earlier? As I mentioned, Fleet doesn't prohibit logic in the template — that discipline is up to the programmer. The way Fleet uses namespaces makes it easy to factor your presentation-oriented code into separate files in your code base, as I've done in this example (you should define helper methods intended to be called from templates in the "helpers" namespace). This also means that it stands out if the template calls other functions that aren't appropriate there, as those methods must be namespace-qualified. Overall, Fleet is an excellent, pragmatic choice for a Clojure-based templating system.

Enlive

Christophe Grand wrote Enlive (see http://github.com/cgrand/enlive), and its approach to the templating task is very different from Fleet's. It also does an excellent job, however. Rather than allowing embedded Clojure code within templates, Enlive templates contain only HTML. More correctly, an Enlive template has two pieces: an HTML file and a transformation function. Enlive supports a convenient notation for defining transformation functions, based on CSS selector notation. When rendering a template, Enlive calls the transformation function to massage the HTML.

Figure 3 shows the same example I've been using so far, now written using Enlive. (This version uses the same "helpers.clj" file from the Fleet example.)

A function defined with `deftemplate` builds an entire HTML file. If you want to build a fragment to insert into a larger HTML file, use `defsnippet`. Enlive makes it easy to group multiple snippets into a single HTML file, although, in this case, `snippets.html` contains only one snippet.

A transformation function's body contains selectors and transformation expressions. For each selector, Enlive finds the HTML document's matching part and runs the transformation expression against it. The selectors

```
;; file templates/login-box.html.fleet ---------
<div id="login">
  <span class="login-text">
    <(if (is-logged-in)
      "> <(get-user)> -
         <(link-to "sign out" (get-logout-url "/"))> <"
      (link-to "sign in" (get-login-url "/")))>
  </span>
</div>

;; file templates/index.html.fleet -------------
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html14/strict.dtd">
<html>
  <head>
    <link type="text/css" href="/stylesheets/style.css"
          rel="stylesheet"/>
  </head>
  <body>
    <div class="container">
      <(login-box)>
      <div id="content">
        <(str data)>
      </div>
    </div>
  </body>
</html>

;; file src/helpers.clj ----------------------------
; function bodies omitted for brevity

(ns helpers)

(defn is-logged-in [] ...)

(defn get-user [] ...)

(defn link-to [label path] ...)

(defn get-login-url [prefix] ...)

(defn get-logout-url [prefix] ...)

;; file src/view.clj -------------------------------
(ns view
  (:use fleet helpers))

(fleet-ns view "templates")
```

Figure 2. Fleet templates and supporting code. Fleet templates include embedded Clojure expressions, in the style of many other templating systems. This code generates the same page as the code in Figure 1.

```
;; file src/templates/snippets.html ----------------
<div id="login">
  <span class="login-text">Login form or logout link</span>
</div>

;; file src/templates/index.html -------------------
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <link rel="stylesheet" type="text/css"
          href="/stylesheets/style.css"/>
  </head>
  <body>
    <div class="container">
      <div id="content">body text</div>
    </div>
  </body>
</html>

;; file src/view.clj -------------------------------
(ns view
  (:use net.cgrand.enlive-html helpers))

(defsnippet login-box "templates/snippets.html" [:#login] []
  [:div#login :span.login-text]
      (content
        (html-snippet
          (if (is-logged-in)
              (str (get-user) " - "
                   (link-to "sign out" (get-logout-url "/")))
              (link-to "sign in" (get-login-url "/"))))))

(deftemplate index "templates/index.html" [body-text]
  [:div.container] (prepend (login-box))
  [:div#content] (content body-text))
```

Figure 3. Enlive templates and supporting code. Instead of allowing embedded Clojure code, Enlive transforms HTML files using transformation functions. Again, this code generates the same page as the code in Figures 1 and 2.

use concepts from CSS selectors, although the syntax is different. The transformations replace, insert, or remove portions of the HTML, using Enlive-supplied functions like `content` and `prepend`.

With Enlive set up as I've described, you can render an index page in almost the same way as with Fleet. The only difference is that the template function returns a collection of pieces rather than a single string, so you have to concatenate them together with the `str` function using `(apply str (view/index "Hello, World!"))`.

Obviously, Enlive keeps the HTML files free of logic. Although it doesn't make explicit use of namespaces the way Fleet does, it naturally obeys Clojure's namespace rules, and it's easy to group templates and helper functions so that your presentation code stays clean and well factored. The Enlive model of HTML files and transformation functions feels like an excellent fit for a functional language like Clojure. Its only significant limitation is that you can use it with only XML or HTML files; the reliance on CSS selectors means that you can't use Enlive to massage other kinds of files.

My experience using StringTemplate in a Clojure Web application taught me something that really should have been obvious: a templating system designed for an object-oriented language really isn't a very good fit for a functional language.

But excellent alternatives are available that work very well with Clojure — namely, Fleet and Enlive. Although their philosophies are very different, they both acknowledge the need for presentation-oriented code, and both work well with Clojure's namespaces to enable good separation of presentation code from other parts of the system and from the HTML templates themselves. And, of course, a low-level HTML-generation library such as clj-html has a place in any Web project, serving as a complement to a full-fledged templating system.

If you're interested in Clojure for Web programming, I encourage you to start with the Google App Engine application Bedra wrote about in his article[1] and then expand from that base using either Fleet or Enlive, depending on your taste. Each has a lively user community, making Clojure a vibrant platform for functional Web development.

Reference

1. A. Bedra, "Getting Started with Google App Engine and Clojure," *IEEE Internet Computing*, vol. 14, no. 4, 2010, pp. 85–88.

Glenn Vanderburg is chief scientist at InfoEther (http://infoether.com), where he's a programmer, technical lead, speaker, trainer, and hands-on architect. Contact him at glv@vanderburg.org.