Extracted from:

Seven Web Frameworks in Seven Weeks

Adventures in Better Web Apps

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Seven Web Frameworks in Seven Weeks

Adventures in Better Web Apps

Jack Moffitt
and Fred Daoud

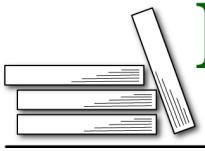
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Seven Web Frameworks in Seven Weeks

Adventures in Better Web Apps

Jack Moffitt
Fred Daoud

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Bruce A. Tate (series editor)
Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93778-563-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2014

Day 3: Other Ways to Build

Our tour thus far has been of a typical stack a Ring app would use. We'll now explore a few side alleys and an alternative templating library to give you a flavor of the myriad of options available. Clojure programmers have been refining ideas from other languages and frameworks, as well as striking out on paths less travelled in their search for web programming enlightenment.

You saw Ring middleware briefly on previous days, and today we'll dive a little deeper so that you can write your own middleware. The pattern used by Ring middleware appears in other Clojure libraries as well—notably in nREPL, the network REPL library—and it is a useful one to know.

Hiccup templating is convenient and fast for Clojure programmers, but it probably isn't the best interface for designers. We'll look at Enlive, the other big name in Clojure templating libraries, and see what solutions it can offer to this problem.

Testing is important, and we'll wrap up with a basic example test. Let's jump right in!

Ring Middleware

On the first day of your Ring journey, we suggested thinking about Ring middleware as simple data transformation functions that modified either the request or response maps provided by Ring. The reality is slightly more complicated, due to the fact that instead of transforming maps directly, we're building a pipeline of transformations to be executed whenever requests are handled. Because Clojure is a functional language, it probably won't surprise you that middleware actually manipulates *functions*.

Functional programming has function right there in its name, so you know it must be important. In functional languages, functions themselves are first-class values just like integers or objects. This means that functions can return functions, variables can be bound to functions, and functions can be passed as arguments to other functions. This is a very powerful concept, and you may have gotten a taste of it from other languages that have borrowed ideas from Lisp.

Let's look at some examples of functions returning functions. To motivate the example, we'll start with a simple goal: write a function that modifies an input list by changing strings to keywords.

clojure/examples/src/examples/function_return.clj

```
(defn keywordize [l]
```

```
  (for [elem l]
    (if (string? elem)
      (keyword elem)
      elem)))

(keywordize [1 2 "foo" 3 "bar"])
;;=> (1 2 :foo 3 :bar)
```

Pretty straightforward, right? Imagine that you've been handed some function that transforms a list, and you'd like to add the keywordize transformation as well. Easy! Just call the original function on the list and then pass that result to keywordize.

clojure/examples/src/examples/function_return.clj

```
(keywordize (other-transform [1 2 "foo" 3 "bar"]))
;; or
(-> [1 2 "foo" 3 "bar"]
    other-transform
    keywordize)
```

A problem occurs if you don't have the input on hand and ready. Given a transforming function, we'd like to modify it by adding more transformations and then return a new transforming function that can be used by other parts of the code—sometime later, when input is available. To do this, we need to return a function instead of simply applying a function:

clojure/examples/src/examples/function_return.clj

```
(defn wrap-keywordize [f]
  (fn [l]
    (keywordize (f l))))
```

If wrap-keywordize is called on a transforming function, it returns a new function that runs the original transformation and then runs keywordize on the result. You'll notice that keywordize is operating on f's output, but what if we wanted it to operate on its input? Perhaps f's transformations already expect strings to be replaced by keys.

clojure/examples/src/examples/function_return.clj

```
(defn wrap-keywordize-output [f]
  (fn [l]
    (keywordize (f l))))

(defn wrap-keywordize-input [f]
  (fn [l]
    (f (keywordize l))))
```

With the -output and -input versions of the wrappers, we can now wrap either way. And nothing stops you from wrapping both input and output.

This function manipulation is mind-bending stuff, so go through the steps a few times if it hasn't quite sunk in fully. The change from applying functions to returning functions is simple, but it is disproportionately powerful. Instead of adding a wing to an airplane, it can add the power to fly to anything you give it.

If you've made it this far, then congratulations—you understand how Ring middleware works! Of course, instead of lists as input, Ring passes the request map, and instead of lists as output, the function at the center of the transformation generates a response map. Ring middleware can choose to transform one or both of these maps, just as our wrapping functions transformed the input and output.

We can put middleware to use to solve a problem in Zap. All of the API routes need to serialize their response to JSON. It would be nice if we could wrap API functions with middleware that did that transformation for us.

Think for a bit about how you might do this, and then take a peek at the following solution:

clojure/examples/src/examples/function_return.clj

```
(defn wrap-json-response [f]
  (fn [req]
    (let [resp (f req)
          body (:body resp)]
      (assoc req :body (json/write-str body)))))
```

Adding wrap-json-response to the list of Ring middleware for your application will change responses that are normal Clojure data structures into their JSON representations. We no longer need to call json/write-str in every route.

Enlive

Let's climb up from the bottom of the stack in the caves of Ring, right up to the mountain summit that is HTML templating. In *HTML Is Data with Hiccup*, we played with Hiccup by writing HTML in Clojure data structures. Let's look now at Enlive, which manipulates existing HTML content to produce output.

Enlive is HTML origami, and it is just as clever and beautiful. You first create a mockup of the HTML output you'd like, and then you use that mockup directly as the HTML template, transforming it via simple changes to the output you want. For example, the mockup HTML might contain a list of dummy projects. In the transformation, the first <li> element would be used as a template to populate with a single project's data, and then one list item would be created for each project, replacing the dummy items.

Even better, the transformations are written using CSS selectors and Hiccup-like data structures. Instead of dealing with partials, which are just templates of pieces of HTML instead of whole HTML documents, templates can be extracted from the mockup output and reused anywhere. HTML designers never have to use anything but HTML, and their mockups never need to be translated into templates at all. Instead, you write the transformations needed to change the mockup into real output in situ.

Before we can transform anything, we need some HTML to work on. Let's pretend that an amazing designer friend of ours has handed us a mockup of the Projects page for Zap:

clojure/zap/day3/resources/templates/projects.html

```
<!DOCTYPE html>
<html>
  <head>
    <link href="/css/bootstrap.min.css" rel="stylesheet" type="text/css">
    <link href="/css/zap.css" rel="stylesheet" type="text/css">
    <title>Projects - Zap</title>
  </head>
```

```
  <body>
    <div class="navbar navbar-inverse">
      <div class="navbar-inner">
        <a class="brand" href="/">Zap!</a>
        <form class="navbar-form pull-right">
          <input class="search-query" placeholder="Search" type="text">
        </form>
      </div>
    </div>

    <div class="container">
      <div class="row admin-bar">
        <a href="/projects/new">Add Project</a>
      </div>

      <h1>Project List</h1>

      <ol>
        <li>
          <a href="/project/1/issues">Zap</a>
        </li>
        <li>
          <a href="/project/2/issues">Website</a>
        </li>
      </ol>
    </div>
  </body>
</html>
```

Enlive provides a macro deftemplate that creates a template function from an HTML file and a list of transformation rules. Let's look at a simple template that just changes the page title:

clojure/examples/src/examples/enlive.clj

```
(use 'net.cgrand.enlive-html)

(deftemplate page-with-title "templates/projects.html"
  [title]

  [:title] (content (str title " - Zap")))
```

deftemplate is similiar to defn. It takes a name for the new function, the path to the template relative to the project's resources directory, the parameters for the function, and finally the body. The body of a template consists of transformations, each of which is a CSS selector and transformation function pair. CSS selectors are written as keywords in a vector. This example selects all elements whose tag is title. The content transformation function replaces the content of the matched elements.

Enlive supports most CSS selectors. A selector of #content .big a would be written [:#content :.big :a]. Enlive's selector syntax reference contains more details.[7] With these, you can easily express targets for transformations.

In addition to the function content, Enlive also provides set-attr for setting attributes on elements, add-class for adding new CSS classes to elements, remove-attr, and do->, which strings together several transformations at once. You can also wrap any of these functions in normal Clojure expressions like if or cond to get dynamic behavior.

You can also make *snippets* in Enlive, which are like templates but created from an extracted piece of a document. Let's make a snippet we can reuse for the project list's <li> elements:

clojure/examples/src/examples/enlive.clj

```
(defsnippet project-item
  "templates/projects.html" [:.container :ol [:li first-child]]
  [proj]

  [:a] (do->
        (set-attr :href (str "/project/" (:id proj) "/issues"))
        (content (:name proj))))
```

❶ The only difference between deftemplate and defsnippet is the extra selector to choose what elements to extract from the base HTML. Notice the new selector notation [:li first-child] in the snippet definition; this is equivalent to CSS's li:first-child. In order to express this compound selector in Enlive, it gets wrapped in its own vector. Enlive's first-child selector predicate is just one of many choices. Some other interesting predicates are nth-child, attr?, and text-pred.

❷ Here's do-> in action combining several transformations into a single one.

That covers the basics of Enlive. Let's put it to use and rewrite the whole project list view for Zap:

clojure/zap/day3/src/zap/views.clj

```
(deftemplate base-page "templates/projects.html"
  [title & body]
  [:title] (content title)
  [:.container] (content body))

(defsnippet admin-bar
  "templates/projects.html" [:.container :.admin-bar]
  [links]
```

7. http://enlive.cgrand.net/syntax.html

```
(2)  [:a (but first-child)] nil
(3)  [:a] (clone-for [[url title] links]
            (do->
             (set-attr :href url)
             (content title))))

(defsnippet project-item
  "templates/projects.html" [:.container :ol [:li first-child]]
  [proj]
  [:a] (do->
        (set-attr :href (str "/project/" (:id proj) "/issues"))
        (content (:name proj))))

(defn projects []
  (base-page
   "Projects - Zap"

   (admin-bar {"/projects/new" "Add Project"})
(4)  (html [:h1 "Project List"])
   (map project-item (models/all-projects))))
```

(1) The base page is created from the mockup, replacing just a few key sections.

(2) The predicate but negates a predicate. Here all children but the first are deleted so that only one element is actually cloned by the next rule.

(3) clone-for clones an element once for each item in a collection. It mirrors Clojure's for syntax, except the body in this case is a transformation.

(4) html creates new DOM nodes in Enlive using Hiccup syntax.

The end result is slightly more verbose than straight Hiccup code, but your designer can now work completely independently in a familiar medium. Integration work only requires agreement on page structure, and it's easy to change CSS selectors in the code if the page structure changes.

A Little About Testing

As the final leg of our tour, let's look very quickly at one way you can test your Ring app. Clojure has great testing facilities, including clojure.test for unit testing, as well as test.generative, which generates random test cases from simple rules. For testing Ring applications, there is Kerodon.[8]

Kerodon adds the ability to interact with Ring apps inside of tests in a convenient way. You can simulate browsing to pages, filling in forms, pressing buttons, and even following redirects. Here's a small example:

8. https://github.com/xeqi/kerodon

clojure/zap/day3/test/zap/basic.clj
```
(deftest projects-page-exists
  (-> (session zap/app)
      (visit "/projects")
      (has (status? 200) "page exists")
      (within [:h1]
        (has (text? "Project List") "header is there"))))
```

You can run your tests with lein test:

```
$ lein test
lein test zap.basic

Testing zap.basic

Ran 1 tests containing 2 assertions.
0 failures, 0 errors.
```

This test visits /projects, checks the HTTP status code of the response, and ensures some known text is present in the page. Behind the scenes Kerodon is building Ring request maps on your behalf and checking that the response maps meet your expectations. Other libraries for testing Ring exist as well, and several other styles of testing frameworks are also available, so try a few and find your favorite.

What We Learned on Day 3

Today was a quick overview of some interesting pieces of the Ring ecosystem. The Clojure community innovates rapidly, and many libraries are competing for their place in your toolbox. There are too many such tools to cover in an entire book, let alone a single chapter, but you should now have a taste for what's out there.

We looked at Ring middleware, which is an important layer in the application stack, stringing together transformations of HTTP requests and responses. Ring middleware is yet another place where Clojure's power of composition shines brightly. The compositional patterns in Clojure can be tricky to wrap your mind around if you aren't used to that level of abstraction.

Enlive is a radical departure from traditional templates in other frameworks, but it is one that works well, especially in teams where HTML is written by nondevelopers. It expresses CSS selectors as data, giving you easy access to manipulate HTML.

Last, but certainly not least, we looked at one approach to testing. Since Ring applications are, at heart, simple functions that take and return maps, testing them is surprisingly easy, even without the help of great libraries like Kerodon.