An Introduction to Enlive

Though Christophe Grand's Enlive has been around for sometime now the Clojure community has been slow to embrace this useful library. I believe this is due simply to the lack of good introductory documentation based on real examples. Please let me know if you find this tutorial useful or helpful in any way. Feel free to suggest additions, corrections, and improvements (even better fork the repo and send me a patch).

What You need to Know

Not much. This tutorial assumes little about your exposure to Clojure. At the very least you'll need to have the Java Virtual Machine (JVM) installed. I won't go into great detail about Clojure's features but I'll try to explain any concepts which may impede your understanding of Enlive's functionality.

As the tutorials progress they will be more useful to you if you have some experience with a modern webframework that ships with a HTML templating library. If you do know Clojure the tutorials are fairly amenable to skimming. My only real assumption is that you have some experience at the command line.

Enlive has many features and I can't possibly cover them all here. Fortunately the Clojure community is vibrant and helpful. If you run into issues or have questions join the Enlive mailing list or jump on the #clojure channel on irc.freenode.net.

HTML Templating

There are two real camps in HTML templating. The first which almost everyone is familiar with is the PHP style template:

```
<?php
for($i = 0; $i < $len; $i++) {
?><p>Foo <?echo $i?></p><?
}
?>
```

This is of course enough to drive anyone insane and some templating solutions like the one in Django have made marginal improvements:

```
{% for i in foo %}
<p>Foo {{ i }}</p>
{% endfor %}
```

This is a bit easier on the eyes. But this isn't very composable and you're stuck with a limited subset of your programming language. By not composable I mean that building pages is largely a copy and paste affair even when templating solutions support inheritance.

This has driven many programmers to discard templating DSLs and to generate markup directly in code. While this is fast and flexible as you now have the power of function composition, it also means that you're putting quite a bit of distance between yourself and a designer comfortable writing HTML and CSS.

There are a few existing solutions that have a novel approach to this problem such as Pure.

Which brings us to Enlive. Enlive gives you the advantages of designer accessible templates (since they're just HTML) without losing the power of function composition. As a result, your designer can create all the various widgets for your website using only HTML and CSS and you can compose your pages from any combination of their designs.

Why Enlive?

Enlive presents a different approach from the more popular templating solutions:

- Code and markup are completely separate.
- You get to use CSS like syntax to manipulate HTML.
- Template inheritance isn't some fancy trick, it's just function composition.
- You have access to the full power of Clojure to manipulate your templates (yes, macros!).

When working with the standard templating solutions you generally need to answer one of two questions, either "What type of text am I going to generate?" or "What type of HTML emitting functions should I write?"

In contrast, with Enlive you usually break down the problem thus:

1. Determine which selectors match the part of the HTML document that you care about.
2. Determine which templates and snippets you need to write to compose your pages.

What We'll Cover

There are six examples in total.

The first one covers grabbing the headlines and points from Hackers News. The next one shows how to make the code less redundant. The third scrapes the New York Times front page since that presents more challenges than Hacker News. The fourth example shows how to use Ring and Enlive together. The fifth example shows how things like looping are achieved without writing any code into the markup. The sixth example shows that Enlive can do all the fancy template inheritance magic you might be used to if you're coming from Django or some other popular modern webframework.

Clone This Repo

The usual:

```
git clone git://github.com/swannodette/enlive-tutorial.git
```

Install Leiningen

In order to start playing around as fast possible you should use Leiningen. It'll take only a couple minutes to get through the instructions here. Leiningen is the easy_install (Python) and gems (Ruby) of the Clojure world. Phil Hagelberg and Co. have done a considerable amount of excellent work to make dependency management simple. I truly envy the new Clojurians who do not know the dark times before **lein repl** and **lein swank** :)

Once you have Leiningen installed, switch into this repository's directory. From there run the following command:

```
lein deps
```

This will install all of the dependencies required for getting through the tutorial. This might take a minute and and will probably generate *a lot* of output. While this may seem disconcerting, this means you'll have a fully functioning Clojure setup without needing to bother with installing Emacs, configuring VIM, or an mucking around with an IDE. Once the the dependencies are installed enter the following command at your terminal:

```
lein repl
```

This will launch a Clojure REPL (Read-Eval-Print-Loop) that has the classpath set properly. Be very thankful if you don't know what the last sentence means. Managing the classpath is one of the few real annoyances when programming Clojure and it's largely Java's fault.

Your First Scrape with Enlive – Hacker News

Enlive is fantastic for scraping the content of webpages. It allows you to scrape content by using a syntax very similar to CSS selectors. In the REPL type the following lines (note that **user=>** is the REPL prompt, not something you type in):

```
user=> (load "tutorial/scrape1")
nil
user=> (in-ns 'tutorial.scrape1)
nil
tutorial.scrape1=> *base-url*
"http://news.ycombinator.com/"
```

The first line loads the file. Note that we need to specify that the **scrape1** file is to be found in the **tutorial** directory which is under the **src** directory of the repo. **src** is put onto the classpath by **lein** so we don't need to specify it. Also take care to note that we left off the **.clj** extension. Unlike many scripting languages, loading a file actually *compiles* it. Clojure is not interpreted.

We then use `in-ns` to put ourselves into that tutorial's namespace. By switching into the tutorial namespace we can use functions defined in the tutorial without having to qualify them. It's much easier to type `*base-url*` then `tutorial.scrape1/*base-url*`.

Let's see what's in that file. Open up **scrape1.clj** with your favorite text editor (you can find it in **your-tutorial-clone/src/tutorial/**). You'll see it's a fairly short program.

At the top of this file is the namespace declaration. This keeps your code from clashing with other people's code when they try to use your library. The namespace declaration also includes another library, Enlive, via `:require`. In this case we are generating an alias so we don't have to type the very long namespace for Enlive.

The function `fetch-url` grabs the contents of a url synchronously. `fetch-url` uses `html/html-resource` (remember we aliased **net.cgrand.enlive-html** to **html** for convenience) another handy function defined in the Enlive library. It takes raw HTML and converts it into a nested data structure (think DOM minus tediousness).

Note that the function `hn-headlines` uses `fetch-url`. But it's also surrounded by a lot of funny stuff. You might have noticed `html/select`; `html/select` takes parsed html content and selects the nodes specified by a Clojure vector that looks very similar to a CSS selector.

```
[:td.title :a]
```

Now that looks kind of weird. But if you squint a little it might remind you of this:

```
td.title a
```

This is a CSS selector for matching all links inside of table elements that have the CSS class “title”. If you’re a Javascript hacker you should know this stuff by heart.

So let’s break this down. `fetch-url` grabs the contents of the url and parses it into a data structure. `html/select` takes it and extracts only those nodes that match the selector – it always returns a vector of nodes. We then use Clojure’s `map` function to iterate over the vector’s elements applying a function to extract each nodes’ text-node, in this case `html/text` (`map` is actually lazy, but we’re not going to get into what that means in this Enlive tutorial).

Believe it or not, these 10 lines of code are enough to extract all of the headlines from the Hacker News front page. Let’s try it out at the REPL now.

```
tutorial.scrape1=> (hn-headlines)
("A 'lorem ipsum' for images." "Google Reader Can Now Track Changes to Any Website - Even
Without a Feed" "jQuery 1.4.1 Released" ... "More")
```

Nice. After this the next function `hn-points` should make a lot more sense. It does the same thing but we grab the score from a different place in the markup. Try to run this function as well.

```
tutorial.scrape1=> (hn-points)
... output ...
```

The last function takes the output of the two different functions and prints out the headline and score for each item on Hackers News.

```
tutorial.scrape1=> (print-headlines-and-points)
... output ...
```

print-headlines-and-points looks like a doozy doesn’t it?

```
(defn print-headlines-and-points []
  (doseq [line (map #(str %1 " (" %2 ")") (hn-headlines) (hn-points))]
    (println line)))
```

Let’s break it down. Again we have `map`. We know that it maps a function over a vector to return a new vector of elements with that function applied.

```
#(str %1 " (" %2 ")") ; is just shorthand for
(fn [arg1 arg2] (str arg1 " (" arg2 ")")
```

This is an anonymous function. I’m not going to explain that here, they’re pretty popular these days. `str` is a built in function for doing string concatenation.

Oddly this `map` is accepting not one list of things, but two! Check this out:

```
tutorial.scrape1=> (map + [1 2 3] [4 5 6])
(5 7 9)
```

Wow you can map two different vectors into one! Finally we have `doseq`. `doseq` is just a convenient way to work with lists when you’re dealing with **side effects** like printing to the REPL. I’m not going to get into that here. All it does is say take a list of things, assign each thing one at time to a variable, and then execute the following expressions (hopefully you’re actually doing something with that variable!)

Not bad for 17 lines of code. One obvious problem here is that we make two separate requests for the Hacker News front page. Let's fix this now.

Your Second Scrape – Improvements

Take a look at **scrape2.clj**. It's also about 17 lines of code and it looks pretty much the same except that we no longer have one function to grab headlines and another for article points.

```
(defn hn-headlines-and-points []
  (map html/text
       (html/select (fetch-url *base-url*)
                    #{[:td.title :a] [:td.subtext html/first-child]})))
```

This select grabs what we're interested at the same time.

```
#{[:td.title :a] [:td.subtext first-child]}
```

It's pretty much the same as:

```
td.title a, td.subtext:first-child
```

Lets try out the functions. Start up the REPL with **lein repl** if you've shut it down and run the following.

```
tutorial.scrape1=> (load "scrape2")
nil
tutorial.scrape1=> (in-ns 'tutorial.scrape2)
nil
tutorial.scrape2=> (hn-headlines-and-points)
... output ...
```

The above assumes you've continued from the first tutorial. If you're starting from scratch you'll need to be more specific about your loading, use `(load "tutorial/scrape2")` instead.

The results are interleaved so we can use Clojure's partition function to pair them up and output them just like we did in the previous scrape. The map looks a little bit different:

```
(defn print-headlines-and-points []
  (doseq [line (map (fn [[h s]] (str h " (" s ")"))
                    (partition 2 (hn-headlines-and-points)))]
    (println line)))
```

To get a sense of what partition does let's use the REPL again:

```
tutorial.scrape2=> (partition 2 [1 2 3 4 5 6 7 8 9 0])
((1 2) (3 4) (5 6) (7 8) (9 0))
```

Neat, it lets us pair things together. Exactly what we need.

But what's up with the **fn** this time?

```
(fn [[h s]] (str h " (" s ")"))
```

Say hello to destructuring. A lot of popular languages allow you to destructure but probably not as ubiquitously as Clojure does. Here we know that we are going to receive a two element vector for each item in the vector we're mapping over. So we're just saying that we want to assign the first element of

that pair to the local variable **h** and the other to **s**.

The rest of the function should be clear from the last tutorial.

Your Third Scrape – The New York Times

Our third scrape tackles the New York Times whose front page structure is considerably more complicated than Hacker News. Now to be clear this not that useful since the New York Times provides a fairly comprehensive list of RSS feeds.

Take a look at **scrape3.clj**. This is a bit longer. Before we dive in let's see how it works. Start up the Clojure REPL if it's not already up and running.

```
tutorial.scrape2=> (load "scrape3")
nil
tutorial.scrape2=> (in-ns 'tutorial.scrape3)
nil
tutorial.scrape3=> (print-stories)
... output ...
```

If you're not continuing from a previous tutorial you'll need to more specific about your loading, using `(load "tutorial/scrape3")` instead.

Now this isn't perfect for a variety of reasons but it works well enough for the purposes of demonstration. Let's look at the code. At the top of the file we see that we have a variety of selectors.

```
(def *story-selector*
     [[:div.story
       (html/but :.advertisement)
       (html/but :.autosStory)
       (html/but :.adCreative)]])
```

Here we are matching any div with the CSS class story that does not also have any of the other classes specified.

```
(def *headline-selector*
     #{[html/root :> :h2 :a],
       [html/root :> :h3 :a]
       [html/root :> :h5 :a]}))
```

Here we know from looking at the markup of the page that headlines might match any of these three selectors. The selectors will only match headline tags that are children of the root element. We do this because there are story divs on the New York Times webpage that actually have multiple headlines underneath. The byline and story summary selectors are pretty much the same.

```
(defn extract [node]
  (let [headline (first (html/select [node] *headline-selector*))
        byline   (first (html/select [node] *byline-selector*))
        summary  (first (html/select [node] *summary-selector*))
        result   (map html/text [headline byline summary])]
    (zipmap [:headline :byline :summary] (map #(re-gsub #"\n" "" %) result))))
```

Here we take a node and extract the match. Note that we have to call first on the result of `html/select` because `html/select` always returns a sequence of nodes and not a single node. `zipmap` is a handy function, it allows us to take two sequences and zip them up into a hash-map. So

here we take only the text nodes from the matches and remove any newline characters before we finally zip it up into a tidy hash-map.

Because this scrape is not comprehensive we might match empty stories, so we define a function `empty-story?` that checks for that. We use this to filter out any empty stories:

```
(defn print-stories []
  (doseq [story (remove empty-story? (map extract (stories)))]
    (print-story story)))
```

Hopefully by this point you can begin to make sense of the last few functions. If one of the functions seems unclear I suggest calling that function at the REPL with some dummy input to get a better sense of what it does.

So that's it for scraping. It's time to move on to how Enlive is useful for building your own pages.

Your First Template – The Basics

This is where things begin to get really interesting. We're going to use Ring, an ultralight HTTP framework. If you're familiar with Rack or CherryPy you will feel right at home.

Let's get started. If you aren't running a REPL be sure to start one up from the repo directory with **lein repl**.

Once you see the REPL prompt type the following:

```
tutorial.scrape3=> (load "template1")
nil
tutorial.scrape3=> (in-ns 'tutorial.template1)
nil
```

If you're not continuing from a previous tutorial you'll need to be more specific about your loading, use `(load "tutorial/template1")` instead.

You should see some output that lets you know that Ring is starting up a webserver on port 8080. Point your browser at **http://localhost:8080**. You should see a very boring page. Point your browser at **http://localhost:8080/change**. You should see something slightly different.

First open **template1.html** and take a look at it. If you're used to other templating solutions the most shocking thing should be that there is absolutely no Clojure code in this file. And there never will be. Period.

Now let's take a look at the code in **template1.clj**. By now the namespace part should be familiar so we'll skip over that. After the namespace declaration we'll see our first template definition:

```
(html/deftemplate index "tutorial/template1.html"
  [ctxt]
  [:p#message] (html/content (:message ctxt)))
```

Every template has the argument list `[name source args & forms]`. An Enlive template is a macro that when compiled will create a function with the same name. This function will have the same signature as defined by **args**. **forms** consists of pairs of Enlive selectors and a function to execute for each node that matches the selector.

Here our template will find all **p** elements with the CSS id **message**. CSS ids should be unique so ideally this will only match a single element. Then we have the function which will receive this

matching element.

```
(html/content (:message ctxt))
```

This means we'll replace the content of any matching node with the value for the key `:message` in the `ctxt` hash-map that was passed as parameter to this template. The important thing to grasp here is that `html/content` is a function which *returns* a function which will receive the matched element.

For example what if we want a default message if there is no value for `:message` in `ctxt`? It would look something like this:

```
(html/deftemplate index "tutorial/template1.html"
  [ctxt]
  [:p#message] (fn [match]
                 (if-let [msg (:message ctxt)]
                   ((html/content msg) match)
                   ((html/content "Nothing to see here!") match))))
```

It should be clear that `html/content` returns a function which will receive the matching element and modify it. This could be made slightly less verbose like so:

```
(html/deftemplate index "tutorial/template1.html"
  [ctxt]
  [:p#message] (html/content (get ctxt :message "Nothing to see here"))
```

This is a considerable improvement and shows off a couple nice Clojure features. However even this is kinda meh. Why? Because what we really want is not just a way to specify a default. Honestly the default value will probably be in the markup itself! It would be much cooler to leave the content of the node *unchanged* if we for some reason hand it nil for it's content value. This would allow us to easily implement template inheritance which we'll talk about later (grin).

While Enlive does not have a great shortcut for expressing this pattern of "change the content of this node only if given a non-nil value", since Clojure is a competent Lisp, it's easy to write macros to remove the boilerplate. I've included a handy macro called `maybe-content` which allows us to write the following instead:

```
(html/deftemplate index "tutorial/template1.html"
  [ctxt]
  [:p#message] (maybe-content (:message ctxt) "Nothing to see here!"))
```

Pretty slick eh? ;) We get the terseness of the `get` as well as the plumbing for template inheritance. While macros are too advanced of a topic to delve into here, having them around when you're templating HTML is incredibly powerful.

The remainder of **template1.clj** is specific to Ring and Moustache, a routing library. We're not going to get too deep into that because these tutorials are about Enlive, not Ring and Moustache.

```
(def routes
  (app
   [""]       (fn [req] (render-to-response
                         (index {})))
   ["change"] (fn [req] (render-to-response
                         (index {:message "We changed the message!"})))
   [&]        {:status 404
               :body "Page Not Found"}))
```

```
(defonce *server* (run-server routes))
```

This is the Moustache route defining syntax. A couple things to note `render-to-response` is not a function of Enlive, it's something I added via **utils.clj** in the repository. `render-to-response` isn't magic it's just a function that looks like this:

```
(defn render [t]
  (apply str t))

(def render-to-response
     (comp response render))
```

All this does is take a list of strings, concatenates them into a single string, and serve back a proper Ring response. This is because when an Enlive template function is called it returns a list of strings.

Also note that our template function `index` must be called with at least one parameter. The last bit of **template1.clj** is just boilerplate for starting and stopping the server.

Well that's about it! You've seen your first Enlive template. While it may not seem like much yet, ***there was absolutely no mixing of code and HTML***. If you bear with me till the third template tutorial, I think you'll see just some how powerful this can be.

Your Second Template – Looping

A common operation when generating web pages is looping over some piece of HTML because you need to present a list of items to the user. People just love lists. How can Enlive create lists of HTML when there's no code in the template?! We'll get into this in this tutorial.

If you don't have a Clojure REPL running start a new one with **lein repl** at the commandline from the tutorial repo's directory. Enter the following (if you're continuing from the previous tutorial you should should stop the Ring app for that tutorial first):

```
tutorial.template1=> (.stop *server*)
nil
tutorial.template1=> (load "template2")
nil
tutorial.template1=> (in-ns 'tutorial.template2)
nil
```

If you're not continuing from a previous tutorial you can ignore `(.stop *server*)` and you'll need to be more specific about your loading, use `(load "tutorial/template2")` instead.

Open up the file **template2.html** in your text editor and give it a quick look over. Then open the file **template2.html** in your favorite web browser. It's just page with a list of links, not that special. Point your browser at **http://localhost:8080/**. You should see pretty much the same thing except that we've dynamically inserted links.

How did we do that if we have no inline code to define the loop? Let's get into the code. Open up **template2.clj** in your favorite text editor. At the top of the file you should see the by now familiar namespace declaration. One thing we've changed is how we import Enlive functionality.

```
(:use [net.cgrand.enlive-html
         :only [deftemplate defsnippet content clone-for
                nth-of-type first-child do-> set-attr sniptest at emit*]]
        [net.cgrand.moustache :only [app]]
```

```
        [tutorial.utils :only [run-server render-to-response page-not-found]])
```

In this tutorial we'd rather use the Enlive functions without having to qualify them. So we import them using `:use` and specify that we only want to import a specific set of definitions.

After that we declare a variable for holding a dummy context which we're going to pass to our template.

```
(def *dummy-context*
     {:title "Enlive Template2 Tutorial"
      :sections [{:title "Clojure"
                  :links [{:text "Macros"
                           :href "http://www.clojure.org/macros"}
                          {:text "Multimethods & Hierarchies"
                           :href "http://www.clojure.org/multimethods"}]}
                 {:title "Compojure"
                  :links [{:text "Requests"
                           :href "http://www.compojure.org/docs/requests"}
                          {:text "Middleware"
                           :href "http://www.compojure.org/docs/middleware"}]}
                 {:title "Clojars"
                  :links [{:text "Clutch"
                           :href "http://clojars.org/org.clojars.ato/clutch"}
                          {:text "JOGL2"
                           :href "http://clojars.org/jogl2"}]}
                 {:title "Enlive"
                  :links [{:text "Getting Started"
                           :href "http://wiki.github.com/cgrand/enlive/getting-started"}
                          {:text "Syntax"
                           :href "http://enlive.cgrand.net/syntax.html"}]}]})
```

This of course would be something that we probably would have read out of a database. The take away here is that Clojure makes it easy to define nested data structures. `*dummy-context*` is just a hash-map (aka dictionary, aka associative array) of two key-value pairs. The first pair is for the title of the page. The second pair is the list of sections. Each section also has a title as well as a list of links. Each link has some text and url. If you're used to building up JSON data structures from database results this should be pretty familiar to you.

Figuring out your selectors

Using Enlive for templating usually involve two steps. The first step is figuring out which part of the markup you want to make into a component. Each component will become a snippet. A snippet is a reusable mini-template that you can use when constructing larger templates. In order to create a working snippet you need to determine the CSS selector which will allow you to match exactly that part of the document.

Consider our situation. Our designer has handed us some nice markup and some CSS. To better convey the final result they have included some dummy content. With a traditional templating solution this is a big no no. With Enlive, working around it requires a minimal amount of effort. So the key here is to identify the "model" element.

In our case we have two distinct models, the first is the pair of the section title and the links for that section. The second is the individual link. In a templating DSL we would probably do something like the following:

```
{% for section in sections %}
<h2 class="title">{{ section.title }}</h2>
<ul class="content">
 {% for link in section.links %} <!-- Inner Loop -->
 <a target="new" href="{{ link.href }}">{{ link.text }}</a>
 {% endfor %}
</ul>
{% endfor %}
```

First, we want to be able handle the inner loop. On one level, as you're about see, there's a little more typing involved upfront when using Enlive. But you will end up with something that's considerably more reusable. In the traditional template the inner link loop and the outer section loop are hopelessly interwined. You may have many pages on your site that use the same section pattern but not the internal link pattern. But since these can't be separated you'll have to do some copy and paste. Not so with Enlive.

So let's define our link component. We don't want the dummy content so we really only want to match the very first link that satisfies our need, the selector looks something like this:

```
(def *link-sel* [[:.content (nth-of-type 1)] :> first-child])
```

We only want to match the first ul element that we find that has the content class and only the very first child inside that. This is the selector that gets the job done. It's analogous to:

```
.content:nth-of-type(1) > *:first-child
```

It's important to note that using `nth-of-type` requires an extra pair of brackets around the element that matches `:.content`. This extra pair of brackets is easy to forget. Whenever you want to be more specific about what type of element you want to match (beyond matching on CSS id or class) you'll need an extra pair of brackets.

Now that we have our selector `defsnippet` will look like the following:

```
(defsnippet link-model "tutorial/template2.html" *link-sel*
  [{text :text href :href}]
  [:a] (do->
        (content text)
        (set-attr :href href)))
```

Snippets are like templates with two main differences. First, snippets take a selector. This means that they can match only specific parts of an HTML document. The function produced by a `defsnippet` returns transformed content, *not* a list of strings the way `deftemplate` does. This snippet destructures it's first argument (a hash-map) to extract the value of the keys `:text` and `:href`. We're also introduced to `do->`. This is a convenience, we often want to take the matched element and apply a series of transformations to it. In this case we want to set the content of the node as well as its href attribute.

Let's try out our snippet to see that it worked:

```
tutorial.template2> (render (emit* (link-model {:href "bar" :text "foo"})))
```

Here we have to use `emit*` because snippets return a sequence of nodes not strings the way templates do. `render` is just a utility function for taking a number of strings and creating a single string.

Okay now we want to loop over the sections. A section is a **h2** tag followed by a **ul** tag. Again we need

to figure out the correct selector. This time we're trying to emulate the following popular pattern for the outer loop:

```
{% for section in sections %} <!-- OUTER LOOP -->
<h2>{{ section.title }}</h2>
<ul>
  {% for link in section.links %}
  <a target="new" href="{{ link.href }}>{{ x.text }}</a>
  {% endfor %}
</ul>
{% endfor %}
```

Note that unlike the previous example what we're looping over has no "container". That is, there is no surrounding element for the adjacent **h2** and **ul** tags. Enlive recently added support for "ranges" making it simple to express this pattern with Enlive templates.

Again our HTML has some dummy content again. We only care about the first range of **h2** and **ul** tags, we don't want to match any more than that. We can define a selector to do this like so:

```
(def *section-sel* {[:.title] [[:.content (nth-of-type 1)]]})
```

There is no CSS selector that can represent this. Again take care to note that since we want to select only the first **ul** that we find, we need an extra pair of brackets around `:.content`. This is a common mistake to leave these out.

Now that we have our selector we can define our section snippet like so. Pretty straightforward. Remember `defsnippet` just creates a function which can take whichever arguments you specify and returns the transformed markup. We're creating links using `link-model` and putting those links inside of the **ul** in the section.

```
(defsnippet section-model "tutorial/template2.html" *section-sel*
  [{:keys [title data]} model]
  [:.title]   (content title)
  [:.content] (content (map model data)))
```

Now let's look at the template to see how we put this all together:

```
(deftemplate index "tutorial/template2.html"
  [{:keys [title sections]}]
  [:#title] (content title)
  [:body]   (content (map #(section-model % link-model) sections)))
```

As you can see it looks really similar to `section-model`. Again the main difference is that templates don't take selectors and the function they define returns a list of strings.

That's it. While we've seen some interesting features and while HTML and code separation is cool, so far you may think Enlive involves *more* work then it actually saves. That's because we're showing a very trivial example. In the third example we'll demonstrate just how much time Enlive can save you when building something a little more real world.

Your Third Template – Template Inheritance

We now have a basic working idea of how templates work in Enlive. Templates are simply functions. Now it's still unclear if there is any real advantages to the Enlive way. Hopefully in this tutorial we can prove it's immense power.

Start a REPL if you don't already have one running with **lein repl**. Type the following:

```
tutorial.template2=> (.stop *server*)
nil
tutorial.template2=> (load "template3")
nil
tutorial.template2=> (in-ns 'tutorial.template3)
nil
```

If you're not continuing from a previous tutorial you should ignore `(stop-app)` and you'll need to be more specific about your loading, use `(load "tutorial/template3")` instead.

Point your favorite web browser to **http://localhost:8080/base.html**. You should see a fairly plain page. This is not a template. You can try opening up **base.html** as a file in your browser and see that it's identical to what is being served by Ring. Now point your browser at **http://localhost:8080/3col.html**. You should see another page that has a 3 column layout. Now point your browser at **http://localhost:8080/a/**. The code required to do this follows:

```
(defn viewa []
  (base {:title "View A"
         :main (three-col {})}))
```

If you look at the markup for **base.html** and **3col.html** you will see that there is not one line of code! So how did we magically put these two things together with so little code! Once you understand what's going, you'll see that template inheritance in Enlive is nothing more than combining some functions.

Take a look at **http://localhost:8080/navs.html**. You should see some truly ugly nav bars ;) Now point your browser at **http://locahost:8080/b/**. You can see it's easy to define a site wide layout, a 3 column middle main layout, and customize the contents of each column. Again there's absolute no code in the markup, only the following code is needed to construct this page:

```
(defn viewb []
  (let [navl (nav1)
        navr (nav2)]
   (base {:title "View B"
          :main (three-col {:left  navl
                            :right navr})})))
```

Pretty slick. Templating with Enlive is just writing some Clojure code. This is different from even the good HTML templating solutions out there- few give you the full power of the language.

One last live example before we dive into the code. Point your browser at **http://localhost:8080/c/**. Huh, looks pretty much like b. Point your browser at **http://localhost:8080/c/reverse**. Notice something different?

We just flipped the two navs! How complicated is doing something like this?

```
(defn viewc
  ([] (viewc nil))
  ([action]
     (let [navs [(nav1) (nav2)]
           [navl navr] (if (= action "reverse") (reverse navs) navs)]
       (base {:title "View C"
              :main (three-col {:left  navl
                                :right navr})}))))
```

Nothing more complicated than reversing a vector ;)

So how does this actually work? Open up **template3.clj** in your favorite text editor.

The Templates and Snippets

The first thing to look at is the **base** template.

```
(html/deftemplate base "tutorial/base.html"
  [{:keys [title header main footer]}]
  [:#title]  (maybe-content title)
  [:#header] (maybe-substitute header)
  [:#main]   (maybe-substitute main)
  [:#footer] (maybe-substitute footer))
```

Remember, `maybe-content` and `maybe-substitute` are not Enlive functions. They are two simple macros I've written for the purposes of this tutorial. `maybe-content` will only set the content of its node if its argument is not nil. `maybe-substitute` will only substitute its node if its argument is not nil.

We do this because we want the ability to handle template inheritance. Base represents the most basic template, and we can then "inherit" from it, overriding only specific elements. Note that this template uses **base.html**. You should look at this file now.

Next is the `three-col` snippet. It should be pretty obvious that this is a snippet for doing three column layout. Note that it uses **3col.html**, you should take a look at this file.

The last bits are the various nav snippets and they are loaded from **navs.html**. Again you should go over this file.

The Pages

Now for the fun part. The pages are just functions no more and no less. The first page `viewa` is just rendering the base template with the title "View A" and setting the main block of the page to the 3 column snippet.

The page `viewb` does pretty much the same thing but this time we've added some navs for flair. Notice how much this function looks like `viewb`.

`viewc` does pretty much the same thing but it checks to see if there is a parameter for reversing the navs. If present, the order of the navs is reversed.

It should be far more clear now what Enlive brings to the table over traditional templating solutions. While preparing your templates and snippets takes a little more work up front, building different pages from these templates and snippets is very, very fast and making changes is just moving a couple of values around in your functions, not mucking around with a crippled DSL. Your designer can create all the various widgets for your website using pure HTML and CSS and you can compose your pages from any combination of their designs.

Common Mistakes & Caveats

Converting Numbers

When outputting numbers you need to convert them with `str`.

```
[:div.foobar] (content (str 1))
```

Since snippets take a selector sometimes you might not have set this value correctly. This is usually the case if you're not seeing any output at all from a snippet. It's really easy to test a snippet – they're just functions.

Template out of date

Your templates do not automatically reload. When you make edits to your HTML or your template code I recommend running the following at the REPL:

```
(load "your-library-name")
```

It's a minor annoyance for all the benefits you reap. It also wouldn't be too hard to create a system that reloaded templates (at least while in development mode) upon page refresh.

Be careful, ***do not include the .clj extension***. Also ***do not use -'s in your file name***. If you want dashes you need to name the actual file using underscores.