# A Brief Overview of the Clojure Web Stack

This article introduces Clojure's web application stack. The heart of this stack is Ring: an interface for conforming libraries, a set of adapters for various HTTP servers, and middleware and utilities. This article aims to help you navigate the increasingly broad range of libraries and choose some solid libraries and get an app moving with Ring.

You will need have at least a basic understand of Clojure (1.2.0), Leiningen and HTTP/Web development to get the most out of this article.

All code in this article is using Clojure 1.2 and Ring 0.3.7.

**A note on *full stack* frameworks:**

If you are familiar with web application frameworks in other languages, such as *Ruby on Rails* or *Django*, you may be looking for (or expecting) a full stack solution. While there is a full stack solution for Clojure, I encourage you to explore the wide range of tools available in the Ring ecosystem with an open mind; many of the reasons you might typically choose a full stack are not as important in Clojure.

## Architecture Overview

- **Application**

  The specific details for your web application.
  - Handler(s)
    Application specific
  - Middleware
    Both application specific and from `ring-core`.
- Ring Adapter
- Web Server

Most of the details we will be examining will be in the *Application* layer of this diagram, after all it is the section specific to your sites. We will briefly look at *adapters* and *servers*; just enough to get going.

## The Basics: Requests, Responses, Handlers & Middleware

> "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures." — Alan J. Perlis

At its most basic, Ring is an interface spec. This spec defines *Request* and *Response* map contents and how a function, called a *handler*, should treat them. A handler is just a function that takes a map and returns a map. In both cases what the keys are, and what their corresponding values are is detailed in the spec. Do look at the spec, and reference it whenever a new request or response key is introduced. To reiterate: there is no magic in a handler function.

The simplest, and traditional, example of a handle is:

```
?
```

```
(defn hello-handler [req] {:body "Hello, World!"
                           :headers {}
                           :status 200})

(hello-handler {:uri "/hello"}) ; => {:body "Hello, World!",
                                ;     :headers {},
                                ;     :status 200}
```

This handler ignores the details of the request (such as the uri and http method) and returns a simple 'Hello, World!' resource. Notice that a web application in Ring is simply a Clojure function, thus it can leverage all the standard Clojure tools for procressing maps and for handling functions. As we will see, this also makes it trivial to test in a repl.

**Spurious request maps**

The observant of you (who have seen the spec) will realise that I am omitting a number of required fields from the request map; it is easier to explore the interface in a REPL. I'll be adding more properties as required, just be aware that these examples are not complete.

As an example of leveraging this power, the following handler uses Clojure's destructuring and a `when` form to check for the correct uri before returning a result:

```
(defn hello-handler-2 [{:keys [uri]}]
   (when (= uri "/hello") {:body "Hello, World!"
                           :headers {}
                           :status 200}))

(hello-handler-2 {:uri "/"})   ; => nil
(hello-handler-2 {:uri "/hello"}) ; => {:body "Hello, World!",
                                  ;     :headers {}
                                  ;     :status 200}
```

We can extract out the concept of checking a path to match a constant so that we can reuse it:

```
(defn wrap-uri-check [expected-uri handler]
   (fn [{:keys [uri] :as req}]
       (when (= uri expected-uri)
             (handler req))))

(def hello-handler-3
    (wrap-uri-check "/hello"
        (fn [req] {:body "Hello, world!"
                   :headers {}
                   :status 200})))

(hello-handler-3 {:uri "/"})   ; => nil
(hello-handler-3 {:uri "/hello"}) ; => {:body "Hello, World!",
                                  ;     :headers {}
                                  ;     :status 200}
```

Now we have a reusable uri checking function. This pattern of decorating a handler function with a

wrapper that processes the request (or response) is known as a *middleware*. Note that middleware are called in the reverse order to the wrapping, e.g. the inner most middleware handles the incoming request last and the outgoing response first, while the outer most middleware handles the incoming request first and the outgoing request last.

One last variation of `hello-handler`:

**Dependencies:** `[ring/ring-core "0.3.7"]`

```
?
1  (use '[ring.util.response :only [response]])
2
3  (defn make-greeting-handler [word greeting]
4      (wrap-uri-check (str "/" word)
5          (constantly (response greeting))))
6
7  (defn first-of
8      [arguments fns]
9      (first (keep #(apply % arguments) fns)))
10
11 (def hallo-handler (make-greeting-handler "hallo" "Hallo Welt!"))
12 (def hello-handler-4 (make-greeting-handler "hello" "Hello, World!"))
13
14 (def greeting-handler
15     #(first-of [%] [hello-handler-4 hallo-handler]))
16
17 (greeting-handler {:uri "/hallo"}) ; => {:status 200,
18                                          :headers {},
19                                          :body "Hallo Welt!"}
20 (greeting-handler {:uri "/hello"}) ; => {:status 200,
21                                          :headers {},
22                                          :body "Hello, World!"}
23 (greeting-handler {:uri "/gidday"}) ; => nil
```

This variation shows how applications can be composed of other applications. We use `constantly` to generate a static handler around a response; We have a factory function that generates new handlers already wrapped with a middleware, and then lastly we compose a couple of applications together into one application using the `first-of` combinator we defined[i].

### `nil` Responses

The definition and use of `first-of` takes advantage of a feature of Ring that isn't mentioned in the spec but is none the less idiomatic: Any ring handler may return `nil` instead of a response map; If the top level handler returns `nil`, then Ring generates a (basic) 404 handler. This can be used (as we have done here) to define handlers that compose easily.

We will return to this simple example later to examine more expressive tools for processing routes and creating responses. Next though, we need to look at how to actually connect our simple hello application to a web server.

**Response Body**

A brief aside on the `:body` property of the response map: So far we have just returned `strings` as the body. However, Ring allows any of `String`, `ISeq`, `File` or `InputStream`. Check out the spec for more details.

### Servers and Adapters

So far we have been examining how the *Application* layer is created with Ring. We know how to define handlers and middleware and how to compose them to create interesting applications. The adapter is responsible for not only connecting a handler to a server, but for abstracting away the details of that particular server. As a result each adapter can vary quite a bit in the details of how it is implemented and used.

There are roughly three models for how your application may be connected to the server and outside world:

- Host an HTTP server, such as Jetty, internally as part of your application. This adapter can be found in `ring.adapter.jetty` which is part of the Ring project.
- Host your application as a servlet inside a container such as Tomcat. This adapter can be found in `ring.util.servlet` which is part of the Ring project.
- Talk over some protocol to an web server outside your JVM, such as Mongrel2 over 0MQ. Mongrel2 is currently supported via `ring-mongrel2-adapter`.

You may find yourself mixing some of those approaches; e.g. hosting a Jetty server inside you application and talking to an external NginX server over HTTP.

For getting started I would suggest that you stick with an embedded Jetty, as you can trivially run it from a REPL and access it directly from your localhost. The following block of code shows how you would connect the `greeting-handler` application from earlier to an internal Jetty:

**Dependencies:** `[ring/ring-jetty-adapter "0.3.7"]`

```
?
1 (use '[ring.adapter.jetty :only [run-jetty]])
2
3 (defonce server (run-jetty #'greeting-handler
4                            {:port 8000 :join? false}))
```

Visit http://localhost:8000/hello and http://localhost:8000/hallo to see your application in action!

The `#'` used above is known as var quote; this allows you rebind `greeting-handler` in your REPL and the server will immediately reflect your changes. You can also start and stop your server from the repl with `(.start server)` and `(.stop server)`.

# Common Stack

With the nuts and bolts of Ring covered, it's time to survey the options for putting together a stack of your own.

While there is a lot of choice available to the Ring programmer, there are particular choices for various layers of the stack that are common. In particular at *route dispatch* and *HTML generation*. The following diagram expands on the one at the top of this article to show how a real application stack might look:

        - **HTML Generation**
        - Enlive
        - Hiccup
- View Handlers
  Application logic, database access etc. The real guts of your application.

        - **Route Dispatch**

  Uses URL and Method to determine view handler and parse URL fragments.
        - Moustache
        - Compojure

        - **Middleware**

        - Session Handling
          Specific implemention depends on your backend. Ring provides an in-memory solution that is appropriate for development only.
        - Form Decoding
        - etc…
- `ring.adapter.jetty`
- Jetty
  This is a jetty server instance running inside your JVM. At the time of writing the default is Jetty 6.x.

## Route Dispatch

The layer I am calling *Route Dispatch* covers mapping a request to the appropriate sub handler based on (at least) the URI and HTTP method. This is like a generalized, and much more powerful, version of the combination of `wrap-uri-check` and `first-of` that were presented in earlier.[ii]

A second major feature of this layer is that these libraries provide convenient tools for unpacking the URI and binding them to names.

## Moustache

Moustache wires together handlers and middleware using a route dispatch based *application* model that determines which handler to call based on the route information in the request. Secondly it provides sophisticated tools for unpacking a uris with literals, regular expression and custom validators.

From the library user's perspective there is only one macro you need to know: `app`; This returns a new handler function that will dispatch your routes to handlers. Not only that, it will create new handlers for routes with constant results. For example the entire `greeting-handler` is written as:

**Dependencies:** `[moustache "1.0.0"]`

?

```
(use '[net.cgrand.moustache :only [app delegate]]))

(def greeting-handler-2
    (app ["hallo"] "Hallo welt!"
         ["hello"] "Hello, world!"))

(greeting-handler-2 {:uri "/hallo"})
    ; => {:status 200,
    ;     :headers {"Content-Type" "text/plain;charset=UTF-8"},
    ;     :body "Hallo welt!"}

(greeting-handler-2 {:uri "/hello"})
    ; => {:status 200,
    ;     :headers {"Content-Type" "text/plain;charset=UTF-8"},
    ;     :body "Hello, world!"}

(greeting-handler-2 {:uri "/"})
    ; => {:status 404}
```

Notice that not only is moustache making the things we were all ready doing easier it has made them more comprehensive too; We have a real 404 response for "/" and `Content-Type` headers for the two matching routes. Most of behaviour is only present when the app is being used to generate plain text. This isnt the most useful for a real application but it is great for getting off the ground quickly.

Lets extend the example to use unpack the route and look up greeting based on the word for "hello" in the route by dispatching to another handler:

?

```
(def greetings {"hello" "Hello, world!" "hallo" "Hallo welt!"})

(def greeting-handler-3
   (app [word] (fn [req] (when-let [greeting (greetings word)]
                           greeting)))
```

Here we have created a new handler inline. The handler has `word` in its lexical scope and bound to the text of `:uri`. This also shows how Moustache facilitates composition of handlers: any ring handler can be the *Right-Hand-Side* of a *route*, *handler* pair in `app`.

As an example of this `app`/handler composition lets look for a moment at a super powered greeter application. This greeter provides a number of ways to get personalised hello world strings, both via http resources and XML-RPC.[iii] Finally, we'll create a simple middleware to make 404's cleaner.

**Dependencies:** `[necessary-evil "1.1.0"]`

?

```
(require '[necessary-evil.core :as xml-rpc])
    (use '[ring.util.response :only [response]]
         '[net.cgrand.moustache :only [app delegate]])

(def rpc-hello (xml-rpc/end-point
    {:hello (fn hello ([] (hello "World"))
                      ([name] (str "Hello, " name "!")))}))

(defn simple-greeting
    "A parameterised application"
    [greeting]
    (app [name] ["" greeting ", " name "!"]))

(defn make-404
    [req]
    (response (str "Sorry, the resource at "
                   (:uri req "??")
                   " was not able to be located")))

(defn wrap-404s
    [handler]
    (fn [req]
        (let [resp (handler req)]
            (if (or (nil? resp) (= (:status resp 404) 404))
                (make-404 req)
                resp))))

;; clearly you wouldnt do this in the real world, but its a nice
;; example
(def greeting-handler-4
    (app wrap-404s
         ["formal" name] (fn [r] (response (str "How do you do,"
                                                name "?")))
         ["everyday" & ] (simple-greeting "Hello")
         ["casual" & ] (simple-greeting "Hi")
         ["xmlrpc"] rpc-hello))
```

This is quite a bit of code compared to previous examples, but you should be able to work out what is going on. The newly introduced constructions we have not seen before are:

- The `xml-rpc/end-point`; This is part of `necessary-evil` and just creates an ordinary Ring handler.
- A parameterised *moustache* application (`simple-greeting`). This is just returning a new handler whenever it is called, binding `greeting` to its argument
- The Moustache string sequence literal notation in `simple-greeting`, e.g. `["" greeting ", " name "!"]`. Note the empty string `""`; this is due to a quirk in the moustache syntax. The first item in the literal vector must be a string literal.
- The `&` in routes; This allows the remainder of the route to be passed on to the RHS handler.

One quirk of moustache is that you cannot have arbitrary code on the RHS of a route pair; you must provide a handler function. However, if the handler is defined elsewhere it will not have the benefit of lexical capture of route parameters. To help with this, Moustache broadens the interface for handler functions using a utility called `delegate`.

`delegate` is best explained by its definition and an example:

```
(defn delegate
 "Take a function and all the normal arguments to f but the first,
  and returns a 1-argument fn."
 [f & args]
  #(apply f % args))
```

And an example:

```
(defn simple-greeting-2
    [req greeting name]
    (response (str greeting ", " name "!")))

(def greeting-handler-5
    (app wrap-404s
         ["formal" name] (fn [r] (response (str "How do you do,"
                                                name "?")))
         ["everyday" name] (delegate simple-greeting "Hello" name)
         ["casual" name] (delegate simple-greeting "Hi" name)
         ["xmlrpc"] rpc-hello))
```

I mentioned that route dispatch needs to be able to select a handler based on the HTTP method of the request. Moustache support a number of ways of handling this. For example the following contrived handler:

```
(def get-post-handler
   (app [fragment] {:get ["this was a get to: " fragment]
                    :post ["this was a post to: " fragment]}))

(get-post-handler {:uri "/foo" :request-method :get})
    ; => {:status 200,
          :headers {"Content-Type" "text/plain;charset=UTF-8"},
          :body "this was a get to: foo"}

(get-post-handler {:uri "/foo" :request-method :post})
    ; => {:status 200,
          :headers {"Content-Type" "text/plain;charset=UTF-8"},
          :body "this was a post to: foo"}

(get-post-handler {:uri "/foo" :request-method :delete})
    ; => {:status 405,
          :headers {"Allow" "GET, POST"}}
```

As you can see we now are passing the `:request-method` in as a keyword. When our method matches one of the ones allowed by our route we get the responses as expected. If we supply an

unsupported method (or omit it while testing) moustache returns a 405 response with the `Allow` header set to the methods that that resource *will* accept. Remember to check the syntax documentation and walkthrough for additional ways of specifying method types.

The last major feature of Moustache we will look at in this article is route validation. Here is a simple application that does some arithmatic, and needs to ensure that the routes are only valid when the variables are valid numbers.

```
(defn integer [s]
  "Taken from the Moustache walkthrough"
  (try (Integer/parseInt s) (catch Exception e)))

(defn math-view
    [req op & args] (response (str (apply op args))))

(def arithmatic-app
    (app ["add" [n integer] [m integer]] (delegate math-view + n m)
         ["sub" [n integer] [m integer]] (delegate math-view - n m)
         ["negate" [n integer]] (delegate math-view * -1 n)))

(arithmatic-app {:uri "/add/1/2"}) ; => {:status 200,
                                   ;     :headers {},
                                   ;     :body "3"}
(arithmatic-app {:uri "/add/1/a"}) ; => {:status 404}
```

By now you should have a good understanding of the scope and style of moustache. Definately check out the read me, walkthrough and syntax guide. Moustache is a little weird to get started with, but the initial learn curve pays off.

## Compojure

Like Moustach Compojure provides route dispatching. While it performs a similar role, the approach is a little different. If you come from a Ruby web background (Sinatra in particular) a lot of Compojure may feel familiar to you.

**Caveat:** I have only dabbled with Compojure, rather than implementing a full site with it like I have with Moustache. As a result aspects of this section are not as detailed as the previous.
**A note on versions:** Compojure predates Ring and has changed dramatically over its life. When reading articles about Compojure be sure to check the publishing date and versions discussed.

The core of the Compojure is the `routes` macro (and the convenience form `defroutes`). `routes` performs a similar role as `app` does in Moustache (see above). In addition to this macro, there are six macros that are used in combination to define routing: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, and `ANY`. These clearly correspond to the main HTTP methods and all take the same arguments: `[path args & body]`.

Lets re-examine `greeting-handler-2` as a Compojure application:

**Dependencies:** `[compojure "0.6.2"]`

```
(use 'compojure.core)

(defroutes greeting-handler-6
    (ANY "/hallo" [] "Hallo welt!")
    (ANY "/hello" [] "Hello, world!"))

(greeting-handler-6 {:uri "/hallo"})
    ; => {:status 200,
    ;     :headers {"Content-Type" "text/html"},
    ;     :body "Hallo welt!"}

(greeting-handler-6 {:uri "/hello"})
    ; => {:status 200,
    ;     :headers {"Content-Type" "text/html"},
    ;     :body "Hello, world!"}

(greeting-handler-6 {:uri "/"})
    ; => nil
```

If you worked through the Moustache section above, this will be familiar. As you can see, the `greeting-handler-6` defines two routes, `/hallo` and `/hello`. Each of these responds to any HTTP method and returns a constant string response. The empty vector is the arguments for local bindings of the `request` and any variables destructured from the path. Because the routes here are returning constant values this has been left empty. Like Moustache, the first matching route is the one that responsds.

Aside from the definition of the routes, the handling of URIs that are not specified in the routes is the biggest difference. This can specifically handled with the `compojure.route/not-found` utility function:

```
?
(require '[compojure.route :as route])

(defroutes greeting-handler-6
    (ANY "/hallo" [] "Hallo welt!")
    (ANY "/hello" [] "Hello, world!")
    (route/not-found "Four Oh Four"))
```

It is important that `not-found` is the last route in your configuration as it will match any and every request that has not otherwise been handled.

The definition of `not-found` is clear and simple example of composition in Compojure:

```
?
(defn not-found
  "A route that returns a 404 not found response, with its argument
  as the response body."
  [body]
  (routes
    (HEAD "*" [] {:status 404})
    (ANY "*" [] {:status 404, :body body})))
```

We know that `route` returns a new ring handler (afterall, we have been using a route as a ring handler

in the previous examples). This route uses a wildcard route to match every request that comes in regardless of `:uri`. `HEAD` is special cased (to not return a body), otherwise any other method is caught by the `ANY` route. As you can see, the body of a rule is allowed to be a raw Ring response map.

This parametric handler generation follows the same pattern we saw previous with Moustache. Lets look at another example by porting the overkill greeter app from Moustache to Compojure:

**Dependencies:** `[necessary-evil "1.1.0"]`

```
(require '[necessary-evil.core :as xml-rpc])
    (use '[ring.util.response :only [response]])

(defn dispatch
    "dispatch is takes a handler and a new uri and returns a new handler"
    [uri handler]
        (fn [req] (handler (assoc req :uri uri))))

(def rpc-hello (xml-rpc/end-point
    {:hello (fn hello ([] (hello "World"))
                      ([name] (str "Hello, " name "!")))}))

(defn simple-greeting-3
    "A parameterised application"
    [greeting]
    (routes (ANY "/:name" [name] (str greeting ", " name "!"))))

(defroutes greeting-handler-7
    (ANY "/formal/:name" [name]
       (response (str "How do you do," name "?")))
    (ANY "/everyday*" [*] (dispatch * (simple-greeting-3 "Hello")))
    (ANY "/casual*" [*] (dispatch * (simple-greeting-3 "Hi")))
    (ANY "/xmlrpc" [] rpc-hello)
    (route/not-found "Nope; not here."))
```

The biggest difference between the two versions of this code is the introduction of the `dispatch` function. To the best of my knowledge there is no Compojure specific way of doing this. This does however demonstrate a difference between Moustache and Compojure: Moustache modifies the `:uri` of the request for us when it matches, and Compojure does not.

**Updated, 7 June 2011:** James Reeves (author of Compojure and Ring contributer) provided the following correction to my claim above and the corrected code snippet:

> There is, in the recently-introduced `context` macro. However, it appears that this isn't a well-known feature.

```
(require '[necessary-evil.core :as xml-rpc])

(def rpc-hello (xml-rpc/end-point
  {:hello (fn hello ([] (hello "World"))
                    ([name] (str "Hello, " name "!")))}))

(defn simple-greeting-3
  "A parameterised application"
  [greeting]
  (ANY "/:name" [name] (str greeting ", " name "!")))

(defroutes greeting-handler-7
  (ANY "/formal/:name" [name]
    (str "How do you do," name "?"))
  (context "/everyday" [] (simple-greeting-3 "Hello"))
  (context "/casual" [] (simple-greeting-3 "Hi"))
  (ANY "/xmlrpc" [] rpc-hello)
  (route/not-found "Nope; not here."))
```

This example also shows some of the routing patterns that Compojure uses for matching. Compojure uses a library called Clout under the hood to handle route matching.

Finally, Compojure allows complex forms as the body of a route. This form also has an implicit `do`. This is probably the biggest casual differentiator between Moustache and Compojure. I recommend examining the implemetation of the `render` function to learn more about the various things you can return from a route body.

#### Route Dispatch Summary

As you can see from the brief surveys of Moustache and Compojure, they provide a similar range of features. While my preference is for Moustache, either is a good choice and switching between them is relatively trivial.

Compojure is more popular and has a more natural syntax to get started with, the cost is that its a some impedance mismatch with composing apps with other Ring libraries. Moustache is conceptually simple and fits nicely into the Ring ecosystem but at the cost of some slightly surprising syntax at times. Neither has particular comprehensive documentation. If you are very new to Clojure or web development, Compojure might be a better choice. Otherwise my suggestion is to choose the one that seems most straight forward to you.

## HTML Generation

HTML Generation is a core requirement of most web applications (see below for notes on JSON). We will briefly survey the two main candidates.

## Enlive

Enlive is a fantastic library from Christophe Grand who also created Moustache (see above). Instead of trying to cover it myself, I suggest that you read (and work) through David Nolen's in-depth tutorial.

Enlive has a steeper learning curve than the common alternative (Hiccup, see below) but it is, in my

opinion, a superior library. Firstly, in addition to just generating HTML, you can use the same tools to manipulate existing documents. For example David Nolen's tutorial starts out using enlive to scrap web pages. Secondly, the seperation between templates and code is clearer than in any tool I have used: the HTML files are pure HTML, no additional markup, and are manipulated with CSS-like selectors.

**A note on teams:** There is a popular idea that enlive is better when you have a team with seperate people in designer and developer roles, and hiccup is better for the one person does it all application. I disagree however, as working iteratively on the design is much easier as I can edit the html and css in textmate (despite using emacs for my Clojure code) and use the built in webkit preview to see my changes without having to reload anything, or have an application running.

## Hiccup

Hiccup is at the complete opposite end of the spectrum from Enlive: everything exists in Clojure code and there are no external template files. Hiccup is a DSL built around a single macro: `html`. The macro takes zero or more forms which may be either literal text, vectors representing elements or lists which are executed. The following example illustrates how this works:

**Dependencies:** `[hiccup "0.3.4"]`

?

```
(use 'hiccup.core)

; literals:
(html "Hello, world!")
    ; => "Hello, world!"

(html 1)
    ; => "1"

; elements:
(html [:p "Hello, world!"])
    ; => "<p>Hello, world!</p>"

(html [:h1 "Hello," " world!"]
      [:p "Greetings from your computer!"])
    ; => "<h1>Hello, world!</h1><p>Greetings from your computer!</p>"

; element with attributes:
(html [:div {:class "grid_8 alpha"}
          [:p "Trendy grid system time"]])
    ; => "<div class=\"grid_8 alpha\"><p>Trendy grid system
time</p></div>"

; the same html using the CSS like shortcuts:
(html [:div.grid_8.alpha
          [:p "Trendy grid system time"]])
    ; => "<div class=\"grid_8 alpha\"><p>Trendy grid system
time</p></div>"

; calling functions:
(html (interpose " " (range 5)))
    ; => "0 1 2 3 4"

(html [:ul (map (fn [name] [:li name])
                ["Croaker" "Raven" "Murgen" "One-Eye"])])
    ; => "<ul><li>Croaker</li><li>Raven</li><li>Murgen</li><li>One-
Eye</li></ul>"
```

This example highlights basicly everything you need to get going with hiccup. You can see that is extremely simple and allows straight forward composition of elements.

One thing to watch out with hiccup is that content is not escaped by default; you need wrap it in `escape-html` or its alias `h`. This is an unfortunate default that you definately need to be aware of if you choose to use hiccup.

?

```
(html "malicious content: <script>while (true) { /* uh oh */ }</script>")
    ; => "malicious content: <script>while (true) { /* uh oh */ }
</script>"

(html (escape-html "malicious content: <script>while (true) { /* uh oh */ }
</script>"))
    ; => "malicious content: &lt;script&gt;while (true) { /* uh oh */ }
&lt;/script&gt;"

(html (h "malicious content: <script>while (true) { /* uh oh */ }
</script>"))
    ; => "malicious content: &lt;script&gt;while (true) { /* uh oh */ }
 &lt;/script&gt;"
```

The code for hiccup is quite straight forward and worth your time reading at least briefly. The library has some additional middlewares and utilities for pages and forms that may make your life easier. In particular `hiccup.page-helpers` contains macros and functions for different doctypes, common elements such as includes for javascript, css, lists and images. `hiccup.form-helpers` has utility functions for most of the major form controls. Reading through these will help you get a feel for idiomatic *hiccup* code. You may also find the Hiccup Cheatsheet useful.

# Other Components

A real web application is more than route dispatch and HTML generation. These aspects are further from ring so we will only look at the them briefly.

## Database Connectivity

You probably want to be able to communication with a database of some description. Clojure has a wide range of options here depending on your needs.

There are no SQL/Relational DB *ORMs* for Clojure for obvious reasons. Depending on the amount of abstraction you want, you probably want to look at `clojure.contrib.sql` or *ClojureQL*.

*ClojureQL* is an implementation of relational algebra as first class Clojure functions. The most significant advantage is that it allows you to write various expressions as functions on a table and then compose them together to create the particular queries you need. Definitely worth a look. Lau Jensen has an example site on his GitHub built with Moustache, Enlive and ClojureQL that shows how you might use ClojureQL.

`clojure.contrib.sql` is a relatively low level abstraction over JDBC. It is used as internally by ClojureQL.

## Forms

This is one area that has relatively weak support currently. Decoding form data from `application/x-www-form-urlencoded`, or `multipart/form-data` encodings is provided by the core middlewares in `ring.middleware.params` and `ring.middleware.multipart-params`.

The following is an extremely simple example of handling a post-back:

```
?
  (defn form-view
1     [r] (response "<html>
2                      <form method=\"post\">
3                         <input type=\"text\" name=\"val\">
4                         <input type=\"submit\">
5                      </form>
6                    </html>"))
7
8  (defn handle-form-view
9     [r] (response (str "<html>val was:"
10                       (-> r :form-params (get "val"))
11                       "</html>")))
12
13 (def ^{:doc "A very simple moustache based handler that uses wrap-params
14 to decode a form postback"}
15     simple-form-handler
16     (app
17          wrap-params
18          [] {:get form-view
19              :post handle-form-vew}))
20
21 (run-jetty #'simple-form-handler {:port 8000 :join?
   false})
```

Now visit http://localhost:8000/ and you should see a simple form. Enter a value and click submit and you will be taken to a page that displays 'val was:' and the value you entered.

What is not supported is form validation or generation. Users of compojure may find Brenton Ashworth's Sandbar useful. Both Enlive and Compojure/Hiccup may gain utilities for generating forms in the future[iv].

At the time of writing a collection libraries have just appeared that may fit this space: Flutter (a Hiccup based library), and `clj-decline`. The author, Joost Diepenmaat has provided a demo application that covers most of the details.

## JSON Generation

JSON is very straight forward in Clojure as the datastructures in JSON have a very straight-forward mapping to the core Clojure structures. The library `ring-json-params` provides a middleware to take care of decoding incoming JSON data. Mark McGranaghan has an tutorial on building simple RESTful apis using JSON that uses this middleware. Note that this was published in August 2010.

## Finding More…

This article has touched on a number of common components you might want to investigate, but there is probably something else that you need for your project's stack. The following are additional resources that you may find useful:

- Ring Libraries — A catalog of libraries built on top of Ring on the Ring Wiki.
- Clojure Toolbox— A categorised directory of libraries and tools for Clojure.

- Clojure Libraries
- Clojars

### See Also

- Mark McGranaghan: One Ring to Bind Them – A video introducing Ring.

# Glossary

Handler
: A function that takes a *request* map and may return a *response* map.

Middleware
: A Middleware takes a *Handler* and wraps it with a new handler that interposes itself between the caller and the handler and operates on either or both of request and response.

Adapter
: Connects a top level Ring *handler* to an HTTP Server.

## Footnotes

1. If you have come from an OO background, you may want to consider how handlers and middleware relate to the *Decorator* and *Composite* patterns. These ideas are central to building applications with Ring.
2. You should definately prefer one of these libraries to `wrap-uri-check` and `first-of`.
3. Shameless self promotion aside, you really shouldnt use xml-rpc unless a legacy client or service requires it. It does however illustrate how you can use a black box Ring handler in your application.
4. Observant readers may note that these two components exist in different layers in my common stack diagram above.


### Thanks

Thanks to Steven Ashley, Matt Wilson and Alex Popescu for reading over drafts and providing feedback.


### Updates

#### June 7, 2011

- Added James Reeve's comment about `context` in Compojure.
- Corrected small errors in code snippets. Thanks to Shashy Dass for spotting these.
- Corrected German “Hello World”, thanks to Philipp Steinwender for the correction.