Getting Started with Compojure

Compojure is a DSL for *quickly* creating *performant* web applications in Clojure with minimal effort:

```
(ns myapp.core
  (:require [compojure.core :refer :all]
            [org.httpkit.server :refer [run-server]])) ; httpkit is a server

(defroutes myapp
  (GET "/" [] "Hello World"))

(defn -main []
  (run-server myapp {:port 5000}))
```

Step 1: Create a project with Leiningen:

```
lein new myapp
```

Step 2: Put the above code in `src/myapp/core.clj`

Step 3: Add some dependencies to `project.clj`:

```
[compojure "1.1.8"]
[http-kit "2.1.16"]
```

Step 4: Run:

```
lein run -m myapp.core
```

View at: http://localhost:5000/

Compojure apps will run on any ring-compatible server, but we recommend http-kit for its performance and massive concurrency.

Routes

In compojure, each route is an HTTP method paired with a URL-matching pattern, an argument list, and a body.

```
(defroutes myapp
  (GET "/" [] "Show something")
  (POST "/" [] "Create something")
  (PUT "/" [] "Replace something")
  (PATCH "/" [] "Modify Something")
  (DELETE "/" [] "Annihilate something")
  (OPTIONS "/" [] "Appease something")
  (HEAD "/" [] "Preview something"))
```

Compojure route definitions are just functions which accept request maps and return response maps:

```
(myapp {:uri "/" :request-method :post})
; => {:status 200
;     :headers {"Content-Type" "text/html; charset=utf-8}
;     :body "Create Something"}
```

The body may be a function, which must accept the request as a parameter:

```
(defroutes myapp
  (GET "/" [] (fn [req] "Do something with req")))
```

Or, you can just use the request directly:

```
(defroutes myapp
  (GET "/" req "Do something with req"))
```

Route patterns may include named parameters:

```
(defroutes myapp
  (GET "/hello/:name" [name] (str "Hello " name)))
```

You can adjust what each parameter matches by supplying a regex:

```
(defroutes myapp
  (GET ["/file/:name.:ext" :name #".*", :ext #".*"] [name ext]
    (str "File: " name ext)))
```

Middleware

Clojure uses Ring for routing. Handlers are just functions that accept a request map and return a response map (Compojure will turn strings into 200 responses for you).

You can easily write middleware that wraps all or part of your application to modify requests or responses:

```
(defroutes myapp
  (GET "/" req (str "Hello World v" (:app-version req))))

(defn wrap-version [handler]
  (fn [request]
    (handler (assoc request :app-version "1.0.1"))))

(defn -main []
  (run-server (wrap-version myapp) {:port 5000}))
```

Ring-Defaults provides some handy middlewares for sites and apis, so add it to your dependencies:

```
[ring/ring-defaults "0.1.1"]
```

Then, you can import it in your ns:

```
(ns myapp.core
  (:require [compojure.core :refer :all]
            [ring.middleware.defaults :refer :all]
            [org.httpkit.server :refer [run-server]]))
```

And use `wrap-defaults` to add the `site-defaults` middleware to your app:

```
(defn -main []
  (run-server (wrap-defaults myapp site-defaults) {:port 5000}))
```

Now, your handlers may utilize query parameters:

```
(defroutes myapp
  (GET "/posts" req
    (let [title (get (:params req) :title)
          author (get (:params req) :author)]
      (str "Title: " title ", Author: " author))))
```

Or, for POST and PUT requests, form parameters as well

```
(defroutes myapp
  (POST "/posts" req
    (let [title (get (:params req) :title)
          author (get (:params req) :author)]
      (str "Title: " title ", Author: " author))))
```

Return values

The return value of a route block determines the response body passed on to the HTTP client, or at least the next middleware in the ring stack. Most commonly, this is a string, as in the above examples. But, you may also return a response map:

```
(defroutes myapp
  (GET "/" []
    {:status 200 :body "Hello World"})
  (GET "/is-403" []
    {:status 403 :body ""})
  (GET "/is-json" []
    {:status 200 :headers {"Content-Type" "application/json"} :body "{}"}))
```

Static Files

To serve up static files, use `compojure.route.resources`. Resources will be served from your project's `resources/` folder.

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

(defroutes myapp
  (GET "/")
  (route/resources "/")) ; Serve static resources at the root path

(myapp {:uri "/js/script.js" :request-method :get})
; => Contents of resources/public/js/script.js
```

Views / Templates

To use templating with Compojure, you'll need a template library. Here are a few:

Stencil

Stencil is a Mustache template library:

```
(require '[stencil.core :refer [render-string]])

(defroutes myapp
  (GET "/hello/:name" [name]
    (render-string "Hello {{name}}" {:name name})))
```

You can easily read in templates from your resources directory. Here's a helper function

```
(require 'clojure.java.io)

(defn read-template [filename]
  (slurp (clojure.java.io/resource filename)))

(defroutes myapp
  (GET "/hello/:name" [name]
    (render-string (read-template "templates/hello.html") {:name name})))
```

Selmer

Selmer is a Django and Jinja2-inspired templating language:

```
(require '[selmer.parser :refer [render-file]])

(defroutes myapp
  (GET "/hello/:name" [name]
    (render-file "templates/hello.html" {:name name})))
```

Hiccup

Hiccup is a library for representing HTML as Clojure code

```
(require '[hiccup.core :as hiccup])

(defroutes myapp
  (GET "/hello/:name" [name]
    (hiccup/html
      [:html
        [:body
          [:h1 {:class "title"}
            (str "Hello " name)]]])))
```

Markdown

Markdown-clj is a Markdown implementation.

```
(require '[markdown.core :refer [md-to-html-string]])

(defroutes myapp
  (GET "/hello/:name" [name]
    (md-to-html-string "## Hello, world")))
```

Further reading:

- Official Compojure Documentation
- Clojure for the Brave and True