Routing

Contents

Luminus uses Compojure to define application routes. The routes are the entry points to your application and are used to establish a communiction protocol between the server and the client.

Routes

Compojure route definitions are just functions that accept request maps and return response maps. Each route is an HTTP method paired with a URL-matching pattern, an argument list, and a body.

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

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

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

Build Tool: lein ▾

Topics

Your First Application
REPL Driven Developme
Application Profiles
HTML Templating
Static Assets
ClojureScript
- **Routing**

RESTful Services
Request types
Response types
Websockets
Middleware
Sessions and Cookies
Input Validation
Security
Component Lifecycle
Database Access
Database Migrations
Logging
Internationalization
Testing
Server Tuning
Environment Variables
Deployment
Useful Libraries
Sample Applications
Upgrading
Clojure Resources

Books

Or, we can just use the request directly by declaring it as the second argument to the route:

```
(GET "/foo" request (interpose ", " (keys request)))
```

The above route reads out all the keys from the request map and displays them. The output will look like the following:

```
:ssl-client-cert, :remote-addr, :scheme, :query-params, :session, :for
:multipart-params, :request-method, :query-string, :route-params, :con
:cookies, :uri, :server-name, :params, :headers, :content-length, :ser
:character-encoding, :body, :flash
```

Route patterns may include named parameters:

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

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

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

Handlers may utilize query parameters:

```
(GET "/posts" []
  (fn [req]
    (let [title (get-in req [:params :title])
          author (get-in req [:params :author])]
      "Do something with title and author")))
```

Or, for POST and PUT requests, form parameters:

```
(POST "/posts" []
  (fn [req]
    (let [title (get-in req [:params :title])
          author (get-in req [:params :author])]
      "Do something with title and author")))
```

Compojure also provides syntax sugar for accessing the form parameters as seen below:

```
(POST "/hello" [id] (str "Welcome " id))
```

In the guestbook application example we saw the following route defined:

```
(POST "/"  [name message] (save-message name message))
```

Note that POST requests must contain a CSRF token by default. Please refer here for more details on managing CSRF middleware.

This route extracts the name and the message form parameters and binds them to variables of the same name. We can now use them as any other declared variable.

It's also possible to use the regular Clojure destructuring inside the route.

```
(GET "/:foo" {{foo "foo"} :params}
  (str "Foo = " foo))
```

Furthermore, Compojure allows destructuring a subset of form parameters and creating a map from the rest.

```
[x y & z]
x -> "foo"
y -> "bar"
z -> {:v "baz", :w "qux"}
```

Above, parameters x and y have been bound to variables, while parameters v and w remain in a map called z. Finally, if we need to get at the complete request along with the parameters we can do the following:

```
(GET "/" [x y :as r] (str x y r))
```

Here we bind the form parameters x and y, and bind the complete request map to the variable r.

Return values

The return value of a route block determines at least 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, we may also return a response map:

```
(GET "/" []
    {:status 200 :body "Hello World"})

(GET "/is-403" []
    {:status 403 :body ""})

(GET "/is-json" []
    {:status 200 :headers {"Content-Type" "application/json"} :body "{
```

Static Resources

By default, any resources located under the `resources/public` directory will be available to the clients. This is handled by the `compojure.route/resources` handler found in the `<app>.handler` namespace:

```
(defroutes base-routes
  (route/resources "/")
  (route/not-found "Not Found"))
```

Any resources found on the classpath of the application can be accessed using `clojure.java.io/resource` function:

```
(slurp (clojure.java.io/resource "myfile.md"))
```

Conventionally, non-source resources should be placed in the `resources` directory of the project.

Handling file uploads

Given page called `upload.html` with the following form:

```
<h2>Upload a file</h2>
<form action="/upload" enctype="multipart/form-data" method="POST">
    {% csrf-field %}
    <input id="file" name="file" type="file" />
    <input type="submit" value="upload" />
</form>
```

we could then render the page and handle the file upload as follows:

```
(ns myapp.upload
  (:use compojure.core)
  (:require [myapp.layout :as layout]
            [ring.util.response :refer [redirect file-response]])
  (:import [java.io File FileInputStream FileOutputStream]))

(def resource-path "/tmp/")

(defn file-path [path & [filename]]
  (java.net.URLDecoder/decode
    (str path File/separator filename)
    "utf-8"))

(defn upload-file
  "uploads a file to the target folder
   when :create-path? flag is set to true then the target path will be
  [path {:keys [tempfile size filename]}]
  (try
    (with-open [in (new FileInputStream tempfile)
                out (new FileOutputStream (file-path path filename))]
      (let [source (.getChannel in)
            dest   (.getChannel out)]
        (.transferFrom dest source 0 (.size source))
        (.flush out)))))

(defroutes home-routes
  (GET "/upload" []
       (layout/render "upload.html"))

  (POST "/upload" [file]
       (upload-file resource-path file)
       (redirect (str "/files/" (:filename file))))
```

```
  (GET "/files/:filename" [filename]
       (file-response (str resource-path filename))))
```

Th `:file` request form parameter points to a map containing the description of the file that will be uploaded. Our `upload-file` function above uses `:tempfile`, `:size` and `:filename` keys from this map to save the file on disk.

A file upload progress listener can be added in the `<app>.middleware/wrap-base` function by updating `wrap-defaults` as follows:

```
(wrap-defaults
  (-> site-defaults
      (assoc-in [:security :anti-forgery] false)
      (dissoc :session)
      (assoc-in [:params :multipart]
                {:progress-fn
                 (fn [request bytes-read content-length item-count]
                   (log/info "bytes read:" bytes-read
                             "\ncontent length:" content-length
                             "\nitem count:" item-count))})))
```

Alternatively, if you're fronting with Nginx then you can use its Upload Progress Module.

Organizing application routes

It's a good practice to organize your application routes together by functionality. Compojure provides the `defroutes` macro which can group several routes together and bind them to a symbol.

```
(defroutes auth-routes
  (POST "/login" [id pass] (login id pass))
  (POST "/logout" [] (logout)))

(defroutes app-routes
  (GET "/" [] (home))
  (route/resources "/")
  (route/not-found "Not Found"))
```

It's also possible to group routes by common path elements using `context`. If you had a set of routes that all shared `/user/:id` path as seen below:

```
(defroutes user-routes
  (GET "/user/:id/profile" [id] ...)
  (GET "/user/:id/settings" [id] ...)
  (GET "/user/:id/change-password" [id] ...))
```

We could rewrite that as:

```
(def user-routes
  (context "/user/:id" [id]
    (GET "/profile" [] ...)
```

```
    (GET "/settings" [] ...)
    (GET "/change-password" [] ...)))
```

Once all your application routes are defined you can add them to the main handler of your application. You'll notice that the template already defined the `app` route group in the `handler` namespace of your application. All you have to do is add your new routes there.

Note that you can also apply custom middleware to the routes using `wrap-routes` as seen with `home-routes`. The middleware will be resolved after the routes are matched and only affect the specified routes as opposed to global middleware that's referenced in the `middleware/wrap-base`.

```
(mount/defstate app
  :start
  (middleware/wrap-base
    (routes
      (wrap-routes #'home-routes middleware/wrap-csrf)
      (route/not-found
        (:body
         (error-page {:code 404
                      :title "page not found"}))))))
```

Further documentation is available on the official Compojure wiki

Restricting access

Some pages should only be accessible if specific conditions are met. For example, you may wish to define admin pages only visible to the administrator, or a user profile page which is only visible if there is a user in the session.

Restricting access based on route groups

The simplest way to restrict access is by applying the `restrict` middleware to groups of routes that should not be publicly accessible. First, we'll add the following code in the `<app>.middleware` namespace:

```
(ns <app>.middleware
  (:require
    ...
    [buddy.auth.middleware :refer [wrap-authentication]]
    [buddy.auth.backends.session :refer [session-backend]]
    [buddy.auth.accessrules :refer [restrict]]
    [buddy.auth :refer [authenticated?]]))

(defn on-error [request response]
  {:status  403
   :headers {"Content-Type" "text/plain"}
   :body    (str "Access to " (:uri request) " is not authorized")})


(defn wrap-restricted [handler]
  (restrict handler {:handler authenticated?
                     :on-error on-error}))
```

```
(defn wrap-base [handler]
  (-> handler
      wrap-dev
      (wrap-authentication (session-backend))
      ...))
```

We'll wrap the authentication middleware that will set the `:identity` key in the request if it's present in the session. The session backend is the simplest one available, however Buddy provides a number of different authentications backends as described here.

The `authenticated?` helper is used to check the `:identity` key in the request and pass it to the handler when it's present. Otherwise, the `on-error` function will be called.

This is the default authentication setup that will be produced using the `+auth` profile when creating a new project.

We can now wrap the route groups we wish to be private using the `wrap-restricted` middleware in the `<app>.handler/app` function:

```
(def app
  (-> (routes
        (-> home-routes
            (wrap-routes middleware/wrap-csrf)
            (wrap-routes middleware/wrap-restricted))
        base-routes)
      middleware/wrap-base))
```

Restricting access based on URI

Using the `buddy.auth.accessrules` namespace from Buddy, we can define rules for restricting access to specific pages based on its URI pattern.

Specifying Access Rules

Let's take a look at how to create a rule to specify that restricted routes should only be accessible if the `:identity` key is present in the session.

First, we'll reference several Buddy namespaces in the `<app>.middleware` namespace.

```
(ns myapp.middleware
  (:require ...
            [buddy.auth.middleware :refer [wrap-authentication]]
            [buddy.auth.accessrules :refer [wrap-access-rules]]
            [buddy.auth.backends.session :refer [session-backend]]
            [buddy.auth :refer [authenticated?]]))
```

Next, we'll create the access rules for our routes. The rules are defined using a vector where each rule is represented using a map. A simple

rule that checks whether the user has been authenticated can be seen below.

```
(def rules
  [{:uri "/restricted"
    :handler authenticated?}])
```

We'll also define an error handler function that will be used when access to a particular route is denied:

```
(defn on-error
  [request value]
  {:status 403
   :headers {}
   :body "Not authorized"})
```

Finally, we have to add the necessary middleware to enable the access rules and authentication using a session backend.

```
(defn wrap-base [handler]
  (-> handler
      (wrap-access-rules {:rules rules :on-error on-error})
      (wrap-authentication (session-backend))
      ...))
```

Note that the order of the middleware matters and `wrap-access-rules` must precede `wrap-authentication`.

Buddy session based authentication is triggered by setting the `:identity` key in the session when the user is successfully authenticated.

```
(def user {:id "bob" :pass "secret"})

(defn login! [{:keys [params session]}]
  (when (= user params)
    (-> "ok"
        response
        (content-type "text/html")
        (assoc :session (assoc session :identity "foo")))))
```