RESTful Services

Contents

The recommended way to write REST services in Luminus is by using the Compojure-API library. The library uses Prismatic Schema to generate and check request and response parameters for the endpoints.

The easiest way to add Swagger support is by using the `+swagger` profile:

```
lein new luminus swag +swagger
```

The resulting project will contain an `<app>.routes.services` namespace with a few example routes defined.

Working with Swagger

We can see that routes are declared using the Compojure-API helpers such `compojure.api.sweet/GET*` as opposed to `compojure.core/GET` that we'd use normally.

The syntax for these endpoints is similar to the standard Compojure syntax except that it also requires us to annotate each service operation as seen below:

```
(GET* "/plus" []
        :return       Long
        :query-params [x :- Long {y :- Long 1}]
        :summary      "x+y with query-parameters. y defaults to 1."
        (ok (+ x y)))
```

The above service operation can be called from ClojureScript as follows:


```
(ns swag.core
  (:require [reagent.core :as reagent :refer [atom]]
            [ajax.core :refer [GET]]))

(defn add [params result]
  (GET "/api/plus"
       {:headers {"Accept" "application/transit+json"}
        :params @params
        :handler #(reset! result %)}))

(defn int-value [v]
  (-> v .-target .-value int))

(defn home-page []
  (let [params (atom {})
        result (atom nil)]
    (fn []
      [:div
       [:form
        [:div.form-group
         [:label "x"]
         [:input
          {:type :text
           :on-change #(swap! params assoc :x (int-value %))}]]
        [:div.form-group
         [:label "y"]
         [:input
          {:type :text
           :on-change #(swap! params assoc :y (int-value %))}]]]
       [:button.btn.btn-primary {:on-click #(add params result)} "Add"
       (when @result
         [:p "result: " @result])])))

(reagent/render-component [home-page] (.getElementById js/document "ap
```

We must specify the return type, the query parameter types, and provide a description for each service operation. When working with complex types we must provide a schema definition for each one:

```
(s/defschema Thingie {:id Long
                      :hot Boolean
                      :tag (s/enum :kikka :kukka)
                      :chief [{:name String
                               :type #{{:id String}}}]})

(POST* "/echo" []
       :return   Thingie
       :body     [thingie Thingie]
       :summary  "echoes a Thingie from json-body"
       (ok thingie))
```

The project is also setup to generate a documentation page for the services using the ring-swagger-ui library. The API documentation is available at the `/swagger-ui` URL.

```
(ring.swagger.ui/swagger-ui
   "/swagger-ui"
   :api-url "/swagger-docs")
```

CSRF

CSRF protection provided by the ring-anti-forgery middleware is enabled by default. The `+swagger` profile creates the `service-routes` that aren't wrapped by CSRF protection.

```
(def app
  (-> (routes
        service-routes ;; no CSRF protection
        (wrap-routes home-routes middleware/wrap-csrf)
        base-routes)
      middleware/wrap-base))
```

In order to add CSRF support for Swagger services, you would need to add the following options to the service routes:

```
(context "/api" []
   :middleware [wrap-anti-forgery]
   :header-params [{x-csrf-token :- String nil}]
   ...)
```

The token will have to be pasted as an optional header-parameter in the UI.

Authentication

Services declared using compojure-api can have their own authentication rules. This is useful if you wish to return different kinds of errors than you would when serving HTML pages.

In order to provide authentication, we'll first need to implement `restructure-param` methods:

```
(ns <<myapp>>.routes.services
  (:require ...
            [compojure.api.meta :refer [restructure-param]]
            [buddy.auth.accessrules :refer [restrict]]
            [buddy.auth :refer [authenticated?]]))

(defn access-error [_ _]
  (unauthorized {:error "unauthorized"}))

(defn wrap-restricted [handler rule]
  (restrict handler {:handler  rule
                     :on-error access-error}))

(defmethod restructure-param :auth-rules
  [_ rule acc]
  (update-in acc [:middleware] conj [wrap-restricted rule]))

(defmethod restructure-param :current-user
  [_ binding acc]
  (update-in acc [:letks] into [binding `(:identity ~'+compojure-api-r
```

The above code creates the `:auth-rules` key that can be used in compojure-api routes. This key will apply the authentication middleware

to the routes using it.

The `:current-user` key will bind the `:identity` from the request and can be used to access the user identity.

We can now define services as follows:

```
(defn admin? [req]
  (and (authenticated? req)
       (#{:admin} (:role (:identity req)))))

(defapi service-routes
  {:swagger {:ui   "/swagger-ui"
             :spec "/swagger.json"
             :data {:info {:version     "1.0.0"
                           :title       "Sample API"
                           :description "Sample Services"}}}}

  (POST "/login" req
    :return {:userid String}
    :body-params [userid :- String pass :- String]
    :summary "User login handler"
    (assoc-in (ok {:userid userid}) [:session :identity] {:userid user

  (context "/api" []
    ;; note the :auth-rules key pointing to the authenticated? rule
    ;; all routes within the context will require that the user is pre
    :auth-rules authenticated?

    ;;authentication can also be specified as a combination of rules
    ; :auth-rules {:or [authenticated? admin?]}
    ; :auth-rules {:and [authenticated? admin?]}

    :tags ["private"]

    (GET "/foo" []
      :current-user user
      (ok user))

    (POST "/logout" []
      :return String
      :summary "remove the user from the session"
      (assoc (ok "ok") :session nil))))
```

In the above example, the `/login` route does not require authentication. Meanwhile, the routes defined within the `/api` context will only be accessible when a user is present in the session.