Published in The Startup

Functional Human Follow

Jun 13, 2019 · 11 min read · ✦ · Listen

Save

Building a REST API in Clojure

Backend API development made simple with Clojure

I am going to write this up for anyone who is just starting to get into Clojure or for anyone else who is curious to see how a simple API server could be built.

Clojure is a really great functional programming language with many cool features. Hopefully you will find this tutorial useful if you have skimmed through the Clojure documentation to get a feel for the basic datatypes and language, or perhaps dabbled in the REPL writing some simple functions.

Photo by Ana Martin on Unsplash

We are going to build a simple REST API to get a feel for what a simple API back end might look like. To do this I will be using the fantastic *HTTPKit* Compojure and data.JSON libraries and you can also clone the final code on my github.

In production you might also choose to use a more feature rich implementation such as Yada, Liberator or Pedestal, but for our simple purposes Compojure, HTTPKit and Ring will suffice, as in many cases these frameworks often share the same basic building blocks.

We will develop a simple API that defines the following basic operations:

```
GET http://127.0.0.1:3000/ - prints "Hello World"
GET http://127.0.0.1:3000/request/ - Shows us the HTTPRequest object
GET http://127.0.0.1:3000/hello?name=fh - prints "Hello, FH"
GET http://127.0.0.1:3000/people/ - Returns JSON of people

Finally, we will also be able to add new people here:
GET http://127.0.0.1:3000/people/add?firstname=john&surname=lennon
```

Getting started...

I am going to assume that you have already set up your Clojure environment with the Java SDK and Leiningen by following other hello-world type tutorials, as the focus today is on creating a simple REST API.

Leiningen is a brilliant tool for Clojure, as it allows us to create various Clojure projects with scaffold to get quickly up and running, be it a simple Clojure app or a ClojureScript single page application. We are going to start by creating a simple Clojure app that will compile to Java.

In a terminal command, navigate to a projects directory of your choice and enter the following command to create a new project

```
lein new app rest-demo
```

This will create our sample application. In your terminal, move into the project subfolder `\rest-demo\rest-demo\` and run the app by typing:

```
lein run
; Hello, World!
```

If you look in the project folders you will see that we have a project.clj file which is Clojure's equivalent of a package.json file. We are going to modify this file to add some extra dependencies into our project.

Replace the dependcies section to add the following libraries:

```
:dependencies [[org.clojure/clojure "1.10.0"]
                ; Compojure - A basic routing library
                [compojure "1.6.1"]
                ; Our Http library for client/server
                [http-kit "2.3.0"]
                ; Ring defaults - for query params etc
                [ring/ring-defaults "0.3.2"]
                ; Clojure data.JSON library
                [org.clojure/data.json "0.2.6"]]
```

Ring - https://github.com/ring-clojure/ring
A web application library inspired by Python's WSGI and Ruby's Rack. You can actually use HTTP kit to create a web server without the ring/ring-defaults dependency, but you really need this to be able to work with query parameters and cookies etc.

Compojure — https://github.com/weavejester/compojure
A routing library for Ring. Routes webpages to the appropriate function.

HTTP Kit — https://www.http-kit.org/
A ring-comptable client and event-driven server library. Supports WebSocket and HTTP long polling/streaming

Data.json — https://github.com/clojure/data.json
Clojure's JSON library which converts Clojure data types, e.g. maps to and from JSON.

Serving up a page...

With our dependencies configured, we can now go to our main application file at */src/rest_demo/core.clj*

The top definition is our namespace, rest-demo.core. However, to use our libraries we are going to need to require them, so lets update it to:

```
(ns rest-demo.core
  (:require [org.httpkit.server :as server]
            [compojure.core :refer :all]
            [compojure.route :as route]
            [ring.middleware.defaults :refer :all]
            [clojure.pprint :as pp]
            [clojure.string :as str]
            [clojure.data.json :as json])
  (:gen-class))
```

Next we are going to define our basic routes (we will our define *simple-body-page* and *request-example* functions in a minute):

```
(defroutes app-routes
  (GET "/" [] simple-body-page)
```

```
  (GET "/request" [] request-example)
  (route/not-found "Error, page not found!"))
```

Then we will update our main- entry function to run a HTTPKit server:

```
(defn -main
  "This is our main entry point"
  [& args]
  (let [port (Integer/parseInt (or (System/getenv "PORT") "3000"))]
    ; Run the server with Ring.defaults middleware
    (server/run-server (wrap-defaults #'app-routes site-defaults)
{:port port})
    ; Run the server without ring defaults
    ;(server/run-server #'app-routes {:port port})
    (println (str "Running webserver at http:/127.0.0.1:" port
"/"))))
```

The main function sets up a local port variable using let which will be passed in from a string using Integer/parseInt or, if present, from the envrionment variable named PORT. The next line then runs our HTTP Kit server. It tells HTTPKit to provide some default Ring middleware functionality known as site-defaults and to serve up our app-routes.

It is also possible to run HTTP Kit without the Ring middleware, and you will see some tutorials start instead with:

```
;  (server/run-server #'app-routes {:port port})
```

However this can lead to problems as we shall see a bit later., The ring defaults middleware is required to access things like query-strings and cookies. So we typically pass it into our run-server function with (wrap-defaults #’app-routes site-defaults).

- `api-defaults` The “api” defaults will add support for urlencoded parameters, but not much else.
- `site-defaults` The “site” defaults add support for parameters, cookies, sessions, static resources, file uploads, and a bunch of browser-specific security headers.

You can also set more secure versions of the above to only use https and force ssl encryption, which is what you would do as a minimum in a production environment:

- `secure-api-defaults`
- `secure-site-defaults`

You can find out more about ring here, but for now just keep in mind that by wrapping our routes in the *site-defaults* function we have just provided a boat load of extra functionality to our event handlers, so we can access query-string parameters using :param etc. Without this, we only have limited information available :query-string.

Creating Event Handlers

With our basic server configured, we can now define some event handlers for our simple-body-page and request-example.

```
; Simple Body Page
(defn simple-body-page [req]
  {:status  200
   :headers {"Content-Type" "text/html"}
   :body    "Hello World"})

; request-example
(defn request-example [req]
     {:status  200
      :headers {"Content-Type" "text/html"}
      :body    (->>
                (pp/pprint req)
                (str "Request Object: " req))})
```

You can see that an event handler has a single request parameter which returns a map. In the map, :status is our HTTP status of 200 ok :headers are where our HTTP headers are defined and :body is the body of the response back to the client. Our request-example is similar, except that we are using Clojure's pipeline transformations to print the request object to the console and return a string representation to the browser.

Run the server with `lein run` and navigate to `http://127.0.0.1:3000/`

If all goes well, you should see *"Hello world".* Cool. We should now take a look at our request page to see what our request map actually looks like:

```
http://127.0.0.1:3000/request
```

You should see a horrible message of information returned in your browser. In your terminal window, you should hopefully see something that is a little easier to read, which will give you an idea of what information is available to our event handlers for each request.

Now lets try passing in some data. Add a hello-name function to your code below:

```
(defn hello-name [req] ;(3)
     {:status  200
      :headers {"Content-Type" "text/html"}
      :body    (->
                (pp/pprint req)
                (str "Hello " (:name (:params req))))})
```

Now we can add our handler to a new route at /hello:

```
(defroutes app-routes
  (GET "/" [] simple-body-page)
  (GET "/request" [] request-example)
  (GET "/hello" [] hello-name)
  (route/not-found "Error, page not found!"))
```

Re-run Lein with `lein run`. You should now be able to access our hello-name function with a query parameter:

```
http://127.0.0.1:3000/hello?name=FunctionalHuman
;  Hello FunctionalHuman
```

What if we forget to Wrap-Defaults?

What happens if we forgot to include the Ring defaults middleware and just run our server without wrap-defaults? Try it, comment out the wrap-defaults in our main

function.

```
(defn -main
  "This is our main entry point"
  [& args]
  (let [port (Integer/parseInt (or (System/getenv "PORT") "3000"))]
    ; Run the server with Ring.defaults middleware
    ;(server/run-server (wrap-defaults #'app-routes site-defaults)
{:port port})
    ; Run the server without ring defaults
    (server/run-server #'app-routes {:port port})
    (println (str "Running webserver at http:/127.0.0.1:" port
"/"))))
```

Here is our example request object for hello-name:

```
; Example request object without ring middleware
; http://127.0.0.1:3000/hello?name=FunctionalHuman

{:remote-addr "127.0.0.1",
 :params {},   ; <---- We are missing :params

:route-params {},
 :headers
 {"host" "127.0.0.1:3000",
  "user-agent"
  "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:67.0) Gecko/20100101
Firefox/67.0",
  "cookie" "ring-session=8f0ec983-c697-40f7-b57a-c95fc9050114",
  "connection" "keep-alive",
  "upgrade-insecure-requests" "1",
  "accept"
  "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
  "accept-language" "en-GB,en;q=0.5",
  "accept-encoding" "gzip, deflate",
  "cache-control" "max-age=0"},
 :async-channel
 #object[org.httpkit.server.AsyncChannel 0x10dc3961
"/127.0.0.1:3000<->/127.0.0.1:59051"],
 :server-port 3000,
 :content-length 0,
 :compojure/route [:get "/hello"],
 :websocket? false,
 :content-type nil,
 :character-encoding "utf8",
 :uri "/hello",
 :server-name "127.0.0.1",
 :query-string "name=FunctionalHuman",  ; <--  plain string :(
 :body nil,
```

```
 :scheme :http,
 :request-method :get
```

Notice the empty :params object and the plain string in the :query-string key? Also note how our *hello-name* function no longer works. We should comment the *wrap-defaults* call back in and try again:

```
:cookies
 {"ring-session" {:value "8f0ec983-c697-40f7-b57a-c95fc9050114"}},
 :remote-addr "127.0.0.1",
 :params {:name "FunctionalHuman"},
 :flash nil,
 :route-params {},
 :headers
 {"host" "127.0.0.1:3000",
  "user-agent"
  "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:67.0) Gecko/20100101
Firefox/67.0",

"cookie" "ring-session=8f0ec983-c697-40f7-b57a-c95fc9050114",
  "connection" "keep-alive",
  "upgrade-insecure-requests" "1",
  "accept"
  "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
  "accept-language" "en-GB,en;q=0.5",
  "accept-encoding" "gzip, deflate",
  "cache-control" "max-age=0"},
 :async-channel
 #object[org.httpkit.server.AsyncChannel 0x15f4f780
"/127.0.0.1:3000<->/127.0.0.1:59074"],
 :server-port 3000,
 :content-length 0,
 :form-params {},
 :compojure/route [:get "/hello"],
 :websocket? false,
 :session/key nil,
 :query-params {"name" "FunctionalHuman"},
 :content-type nil,
 :character-encoding "utf8",
 :uri "/hello",
 :server-name "127.0.0.1",
 :anti-forgery-token

"eUQfoV5SR9lL49LqAyUMZ1QbtVezM3P9QDI/xLbBVKNQNRw3WjiCw3VPOOmk18Drx0H
CZll67+FLkcYN",
 :query-string "name=FunctionalHuman",
 :body nil,
 :multipart-params {},
 :scheme :http,
```

```
    :request-method :get,
    :session {}
```

Woah, look at that! We now have a lot more useful information to work with, including cookies. Not only that, but our :params key is now a map of query parameters.

```
  :params {:name "FunctionalHuman"}
```

This means we can now easily retrieve any query parameters in our event handler functions:

```
  (str "Hello " (:name (:params req)))})
```

Building out our API

Now that we have the basics down, we should try and return some JSON. We will use Clojures data.JSON library to easily create JSON from Clojure maps.

We should start by defining our JSON objects, and then bundle them into a collection of some sort. In our example we are going to create a new atom called people-collection, which will store a vector of people.

```
  ; my people-collection mutable collection vector
  (def people-collection (atom []))
```

An Atom in Clojure is a bit like a supercharged variable with some amazing properties. It is mutable, but it can also be mutated by different threads without race conditions or other problems because it operates Software Transactional Memory. In our web server context, all of our threads can access and write to this atom without us having to worry about loosing data.

Now we will define a helper function, *addperson* that will take *firstname* and *secondname* arguments, create a new person map and add the person into our people-collection with capitalized names:

```
;Collection Helper functions to add a new person
(defn addperson [firstname surname]
  (swap! people-collection conj {:firstname (str/capitalize
firstname) :surname (str/capitalize surname)}))
```

The (*swap!*) function is how we can safely write a value to an atom. In this line we are telling Clojure to swap out the people-collection with a new value Then we can create some example people:

```
; Example JSON objects
(addperson "Functional" "Human")
(addperson "Micky" "Mouse")
```

We can then write a new event handler to output JSON as a content-type:

```
; Return List of People
(defn people-handler [req]
        {:status  200
         :headers {"Content-Type" "text/json"}
         :body    (str (json/write-str @people-collection))})
```

The secret sauce here is the (json/write-str people-collection). This function takes our collection and converts all of the mapped key value pairs into JSON.

Before we can test this, we need to update our routes:

```
; Our main routes
(defroutes app-routes
  (GET "/" [] simple-body-page)
  (GET "/request" [] request-example)
  (GET "/hello" [] hello-name)
  (GET "/people" [] people-handler)
  (route/not-found "Error, page not found!"))
```

Finally, re-run lein and our API is now returning valid JSON.

```
;http://127.0.0.1:3000/people
[{
 "firstname": "Functional",
 "surname": "Human"
}, {
 "firstname": "Micky",
 "surname": "Mouse"
}]
```

What about if we want to allow the user to add new data?

We can take what we have learned so far and use our knowledge to create a helper function that will allow the user to create new people with a simple get request, e.g:

```
http://127.0.0.1:3000/people/add?firstname=john&surname=lennon
```

This will make :*firstname* and :*surname* available in our :*params* map of our request object *req*. To make our code a little bit easier to read, we should create a helper function that will take a *propertyname* and a *request* object, and extract the named property.

```
; Get the parameter specified by pname from :params object in req
(defn getparameter [req pname] (get (:params req) pname))
```

We can then use it like this to return our query parameter:

```
(getparameter req :firstname) ; john
```

Our new helper function works well, but things are going to get cluttered as we have to access both :firstname and :surname. To make our code a bit more readable we can start to apply some functional programming concepts.

In FP there is the concept of *partially* applying a function. In our example, we are going to partially apply the getparameter function, by passing it the req object. We can then assign this to a local variable p which will allow us to retrieve any key in the :params map simply by calling `(p :firstname)`

Here is our new addperson-handler event:

```
; Add a new person into the people-collection
(defn addperson-handler [req]
        {:status  200
         :headers {"Content-Type" "text/json"}
         :body    (-> (let [p (partial getparameter req)]
                        (str (json/write-str (addperson (p
:firstname) (p :surname))))))})
```

Handily our addperson helper message already returns the people-collection, so we can just pass that into our json/write-str to respond with valid json.

Finally we need to add another route:

```
(GET "/people/add" [] addperson-handler)
```

If you run lein again, you should be able to add new users by changing the firstname and surname query parameters:

```
http://127.0.0.1:3000/people/add?firstname=john&surname=lennon
http://127.0.0.1:3000/people/add?firstname=paul&surname=mccartney
http://127.0.0.1:3000/people

[{"firstname":"Functional","surname":"Human"},
{"firstname":"Micky","surname":"Mouse"},
{"firstname":"John","surname":"Lennon"},
{"firstname":"Paul","surname":"Mccartney"}]
```

The really cool part is that our web server can be running in different threads and our people-collection atom will never loose data.

Finally, just to end with something simple. The *defroutes function* can be used with other HTTP operations. In this tutorial so far we have just used GET, but we could also use PUT or POST or ANY. For example, to make addperson-handler a HTTP Post operation, simply change as follows:

```
; Our main routes
(defroutes app-routes
  (GET "/" [] simple-body-page)
  (GET "/request" [] request-example)
  (GET "/hello" [] hello-name)
  (GET "/people" [] people-handler)
  (POST "/people/add" [] addperson-handler)
  (route/not-found "Error, page not found!"))
```

Hopefully this tutorial has given you some food for thought about how a simple RESTful API can be created in Clojure. The HTTPServer, Ring, Compojure and data.JSON libraries are a pretty awesome combination.

Our final code is pretty concise and declarative thanks to Clojure and the awesome libraries. We still need to add some error handling, but it gives you an idea of what is possible.

You can view the sourcecode on my github repository.

Clojure Rest Api Clojurescript Programming