Extracted from:

Web Development with Clojure

Build Bulletproof Web Apps with Less Code

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Web Development with Clojure

Build Bulletproof Web Apps with Less Code

Dmitri Sotnikov

edited by Michael Swaine

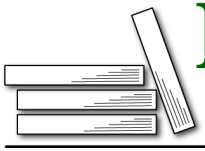
Web Development with Clojure

Build Bulletproof Web Apps with Less Code

Dmitri Sotnikov

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-64-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2014

CHAPTER 3

Liberator Services

In the last chapter we talked about how to write a typical web application and how its components interact with one another. For example, we now know how to manage the routes, write HTML templates, and use sessions for state management. In this chapter we'll look at a different approach to writing applications.

As you've probably noticed, the separation between the client and the server portions of the application is not enforced. If we're not careful we could easily end up with a tightly coupled client and server components. This could become a problem if we wish to add a different client later on—for example, if we decided to create a native mobile version of our application.

In this chapter we'll cover how to use the Liberator library to ensure the separation of concerns between the server and the client.[1] Liberator is a Clojure library for writing RESTful services modeled after webmachine,[2] a popular service framework for Erlang. Its primary feature is that it puts a strong emphasis on decoupling the front end from the back end of your application.

Conceptually, Liberator provides a clean way to reason about your service operations. Each request passes through a series of conditions and handlers defined in the resource. These map to the codes specified by the HTTP RFC 2616, such as 200 - OK, 201 - created, 404 - not found, and so on.

This approach makes it very easy to write standards-compliant services and to group the operations logically. It also means that your services will automatically use the appropriate HTTP codes associated with a particular response.

1. http://clojure-liberator.github.io/liberator/
2. https://github.com/basho/webmachine

Due to its focus on the separation of the front-end and back-end logic, Liberator is a natural choice for writing many types of web applications. These include general-purpose services, single-page applications, and applications that might have nonweb clients, such as mobile applications.

Creating the Project

In this section we'll cover how to create a simple application that serves static resources, provides basic session management, and handles JavaScript Object Notation (JSON) operations.

First let's create a new application called *liberator-service* using the *compojure-app* template.

```
lein new compojure-app liberator-service
```

Once the application is created, add Liberator and Cheshire dependencies to our project.clj dependencies vector:[3]

```
:dependencies
[ ...
  [liberator "0.10.0"]
  [cheshire "5.2.0"]]
```

Cheshire is a fast and easy-to-use JSON parsing library. We'll use it for parsing the requests from the client and generating the responses.

At this point, we should be able to start up the read-evaluate-print loop (REPL) by running the start-server in the liberator-service.repl namespace.

Currently, the application displays the default home route created by the template. Let's look at how we can render a Liberator route instead.

Defining Resources

Liberator uses the concept of resources to interact with the client. The resources are simply Ring-compliant handlers that can be used inside your Compojure routes. These resources are defined using the resource and defresource macros. Let's open the liberator-service.routes.home namespace. We'll remove the reference to layout and add the references for resource and defresource to the declaration:

liberator-snippets/home.clj

```
(ns liberator-service.routes.home
  (:require [compojure.core :refer :all]
            [liberator.core
             :refer [defresource resource request-method-in]]))
```

3. https://github.com/dakrone/cheshire

Now we can replace our "/" route with a resource as follows:

liberator-snippets/home.clj
```
(defroutes home-routes
  (ANY "/" request
       (resource
         :handle-ok "Hello World!"
         :etag "fixed-etag"
         :available-media-types ["text/plain"])))
```

If we reload the page we'll see *Hello World!* displayed. Note that we're using `ANY` Compojure route for our resource. This allows the Liberator resource to handle the request type.

Say we want to name the resource handler; we can use `defresource` instead:

liberator-snippets/home.clj
```
(defresource home
  :handle-ok "Hello World!"
  :etag "fixed-etag"
  :available-media-types ["text/plain"])

(defroutes home-routes
  (ANY "/" request home))
```

The request in the preceding route is simply a map that's described in *What's in the Request Map*, on page ?.

A set of keys defined by the Liberator application programming interface represents each resource. Specific actions are in turn associated with each key. A key can fall into one of four categories:

- Decision
- Handler
- Action
- Declaration

Each key can be associated with either constants or functions. The functions should accept a single parameter that is the current context, and return a variety of responses.

The context parameter contains a map with keys for the *request*, the *resource*, and optionally the *representation*. The *request* key points to the Ring request. The *resource* represents the current state of the resource, and the *representation* contains the results of content negotiation.

Let's take a close look at each of the categories and their purposes.

Making Decisions

The decisions are used to figure out how to handle the client request. The decision keys end with a question mark (?) and their handler must evaluate to a Boolean value.

A decision function can return a Boolean value indicating the result of the decision, or it can return a map or a vector. In case a map is returned, the decision is assumed to have been evaluated to true and the contents of the map are merged with the response map. In case a vector is returned, it must contain a Boolean indicating the outcome, followed by a map to be merged with the response.

When any decision has a negative outcome, its corresponding HTTP code will be returned to the client. For example, if we wanted to mark as unavailable the route we defined earlier, we could add a decision key called service-available? and associate it with a false value:

liberator-snippets/home.clj

```
(defresource home
  :service-available? false
  :handle-ok "Hello World!"
  :etag "fixed-etag"
  :available-media-types ["text/plain"])
```

If we reload the page we'll see the *503* response type associated with the *Service not available* response.

Alternatively, we could restrict access to the resource by using the method-allowed? decision key along with a decision function.

```
(defresource home
  :method-allowed?
  (fn [context]
    (= :get (get-in context [:request :request-method])))
  :handle-ok "Hello World!"
  :etag "fixed-etag"
  :available-media-types ["text/plain"])
```

Since checking the request method is a common operation, Liberator provides a key called :allowed-methods. This key should point to a vector of keywords representing the HTTP methods.

```
(defresource home
  :allowed-methods [:get]
  :handle-ok "Hello World!"
  :etag "fixed-etag"
  :available-media-types ["text/plain"])
```

We can also combine multiple decision functions in the same resource, as seen here:

liberator-snippets/home.clj
```
(defresource home
  :service-available? true

  :method-allowed? (request-method-in :get)

  :handle-method-not-allowed
  (fn [context]
    (str (get-in context [:request :request-method]) " is not allowed"))

  :handle-ok "Hello World!"
  :etag "fixed-etag"
  :available-media-types ["text/plain"])
```

Creating Handlers

A handler function should return a standard Ring response. Handler keys start with the handle- prefix. We saw a handler function when we used the handle-ok key to return the response in our resource.

There are other handlers, such as handle-method-not-allowed and handle-not-found. The full list of handlers can be found on the official documentation page.[4] These handlers can be used in conjunction with the decisions to return a specific response for a particular decision outcome.

For example, if we wanted to return a specific response when the service is not available, we could do the following:

liberator-snippets/home.clj
```
(defresource home
  :service-available? false
  :handle-service-not-available
  "service is currently unavailable..."

  :method-allowed? (request-method-in :get)
  :handle-method-not-allowed
  (fn [context]
    (str (get-in context [:request :request-method]) " is not allowed"))

  :handle-ok "Hello World!"
  :etag "fixed-etag"
  :available-media-types ["text/plain"])
```

Our resource now has custom handlers for each decision outcome.

4. http://clojure-liberator.github.io/liberator/doc/handlers.html

Taking Actions

An action represents an update of the current state by the client, such as a PUT, POST, or DELETE request. The action keys end with an exclamation point (!) to indicate that they're mutating the application's internal state.

Once an action occurs, we can return the result to the client using the handle-created handler.

Writing Declaration

Declarations are used to indicate the resource's capabilities. For example, our resource uses the available-media-types declaration to specify that it returns a response of type text/plain. Another declaration we saw is the etag, allowing the client to cache the resource.

Putting It All Together

Let's look at an example of a service that has a couple of resources that allow the client to read and store some data.

The application will display a list of users and allow the client to add additional users to the list. The client will be implemented in JavaScript and use Ajax to communicate with the service.

To start, let's create a static *HTML* page in our public directory and call it home.html. The page contents will look like this:

liberator-snippets/home.html

```
<html>
  <head>
    <title>Liberator Example</title>
    <script type="text/javascript"
            src="//ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.min.js">
    </script>

    <script type="text/javascript">
      function renderUsers(users) {
        $('#user-list').empty();
        for(user in users)
          $('#user-list').append($('<li/>', {html: users[user]}));
        }
        function getUsers() {
          $.get("/users", renderUsers);
        }
        function addUser() {
          $.post("/add-user", {user: $('#name').val()}, renderUsers);
        }
```

```
        $(function() {getUsers();});
    </script>
  </head>

  <body>
    <h1>Current Users</h1>

    <ul id="user-list"></ul>
    <input type="text" id="name" placeholder="user name"/>
    <button onclick="addUser()">Add User</button>
  </body>

</html>
```

The page contains functions to render a list of users from given a JSON array, get the current users from the /users URI, and add a new user via the /add-user URI. In addition we have a user-list placeholder for displaying the users, and a text field along with the Add User button for adding new users. The page should look like the following image.

We'll now create corresponding resources to handle each of the operations. To serve the data as JSON we'll first have to add a reference to cheshire.core/generate-string in the declaration of our home namespace:

```
(ns liberator-service.routes.home
  (:require ...
            [cheshire.core :refer [generate-string]]))
```

Next we'll create an atom to hold the list of users:

```
(def users (atom ["John" "Jane"]))
```

The first resource will respond to GET requests and return the contents of the users atom as JSON.

liberator-service/src/liberator_service/routes/home.clj
```
(defresource get-users
  :allowed-methods [:get]
  :handle-ok (fn [_] (generate-string @users))
  :available-media-types ["application/json"])
```

In the resource, we use the :allowed-methods key to restrict it to only serve GET requests. We use the available-media-types declaration to specify that the response is of type application/json. The resource will generate a JSON string from our current list of users when called.

The second resource will respond to POST and add the user contained in the form-params to the list of users. It will then return the new list:

liberator-snippets/home.clj
```
(defresource add-user
  :method-allowed? (request-method-in :post)
  :post!
  (fn [context]
    (let [params (get-in context [:request :form-params])]
      (swap! users conj (get params "user"))))
  :handle-created (fn [_] (generate-string @users))
  :available-media-types ["application/json"])
```

Here we check that the method is POST, and use the post! action to update the existing list of users. We then use the handle-created handler to return the new list of users to the client.

Note that with the resource just detailed, the handle-created value *must* be a function.

The following resource will compile without errors. However, when it runs you'll see the old value of users. This is because (generate-string @users) is evaluated *before* the decision graph is run.

liberator-snippets/home.clj
```
(defresource add-user
  :method-allowed? (request-method-in :post)
  :post!
  (fn [context]
    (let [params (get-in context [:request :form-params])]
      (swap! users conj (get params "user"))))
  :handle-created (generate-string @users)
  :available-media-types ["application/json"])
```

It is therefore important to ensure that you provide the :handle-created key with a function that will be run when the decision graph is executed, as we did in the original example.

You'll notice that nothing is preventing us from adding a blank user. Let's add a check in our service to validate the request to add a new user:

liberator-service/src/liberator_service/routes/home.clj

```
(defresource add-user
  :allowed-methods [:post]
  :malformed? (fn [context]
                (let [params (get-in context [:request :form-params])]
                  (empty? (get params "user"))))
  :handle-malformed "user name cannot be empty!"
  :post!
  (fn [context]
    (let [params (get-in context [:request :form-params])]
      (swap! users conj (get params "user"))))
  :handle-created (fn [_] (generate-string @users))
  :available-media-types ["application/json"])
```

Now, if the value of the user parameter is empty, we'll be routed to handle-malformed, which will inform the client that the user name cannot be empty. Next time we try to add an empty user, we'll see a 400 error in the browser:

```
POST http://localhost:3000/add-user 400 (Bad Request)
```

We can now update our page to handle the error and display the message, as follows:

liberator-snippets/home1.html

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
    <title>Liberator Example</title>

    <script type="text/javascript"
            src="//ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.min.js">
    </script>

    <script type="text/javascript">
        function renderUsers(users) {
          $('#user-list').empty();
          for(user in users)
            $('#user-list').append($('<li/>', {html: users[user]}));
        }

        function getUsers() {
          $.get("/users", renderUsers);
        }

        function handleError(xhr) {
          $('#error').text(xhr.statusText + ": " + xhr.responseText);
        }
```

```
        function addUser() {
          var jqxhr = $.post("/add-user", {user: $('#name').val()}, renderUsers)
                       .fail(handleError);
        }

        $(function() {getUsers();});
    </script>
  </head>
  <body>
    <h1>Current Users</h1>
    <p id="error"></p>
    <ul id="user-list"></ul>
    <input type="text" id="name" placeholder="user name"/>
    <button onclick="addUser()">Add User</button>
  </body>
</html>
```

Now, if we click the Add User button without filling in the user name field we'll see the following error:

As a final touch, let's add a home resource that will serve our home.html file. To do that we'll add the lib-noir dependency to our project.clj:

```
:dependencies [... [lib-noir "0.7.2"]]
```

Next we'll add references to noir.io and clojure.java.io to the home namespace declaration:

```
(ns liberator-service.routes.home
  (:require [...
             [noir.io :as io]
             [clojure.java.io :refer [file]]))
```

Now we can create a new resource called home that will serve the home.html file:

liberator-service/src/liberator_service/routes/home.clj
```
(defresource home
    :available-media-types ["text/html"]

    :exists?
    (fn [context]
      [(io/get-resource "/home.html")
       {::file (file (str (io/resource-path) "/home.html"))}])

    :handle-ok
    (fn [{{{resource :resource} :route-params} :request}]
      (clojure.java.io/input-stream (io/get-resource "/home.html")))
    :last-modified
    (fn [{{{resource :resource} :route-params} :request}]
      (.lastModified (file (str (io/resource-path) "/home.html")))))
```

The resource will check whether the file exists and when it was last modified. If the file isn't available then io/get-resource will return a nil and the client will get a 404 error. If the file wasn't changed since the last request, the client will be returned a 304 code instead of the file, indicating that it wasn't modified.

Thanks to this check, the file will be served only if it exists and we made changes to it since it was last requested. We can now add a route to serve home.html as our default resource:

```
(ANY "/" request home)
```

Our home namespace containing the service counterparts to the page should look like this:

liberator-service/src/liberator_service/routes/home.clj
```
(ns liberator-service.routes.home
  (:require [compojure.core :refer :all]
            [liberator.core :refer [defresource resource]]
            [cheshire.core :refer [generate-string]]
            [noir.io :as io]
            [clojure.java.io :refer [file]]))
(defresource home
    :available-media-types ["text/html"]

    :exists?
    (fn [context]
      [(io/get-resource "/home.html")
       {::file (file (str (io/resource-path) "/home.html"))}])

    :handle-ok
    (fn [{{{resource :resource} :route-params} :request}]
      (clojure.java.io/input-stream (io/get-resource "/home.html")))
    :last-modified
```

```
    (fn [{{{resource :resource} :route-params} :request}]
      (.lastModified (file (str (io/resource-path) "/home.html")))))

(def users (atom ["foo" "bar"]))
(defresource get-users
  :allowed-methods [:get]
  :handle-ok (fn [_] (generate-string @users))
  :available-media-types ["application/json"])

(defresource add-user
  :allowed-methods [:post]
  :malformed? (fn [context]
                (let [params (get-in context [:request :form-params])]
                  (empty? (get params "user"))))
  :handle-malformed "user name cannot be empty!"
  :post!
  (fn [context]
    (let [params (get-in context [:request :form-params])]
      (swap! users conj (get params "user"))))
  :handle-created (fn [_] (generate-string @users))
  :available-media-types ["application/json"])

(defroutes home-routes
  (ANY "/" request home)
  (ANY "/add-user" request add-user)
  (ANY "/users" request get-users))
```

As you can see, Liberator ensures separation of concerns by design. With the Liberator model you will have small self-contained functions, each of which handles a specific task.

What You've Learned

So far we've been focusing on the server-client-interaction portion of the application. In the next chapter we'll take a deeper look at connecting to and working with databases.