Extracted from:

Web Development with Clojure, Third Edition

Build Large, Maintainable Web Applications Interactively

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Web Development with Clojure

3rd Edition

Build Large, Maintainable Web Applications Interactively

Dmitri Sotnikov and Scot Brown

edited by Michael Swaine

Web Development with Clojure, Third Edition

Build Large, Maintainable Web Applications Interactively

Dmitri Sotnikov
Scot Brown

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

ISBN-13: 978-1-68050-682-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2021

Build the UI with Reagent

Reagent is more opinionated than React regarding component life cycle. While React is agnostic about the strategy used to trigger component updates; Reagent makes this decision for us. Reagent uses a data-driven approach where components observe the state of reactive atoms. Components just specify the state they're concerned with, and Reagent handles the life cycle for us. As a result, we typically only need to implement the render functions of our components.

Before we can build our UI, we need to add the Reagent dependency to our project:

```
:dependencies
[...
 [reagent "1.0.0"]]
```

Since our dependencies have changed, we also need to restart cljsbuild.

```
...
Successfully compiled "target/cljsbuild/public/js/app.js" in ... seconds.
^C
$ lein cljsbuild auto
Watching for changes before compiling ClojureScript...
...
```

To use Reagent, we have to require it in our namespace just like in Clojure. Let's open up the guestbook.core ClojureScript namespace. Recall that it's found in the src/cljs source folder and not the src folder that contains the Clojure source files. Update the namespace declaration to include the following require form:

```
(ns guestbook.core
  (:require [reagent.core :as r]
            [reagent.dom  :as dom]))
```

Reagent Components

Reagent bases its UI component syntax on the popular templating library called Hiccup.[7] Hiccup uses plain data structures to represent HTML, so you don't have to learn a separate domain-specific language. Let's render a component by calling the reagent.dom/render function, like so:

```
(dom/render
  [:h1 "Hello, Reagent"]
  (.getElementById js/document "content"))
```

7. https://github.com/weavejester/hiccup

reagent.dom/render takes two arguments: a component and the target DOM node.

At this point our namespace should look like this:

```
(ns guestbook.core
  (:require [reagent.core :as r]
            [reagent.dom :as dom]))

(dom/render
  [:h1 "Hello, Reagent"]
  (.getElementById js/document "content"))
```

The HTML nodes are represented using a vector with the structure corresponding to that of the resulting HTML tag, as shown in the following example:

```
[:tag-name {:attribute-key "attribute value"} tag body]

<tag-name attribute-key="attribute value">tag body</tag-name>
```

Let's put our h1 inside a div with id hello and class content.

```
[:div {:id "hello", :class "content"} [:h1 "Hello, Auto!"]]

<div id="hello" class="content"><h1>Hello, Auto!</h1></div>
```

Since setting the id and the class attributes for elements is a common operation, Reagent provides CSS-style shorthand for these actions. Instead of what we wrote earlier, we could simply write our div as follows:

```
[:div#hello.content [:h1 "Hello, Auto!"]]
```

Reagent also provides shorthand for collapsing nested tags into a single tag. The preceding code could be rewritten this way:

```
[:div#hello.content>h1 "Hello, Auto!"]
```

The > indicates that the h1 tag is nested inside the div tag. This notation is very helpful with CSS frameworks, such as Bulma and Bootstrap, that rely on deeply nested tags to style elements.

As you can see, creating Reagent components takes very little code and produces markup that's easy to correlate back to the template definitions.

Reimplementing the Form

By this point you should have an understanding of how you can create different kinds of HTML elements using Reagent. Now let's take a look at what makes Reagent truly special: how you can connect elements to data. We'll bind the form that allows the user to input their name and a message to a Reagent atom that contains the entered values. Also, thanks to Reagent, this binding goes both ways—if the data changes, so will the UI.

So far we've only looked at components that directly represent HTML. However, Reagent allows you to treat any function as a component. The function simply has to return either a Reagent vector or a component that can be rendered by React directly. The latter becomes useful when you want to use React libraries or create your own custom components. Let's not worry about that just yet, though.

In our case, let's create a function called message-form. The function uses a let statement to create a binding for the atom that contains the form data. It then returns a function that generates the form and references the previously defined atom in the input fields of the following component:

guestbook-reagent-start/src/cljs/guestbook/core.cljs

```
(defn message-form []
  (let [fields (r/atom {})]
    (fn []
      [:div
       [:div.field
        [:label.label {:for :name} "Name"]
        [:input.input
         {:type :text
          :name :name
          :on-change #(swap! fields
                             assoc :name (-> % .-target .-value))
          :value (:name @fields)}]]
       [:div.field
        [:label.label {:for :message} "Message"]
        [:textarea.textarea
         {:name :message
          :value (:message @fields)
          :on-change #(swap! fields
                             assoc :message (-> % .-target .-value))}]]
       [:input.button.is-primary
        {:type :submit
         :value "comment"}]])))
```

Here, we're using a closure to create a local state for the fields binding and then returning a function that references it. Reagent first calls the outer function, then calls the returned function whenever it renders the component. This preserves our local state and protects it from any external code, for better or worse. Also note that any arguments referenced in the outer function will retain the value they have when it is first called. This is sometimes useful for initialization tasks, but more often than not, you should get the arguments from the inner function instead. This is *the* most common bug with Reagent, so be careful with your component arguments.

Note that we're using the r/atom from Reagent instead of using the regular ClojureScript atom. Reagent atoms behave the same way as regular atoms, with one important difference: any UI components that reference Reagent atoms will be repainted whenever the value of the atom is changed. When we want to create state, we create an atom to hold it.

This approach automates the process of keeping the UI in sync with the model. With Reagent, we're able to write our UI in a declarative fashion and have it automatically render the current state of our model. Let's see exactly how all this works by implementing the form in our guestbook application using Reagent.

Aside from that, the content of the form should look familiar, since it closely mimics the HTML we used previously. The changes to note are that we've changed the form element to a div, and in addition we've added the :on-change key to bind the input and the textarea elements to functions that are responsible for updating the fields atom with the current values entered by the user. These functions accept the DOM event object as their input and grab the value from its target.

Now, let's create a home function that uses our mesage-form component:

```
(ns guestbook.core
  ...)

(defn message-form []
  ...)

(defn home []
  [:div.content>div.columns.is-centered>div.column.is-two-thirds
    [:div.columns>div.column
      [message-form]]])
```

Note that we place the message-form in a vector instead of calling it as a function. This allows Reagent to decide when the function needs to be evaluated in case the component has to be repainted. Components can now be recomputed and repainted as the state of their corresponding atoms changes. We'll see how this becomes important shortly.

Finally, let's change our render call to use our home function:

```
(ns guestbook.core
  ...)

(defn message-form []
  ...)

(defn home []
  ...)
```

```
(dom/render
  [home]
  (.getElementById js/document "content"))
```

We should now be able to reload the page and see the form rendered there looking very much like the form we had previously. We can even type text in the fields, but we obviously can't see if it's being stored anywhere. Let's modify the form to convince ourselves that the inputs are actually writing the data to our fields atom by adding the following elements to it:

guestbook-reagent-start/src/cljs/guestbook/core.cljs

```
[:p "Name: " (:name @fields)]
[:p "Message: " (:message @fields)]
```

Now we can clearly see that whenever the value of the name or the message field changes, it's immediately reflected in the atom. Once the value of the atom changes, then the component is repainted and we see the new values displayed on the screen.

Talking to the Server

At this point we'd like to take the values of the fields and send them to the server when we click the comment button. We'll use the cljs-ajax[8] library to communicate with the server. The first thing we need to do is to add a dependency for it in our project.clj file.

guestbook-reagent/project.clj

```
[cljs-ajax "0.8.1"]
[org.clojure/clojurescript "1.10.764" :scope "provided"]
[reagent "1.0.0"]
```

Now that we've added the cljs-ajax library, we have to clean out the existing generated JavaScript and restart the ClojureScript compiler. Let's do that by running the following commands:

```
$ lein clean
$ lein cljsbuild auto
Watching for changes before compiling ClojureScript...
```

Now, let's add the following reference in our namespace declaration [ajax.core :refer [GET POST]]:

```
(ns guestbook.core
  (:require [reagent.core :as r]
            [reagent.dom  :as dom]
            [ajax.core :refer [GET POST]]))
```

8. https://github.com/JulianBirch/cljs-ajax

This provides GET and POST functions that we'll use to talk to our server. Let's write a function that submits our form and prints the response:

```
(defn send-message! [fields]
  (POST "/message"
        {:params @fields
         :handler #(.log js/console (str "response:" %))
         :error-handler #(.error js/console (str "error:" %))}))
```

This function attempts to POST to the /message route using the value of the fields atom as the params. The function uses the :handler and the :error-handler keys to handle the success and the error responses, respectively. We're just logging the response for now to see how it behaves. Let's hook this function up to our submit comment button by using the :on-click key so we can test it out:

```
[:input.button.is-primary
  {:type :submit
   :on-click #(send-message! fields)
   :value "comment"}]
```

Let's check the terminal to see that our ClojureScript recompiled successfully and then reload the browser window. When the page reloads, open up the browser console and click the comment button.

```
error:{:status 403,
       :status-text "Forbidden",
       :failure :error,
       :response "<h1>Invalid anti-forgery token</h1>"}
```

We got an error response! This error indicates that we didn't supply the anti-forgery token in our request. If you recall, the server has anti-forgery protection enabled and requires the client to submit the generated token along with any POST request. Previously, we used the {% csrf-field %} tag in our template to supply the token. Since we're now using an Ajax call instead of a form submission, we have to provide this field another way.

First, let's update our home.html template to create a hidden field with the value of the token:

guestbook-reagent/resources/html/home.html

```
{% extends "base.html" %}
{% block content %}
<input id="token" type="hidden" value="{{csrf-token}}">
<div id="content"></div>
{% endblock %}
{% block page-scripts %}
    {% script "/js/app.js" %}
{% endblock %}
```

Now we set the token as an x-csrf-token header on our request. While we're at it, let's also set the Accept header to application/transit+json:

```
(defn send-message! [fields]
  (POST "/message"
        {:format :json
         :headers
         {"Accept" "application/transit+json"
          "x-csrf-token" (.-value (.getElementById js/document "token"))}
         :params @fields
         :handler #(.log js/console (str "response:" %))
         :error-handler #(.log js/console (str "error:" %))}))
```

We set the Accept header to tell the server we'd like to get the response encoded using Transit.[9] With Transit, Clojure data structures are tagged when they're encoded. This means that we can send data structures like keywords, sets, and dates without worrying about them being converted to strings, vectors, or numbers. We can also extend Transit with our own readers and writers, but we won't get into that here.

Let's reload our page and give it another try.

We're no longer getting an error, but we're getting an HTML response instead of data. Our original code for the /message route used the redirect function to display the page after attempting to add the message. This means it's sending us the HTML for the redirect page. This time around, we want to return a Transit response indicating success or failure.

Let's update our save-message! function as follows:

guestbook-reagent/src/clj/guestbook/routes/home.clj

```
(defn save-message! [{:keys [params]}]
  (if-let [errors (validate-message params)]
    (response/bad-request {:errors errors})
    (try
      (db/save-message! params)
      (response/ok {:status :ok})
      (catch Exception e
        (response/internal-server-error
         {:errors {:server-error ["Failed to save message!"]}})))))
```

The updated function returns a 400 response when validation fails, a 200 response when it's saved successfully, and a 500 response in case of errors.

Note that we don't have to manually deserialize the request nor serialize the response in our route. This is handled by the Muuntaja library.[10] The library

9. https://github.com/cognitect/transit-format
10. https://github.com/metosin/muuntaja

checks the Content-Type header in the request and deserializes the content based on that. The response is serialized to the format specified in the Accept header. The cljs-ajax library defaults to using the Transit format. If you recall, we explicitly set the Accept header to Transit as well. No additional work is needed on your part.

Let's reload the page and try it out. When we try to POST invalid parameters, we should see a response in the console that looks similar to the following:

```
error:{:status 400,
       :status-text "Bad Request",
       :failure :error,
       :response {:errors {:message ("message is less than the minimum")}}}
```

When we submit a valid request, we should see the following printed instead:

```
response:{:status :ok}
```

Notice that in the first case our error-handler was triggered, while in the second case our success handler function is triggered. The cljs-ajax library uses the status code in the response to select the appropriate handler.

Now that we're communicating with the server, let's update our code to display the errors on the page. We'll use the clojure.string namespace's join function, so let's add it to our namespace's require statement first:

```
(ns guestbook.core
  (:require ...
            [clojure.string :as string]))
```

Then let's update our code like so:

guestbook-reagent-start/src/cljs/guestbook/core.cljs

```
(defn send-message! [fields errors]
  (POST "/message"
        {:format :json
         :headers
         {"Accept" "application/transit+json"
          "x-csrf-token" (.-value (.getElementById js/document "token"))}
         :params @fields
         :handler (fn [r]
                    (.log js/console (str "response:" r))
                    (reset! errors nil))
         :error-handler (fn [e]
                          (.log js/console (str e))
                          (reset! errors (-> e :response :errors)))}))

(defn errors-component [errors id]
  (when-let [error (id @errors)]
    [:div.notification.is-danger (string/join error)]))

(defn message-form []
  (let [fields (r/atom {})
        errors (r/atom nil)]
    (fn []
      [:div
       [:p "Name: " (:name @fields)]
       [:p "Message: " (:message @fields)]
       [errors-component errors :server-error]
       [:div.field
        [:label.label {:for :name} "Name"]
        [errors-component errors :name]
        [:input.input
         {:type :text
          :name :name
          :on-change #(swap! fields
                             assoc :name (-> % .-target .-value))
          :value (:name @fields)}]]
       [:div.field
        [:label.label {:for :message} "Message"]
        [errors-component errors :message]
        [:textarea.textarea
         {:name :message
          :value (:message @fields)
          :on-change #(swap! fields
                             assoc :message (-> % .-target .-value))}]]
       [:input.button.is-primary
        {:type :submit
         :on-click #(send-message! fields errors)
         :value "comment"}]])))
```

The updated code uses a second atom called errors to store any errors received from the server. We pass the errors to the send-message! function. The function now either clears the errors on success or sets the errors from the response.

We also created a new component called errors-component that accepts the errors and the field ID. It checks if any errors are associated with the ID and returns an alert with the message if that's the case. This component requires the clojure.string library, so remember to require [clojure.string :as string] in your namespace declaration. If the component sees no errors, we simply return a nil. Reagent will handle this intelligently and omit the component in that case. You can see that with the new approach the errors are showing up just as they did in the previous version.

Share Code with the Client

Using the same language on the client and the server allows us to share code between them. Doing so avoids duplication and reduces the chance for errors and unexpected behaviors. Also, we'll be able to run validation client-side and avoid calling the server entirely if the input data is invalid.

Let's update our project to extract validation into a namespace that will be cross-compiled to both Clojure and ClojureScript. The first step is to create a new source folder called src/cljc and update the source paths in project.clj to include it.

guestbook-cljc/project.clj

```
;;Clojure Source Paths
:source-paths ["src/clj" "src/cljc"]
;;...
;;ClojureScript Source Paths
:cljsbuild {:builds
            {:app
             {:source-paths ["src/cljs" "src/cljc"]
              :compiler «compiler options»}}}
```

With the new source folder in place, let's create a new namespace called validation in src/cljc/guestbook/validation.cljc. Note that the file extension is cljc—this hints the compiler that this file will be cross-compiled to both Clojure and ClojureScript.

Next, let's move the validation code from the guestbook.routes.home namespace to the guestbook.validation namespace:

guestbook-cljc/src/cljc/guestbook/validation.cljc

```
(ns guestbook.validation
  (:require
   [struct.core :as st]))
```

```
(def message-schema
  [[:name
    st/required
    st/string]
   [:message
    st/required
    st/string
    {:message "message must contain at least 10 characters"
     :validate (fn [msg] (>= (count msg) 10))}]])

(defn validate-message [params]
  (first (st/validate params message-schema)))
```

The updated guestbook.routes.home namespace will now require the validate-message function from the guestbook.validation namespace:

guestbook-cljc/src/clj/guestbook/routes/home.clj

```
(ns guestbook.routes.home
  (:require
   [guestbook.layout :as layout]
   [guestbook.db.core :as db]
   [clojure.java.io :as io]
   [guestbook.middleware :as middleware]
   [ring.util.response]
   [ring.util.http-response :as response]
   [guestbook.validation :refer [validate-message]]))
```

Let's restart the application to confirm that server-side code still works correctly. Now we can turn our attention to the client, where we can start using the validate-message function to ensure that the message content is valid before sending it to the server.

We require the guestbook.validation namespace the same way we did earlier:

guestbook-cljc/src/cljs/guestbook/core.cljs

```
(ns guestbook.core
  (:require [reagent.core :as r]
            [reagent.dom :as dom]
            [ajax.core :refer [GET POST]]
            [clojure.string :as string]
            [guestbook.validation :refer [validate-message]]))
```

Then we add the validation check in the send-message! function:

guestbook-cljc/src/cljs/guestbook/core.cljs

```
(defn send-message! [fields errors]
  (if-let [validation-errors (validate-message @fields)]
    (reset! errors validation-errors)
    (POST "/message"
        ;;...
        )))
```

The updated code will attempt to validate the message before making the Ajax call. If any errors are returned by the `validate-message` function, those will be set in the `errors` atom, shortcutting the need for server-side validation.

The advantage of this approach is that we're able to reduce server load by avoiding making Ajax calls with invalid data. Of course, the server will still have the final say since it's using the same validation logic.