Extracted from:

Web Development with Clojure, 2nd Edition

Build Bulletproof Web Apps with Less Code

This PDF file contains pages extracted from *Web Development with Clojure, 2nd Edition*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2016 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Web Development with Clojure

2nd Edition

Build Bulletproof Web Apps with Less Code

Dmitri Sotnikov

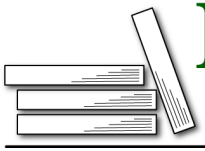
edited by Michael Swaine

Web Development with Clojure, 2nd Edition

Build Bulletproof Web Apps with Less Code

Dmitri Sotnikov

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (index)
Candace Cunningham, Molly McBeath (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2016 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-68050-082-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2016

CHAPTER 5

Real-Time Messaging with WebSockets

In this chapter we'll take a look at using WebSockets for client-server communication. In the traditional Ajax approach, the client first sends a message to the server and then handles the reply using an asynchronous callback. WebSockets provide the ability for the web server to initiate the message exchange with the client.

Currently, our guestbook application does not provide a way to display messages generated by other users without reloading the page. If we wanted to solve this problem using Ajax, our only option would be to poll the server and check if any new messages are available since the last poll. This is inefficient since the clients end up continuously polling the server regardless of whether any new messages are actually available.

Instead, we'll have the clients open a WebSocket connection when the page loads, and then the server will notify all the active clients any time a new message is created. This way the clients are notified in real time and the messages are only sent as needed.

Set Up WebSockets on the Server

WebSockets require support on both the server and the client side. While the browser API is standard, each server provides its own way of handling WebSocket connections. In this section we'll take a look at using the API for the Immutant web server that Luminus defaults to.

Let's start by updating the server-side code in the project to provide a WebSocket connection. Once the server is updated we'll look at the updates required for the client.

• Click HERE to purchase this book now. discuss

Add WebSocket Routes

The first thing we'll do is create a new namespace for handling WebSocket connections. Let's call this namespace guestbook.routes.ws and put the following references in its declaration.

guestbook-websockets/src/clj/guestbook/routes/ws.clj

```
(ns guestbook.routes.ws
  (:require [compojure.core :refer [GET defroutes]]
            [clojure.tools.logging :as log]
            [immutant.web.async :as async]
            [cognitect.transit :as transit]
            [bouncer.core :as b]
            [bouncer.validators :as v]
            [guestbook.db.core :as db]))
```

The immutant.web.async reference is the namespace that provides the functions necessary to manage the life cycle of the WebSocket connection.

The cognitect.transit namespace provides the functions to encode and decode messages using the transit format. When we used Ajax, the middleware was able to serialize and deserialize the messages automatically based on the content type; however, we'll have to do that manually for messages sent over the WebSocket.

Since we'll be saving messages, we need to reference the bouncer and the database namespaces so that we can move over the validate-message and the save-message! functions that we originally used in the guestbook.routes.home namespace.

The server needs to keep track of all the channels for the clients that are currently connected in order to push notifications. Let's use an atom containing a set for this purpose.

guestbook-websockets/src/clj/guestbook/routes/ws.clj

```
(defonce channels (atom #{}))
```

Next, we need to implement a callback function to handle the different states that the WebSocket can be in, such as when the connection is opened and closed. We want to add the channel to the set of open connections when a client connects, and we want to remove the associated channel when the client disconnects.

guestbook-websockets/src/clj/guestbook/routes/ws.clj

```
(defn connect! [channel]
  (log/info "channel open")
  (swap! channels conj channel))
```

```
(defn disconnect! [channel {:keys [code reason]}]
  (log/info "close code:" code "reason:" reason)
  (swap! channels clojure.set/difference #{channel}))
```

As mentioned earlier, the messages have to be encoded and decoded manually. Let's create a couple of helper functions for that purpose.

guestbook-websockets/src/clj/guestbook/routes/ws.clj

```
(defn encode-transit [message]
  (let [out    (java.io.ByteArrayOutputStream. 4096)
        writer (transit/writer out :json)]
    (transit/write writer message)
    (.toString out)))

(defn decode-transit [message]
  (let [in (java.io.ByteArrayInputStream. (.getBytes message))
        reader (transit/reader in :json)]
    (transit/read reader)))
```

When the client sends a message, we'll want to validate it and attempt to save the message, as we did earlier. Let's take the save-message! and the validate-message function from the guestbook.routes.home and move them over to the new namespace. The save-message! function no longer needs to generate a Ring response, so we have it return the result directly instead.

guestbook-websockets/src/clj/guestbook/routes/ws.clj

```
(defn validate-message [params]
  (first
    (b/validate
      params
      :name v/required
      :message [v/required [v/min-count 10]])))

(defn save-message! [message]
  (if-let [errors (validate-message message)]
    {:errors errors}
    (do
      (db/save-message! message)
      message)))
```

Finally, we create the handle-message! function that will be called when the client sends a message to the server. When the message is saved successfully, we notify all the connected clients; when any errors occur we notify only the client that sent the original message.

guestbook-websockets/src/clj/guestbook/routes/ws.clj

```
(defn handle-message! [channel message]
  (let [response (-> message
                     decode-transit
```

```
                    (assoc :timestamp (java.util.Date.))
                    save-message!)]
    (if (:errors response)
      (async/send! channel (encode-transit response))
      (doseq [channel @channels]
        (async/send! channel (encode-transit response))))))
```

The function accepts the channel of the client that sent the message along with the message payload. The message has to be decoded using the decode-transit function that we wrote earlier. The result should be a map with the same keys as before. Let's associate the timestamp and attempt to save the message to the database using the save-message! function.

When the response map contains the :error key, we notify the client on the channel that was passed in; otherwise we notify all clients in the channels atom. The response is sent using the async/send! call that accepts the channel and the message as a string, so we have to call encode-transit on the response before it's passed to async/send!.

Now that we've implemented all the callbacks, let's put these in a map and pass it to the async/as-channel function that will create the actual WebSocket channel. This is done in the ws-handler function that follows.

guestbook-websockets/src/clj/guestbook/routes/ws.clj

```
(defn ws-handler [request]
  (async/as-channel
    request
    {:on-open    connect!
     :on-close   disconnect!
     :on-message handle-message!}))
```

All that's left to do is create the route definition using the defroutes macro, just as we would with any other Compojure routes.

guestbook-websockets/src/clj/guestbook/routes/ws.clj

```
(defroutes websocket-routes
           (GET "/ws" [] ws-handler))
```

Note that we could define multiple WebSockets and assign them to different routes. In our case we have just a single /ws route for our socket.

Now that we've migrated the code for saving messages to the guestbook.routes.ws namespace, we can clean up the guestbook.routes.home namespace as follows.

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

```
(ns guestbook.routes.home
  (:require [guestbook.layout :as layout]
            [guestbook.db.core :as db]
```

```
            [bouncer.core :as b]
            [bouncer.validators :as v]
            [compojure.core :refer [defroutes GET POST]]
            [ring.util.response :refer [response status]]))

(defn home-page []
  (layout/render "home.html"))

(defn about-page []
  (layout/render "about.html"))

(defroutes home-routes
  (GET "/" [] (home-page))
  (GET "/messages" [] (response (db/get-messages)))
  (GET "/about" [] (about-page)))
```

Update the Handler

Now that we've added the new routes, we need to navigate to the guestbook.handler namespace, reference the new namespace, and add the routes to the app-routes definition.

```
(ns guestbook.handler
  (:require ...
            [guestbook.routes.ws :refer [websocket-routes]]))
```

guestbook-websockets/src/clj/guestbook/handler.clj

```
(def app-routes
  (routes
    #'websocket-routes
    (wrap-routes #'home-routes middleware/wrap-csrf)
    (route/not-found
      (:body
        (error-page {:status 404
                     :title "page not found"}))))))
(def app (middleware/wrap-base #'app-routes))
```

We're now done with all the necessary server-side changes to facilitate WebSocket connections. Let's turn our attention to the client.

Make WebSockets from ClojureScript

Now that we've created a WebSocket route on the server, we need to write the client-side portion of the socket. Once that's done we'll have full-duplex communication between the server and the client.

Create the WebSocket

Let's start by creating a namespace called guestbook.ws for the WebSocket client. This namespace will be responsible for creating a socket as well as for sending

and receiving messages over it. In the namespace declaration, let's add a reference to cognitect.transit.

guestbook-websockets/src/cljs/guestbook/ws.cljs

```
(ns guestbook.ws
  (:require [cognitect.transit :as t]))
```

Next, let's create an atom to house the channel for the socket and add helpers for reading and writing transit-encoded messages.

guestbook-websockets/src/cljs/guestbook/ws.cljs

```
(defonce ws-chan (atom nil))
(def json-reader (t/reader :json))
(def json-writer (t/writer :json))
```

We can now add functions to receive and send transit-encoded messages using the channel. The receive-message! function is a closure that accepts a handler function and returns a function that deserializes the message before passing it to the handler.

guestbook-websockets/src/cljs/guestbook/ws.cljs

```
(defn receive-message! [handler]
  (fn [msg]
    (->> msg .-data (t/read json-reader) handler)))
```

The send-message! function checks if there's a channel available and then encodes the message to transit and sends it over the channel.

guestbook-websockets/src/cljs/guestbook/ws.cljs

```
(defn send-message! [msg]
  (if @ws-chan
    (->> msg (t/write json-writer) (.send @ws-chan))
    (throw (js/Error. "WebSocket is not available!"))))
```

Finally, let's write a function to initialize the WebSocket. The function calls js/WebSocket with the supplied URL to create the channel. Once the channel is created, it sets the onmessage callback to the supplied handler function and puts the channel in the ws-chan atom.

guestbook-websockets/src/cljs/guestbook/ws.cljs

```
(defn connect! [url receive-handler]
  (if-let [chan (js/WebSocket. url)]
    (do
      (set! (.-onmessage chan) (receive-message! receive-handler))
      (reset! ws-chan chan))
    (throw (js/Error. "WebSocket connection failed!"))))
```

As you can see, setting up a basic WebSocket connection is no more difficult than using Ajax. The main differences are that we have to manually handle serialization and that the messages received by the receive-message! function are not directly associated with the ones sent by the send-message! function.

Let's navigate back to the guestbook.core to use a WebSocket connection to communicate with the server instead of Ajax for saving and receiving messages. Noting that WebSockets and Ajax are not mutually exclusive, and we can continue using the existing Ajax call to retrieve the initial list of messages.

First, let's update the namespace declaration to remove the unused POST reference and add the guestbook.ws that we just wrote.

guestbook-websockets/src/cljs/guestbook/core.cljs

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

The functions message-list, get-messages, and errors-component remain unchanged. However, we no longer need the old send-message! function, because the messages are sent using the send-message! function from the guestbook.ws namespace.

guestbook-websockets/src/cljs/guestbook/core.cljs

```
(defn message-list [messages]
  [:ul.content
   (for [{:keys [timestamp message name]} @messages]
     ^{:key timestamp}
     [:li
      [:time (.toLocaleString timestamp)]
      [:p message]
      [:p " - " name]])])

(defn get-messages [messages]
  (GET "/messages"
       {:headers {"Accept" "application/transit+json"}
        :handler #(reset! messages (vec %))}))

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

The message-form function no longer needs to update the message list; it gets updated by the callback that we use to initialize the WebSocket. Conversely, we can no longer set the values of the fields and the errors, so the atoms that hold these values are passed in instead.

The form sets the values in the fields atom and displays the currently populated values in the errors atom. The comment button now sends the current value of the fields atom to the server by calling ws/send-message!.

guestbook-websockets/src/cljs/guestbook/core.cljs

```
(defn message-form [fields errors]
  [:div.content
   [:div.form-group
    [errors-component errors :name]
    [:p "Name:"
     [:input.form-control
      {:type      :text
       :on-change #(swap! fields assoc :name (-> % .-target .-value))
       :value     (:name @fields)}]]
    [errors-component errors :message]
    [:p "Message:"
     [:textarea.form-control
      {:rows      4
       :cols      50
       :value     (:message @fields)
       :on-change #(swap! fields assoc :message (-> % .-target .-value))}]]
    [:input.btn.btn-primary
     {:type     :submit
      :on-click #(ws/send-message! @fields)
      :value    "comment"}]]])
```

Now let's add the response-handler function that receives the messages from the server and sets the values of the messages, the fields, and the errors atoms accordingly. Specifically, if the :errors key is present in the response, then the errors atom is set with its value. Otherwise, the errors and fields are cleared and the response is added to the list of messages.

guestbook-websockets/src/cljs/guestbook/core.cljs

```
(defn response-handler [messages fields errors]
  (fn [message]
    (if-let [response-errors (:errors message)]
      (reset! errors response-errors)
      (do
        (reset! errors nil)
        (reset! fields nil)
        (swap! messages conj message)))))
```

The home function now initializes all the atoms and then passes these to the response-handler, which in turn is passed to the ws/connect! function and used to handle responses. The URL for the WebSocket is composed of the ws:// protocol definition, the host of origin, and the /ws route that we defined earlier.

Next, the function calls get-messages to load the messages currently available on the server and return a component that is used to render the page. This function remains largely unchanged aside from the fact that it passes the updated arguments to the message-form component.

guestbook-websockets/src/cljs/guestbook/core.cljs

```
(defn home []
  (let [messages (atom nil)
        errors   (atom nil)
        fields   (atom nil)]
    (ws/connect! (str "ws://" (.-host js/location) "/ws")
                 (response-handler messages fields errors))
    (get-messages messages)
    (fn []
      [:div
       [:div.row
        [:div.span12
         [message-list messages]]]
       [:div.row
        [:div.span12
         [message-form fields errors]]]])))
```

With these changes implemented we should be able to test that our app behaves as expected by running it as we did previously:

```
lein cljsbuild once
lein run
```

Everything should look exactly the same as it did before; however, we're not done yet. Now that we're using WebSockets, we should be able to open a second browser and add a message from there. The message will now show up in both browsers as soon as it's processed by the server!