Extracted from:

Web Development with Clojure, Third Edition

Build Large, Maintainable Web Applications Interactively

This PDF file contains pages extracted from *Web Development with Clojure, Third 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 © 2021 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.

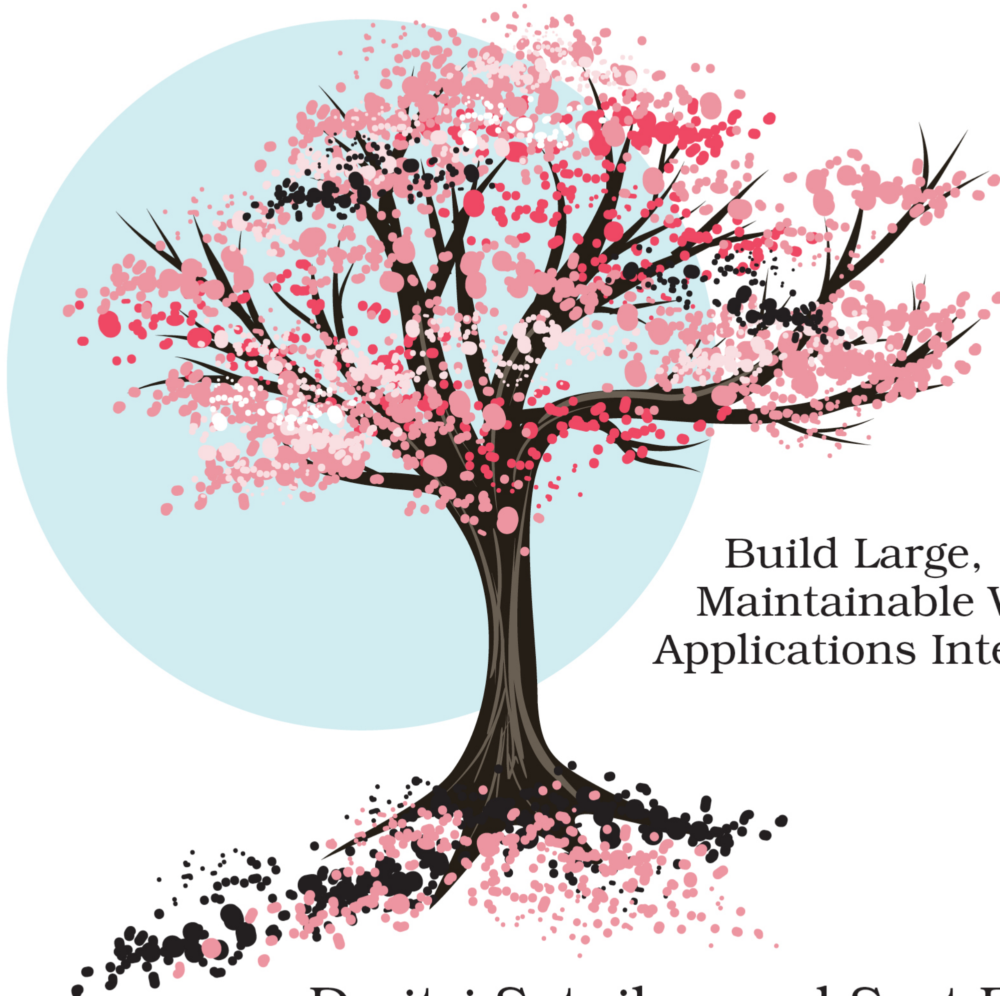
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Web Development with Clojure

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

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.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

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

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

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

Copyright © 2021 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.

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

Multi-User with WebSockets

Instead of just having a refresh messages button, let's add a push notification for new messages using WebSockets.

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 allow the web server to initiate the message exchange with the client.

Currently, our guestbook application doesn't 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 anytime a new message is created. This way the clients are notified in real time and the messages are only sent as needed.

Configuring 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 HTTP Kit 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.

First we add a new namespace for our WebSocket connection handler. Let's call it `guestbook.routes.websockets` and add the following requirements:

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

```
(ns guestbook.routes.websockets
  (:require [clojure.tools.logging :as log]
            [org.httpkit.server :as http-kit]
            [clojure.edn :as edn]
            [guestbook.messages :as msg]))
```

We'll need `org.httpkit.server` for managing our WebSocket connections.

While we figure out the basics of WebSockets, we'll just use `pr-str` and `edn/read-string` to serialize and de-serialize our data.

If we want to communicate to clients, the first thing we need to do is keep track of our connections. Let's create an atom containing our open connections and write connect! and disconnect! functions to manage it.

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

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

(defn connect! [channel]
  (log/info "Channel opened")
  (swap! channels conj channel))

(defn disconnect! [channel status]
  (log/info "Channel closed: " status)
  (swap! channels disj channel))
```

To keep it simple, let's assume our WebSocket will only receive save-message! messages. Let's copy our logic from guestbook.routes.services but replace HTTP responses with maps and add serialization and de-serialization where necessary.

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

```
(defn handle-message! [channel ws-message]
  (let [message (edn/read-string ws-message)
        response (try
                   (msg/save-message! message)
                   (assoc message :timestamp (java.util.Date.))
                   (catch Exception e
                     (let [{id     :guestbook/error-id
                            errors :errors} (ex-data e)]
                       (case id
                         :validation
                         {:errors errors}
                         ;;else
                         {:errors
                          {:server-error ["Failed to save message!"]}}))))]
    (if (:errors response)
      (http-kit/send! channel (pr-str response))
      (doseq [channel @channels]
        (http-kit/send! channel (pr-str response))))))
```

Finally, we need to write a connection handler:

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

```
(defn handler [request]
  (http-kit/with-channel request channel
    (connect! channel)
    (http-kit/on-close channel (partial disconnect! channel))
    (http-kit/on-receive channel (partial handle-message! channel))))

(defn websocket-routes []
  ["/ws"
   {:get handler}])
```

We reference it from guestbook.handler, exposing a route in our app for accepting connections:

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

(mount/defstate app
  :start
  (middleware/wrap-base
    (ring/ring-handler
      (ring/router
        [;;...
        (websocket-routes)])
      ;;...
      )))
```

Now, let's connect from the client.

Connecting from ClojureScript

To connect on the client side, we use js/WebSocket to create a WebSocket connection object, which we use to communicate.

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

```
(ns guestbook.websockets
  (:require [cljs.reader :as edn]))

(defonce channel (atom nil))

(defn connect! [url receive-handler]
  (if-let [chan (js/WebSocket. url)]
    (do
      (.log js/console "Connected!")
      (set! (.-onmessage chan) #(->> %
                                     .-data
                                     edn/read-string
                                     receive-handler))
      (reset! channel chan))
    (throw (ex-info "Websocket Connection Failed!"
                    {:url url}))))

(defn send-message! [msg]
  (if-let [chan @channel]
    (.send chan (pr-str msg))
    (throw (ex-info "Couldn't send message, channel isn't open!"
                    {:message msg}))))
```

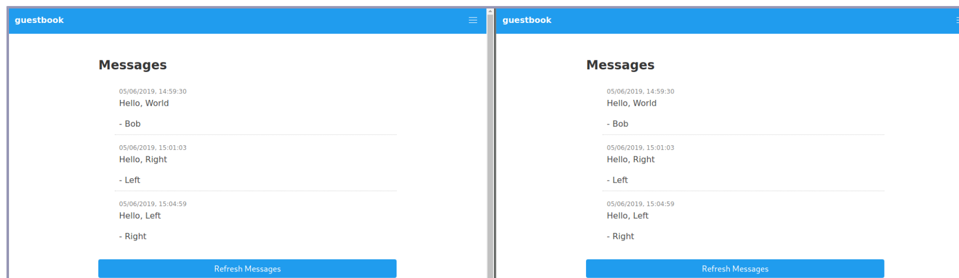
In our core namespace, everything stays mostly the same. The only things we need to do are: update our :message/send event to use our WebSocket, add a handle-response! function that will deal with responses, and call ws/connect! from our init! function.

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

```
(rf/reg-event-fx
 :message/send!
 (fn [{:keys [db]} [_ fields]]
   (ws/send-message! fields)
   {:db (dissoc db :form/server-errors)}))

(defn handle-response! [response]
  (if-let [errors (:errors response)]
    (rf/dispatch [:form/set-server-errors errors])
    (do
      (rf/dispatch [:message/add response])
      (rf/dispatch [:form/clear-fields response]))))
(defn init! []
  (.log js/console "Initializing App...")
  (rf/dispatch [:app/initialize])
  (ws/connect! (str "ws://" (.-host js/location) "/ws")
               handle-response!)
  (mount-components))
```

Let's try it out in the browser. Let's respond to our earlier message via Web-Sockets.



It automatically loads! Since re-frame is event driven, using WebSockets is actually a lot simpler! We just dispatch our events from our WebSocket message handler instead of from our Reagent components. If we look at re-frame-10x's history in the left browser, we can see that [:message/add ...] was indeed triggered even though we didn't submit anything.

From here, we could do a lot to improve our re-frame app, but let's swap out our from-scratch WebSocket implementation for a much richer one—Sente.

Upgrading to Sente

Now that we've done a scratch implementation of WebSockets, let's look at a popular Clojure(Script) WebSockets library: Sente.[15]

Sente is a little more complicated than our toy implementation, but it brings a lot of great features in exchange.

Ajax Fallback Support
: Sente automatically switches to Ajax polling if WebSockets aren't available.

Keep-Alives
: Sends messages periodically to prevent connections from dropping and to kill stale connections.

Message Buffering
: Leverages `core.async` to buffer messages for us.

Encoding
: Serializes and de-serializes data for us.

Security
: Supports Ring anti-forgery middleware.

Let's add the dependency to our `project.clj` and get started:

guestbook-sente-setup/project.clj
```
[com.taoensso/sente "1.16.0"]
```

Upgrading the Server

We can now update the `guestbook.routes.websockets` namespace to use Sente to manage the server-side WebSocket connection. Let's update the dependencies to add `taoensso.sente` and `taoensso.sente.server-adapters.http-kit` references. Also, since

15. https://github.com/ptaoussanis/sente

Sente manages the serialization of data and the management of our connections, let's remove clojure.edn and org.httpkit.server.

guestbook-sente-setup/src/clj/guestbook/routes/websockets.clj

```
(ns guestbook.routes.websockets
  (:require
   [clojure.tools.logging :as log]
   [guestbook.messages :as msg]
   [guestbook.middleware :as middleware]
   [mount.core :refer [defstate]]
   [taoensso.sente :as sente]
   [taoensso.sente.server-adapters.http-kit :refer [get-sch-adapter]]))
```

We initialize Sente by calling the sente/make-channel-socket! function. This function accepts the server adapter and a map of initialization options. We pass in the HTTP Kit server adapter, since that's the server we're using, and we set the :user-id-fn option to use the :client-id key in the request parameters. The reason we have to specify our :user-id-fn is that Sente defaults to using the :uid key from the session. Since we aren't creating Ring sessions for our clients, we'll need to use something else. The :client-id is a UUID that's automatically generated for each Sente client, so it's a perfect fit for us.

guestbook-sente-setup/src/clj/guestbook/routes/websockets.clj

```
(defstate socket
  :start (sente/make-channel-socket!
          (get-sch-adapter)
          {:user-id-fn (fn [ring-req]
                         (get-in ring-req [:params :client-id]))}))

(defn send! [uid message]
  (println "Sending message: " message)
  ((:send-fn socket) uid message))
```

The sente/make-channel-socket! function returns a map that contains a number of variables that were initialized.

:ajax-post-fn
: The function that handles Ajax POST requests.

:ajax-get-or-ws-handshake-fn
: The function that negotiates the initial connection.

:ch-recv
: The receive channel for the socket.

:send-fn
: The function that's used to send push notifications to the client.

:connected-uids
An atom containing the IDs of the connected clients.

We'll usually access the keys on our socket map using helper functions such as send!.

In our first implementation, we sent our guestbook message as a map of fields with no metadata. This restricted us to only one type of message. We should change this so that we can accept multiple message types. We also *must* change this because Sente will call our handler function whenever an event occurs, passing it a map with a bunch of metadata describing the event.

Let's do this with a *multimethod*:

guestbook-sente-setup/src/clj/guestbook/routes/websockets.clj
```
(defmulti handle-message (fn [{:keys [id]}]
                           id))

(defmethod handle-message :default
  [{:keys [id]}]
  (log/debug "Received unrecognized websocket event type: " id))

(defmethod handle-message :message/create!
  [{:keys [?data uid] :as message}]
  (let [response (try
                   (msg/save-message! ?data)
                   (assoc ?data :timestamp (java.util.Date.))
                   (catch Exception e
                     (let [{id     :guestbook/error-id
                            errors :errors} (ex-data e)]
                       (case id
                         :validation
                         {:errors errors}
                         ;;else
                         {:errors
                          {:server-error ["Failed to save message!"]}}))))]
    (if (:errors response)
      (send! uid [:message/creation-errors response])
      (doseq [uid (:any @(:connected-uids socket))]
        (send! uid [:message/add response])))))

(defn receive-message! [{:keys [id] :as message}]
  (log/debug "Got message with id: " id)
  (handle-message message))
```

We've replaced our old handle-message! function with a receive-message! function and a handle-message multimethod. We place any logic that applies to *all* events in our receive-message! wrapper function. It will call handle-message, which will dispatch to different methods based on the :id of the message.

Sente's event-message maps have some useful keys on them besides just :id.

event
: The full event vector.

id
: The ID keyword (first event).

?data
: The data sent in the event (second event).

send-fn
: A function to send a message via the socket this message was received from.

?reply-fn (Server Only)
: Sends an arbitrary response body to the callback function specified client-side (only exists if the client specified a callback function).

uid (Server Only)
: A user-id (that is, may correspond to one or many connections. Is managed based on :user-id-fn; compare :client-id).

ring-req (Server Only)
: The Ring request received in an Ajax post or the initial WebSocket handshake.

client-id (Server Only)
: A client-id that is specific to a single connection.

Since send! only communicates with a single user, we must use a doseq if we want to broadcast messages. Now that we've got the meat of it set up, let's update how it connects to the rest of our application.

guestbook-sente-setup/src/clj/guestbook/routes/websockets.clj

```
(defstate channel-router
  :start (sente/start-chsk-router!
          (:ch-recv socket)
          #'receive-message!)
  :stop (when-let [stop-fn channel-router]
          (stop-fn)))

(defn websocket-routes []
  ["/ws"
   {:middleware [middleware/wrap-csrf
                 middleware/wrap-formats]
    :get (:ajax-get-or-ws-handshake-fn socket)
    :post (:ajax-post-fn socket)}])
```

In addition to managing our socket, Sente provides a helper for creating a message router that passes incoming messages to our handler function. Since

our router depends on the initialization of our socket, we have to define it as a defstate so that Mount knows to start our socket first before starting our router. Once we have our router initialized, the last step is to connect the socket to our web server. Unlike our from-scratch implementation Sente works well with Ring middlewares, so we use wrap-csrf and wrap-formats. It also has both :get and :post handler functions so that the client can use Ajax if it doesn't support WebSockets.

Upgrading the Client

Now that we've changed our server to use Sente, we need to update the client as well. Our client WebSocket connection will look similar to our server, with a few key differences. We'll use Mount on the client, as well, so we won't have to manually connect in our init! function. We'll have two defstate definitions: our socket and our router. We'll have a receive-message! function wrapping a handle-message multimethod. And we'll have a send! function for sending messages over our socket.

Let's start by updating our namespace declaration and creating our socket and send! function:

guestbook-sente-setup/src/cljs/guestbook/websockets.cljs

```
(ns guestbook.websockets
  (:require-macros [mount.core :refer [defstate]])
  (:require [re-frame.core :as rf]
            [taoensso.sente :as sente]
            mount.core))

(defstate socket
  :start (sente/make-channel-socket!
          "/ws"
          (.-value (.getElementById js/document "token"))
          {:type :auto
           :wrap-recv-evs? false}))

(defn send! [message]
  (if-let [send-fn (:send-fn @socket)]
    (send-fn message)
    (throw (ex-info "Couldn't send message, channel isn't open!"
                    {:message message}))))
```

Our call to make-channel-socket! looks a bit different. As the first argument, the web server adapter is replaced by a URL. The second argument, our CSRF token, is new. Since we're in the browser now, we need to send our CSRF token to ensure that our connection is secure. The last argument is an options map like we had on the server, but it has a different set of options available. The only ones we're passing for now are :type and wrap-recv-evs?. The :type option

determines whether we use WebSockets or Ajax as the underlying connection method. Choose :auto to let Sente use whichever method it prefers. The wrap-recv-evs? option specifies whether we want to receive all application messages wrapped in an outer :chsk/recv event. Turn this off by passing false so that our client events are structured like our server events.

Our send! function looks similar to the server, but not quite the same. We're dereferencing our socket before using its :send-fn. This is a minor detail of how Mount works when targeting JavaScript rather than Java. This isn't for any interesting reason, so we'll just have to remember that we need to dereference any Mount states before using them in cljs.

Now that we have our socket set up, let's write our handle-message and receive-message! functions and connect them to a channel-router.

• Click HERE to purchase this book now. discuss

guestbook-sente-setup/src/cljs/guestbook/websockets.cljs

```
(defmulti handle-message
  (fn [{:keys [id]} _]
    id))

(defmethod handle-message :message/add
  [_ msg-add-event]
  (rf/dispatch msg-add-event))


(defmethod handle-message :message/creation-errors
  [_ [_ response]]
  (rf/dispatch
   [:form/set-server-errors (:errors response)]))

;; ---------------------------------------------------------------------------
;; Default Handlers

(defmethod handle-message :chsk/handshake
  [{:keys [event]} _]
  (.log js/console "Connection Established: " (pr-str event)))

(defmethod handle-message :chsk/state
  [{:keys [event]} _]
  (.log js/console "State Changed: " (pr-str event)))


(defmethod handle-message :default
  [{:keys [event]} _]
  (.warn js/console "Unknown websocket message: " (pr-str event)))

;; ---------------------------------------------------------------------------
;; Router

(defn receive-message!
  [{:keys [id event] :as ws-message}]
  (do
    (.log js/console "Event Received: " (pr-str event))
    (handle-message ws-message event)))

(defstate channel-router
  :start (sente/start-chsk-router!
          (:ch-recv @socket)
          #'receive-message!)
  :stop (when-let [stop-fn @channel-router]
          (stop-fn)))
```

Our handle-message function is structured similarly to the one on our server, but instead of interacting with the database, it dispatches re-frame events based on the message received.

We also have to handle a couple Sente-specific events. The :chsk/handshake and :chsk/state events are related to the status of our connection. While they're important for notifying users about lost or spotty connections, we'll just log them in the console for now.

Now that we've sorted out our guestbook.websockets namespace, let's update guestbook.core to use it correctly so that we can get back to coding interactively.

guestbook-sente-setup/src/cljs/guestbook/core.cljs

```
(ns guestbook.core
  (:require
   ;;...
   [mount.core :as mount]))
;;...
(rf/reg-event-fx
 :message/send!
 (fn [{:keys [db]} [_ fields]]
   (ws/send! [:message/create! fields])
   {:db (dissoc db :form/server-errors)}))
;;...
(defn init! []
  (.log js/console "Initializing App...")
  (mount/start)
  (rf/dispatch [:app/initialize])
  (mount-components))
```

We required mount.core, called mount/start from our init! function and changed the value we sent from ws/send! in our :message/send! event. We finally have our app functionally migrated from our from-scratch WebSockets implementation over to Sente. Since we required a new library, we need to restart our app to load the new dependency. Once that's done, let's try it out. You might've noticed that our fields don't clear when we submit our form. That was intentionally left out because it's well suited to the callback feature of Sente.

Leveraging Sente Callbacks

Sente is primarily a WebSockets library, but it allows you to get the best of both the Ajax and WebSocket workflows. Generally speaking, having the server *push* the results of an action to all concerned parties is incredibly powerful, as it separates the concerns of keeping state synchronized from the logistics of client-server communication. But this starts to break down when we want information *about* our actions. In this case, the request/response model of Ajax is a better fit. Sente allows you to mix the two by specifying a callback function when you send a message from a client.

We have two behaviors that are related to the state of our actions: clearing fields after a message is successfully sent and displaying server errors after a message is rejected.

Let's update our code to use a reply function to accomplish these tasks.

First, we need to allow our client-side send! function to take multiple arguments:

guestbook-sente-cb/src/cljs/guestbook/websockets.cljs

```
(defn send! [& args]
  (if-let [send-fn (:send-fn @socket)]
    (apply send-fn args)
    (throw (ex-info "Couldn't send message, channel isn't open!"
                    {:message (first args)}))))
```

Next, we need to require [guestbook.websockets :as ws] and pass our timeout and our callback function from :message/send!:

guestbook-sente-cb/src/cljs/guestbook/core.cljs

```
(rf/reg-event-fx
 :message/send!
 (fn [{:keys [db]} [_ fields]]
   (ws/send!
    [:message/create! fields]
    10000
    (fn [{:keys [success errors] :as response}]
      (.log js/console "Called Back: " (pr-str response))
      (if success
        (rf/dispatch [:form/clear-fields])
        (rf/dispatch [:form/set-server-errors errors]))))
   {:db (dissoc db :form/server-errors)}))
```

Finally, we need to update our server to invoke the :?reply-fn on the message:

guestbook-sente-cb/src/clj/guestbook/routes/websockets.clj

```
(defmulti handle-message (fn [{:keys [id]}]
                           id))

(defmethod handle-message :default
  [{:keys [id]}]
  (log/debug "Received unrecognized websocket event type: " id)
  {:error (str "Unrecognized websocket event type: " (pr-str id))
   :id    id})

(defmethod handle-message :message/create!
  [{:keys [?data uid] :as message}]
  (let [response (try
                   (msg/save-message! ?data)
                   (assoc ?data :timestamp (java.util.Date.))
                   (catch Exception e
                     (let [{id     :guestbook/error-id
                            errors :errors} (ex-data e)]
                       (case id
                         :validation
                         {:errors errors}
                         ;;else
                         {:errors
                          {:server-error ["Failed to save message!"]}}))))]
```

```
    (if (:errors response)
      (do
        (log/debug "Failed to save message: " ?data)
        response)
      (do
        (doseq [uid (:any @(:connected-uids socket))]
          (send! uid [:message/add response]))
        {:success true}))))

(defn receive-message! [{:keys [id ?reply-fn]
                         :as   message}]
  (log/debug "Got message with id: " id)
  (let [reply-fn (or ?reply-fn (fn [_]))]
    (when-some [response (handle-message message)]
      (reply-fn response))))
```

There we go, that's much better. Our handle-message multimethod now returns a map that gets passed to the connection's reply-fn by receive-message! if it exists. Not only does this change handle the clearing of our fields, but it handles a bug we might've encountered later. We were sending the message errors to the :uid of the sender, which is correct as long as a :uid is exactly one connected client. This is currently the case, but when we add user accounts it won't be. A connection's :uid corresponds to *all* connections belonging to that user. This means that if a user were to submit a message from their desktop, errors would display in every one of their active connections (for example, their phone, their five other browser tabs,…). It would be even worse if we'd implemented clearing fields on success in the same way—users could lose long drafted messages in other windows on other devices! Luckily, using :?reply-fn is an easy and elegant way to handle *client*-specific concerns.

• Click HERE to purchase this book now. discuss