Websockets

Contents

This section will cover an example of using websockets for client/server communication with Immutant and HTTP Kit servers in Luminus. As the first step, create a new application. Our example uses Reagent components, so we'll include the `+cljs` profile.

To create an Immutant based application use the default profile:

```
lein new luminus multi-client-ws +cljs
```

To create an HTTP Kit based application use the `+http-kit` profile instead:

```
lein new luminus multi-client-ws +cljs +http-kit
```

Once the application is created we'll need to startup the server and Figwheel. To do that, we'll need to run the following commands in separate terminals.

```
lein run
```

```
lein figwheel
```

The Server

Immutant

Build Tool: lein ▾

Topics

Your First Application
REPL Driven Developme
Application Profiles
HTML Templating
Static Assets
ClojureScript
Routing
RESTful Services
Request types
Response types
- **Websockets**

Middleware
Sessions and Cookies
Input Validation
Security
Component Lifecycle
Database Access
Database Migrations
Logging
Internationalization
Testing
Server Tuning
Environment Variables
Deployment
Useful Libraries
Sample Applications
Upgrading
Clojure Resources

Books

Let's create a new namespace called `multi-client-ws.routes.websockets` and add the following references there:

```
(ns multi-client-ws.routes.websockets
  (:require [compojure.core :refer [GET defroutes wrap-routes]]
            [clojure.tools.logging :as log]
            [immutant.web.async       :as async]))
```

Next, we'll create the `websocket-callbacks` map that will specify the functions that should be triggered during different websocket events:

```
(def websocket-callbacks
  "WebSocket callback functions"
  {:on-open connect!
   :on-close disconnect!
   :on-message notify-clients!})
```

We'll create an websocket handler function and Compojure route for our websocket route:

```
(defn ws-handler [request]
  (async/as-channel request websocket-callbacks))

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

We'll now create an atom to store the channels and define the `connect!` function that will be called any time a new client connects. The function will add the channel to the set of open channels.

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

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

The function will log that a new channel was opened and add the channel to the set of open channels defined above.

When the client disconnects the `disconnect!` function will be called. This function accepts the channel along with a map conatining the `code` and the `reason` keys. It will log that the client disconnected and remove the channel from the set of open channels.

```
(defn disconnect! [channel {:keys [code reason]}]
  (log/info "close code:" code "reason:" reason)
  (swap! channels #(remove #{channel} %)))
```

Finally, we have the `notify-clients!` function that's called any time a client message is received. This function will broadcast the message to all the connected clients.

```
(defn notify-clients! [channel msg]
  (doseq [channel @channels]
    (async/send! channel msg)))
```

That's all we need to do to manage the lifecycle of the websocket connections and to handle client communication.

HTTP KIT

Let's create a new namespace called `multi-client-ws.routes.websockets` and add the following references there:

```
(ns multi-client-ws.routes.websockets
 (:require [compojure.core :refer [GET defroutes]]
           [org.httpkit.server
            :refer [send! with-channel on-close on-receive]]
           [cognitect.transit :as t]
           [clojure.tools.logging :as log]))
```

Next, we'll create a Compojure route for our websocket handler:

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

Where the `ws-handler` function will look as follows:

```
(defn ws-handler [request]
 (with-channel request channel
               (connect! channel)
               (on-close channel (partial disconnect! channel))
               (on-receive channel #(notify-clients %))))
```

The function accepts the request and passes it to the `org.httpkit.server/with-channel` macro provided by the HTTP Kit API. The macro creates a handler that accepts the request as its argument and binds the value of the `:async-channel` key to the second paramets representing the name of the channel. The statement following the channel name will be called once when the channel is created. In our case we'll call the `connect!` function defined below any time a new client connects:

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

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

The function will log that a new channel was opened and add the channel to the set of open channels defined above.

When the client disconnects the `on-close` function will be called. This function accepts the channel along with a handler. The handler should accept the channel and the disconnect status. Our handler will log that the channel disconnected and remove it from the set of open channels.

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

```
  (swap! channels #(remove #{channel} %)))
```

Finally, we have the `on-receive` function that's called any time a client message is received. We'll pass it the `notify-clients` function as the handler. This function will broadcast the message to all the connected clients.

```
(defn notify-clients [msg]
 (doseq [channel @channels]
     (send! channel msg)))
```

That's all we need to do to manage the lifecycle of the websocket connections and to handle client communication.

Adding the routes to the handler

Next, We'll need to add the routes in our `multi-client-ws.handler` namespace:

```
(ns multi-client-ws.handler
 (:require
   ...
   [multi-client-ws.routes.websockets :refer [websocket-routes]]))

(def app
 (-> (routes
       websocket-routes
       (wrap-routes home-routes middleware/wrap-csrf)
       base-routes)
     middleware/wrap-base))
```

The Client

We'll start by creating a `multi-client-ws.websockets` in the `src/cljs/multi_client_ws` folder. The namespace will require the transit library:

```
(ns multi-client-ws.websockets
 (:require [cognitect.transit :as t]))
```

Next, we'll define an atom to hold our websocket channel and a couple of helpers for reading and writing the JSON encoded transit messages.

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

We'll now create a function to handle received messages. The function will accept the callback handler and return a function that decodes the transit message and passes it to the handler:

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

We'll also create a function that sends messages to the socket if it's open:

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

Finally, we'll add a function to create a new websocket given the URL and the received message handler:

```
(defn make-websocket! [url receive-handler]
 (println "attempting to connect websocket")
 (if-let [chan (js/WebSocket. url)]
   (do
     (set! (.-onmessage chan) (receive-transit-msg! receive-handler))
     (reset! ws-chan chan)
     (println "Websocket connection established with: " url))
   (throw (js/Error. "Websocket connection failed!"))))
```

The UI

We'll now navigate to the `multi-client-ws.core` namespace and remove the code that's already there. We'll set the `ns` definition to the following:

```
(ns multi-client-ws.core
 (:require [reagent.core :as reagent :refer [atom]]
           [multi-client-ws.websockets :as ws]))
```

Next, we'll create an atom to keep a list of messages and a Reagent component that renders it:

```
(defonce messages (atom []))

(defn message-list []
 [:ul
  (for [[i message] (map-indexed vector @messages)]
    ^{:key i}
    [:li message])])
```

We'll now create a `message-input` component that will allow us to type in a message and send it to the server. This component creates a local atom to keep track of the message being typed in and sends the message to the server when the `enter` key is pressed.

```
(defn message-input []
 (let [value (atom nil)]
   (fn []
```

```
      [:input.form-control
       {:type :text
        :placeholder "type in a message and press enter"
        :value @value
        :on-change #(reset! value (-> % .-target .-value))
        :on-key-down
        #(when (= (.-keyCode %) 13)
           (ws/send-transit-msg!
            {:message @value})
           (reset! value nil))}])))
```

We can now create the `home-page` component that looks as follows:

```
(defn home-page []
 [:div.container
  [:div.row
   [:div.col-md-12
    [:h2 "Welcome to chat"]]]
  [:div.row
   [:div.col-sm-6
    [message-list]]]
  [:div.row
   [:div.col-sm-6
    [message-input]]]])
```

We'll also create an `update-messages!` function that will be used as the handler for the received messages. This function will append the new message and keep a buffer of 10 last received messages.

```
(defn update-messages! [{:keys [message]}]
  (swap! messages #(vec (take 10 (conj % message)))))
```

All that's left to do is mount the `home-page` component and create the websocket in the `init!` function:

```
(defn mount-components []
 (reagent/render-component [#'home-page] (.getElementById js/document

(defn init! []
 (ws/make-websocket! (str "ws://" (.-host js/location) "/ws") update-m
 (mount-components))
```

We should now be able to open multiple browser windows and any messages typed in one window should show up in all the open windows.

A complete source listing is available on GitHub.