Article
Fabian Haupt · Feb 12, 2016 8m read

Asynchronous Websockets -- a quick tutorial

Intro

Please note, this article is considered **deprecated,** check out the new revision over here: https://community.intersystems.com/post/tutorial-websockets

The goal of this post is to discuss working with Websockets in a Caché environment. We are going to have a quick discussion of what websockets are and then talk through an example chat application implemented on top of Websockets.

Requirements:

- Caché 2016.1+
- Ability to load/compile csp pages and classes

The scope of this document doesn't include an in-depth discussion of Websockets. To satisfy your thirst for details, please have a look at RFC6455[1]. Wikipedia also provides a nice high-level overview[2].

The quick'n'dirty version: Websockets provide a full-duplex channel over a TCP connection. This is mainly focused on, but not limited to, facilitating a persistent two-way communication channel between a web client and a webserver. This has a number of implications, both on the possiblities in application design as well as resource considerations on the server side.

The application

One of the standard examples, kind of the 'hello world' of Websockets is a chat application. You'll find numerous examples of similar implementations for many languages.

First we'll define a small set of goals for our implementetation. Some of these might sound basic, but keep in mind, any project lives and dies with a proper scope definition:

- Users to send messages
- Messages sent by one user should be broadcasted to all connected users

- Provide different chat rooms
- A user should get a list of currently connected users
- A user should be able to set their own nickname

A production like chat application will have many more requirements, but the goal here is to demonstrate basic Websocket programming and not to replace IRC ;)

You'll find the code we are about to discuss in the repository (https://github.com/intersystems/websockets-tutorial). It contains:

- ChatTest.csp -- The CSP page actually showing the chat
- Chat.Server.cls -- Server side class managing the Websocket connection

To be able to manage the chatroom and the communication between the client and the server side, we are defining a couple of message formats we are going to use. Please note, that this is purely up to you. Websockets do not prescribe any format of the data being sent over it.

A chat message:

```
{
        "Type":"Chat",
        "Message":"<msg>"
}
```

A status update, informing the client of its websocketID

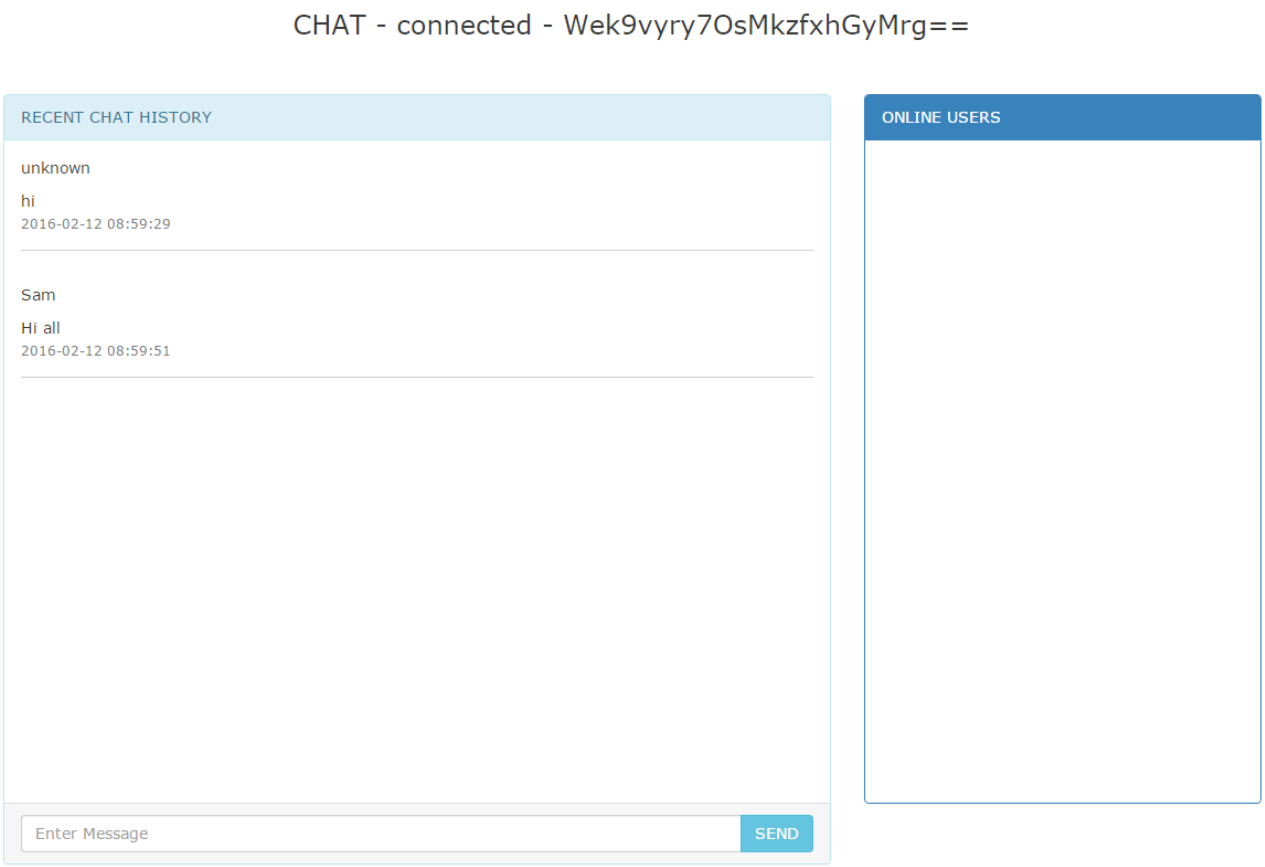
```
{
        "Type":"Status",
        "WSID":"<websocketid>"
}
```

The list of currently connected users (usually updated when a new user connects/disconnects):

```
{
  "Type":"userlist",
  "Users": [ {"Name":"<username>"},
                            ....
               ]
}
```

The Client side

We are going to render the client side with a simple single html5 page. This is basic html with a little bit of css to make it look nice. For details, look at the implementation of ChatTest.csp itself. For simplicity we're pulling in jquery, which will allow us to make dynamic updates to the page a little easier.



Down the road we are interested only in a couple of dom elements, identified by:

- #chat -- the ul holding the chatmessages
- #chatdiv -- the div holding #chat (used for automatic scrolling to the last message)
- #userlist -- the ul holding the list of currently active users
- #inputline -- the input field where the user is going to type messages

There are a couple of lines of javascript code in init() which are dealing with setting up events and handling input which we will not discuss. The first interesting bit is opening the WebSocket and binding function handlers to the relevant events:

In function init(): [..]

```
 ws = new WebSocket(((window.location.protocol == "https:") ? "wss:" : "ws:") + "//"
+ window.location.host + " #($system.CSP.GetDefaultApp($namespace))#/Chat.Server.cls"
+"?room="+ROOM);
```

This opens a websocket connection to our server side class, either using ws or secured wss protocol, depending on the way we connected to our page.

```
  ws.onopen = function(event) {
     $("#headline").html("CHAT - connected");
  };
```

Once the WebSocket has been opened, we update the headline to let us know we are connected. Note that we're

using the CSP expression "#($system.CSP.GetDefaultApp($namespace))#" to get the path of the current namespace. This will only work if you're using Caché to actually serve those pages. If you're only connecting to Caché as a backend and are planning on serving the page through a webserver directly, you'll have to hardcode the path to the websocket class (i.e. /csp/users/Chat.Server.cls)

```
  ws.onmessage = function(event) {
                  var d=JSON.parse(event.data);
                //var msg=d.Message;
                if (d.Type=="Chat") {
                        $("#chat").append(wrapmessage(d));
                        $("#chatdiv").animate({ scrollTop: $('#chatdiv').prop("scroll
Height")}, 1000);
                } else if(d.Type=="userlist") {
                        $("#userlist").html(wrapuser(data.Users));
                } else if(d.Type=="Status") {
                        document.getElementById("headline").innerHTML = "CHAT - conne
cted - "+d.WSID;
                }
  };
```

This function handles an incoming message. We parse the JSON formatted data and then simply act based on the different types of messages as we've defined them earlier: * Chat: using the helper function wrapmessage, we add the new message to the #chat ul * userlist: using the helper function wrapuser, we generate and update the #userlist * Status: is a general status update, and we'll put the websocket ID into the headline

```
  ws.onerror = function(event) { alert("Received error"); };
```

This handles an error, we're simply displaying a message.

```
  ws.onclose = function(event) {
          ws = null;
          $("#headline").html("CHAT - disconnected");
  }
```

Closing the websocket leads to updating the headline again.

The one function left is

```
  function send() {
          var line=$("#inputline").val();
          if (line.substr(0,5)=="/nick"){
                  nickname=line.split(" ")[1];
                  if (nickname==""){
                          nickname="default";
                  }
          } else {
                  var msg=btoa(line);
                  var data={};
                  data.Message=msg;
                  data.Author=nickname;
             if (ws && msg!="") {
                        ws.send(JSON.stringify(data));
```

```
            }
          }
     $("#inputline").val("");
  }
```

Here we handle setting the nickname via "/nick newnickname" and sending a new chatmessage to the server. For that we b64 encode the textmessage and wrap it into an object, which we'll send json encoded to the server. We're doing the b64 encoding to avoid having to deal with special characters.

Server side.

The server side code is a simple class extending %CSP.WebSocket. We'll discuss the 5 functions in our implementation now.

```
Method OnPreServer() As %Status
{
  set ..SharedConnection=1
  set room=$GET(%request.Data("room",1),"default")
  set:room="" room="default"
  if (..WebSocketID'=""){
    set ^CacheTemp.Chat.WebSockets(..WebSocketID)=""
    set ^CacheTemp.Chat.Room(..WebSocketID)=room
  } else {
    set ^CacheTemp.Chat.Error($INCREMENT(^CacheTemp.Chat.Error),"no websocketid defin
ed")=$HOROLOG
  }

  Quit $$$OK
}
```

is the hook that is getting called for a new WebSocket connection. Here we set ..SharedConnection=1 to indicate that we want to be able to write to this socket from multiple processes. We are also recording the new socket ID and association with a chatroom into globals. For this example we're using ^CacheTemp.Chat.* in the hopes to not conflict with anything. Obviously these can be replace by other mechanisms.

```
Method Server() As %Status
{

        job ..StatusUpdate(..WebSocketID)
        for {
        set data=..Read(.size,.sc,1)
         if ($$$ISERR(sc)){
            if ($$$GETERRORCODE(sc)=$$$CSPWebSocketTimeout) {
                                  //$$$DEBUG("no data")
              }
              If ($$$GETERRORCODE(sc)=$$$CSPWebSocketClosed){
                      kill ^CacheTemp.Chat.WebSockets(..WebSocketID)
                      kill ^CacheTemp.Chat.Room(..WebSocketID)
                      do ..EndServer()
                      Quit  // Client closed WebSocket
              }
         } else {
                 set mid=$I(^CacheTemp.Chat.Message)
                 set sc= ##class(%ZEN.Auxiliary.jsonProvider).%ConvertJSONToObject(da
```

```
ta,"%Object",.msg)
                set msg.Room=$G(^CacheTemp.Chat.Room(..WebSocketID))
                set msg.WSID=..WebSocketID //meta data for the message
                set ^CacheTemp.Chat.Message(mid)=msg.$toJSON()
                job ..ProcessMessage(mid)
         }
        }

        Quit $$$OK
}
```

The Server() as %Status method is being called for a WebSocket afterwards. This holds our main loop for incoming messages from the client. An incoming message gets stored into a global after which we job off a updater (job ..ProcessMessage(mid)). This is all we're doing in our main loop.

```
/// clients for this room.
ClassMethod ProcessMessage(mid As %String)
{
  set msg = ##class(%Object).$fromJSON($GET(^CacheTemp.Chat.Message(mid)))
  set msg.Type="Chat"

  set msg.Sent=$ZDATETIME($HOROLOG,3)
  set c=$ORDER(^CacheTemp.Chat.WebSockets(""))
  while (c'="") {
    set ws=..%New()
    set sc=ws.OpenServer(c)
    if $$$ISERR(sc){
      set ^CacheTemp.Chat.Error($INCREMENT(^CacheTemp.Chat.Error),"open failed for",c
)=sc
    }
    set sc=ws.Write(msg.$toJSON())
    set c=$ORDER(^CacheTemp.Chat.WebSockets(c))

  }
}
```

ProcessMessage is getting a message id passed in. It will parse the received json data into an %Object. We now $Order through our ^CacheTemp.Chat.WebSockets global to send the message to all connected chat clients for this room.

The BroadCast method demonstrates how to send a chatmessage to all connected clients. It's not being used on normal operations.

Wrapup

Exercise left for the user:

Implement the tracking of usernames on the server side and send a userlist message at the appropriate times.

Caveats, or why this isn't production code.

For a production ready system the inputs would need to be sanitized. We also hardly added any error trapping, access control, etc .

[1]https://tools.ietf.org/html/rfc6455

[2]https://en.wikipedia.org/wiki/WebSocket

Feedback

Please feel free to provide feedback in the comments below. I'll also try and answer any questions!

#Frontend #Tutorial #Caché

Source URL:https://community.intersystems.com/post/asynchronous-websockets-quick-tutorial