Article
Lily Taub · Mar 19, 2019 9m read

A Tutorial On WebSockets

Intro

Most server-client communication on the web is based on a request and response structure. The client sends a request to the server and the server responds to this request. The WebSocket protocol provides a two-way channel of communication between a server and client, allowing servers to send messages to clients without first receiving a request. For more information on the WebSocket protocol and its implementation in InterSystems IRIS, see the links below.

- WebSocket protocol
- WebSockets in InterSystems IRIS documentation

This tutorial is an update of "Asynchronous Websockets -- a quick tutorial" for Caché 2016.2+ and InterSystems IRIS 2018.1+.

Asynchronous vs Synchronous Operation

In InterSystems IRIS, a WebSocket connection can be implemented synchronously or asynchronously. How the WebSocket connection between client and server operates is determined by the " SharedConnection " property of the %CSP.WebSocket class.

- SharedConnection=1 : Asynchronous operation
- SharedConnection=0: Synchronous operation

A WebSocket connection between a client and a server hosted on an InterSystems IRIS instance includes a connection between the IRIS instance and the Web Gateway. In synchronous WebSocket operation, the connection uses a private channel. In asynchronous WebSocket operation, a group of WebSocket clients share a pool of connections between the IRIS instance and the Web Gateway. The advantage of an asynchronous implementation of WebSockets stands out when one has many clients connecting to the same server, as this implementation does not require that each client be handled by an exclusive connection between the Web Gateway and IRIS instance.

In this tutorial we will be implementing WebSockets asynchronously. Thus, all open chat windows share a pool of connections between the Web Gateway and the IRIS instance that hosts the WebSocket server class.

Chat Application Overview

The " hello world " of WebSockets is a chat application in which a user can send messages that are broadcast to all users logged into the application. In this tutorial, the components of the chat application include:

- Server: implemented in a class that extends %CSP.WebSocket
- Client: implemented by a CSP page

The implementation of this chat application will achieve the following:

- Users can broadcast messages to all open chat windows
- Online users will appear in the " Online Users " list of all open chat windows
- Users can change their username by composing a message starting with the " alias " keyword and this message will not be broadcast but will update the " Online Users " list
- When users close their chat window they will be removed from the " Online Users " list

For the chat application source code, please visit this GitHub repository.

The Client

The client side of our chat application is implemented by a CSP page containing the styling for the chat window, the declaration of the WebSocket connection, WebSocket events and methods that handle communication to and from the server, and helper functions that package messages sent to the server and process incoming messages.

First, we ' ll look at how the application initiates the WebSocket connection using a Javascript WebSocket library.

```
ws = new WebSocket(((window.location.protocol === "https:")? "wss:":"ws:")
                + "//"+ window.location.host + "/csp/user/Chat.Server.cls");
```

new creates a new instance of the WebSocket class. This opens a WebSocket connection to the server using the "wss " (indicates the use of TLS for the WebSocket communication channel) or " ws " protocol. The server is specified by the web server port number and host name of the instance that defines the Chat.Server class (this information is contained in the window.location.host variable). The name of our server class (Chat.Server.cls) is included in the WebSocket opening URI as a GET request for the resource on the server.

The ws.onopen event fires when the WebSocket connection is successfully established, transitioning from a ***connecting*** to a ***open*** state.

```
ws.onopen = function(event){
    document.getElementById("headline").innerHTML = "CHAT - CONNECTED";
};
```

This event updates the header of the chat window to indicate that the client and server are connected.

Sending Messages

The action of a user sending a message triggers the send function. This function serves as a wrapper around the ws.send method, which contains the mechanics for sending the client message to the server over the WebSocket connection.

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

send packages the information to be sent to the server in a JSON object, defining key/value pairs according to the type of information being sent (alias update or general message). btoa translates the contents of a general message into a base-64 encoded ASCII string.

Receiving Messages

When the client receives a message from the server, the ws.onmessage event is triggered.

```
ws.onmessage = function(event) {
    var d=JSON.parse(event.data);
    if (d.Type=="Chat") {
        $("#chat").append(wrapmessage(d));
            $("#chatdiv").animate({ scrollTop: $('#chatdiv').prop("scrollHeight")}, 1
000);
    } else if(d.Type=="userlist") {
        var ul = document.getElementById("userlist");
        while(ul.firstChild){ul.removeChild(ul.firstChild)};
        $("#userlist").append(wrapuser(d.Users));
    } else if(d.Type=="Status"){
        document.getElementById("headline").innerHTML = "CHAT - connected - "+d.WSID;
    }
};
```

Depending on the type of message the client receives (“ Chat ” , “ userlist ” , or “ status ”), the onmessage event calls wrapmessage or wrapuser to populate the appropriate sections of the chat window with the incoming data. If the incoming message is a status update the status header of the chat window is updated with the WebSocket ID,

which identifies the bidirectional WebSocket connection associated with the chat window.

Additional Client Components

An error in the communication between the client and the server triggers the WebSocket onerror method, which issues an alert that notifies us of the error and updates the page's status header.

```
ws.onerror = function(event) {
    document.GetElementById("headline").innerHTML = "CHAT - error";
    alert("Received error");
};
```

The onclose method is triggered when the WebSocket connection between the client and server is closed and updates the status header.

```
ws.onclose = function(event) {
    ws = null;
    document.getElementById("headline").innerHTML = "CHAT - disconnected";
}
```

The Server

The server side of the chat application is implemented by the Chat.Server class, which extends %CSP.WebSocket. Our server class inherits various properties and methods from %CSP.WebSocket, a few of which I ' ll discuss below. Chat.Server also implements custom methods to process messages from and broadcast messages to the client(s).

Before Starting the Server

OnPreServer() is executed before the WebSocket server is created and is inherited from the %CSP.WebSocket class.

```
Method OnPreServer() As %Status
{
    set ..SharedConnection=1
    if (..WebSocketID '= ""){
        set ^Chat.WebSocketConnections(..WebSocketID)=""
    } else {
        set ^Chat.Errors($INCREMENT(^Chat.Errors),"no websocketid defined")=$HOROLOG
    }
    Quit $$$OK
}
```

This method sets the SharedConnection class parameter to 1, indicating that our WebSocket connection will be asynchronous and supported by multiple processes that define connections between the InterSystems IRIS instance and the Web Gateway. The SharedConnection parameter can only be changed in OnPreServer(). OnPreServer() also stores the WebSocket ID associated with the client in the ^Chat.WebSocketConnections global.

The Server Method

The main body of logic executed by the server is contained in the Server() method.

```
Method Server() As %Status
{
    do ..StatusUpdate(..WebSocketID)
    for {
        set data=..Read(.size,.sc,1)
        if ($$$ISERR(sc)){
            if ($$$GETERRORCODE(sc)=$$$CSPWebSocketTimeout) {
                //$$$DEBUG("no data")
            }
            if ($$$GETERRORCODE(sc)=$$$CSPWebSocketClosed){
                kill ^Chat.WebSocketConnections(..WebSocketID)
                do ..RemoveUser($g(^Chat.Users(..WebSocketID)))
                kill ^Chat.Users(..WebSocketID)
                quit  // Client closed WebSocket
            }
        } else{
            if data["User"{
                do ..AddUser(data,..WebSocketID)
            } else {
                set mid=$INCREMENT(^Chat.Messages)
                set ^Chat.Messages(mid)=data
                do ..ProcessMessage(mid)
            }
        }
    }
    Quit $$$OK
}
```

This method reads incoming messages from the client (using the Read method of the %CSP.WebSockets class), adds the received JSON objects to the ^Chat.Messages global, and calls ProcessMessage() to forward the message to all other connected chat clients. When a user closes their chat window (thus terminating the WebSocket connection to the server), the Server() method ' s call to Read returns an error code that evaluates to the macro $$$CSPWebSocketClosed and the method proceeds to handle the closure accordingly.

Processing and Distributing Messages

ProcessMessage() adds metadata to the incoming chat message and calls SendData(), passing the message as a parameter.

```
ClassMethod ProcessMessage(mid As %String)
{
    set msg = ##class(%DynamicObject).%FromJSON($GET(^Chat.Messages(mid)))
    set msg.Type="Chat"
    set msg.Sent=$ZDATETIME($HOROLOG,3)
    do ..SendData(msg)
}
```

ProcessMessage() retrieves the JSON formatted message from the ^Chat.Messages global and converts it to an InterSystems IRIS object using the %DynamicObject class' %FromJSON method. This allows us to easily edit the data before we forward the message to all connected chat clients. We add a Type attribute with the value " Chat, " which the client uses to determine how to deal with the incoming message. SendData() sends out the message to all the other connected chat clients.

```
ClassMethod SendData(data As %DynamicObject)
{
    set c = ""
    for {
        set c = $order(^Chat.WebSocketConnections(c))
        if c="" Quit
        set ws = ..%New()
        set sc = ws.OpenServer(c)
        if $$$ISERR(sc) { do ..HandleError(c,"open") }
        set sc = ws.Write(data.%ToJSON())
        if $$$ISERR(sc) { do ..HandleError(c,"write") }
    }
}
```

SendData() converts the InterSystems IRIS object back into a JSON string (data.%ToJSON()) and pushes the message to all the chat clients. SendData() gets the WebSocket ID associated with each client-server connection from the ^Chat.WebSocketConnections global and uses the ID to open a WebSocket connection via the OpenServer method of the %CSP.WebSocket class. We can use the OpenServer method to do this because our WebSocket connections are asynchronous – we pull from the existing pool of IRIS-Web Gateway processes and assign one the WebSocket ID that identifies the server ' s connection to a specific chat client. Finally, the Write() %CSP.WebSocket method pushes the JSON string representation of the message to the client.

Conclusion

This chat application demonstrates how to establish WebSocket connections between a client and server hosted by InterSystems IRIS. For continue reading on the protocol and its implementation in InterSystems IRIS, take a look at the links in the introduction.

#Best Practices #Frontend #JavaScript #Node.js #ObjectScript #Tutorial #Web Gateway #Caché #InterSystems IRIS #Open Exchange