Lab 11: Web Sockets

Credits: this lab is based on a tutorial originally by Somchok Sakjiraphong. The information on the history of HTML5 is from Wikipedia.

About WebSockets

Introduction

A consortium of Web technologists including the W3C started developing the HTML5 standard in 2004, and the standard finally reached Recommendation (final) stage in 2014. Though it was only recently standardized, HTML 5 functionality has been implemented in browsers for much longer.

From the developer's point of view, the main goal of HTML5 is to allow the use of modern multimedia without plugins. It includes elements such as `<video>`, `<audio>`, and `<canvas>`. It includes a complete specification for the DOM, unlike previous HTML specifications, and it provides support for applications to run offline when the network is disconnected. Associated technology specifications include client-side databases and file storage.

One of the most useful new components of the HTML5 standard is the *WebSockets* API (`http://dev.w3.org/html5/websockets/`), standardized as RFC 6455 in 2011. Besides many other possible applications, WebSockets allows us to solve the problem of *server push* in Ajax applications without client polling and without the "long polling" hacks based on the XmlHttpRequest API used in Comet/Reverse Ajax. The idea is to initiate an HTTP connection to the server and request a WebSocket connection. If the server agrees, then the client and server establish a bidirectional communication channel that remains open as long as the parties desire. See Figure 1 below.

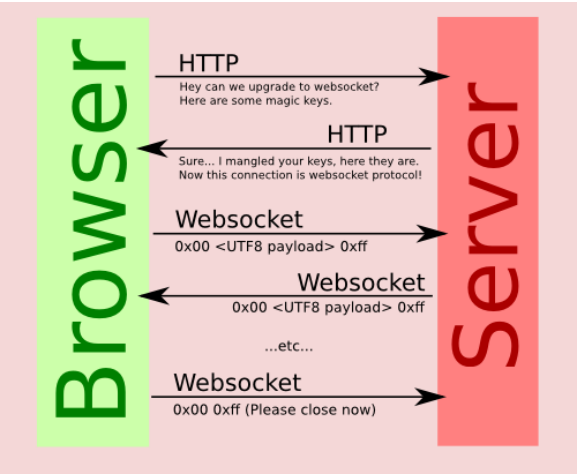

Figure 1: From Nitesh (2011), *Websockets in HTML5*, blog post available at `http://www.mobicules.com/web-application-development/websockets-in-html5/`.

Here is a sample of the client initiation of a WebSocket connection, from `http://en.wikipedia.org/wiki/WebSocketWikipediaarticle`:

```
GET /demo HTTP/1.1
```

```
Upgrade: WebSocket
Connection: Upgrade
Host: example.com
Origin: http://example.com
Sec-WebSocket-Key1: 4 @1  46546xW%0l 1 5
Sec-WebSocket-Key2: 12998 5 Y3 1  .P00

^n:ds[4U
```

Here is a sample of the server's response:

```
Server response:

HTTP/1.1 101 WebSocket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Origin: http://example.com
Sec-WebSocket-Location: ws://example.com/demo
Sec-WebSocket-Protocol: sample

8jKS'y:G*Co,Wxa-
```

On the client side, we use a Javascript API to make the connection, install a message receipt handler, then perform asynchronous sends and receives on the connection. Here is the Javascript client API, from the W3C working draft standard:

```
[Constructor(DOMString url, optional DOMString protocols),
 Constructor(DOMString url, optional DOMString[] protocols)]
interface WebSocket : EventTarget {
  readonly attribute DOMString url;

  // ready state
  const unsigned short CONNECTING = 0;
  const unsigned short OPEN = 1;
  const unsigned short CLOSING = 2;
  const unsigned short CLOSED = 3;
  readonly attribute unsigned short readyState;
  readonly attribute unsigned long bufferedAmount;

  // networking
  [TreatNonCallableAsNull] attribute Function? onopen;
  [TreatNonCallableAsNull] attribute Function? onerror;
  [TreatNonCallableAsNull] attribute Function? onclose;
  readonly attribute DOMString extensions;
  readonly attribute DOMString protocol;
  void close([Clamp] optional unsigned short code, optional DOMString reason);

  // messaging
  [TreatNonCallableAsNull] attribute Function? onmessage;
           attribute DOMString binaryType;
  void send(DOMString data);
  void send(ArrayBuffer data);
  void send(Blob data);
};
```

As of 2016, the client API is supported by all major browsers (Firefox, Chrome, Opera, Internet Explorer, and Safari).

EventMachine

EventMachine is an event-driven asynchronous I/O library for Ruby. This probably doesn't make much sense but looking at some examples might help. Consider the following piece of code.

```
require 'rubygems'
require 'eventmachine'

EventMachine::run do
  puts "Starting the run now: #{Time.now}"
  EventMachine::add_periodic_timer(10) do
    puts "Executing timer event: #{Time.now}"
  end
end
```

This prints the time every 10 seconds. How is it different from writing something like the following?

```
loop {
  puts "Executing timer event: #{Time.now}"
  sleep 10
}
```

The difference is blocking and non-blocking. When the `sleep` function is executed, it pauses and the thread is blocked. Which means we are just waiting for 10 seconds to pass by without doing anything. With EventMachine, we can easily execute other functions during the 10 seconds idle time and when the 10 seconds is over, the code in `add_periodic_timer` will be executed.

Since EventMachine is a very efficient Ruby networking library, we will use it to write our WebSocket server.

Prerequisites

Install the necessary software for the lab:

1. Download the sample blog application from the course Web site and get it up and running.
2. Install 4 gems: `eventmachine`, `eventmachine_httpserver`, `em-websocket`, and `json`.

Instructions

In this lab we will use the example of a blog application in which users can comment on a blog entry. Suppose there are two users, A and B, viewing a blog entry at the same time. If A decides to post a comment, B will not know that there is a new comment posted unless he or she first reloads the page. Traditionally, we would handle such updates by having a client script use `XmlHttpRequests` to poll the server and update the document when any change is found. However, it would be much better if it is possible that when A posts a comment, the server could push the new comment to every client immediately. Let's implement this cool feature.

1. Get the blog application up and running.
2. Create article containing title as string and content as text and also comment containing name as string and comment as text. Note that there is a relationship between article and comment.
3. Before we start, let's check that the basics work. Create a file `ws_test.rb` under `lib/`. Write the following code.

   ```
   require 'rubygems'
   require 'eventmachine'
   ```

```
EventMachine.run do
  EventMachine.add_periodic_timer(1) do
    puts "Hello world"
  end
end
```

Run `ruby ws_test.rb`. You should get a hello message every second. Once you understand the code, stop it and move to the next step.

4. Now we are going to create our WebSocket server. Add the following code to a new file `ws_server.rb` under `lib/`.

```
require 'em-websocket'

EM.run {
  websocket_connections = []
  EM::WebSocket.run(:host => "0.0.0.0", :port => 8080) do |ws|
    ws.onopen do
      puts "WebSocket connection open"
    end

    ws.onclose do
      puts "Connection closed"
    end

    ws.onmessage do |msg|
      puts "Recieved message: #{msg}"
    end
  end
}
```

We have created a WebSocket server on 0.0.0.0 port 8080. We will keep it running together with WEBrick. When a client connects, we will print a message and store the client information in the variable `websocket_connections`. When a client terminates a connection we remove it from the `websocket_connections` array.

5. To test our WebSocket server, let's add a simple script that connects to the server. We can just add it to `app/assets/javascripts/application.js`.

```
$(function(){
  function debug(str){
    console.log(str);
  };
  ws = new WebSocket("ws://0.0.0.0:8080");
  ws.onmessage = function(message) {
  };
  ws.onclose = function() {
    debug("WebSocket connection close");
  };
  ws.onopen = function() {
    debug("WebSocket connection open");
    ws.send("Hello Server");
  };
});
```

Use Firebug to check whether the connection is successful. You should see the message `Socket opened` on the console. When you stop the WebSocket server, you should see the message `Socket closed`.

6. Next we add the following line to the callback function `onopen` on the server:

   ```
   ws.send("Hello World")
   ```

   and the following to `onmessage` on the client

   ```
   alert(message.data);
   ```

   Every time the page reloads you should get an alert message.

7. Whenever a new comment is created, we want to notify the WebSocket server so that it can send the new comment to the client. This is a bit tricky since our Rails application and our WebSocket server are two separate processes. Since we are already using EventMachine and writing servers in EventMachine is not very difficult. We will create an HTTP server in the same event loop (inside `EventMachine.run` {...} probably below the code we start the WebSocket server).

   ```
   EM.start_server('0.0.0.0', 3001, MyHttpServer) { |conn|
     conn.instance_variable_set(:@websocket_connections , websocket_connections)
   }
   ```

   What we have done here is created an HTTP Server running on 0.0.0.0:3001. `MyHttpServer` is the class that handles the HTTP request. Add the following code before the event loop.

   ```
   require 'evma_httpserver'

   class MyHttpServer < EventMachine::Connection
     include EventMachine::HttpServer

     def process_http_request
       # the http request details are available via the following instance variables:
       # @http_protocol
       # @http_request_method
       # @http_cookie
       # @http_if_none_match
       # @http_content_type
       # @http_path_info
       # @http_request_uri
       # @http_query_string
       # @http_post_content
       # @http_headers
       puts @http_post_content
       response = EventMachine::DelegatedHttpResponse.new(self)
       response.status = 200
       response.content_type 'text/html'
       response.content = '<center><h1>Hi there</h1></center>'
       response.send_response
     end
   end
   ```

   Our HTTP server just prints the content posted and send some basic HTML as a response back to the client. Let's test our HTTP server by sending some POST request. We will use a gem called `httparty` to make some POST request, so add it to the Gemfile. Run `bundle install`, fire up `irb`, and try the following:

```
require 'httparty'
response = HTTParty.post("http://0.0.0.0:3001",
                         body: {name: "mdailey", age: "25"})
```

Our HTTP server should print out something like `name=mdailey&age=25`.

8. With the technique mentioned above, we can send a POST request to our HTTP server every time a new comment is created. Add the following method to our comment controller.

```
def post_comment(comment)
  require 'net/http'
  req = Net::HTTP::Post.new('/', initheader = {'Content-Type' => 'application/json'})
  req.body = comment.to_json(only: [ :id, :name, :comment, :article_id ],
                             methods: :pretty_date)
  response = Net::HTTP.new('0.0.0.0', '3001').start {|http| http.request(req) }
  return
end
```

We then slightly modify the create action, so it will look like this below.

```
def create
  @comment = Comment.new(comment_params)
  respond_to do |format|
    if @comment.save
      post_comment(@comment)
      format.html { redirect_to @comment.article, notice: 'Comment was successfully created.' }
      format.json { render :show, status: :created, location: @comment }
    else
      format.html { render :new }
      format.json { render json: @comment.errors, status: :unprocessable_entity }
    end
  end
end
```

Here we just call the function `post_comment` when the comment is saved to the database. Also add the following piece of code to the comment model to format our date:

```
def pretty_date
  created_at.strftime('%b %d, %Y at %I:%M %p')
end
```

Note that we are using Net::HTTP which is a part of Ruby standard library. Actually under the hood, HTTParty also makes use of Net::HTTP. Additionally, we choose to send a JSON since we can use the same JSON to send the every client which is every efficient as modern browser can parse JSON.

9. Now all we have to do is send the JSON to all the client when ever we receive a POST request of a new comment. Add the following piece of code inside the method `process_http_request` in our `MyHttpServer` class.

```
@websocket_connections.each do |socket|
  socket.send(@http_post_content)
end
```

Note: We do not need the `ws.send("Hello World")` in the callback function `onopen` anymore, remove or comment it out and replace with `websocket_connections << ws`.

10. At the client side we need to parse the JSON and write a little bit of JavaScript to create some HTML that will be appended to the list of comments. Therefore, we replace `alert(message.data)` in the callback function `onmessage` with the following code.

```
var comment = eval('(' + message.data + ')');
if (window.location.pathname == "/articles/" + comment["article_id"]) {
  var commentHtml = "<div class=\"comment\">" +
                    "<strong>" + comment["name"] + "</strong>" +
                    "<em> on " + comment["pretty_date"] + "</em>" +
                    "<p>" + comment["comment"] + "</p>";
  $('#comments').append(commentHtml);
}
```

11. Then go to `views/articles/show.html.erb` and add these lines

```
<div id="comments">
  <% @article.comments.each do |comment| %>
    <div class="comment" data-time="<%= comment.created_at.to_i %>">
      <strong><%= comment.name %></strong>
      <em>on <%= comment.created_at.strftime('%b %d, %Y at %I:%M %p') %></em>
      <p><%= comment.comment %></p>
    </div>
  <% end %>
</div>
```

That's it! Congratulations. You have implemented a real-time server push in your blog application.

Testing

One way to test our application is to open two tabs. Insert the comment in one tab and you should see that that the new comment is also added in the other tab.

Remember that WebSocket/MozWebSocket may not work on the some version of Firefox so if you try loading the page on Firefox you would get a JavaScript error. You might want to check out Socket.io which is a JavaScript library for WebSocket, it degrades gracefully to Flash on browsers that do not support WebSocket.