Building real-time web applications using WebSockets - Part 2.

**Contents:**

1. Installing a Web Framework:
   1. Installing Node, Express and Socket.IO packages
2. Integrating Socket.IO
   1. Server-side
   2. Client-side
3. Emitting Events - by client.
4. Broadcasting - by server.
5. Updating chart components.
6. Monitoring WebSocket Traffic.

**Installing a Web Framework:**

The first step in building our real-time application is to set up a Web framework, which can serve our html files. We are going to use express, the most popular Node.js Web framework.

Please follow below steps for setup:

1. [Download and install Node.js](https://nodejs.org/en/download/) (In case you already have node.js installed, please skip this step).
2. Create a package.json manifest file that describes our project in an empty directory (where you want to place your all project files). Optionally you can also create this file using cmd:

npm init

It will ask about your basic project details like name, version, description etc. and file will be created at the end.

{

"name": "myproject",

"version": "1.0.0",

"description": "A Demo Project.",

"author": "",

"license": "ISC"

}

3. Now, we need to install Node.js Web framework express:

npm install express --save-dev

4. Now that express is installed, we will create Server.js with below content which will setup our application:

//Express initializes ‘app’ to be a function handler that we can supply to an HTTP server

var express = require("express");

var app = express();

var http = require('http').Server(app);

var io = require('socket.io')(http);

// serving all client side html/css/js files, placed in public folder.

app.use(express.static(\_\_dirname + '/public'));

//We make the http server listen on port 8080.

http.listen(3000, function () {

console.log('listening on \*:3000');

});

5. Now, if we run node server.js , it should print listening on \*:3000 in cmd.

If you are able to see this, your express has been configured and serving html properly.

You can check by opening [http://localhost:8080/dashboard.html](http://localhost:8080/index.html) in your browser(assuming you have a dashboard.html file in your public folder).

**Integrating Socket.IO:**

Socket.IO can be installed by using command :

npm install socket.io --save-dev

This command will install Socket.IO module and will also update the dependency in package.json file.

Socket.IO is composed of two parts:

1. A Server which integrates with the Node.js HTTP Server : Socket.IO
2. A Client side library that loads on the browser side.: Socket.IO-client

We will see both client and server side implementation of Socket.io one by one.

***Server Side:***

var express = require("express");

var app = express();

var http = require('http').Server(app);

var io = require('socket.io')(http);

app.use(express.static(\_\_dirname + '/public'));

io.on('connection', function(socket){

console.log('a user connected');

socket.on('disconnect', function(){

console.log('user disconnected');

});

});

http.listen(3000, function () {

console.log('listening on \*:3000');

});

Notice, I initialized a new instance of Socket.IO by passing the http (the HTTP server) object. Then I listen on the connection event for incoming sockets and i log it to the console.

***Client Side:***

We just need to add this script tag in our html page just above </body>.

<script src="/socket.io/socket.io.js"></script>

And in app.js: var socket = io();

That’s all it takes to install socket.io-client, which exposes a io global, and then connect.

**Emitting Events - by client.:**

The idea behind using Socket.IO is that we can send and receive any events we want and with any data we want. Any objects that can be encoded as json or even binary data is supported.

In our application, client will emit the event with some data which is received by the server, then the server gets to know that there is some update and it broadcasts to all of its connected clients, making the experience real-time.

So, Let’s make the user emit the event on submit of new data, so that server can get it.

// in user.js

$('#pieform').submit(function () {

socket.emit('pie-chart-data', $('#inputData').val());

$('#inputData').val('');

return false;

});

**Broadcasting event - by server:**

Once event is emitted from any client, Server has to receive it and broadcast it to all connected clients.

// in Server.js

io.on('connection', function(socket){

socket.on('disconnect', function(){

console.log('user disconnected');

});

// Broadcasting by server.js

socket.on('pie-chart-data', function(msg){

io.emit('pie-chart-data', msg);

});

});

**Updating Chart Components:**

And, in app.js file, we need to listen this event, if any sent by server and update our D3 chart with new data.

// in app.js

// Making WebSocket connection with server

var socket = io();

socket.on('pie-chart-data', function(msg) {

// run angular digest cycle to update chart components

$scope.$apply(function() {

$scope.donutData.push(parseInt(msg));

});

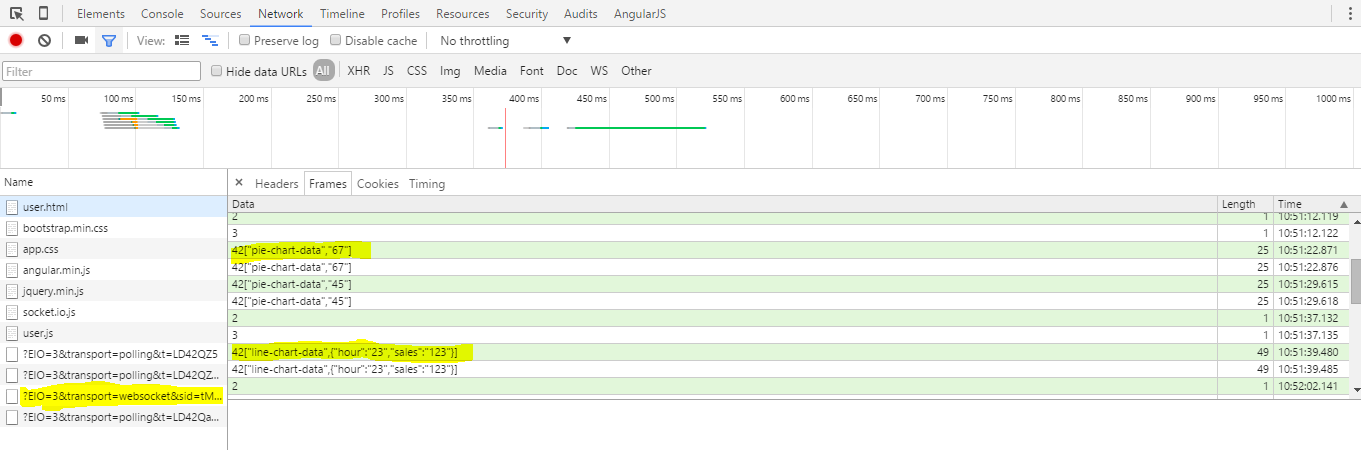
});

$scope.$apply runs the digest cycle, visiting watchers along the way and subsequently updating D3 chart components whose model has been updated(in this case model is $scope.donutData).

**Monitoring WebSocket Traffic:**

The developer tools in Google chrome include a feature for monitoring WebSocket traffic. You can access this tool by following below steps:

1. Open up the Developer tool (by pressing F12).
2. Go to the Network tab.
3. Click on the entry for your WebSocket connection.
4. Switch to the Frames tab.



This tool will show the summary of all the data sent through the connection.

**Conclusion:**

WebSockets represents a big step in the revolution of internet. By opening a bi-directional, low latency connection, it enables a whole new generation of real-time web applications.