FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Departamento de Engenharia Electrotécnica

REDES INTEGRADAS DE TELECOMUNICAÇÕES II

2015 / 2016

Mestrado Integrado em Engenharia Electrotécnica
e de Computadores

4º ano
8º semestre

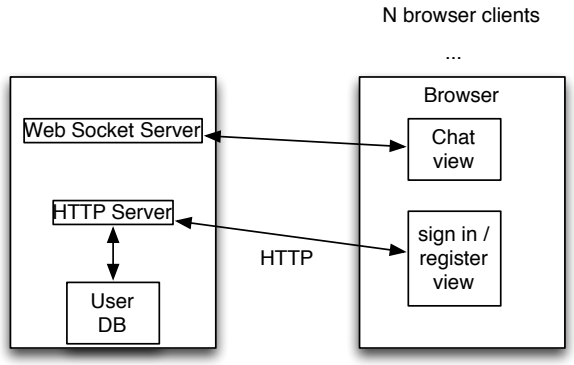
2º Project:
WEB application: Chat using Ajax REST APIs and Websockets

http://tele1.dee.fct.unl.pt **Pedro Amaral**

1. GOALS

Create a Web chat application. The ideia is to create a web chat application. The application is built around a Web server that contains a database of users.

The front-end runs in a web browser. There is a first view were a user can sign in if it already has an account. If the user still does not have an account it can create a new one. After a successful sign in the application switches to the chat view and a list of online users is displayed the user can then click in one of the online users and start a chat session with that use.



The web app has a server side implementation and a client side implementation.

The server consists in a web server and a web socket server. The web server serves http requests from the client to deal with the sign in interface and the new user register interface.

The web socket server deals with the real time communication between the clients in the chat.

The clients are dynamic webpages loaded in web browsers. The webpage has three main views: the sign in view; the user registration view and the chat view.

The application functions according to the following steps:

1. The user inserts the app URL in the browser.
2. The browser issues an HTTP GET to the HTTP server. In response the server renders a dynamic page containing the interface skeleton and links to a series of java script files.
3. The client receives the html file and performs GET requests to received the java script files and other assets of the page (e.g images etc..)
4. After receiving all the assets the browser executes the JavaScript code and renders the sign in page presenting it to the user.
5. If the user already has an account it can proceed with the sign in. The contents of the form are sent via an HTTP POST method to the server. The server checks the values against the user database and returns a token object in Java Object Notation format to the client. This step can be considered an AJAX call since the server does not return a web page but instead it returns the result of a computation in the token Object.
6. The client checks if the response was successful and stores the token Object. It then issues a uses web socket connection request to the server using the token to authenticate itself.
7. Upon success in the connection the client issues an event via the web socket to inform the server of the new user login

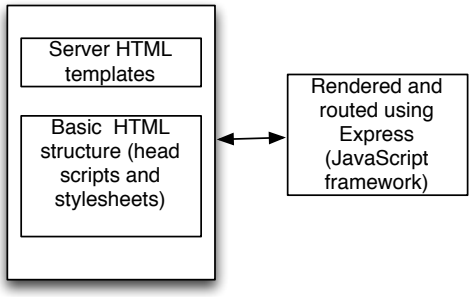
8. The server acesses the database and responds via the web socket with an JSON object containing the list of the other online usernames.

9. Upon reception of that information the client runs some JavaScript code and renders an html chat view. The user may then click in a username to start a chat.

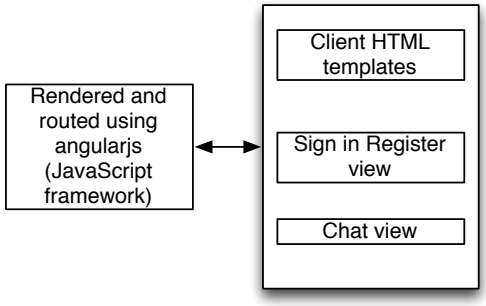
10. The clients and server use the web sockets to issue control and data messages between the clients.

11. The clients can disconnect from a chat and logout from the app, after this re-authentication is needed.

Both server side and client side scripting will be implemented in JavaScript. The Server Side however will use the latest version of the ECMAScript Standard called ECMAScript2015 also known as ES6 while the client side will still use ECMAScript 5 a.k.a ES5 (usually known as JavaScript). You will also use a mix of AJAX for the sign in process and user creation in the database and real time communication via web sockets for the control of the chat.



In the server side a template is used to render the basic HTML (head, script and stylesheets tags). Upon reception of an HTTP GET request the server routes the request (i.e it checks the file name in the GET) and renders the corresponding page.



In the client side the downloaded scripts are run by the browser to show the sign in page. When a user clicks on a button submits a FORM or clicks on a link of the rendered page the client side scripts routes those clicks. One of two things can happen: the link is routed locally and the client renders another page locally; or the link is external and an HTTP request is issued by the browser to a HTTP server.

The locally rendered pages might use information that must be obtained by the client from the server. That information can be obtained via na HTTP AJAX request or via received events in the web socket.

In the course website you can find a boilerplate for the implementation of the app. The boilerplate uses the Express framework for node.js in the server side and the Angular.js framework for the client side. Both

frameworks are in JavaScript. In the following sections we provide some background both on JavaScript as well as on each of this frameworks features with relevant code examples to the execution of the project.

2. Background

2.1 JavaScript

Tools

Google Chrome JavaScript console

Google Chrome Javascript console is a great way to test JavaScript right within any webpage. You can access from the menubar, View -> Developer -> JavaScript Console. Or by pressing the Option+Command +J keys.

NodeJS in Terminal/Shell

With NodeJS installed, you can run JavaScript files within the Terminal application. Within Terminal, navigate to the code directory that contains your .js file and run the file.

Variables

Basics

```
var myStr = "Hello world"; // this is a string
var answer = 42; // a number
var isNYU = true; // boolean
```

Arrays

```
var favPies = [‘blueberry’,'cherry','apple','grape']; //an array.
favPies.push(‘pumpkin'); // this dynamically inserts a value in the array
favPies.splice(2,1); //removes the ‘apple’ element from the array
```

Classes

Classes do not exist in JavaScript ES5 (but they do exist in the ECMAScript 2015 (ES6) version) only concrete instances of objects, therefore there are three ways to obtain an object in ES5.

Using a function

You define a normal JavaScript function and then create an object by using the new keyword. To define properties and methods for an object created using function(), you use the this keyword, as seen in the following example.

```
function Apple (type) {
    this.type = type;
    this.color = "red";
    this.getInfo = function() {
        return this.color + ' ' + this.type + ' apple';
    };
}
```

In ECMAScript 6 you can use a more Java like syntax to achieve the same result

```
class Apple {
   constructor(type){
    this.type = type;
    this.color = "red";
   }
    getInfo() {
        return this.color + ' ' + this.type + ' apple';
    }
}
```

To instantiate an object using the Apple constructor function, set some properties and call methods you can do the following:

```
var apple = new Apple('macintosh');
apple.color = "reddish";
alert(apple.getInfo());
```

A second way is using object literals

Literals are shorter way to define objects and arrays in JavaScript. So you can create an instance (object) immediately. Here's the same functionality as described in the previous examples, but using object literal syntax this time:

```
var apple = {
    type: "macintosh",
    color: "red",
    getInfo: function () {
        return this.color + ' ' + this.type + ' apple';
    }
}
```

In ECMAScript 6 the only difference is that there is now a method definition in object literals

```
var apple = {
    type: "macintosh",
    color: "red",
    getInfo() {
        return this.color + ' ' + this.type + ' apple';
    }
}
```

Note: ECMAScript is a superset of JavaScript (ECMA5) so every JavaScript code will run in an ECMAScript 6 engine however if you use the new ECMAScript 6 features in a browser that does not still have an ECMAScript 6 engine they will not work unless you use a ECMAScript 6 to JavaScript compiler.

In this case you simply start using this instance:

```
apple.color = "reddish";
alert(apple.getInfo());
```

Finally you can mix both cases defining the object as a function but calling the new right away:

```
var apple = new function() {
    this.type = "macintosh";
    this.color = "red";
    this.getInfo = function () {
        return this.color + ' ' + this.type + ' apple';
    };
}
```

In this case you are defining an anonymous constructor function and invoking it with new. You would use the object in the same way as in the previous case.

Functions

JavaScript functions are variables too! Declare them with ‘var'.

```
var sayHello = function() {
    console.log("Hello");
};
sayHello();
```

In ES6 there is a short hand notation for functions called the arrow notation =>

```
var sayHello =() => {
    console.log("Hello");
};
sayHello();
```

Arrow functions if defined as a property of an object have a different this scope however, if you need to use the this scope in the function code to refer to a property of the object you should use the standard function definition.

Since vars are dynamically typed arguments are passed without specifying their type.

```
var repeatYourself = function(words,number) {
    for (counter=0; counter<number; counter++) {
        console.log(counter + ". " + words);
    }
};

repeatYourself('yo',10); //say 'yo' ten times
```

Here an example where an object is passed into a function

```
var printObject = function(myObj) {
```

```
        console.log("Let's look inside the object...");
        for (p in myObj) {
            console.log(p + " = " + myObj[p]);
        }
    };

    var movie = {
        title : 'The Explorers',
        year: 1985,
        director : 'Joe Dante',
        cast : ['Ethan Hawke','River Phoenix','Bobby Fite'],
        description: "This adventurous space tale stars Ethan Hawke and young
star River Phoenix as misfit best friends whose dreams of space travel
become a reality when they create an interplanetary spacecraft in their
homemade laboratory."
    };

    printObject(movie);
```

and two examples on how to return data or objects from a function:

```
    var square = function(num) {
        return num * num;
    };

    var cube = function(num){
        return num * square(num);
    }

    fivesq = square(5);
    fivecube = cube(5);
    console.log( fivesq );
    console.log( fivecube );
```

```
    var createMessage = function(recipient, message) {
        mailObj = {
            to : recipient,
            message : message,
            date : new Date(),
            hasSent : false
        };
        return mailObj;
    };

    myMessage = createMessage('Red','Thanks for ITP.');
    console.log(myMessage);
```

Callbacks

JavaScript magic comes from callbacks. Callbacks allow the execution of a function to include 'next' steps when the requested function finishes. They are functions passed in the arguments of another function:

```
    var say = function(word) {
      console.log(word);
    }
    var computerTalk = function(words, theFunction) {
      theFunction(words);
```

```
    }
    computerTalk("Hello", say);
```

In the computerTalk function the theFunction argument is a function. When we invoke computer talk with “say” in the second argument we are telling to computerTalk to execute say in the `computerTalk("Hello", say);` line.

The same logic can be done with anonymous functions:

```
    // define our function with the callback argument
    function some_function(arg1, arg2, callback) {
        // this generates a random number between
        // arg1 and arg2
        var my_number = Math.ceil(Math.random() * (arg1 - arg2) + arg2);
        // then we're done, so we'll call the callback and
        // pass our result
        callback(my_number);
    }
    // call the function
    some_function(5, 15, function(num) {
        // this anonymous function will run when the
        // callback is called
        console.log("callback called! " + num);
    });
```

It might seem silly to go through all that trouble when the value could just be returned normally, but there are situations where that’s impractical and callbacks are necessary.

Traditionally functions work by taking input in the form of arguments and returning a value using a return statement (ideally a single return statement at the end of the function: one entry point and one exit point). This makes sense. Functions are essentially mappings between input and output.

Javascript gives us an option to do things a bit differently. Rather than wait around for a function to finish by returning a value, we can use callbacks to do it asynchronously. This is useful for things that take a while to finish, like making an AJAX request, because we aren’t holding up the browser. We can keep on doing other things while waiting for the callback to be called. In fact, very often we are required (or, rather, strongly encouraged) to do things asynchronously in Javascript.

2.2 Server Side (Node.JS using Express, JADE, Socket.io and Moongose)

The server-side development is done using Node.Js (nodejs.org) and Express. Node.js ia a platform built on top of the Google V8 JavaScript engine that executes the code it contains built-in libraries that make it possible to run a web server. It uses an event-driven, non-blocking I/O model. Express is a web application development framework that is built on top of Node.Js. Web frameworks provide commonly used functions for URL routing, HTML templates, sessions and extending functionality with third party libraries.

You can install Node.js and Express and use a simple text editor for coding.

Express apps are usually structured with several files and directories.

- /src this is the directory where the server side code exists there is a first file called app.js - this is the main application code that starts the server, requires all the needed libraries, connects to remote services like a database and general configuration for how the server will operate.

- /src/controllers this directory will contain the .js files that define the callback functions for your URL routes and the the code to receive and emit events using the web_socket.
-
- /views - this directory will contain your templates. We are using the Jade template engine.
- /public - all static assets go here. CSS, client side JavaScript files, images, etc.
- /node_modules - this directory contains all node.js models used in the app
- /models - This directory contains js files defining database data structures.
- /lib - This directory will be automatically updated by the ES6 transpiller used to transpile the ES6 code do standard ES5 to be ruined by node.js.

Lets analyse each one of these elements in more detail:

2.2.1 app.js

This file contains the main application code it starts with a series of var declarations that create object instances for several needed services:

```
    import express from 'express';
    import favicon from 'serve-favicon'; // middleware for tab icon
    import errorHandler from 'errorhandler';
    import logger from 'morgan'; // Logs each server request to the console
    import bodyParser from 'body-parser'; // Takes information from POST
requests and puts it into an object
    import methodOverride from 'method-override'; // Allows for PUT and DELETE
methods to be used in browsers where they are not supported
    import http from 'http';
    import mongoose from 'mongoose'; // Wrapper for interacting with MongoDB
    import expressJwt from 'express-jwt'; //for authentication based in tokens
    import socketioJwt from 'socketio-jwt'; //for token based authentication
in the websocket
    import socketio from 'socket.io'; //websocket communications
    import path from 'path' // File path utilities to make sure we're using
the right type of slash (/ vs \)
    import HttpController from './controllers/http' // controller that deals
with HTTP requests for the REST API
    import SocketController from './controllers/socket' //controller that
deals with the Websocket events
```

The above code imports the express object (requiring that the express module exists in the / node_modules directory) the http and path objects.

An app object is then created calling the express() function

```
    let app = express(); // app for http server
```

The next code:

```
    app.set('port', process.env.PORT || 3000); // you can change the port to
another value here
    app.set('views', path.join(__dirname, '..','views')); // sets up the path
for the Jade Templates
    app.set('view engine', 'jade'); //setup template engine - we're using jade
```

```
    app.use(express.static(path.join(__dirname, '..', 'public')));// setting
up the public dir
    app.use(favicon(path.join(__dirname, '..', 'public/img/favicon.ico'))); //
indicate where to find the tab icon
    app.use(logger('dev')); // use developer logs
    app.use(methodOverride()); // Allow PUT/DELETE
    app.use(bodyParser.json()); // Parse JSON data and put it into an object
which we can access
    app.use(bodyParser.urlencoded({ extended: true }))
```

sets a series of application environment parameters, namely the http server port the directory where the template files are the used template engine and several express APIs.

Finally the server can be started with the following code:

```
    server.listen(app.get('port'), function(){
           console.log('Express server listening on port ' + app.get('port'));
         });
```

2.2.2 Routes

Express provides a mechanism to route incoming HTTP requests according to the URLs to specific callback functions that are to be executed, it is a good practice to have the callbacks code in a different js file. In order to do that you have to import the external file to a HttpController object:

```
    import HttpController from './controllers/http' // controller that deals
with HTTP requests for the REST API
```

You can then refer to a function inside the file. For example the following code calls the get function of the app object this function takes to input parameters the first is the url that should be matched with the request the second is the callback function to execute when a request to that url is received in the server. So this means that we are passing the Signin function inside the http.js file as the callback function.

```
    app.get('/', HttpController.SignIn);
```

To complete this inside the http.js file we must create an HttpController object and export it

```
    let HttpController = {

    ….(other object code)

    SignIn: (req, res) => {
              let templateData = {
                  angularApp : "chatapp",
                  pageTitle : "RIT2 Chat"
                  }
              res.render('index', templateData);
          },

    }
    //expose the object to be imported
    export default HttpController
```

The SigIn function receives a req and a res object because it is the implementation of the callback function of an http get. The req object contains the http request and the res object will have the http reply, in the above example the reply contents are generated from the jade template index and the object templateData contains some data that is used to personalise the reply. This brings us to the next point templates.

2.2.3 Templates /views

Templates are files that are used by an engine to render html files, in this project we will use the jade template engine http://jade-lang.com/. In this project most of the html viewed in the app will be render in the client side. In the server the skeleton of the html page is rendered including the head and script tags and the header image. The following jade template is used:

```
doctype html
html(class="no-js", lang="en", ng-app="#{angularApp}")
  head
    meta(charset="utf-8")
    meta(name="viewport", content="width=device-width, initial-scale=1.0")
    title #{pageTitle}
    link(rel="stylesheet", href="stylesheets/foundation.css")
    script(src="js/modernizr.js")
    script(src="js/Angular/Angular.js")
    script(src="js/Angular/Angular-resource.min.js")
    script(src="js/Angular/Angular-route.js")
    script(src="js/#{angularApp}.js")
    script(src="js/controllers#{angularApp}.js")
    script(src="/socket.io/socket.io.js")
    script(src="js/angular/socket.js")


  body
    div(class="row")
      div(class="medium-12 columns")
        p: img(src="img/header.png")
      div(class="medium-12 columns")
        block content
    footer(class="row")
      div(class="medium-12 columns")
        p © DEE - FCT/UNL
    script(src="js/jquery.js")
    script(src="js/foundation.min.js")
    script $(document).foundation();
```

Jade uses a different syntax than html, tags are not closed and indentation is used to delimit each tag. So be very careful in the number of spaces that you use. You should use 2 spaces inside to include a tag inside another tag for example the following means that the paragraph will be included in the div tag.

```
div(class="medium-12 columns”)
  p: img(src=“img/header.png")
```

The template is a file called layout.jade which is the default name for the file used to render a response. As you can see the template contains the tags in the head part of the html including the title the stylesheet file and a bunch of scripts. The body part of the template only contains the header image and a footer. The works “block content” indicate where a partial template can be introduced.

This allows us to have a main layout for the pages and then a changing part where we can insert html rendered via another template file.

In our case the block content is rendered from the index.jade file when the instruction res.render('index', templateData); from the index.js file is executed. The contents of the index.jade file are:

```
extends layout

block content
  div(class="row", ng-view)
```

In this case a very simple layout that only has a div tag. Since we only render this page skeleton in the server these are the only template files needed.

2.2.4 Public / static assets

In the public directory you will have the images, stylesheets (.css files) and JavaScript files that need to be served to the clients. The `app.use(express.static(path.join(__dirname, 'public')));` instruction in the app.js file sets up this directory as the home directory for every url that we send inside the html to the client.

Lets re-examine the above jade template for the page skeleton it contains in the head part a series of included files.

```
link(rel="stylesheet", href="stylesheets/foundation.css")
    script(src="js/modernizr.js")
    script(src="js/Angular/Angular.js")
    script(src="js/Angular/Angular-resource.min.js")
    script(src="js/Angular/Angular-route.js")
    script(src="js/#{angularApp}.js")
    script(src="js/controllers#{angularApp}.js")
    script(src="/socket.io/socket.io.js")
    script(src="js/Angular/socket.js")
```

The stylesheet is in the file /stylesheets/foundation.css, this because we are using an html5 responsive front-end framework (a collection of stylesheet files and client side javascript that helps building responsive html5 pages) called foundation http://foundation.zurb.com/. The next lines include a series of JavaScript files that also must be in the public directory. Those scripts will be executed in the browser (client), in this project we use the client side framework angularjs http://angularjs.org/ to develop the front end so we need to include all the necessary files, including our one client side js files whose name will be rendered by jade using the angularApp variable so in case you render the template with the value angularApp : “chatapp" the files are controllerschatapp.js and chatapp.js we will talk about them later. Finally we include two JavaScript files that are needed to use the web socket on the client side.

2.2.5 Web Sockets

The web socket protocol is an IETF standard http://tools.ietf.org/html/rfc6455 that allows two way real-time communications on top of TCP. It is supported by all major web browsers and it is an independent protocol from HTPP. The only relationship is that the initial handshake request is sent to the

server as an HTTP upgrade request for which the server returns a response, the reason for this is to allow servers to handle HTTP requests and web socket requests in the same port. There are several modules available for Express to aid in the implementation of a web socket. We will use socket.io http://socket.io/. In order to be able to use socket.io we must install it in the Express modules directory. The installation of modules is controlled using a special file called package.json which describes in the JSON format the application and the modules that it uses.

```
{
  "name": "ChatApp",
  "version": "1.0.0",
  "description": "A chat app using MEAN with server side ES6”,
  "main": "app.js",
  "engines": {
    "node": "5.10.1"
  },
  "scripts": {
    "babel": "babel src --out-dir lib",
    "babel:w": "babel -w src --out-dir lib",
    "postinstall": "npm run babel",
    "start": "node lib/app"
  },
  "author": "Pedro Amaral",
  "license": "ISC",
  "dependencies": {
    "babel-cli": "^6.6.5",
    "babel-core": "^6.7.2",
    "babel-loader": "^6.2.4",
    "babel-preset-es2015": "^6.6.0",
    "body-parser": "^1.15.0",
    "css-loader": "^0.23.1",
    "errorhandler": "x.x",
    "es6-promise": "^3.1.2",
    "es6-shim": "^0.35.0",
    "express": "^4.13.4",
    "express-jwt": "^3.3.0",
    "file-loader": "^0.8.5",
    "foundation-sites": "x.x",
    "google-maps": "x.x",
    "jade": "^1.11.0",
    "method-override": "^2.3.5",
    "mongoose": "^4.4.7",
    "morgan": "^1.7.0",
    "reflect-metadata": "0.1.2",
    "rxjs": "^5.0.0-beta.5",
    "script-loader": "^0.6.1",
    "serve-favicon": "x.x",
    "socket.io": "^1.4.5",
    "socketio-jwt": "^4.3.4",
    "style-loader": "^0.13.0",
    "systemjs": "^0.19.24",
    "typescript": "x.x",
    "webpack": "^1.12.14",
    "zone.js": "0.6.6"
  }
}
```

The file starts by defining the apps name and version. The scripts part define the use of babel the transpiller of ES6 code to ES5. The dependencies part has all the modules that are to be installed and there you find socket.io as well as the jade tempting engine that we already talked about. In Eclipse you can right click the file can choose “npm install” this runs a package manager for node.js that install all the needed packets and dependencies. The skeleton project already contains the necessary node_modules, so unless you decide to use some other module you do not need to run pm install.

In order to use the web socket we must set it up in the main application file app.js.

```
var io = require('socket.io').listen(server); //creates the websocket
```

The io object requires the socket.io module code and represents a web socket that is listening in the same port of the server object that represents the http server and was already created.

We then can create a separate .js file to deal with the web socket communications that exports a SocketController object. In the app.js file we can do:

```
import socketio from 'socket.io'; //websocket communications

….
let server = http.createServer(app); //HTTP server Object
let io = socketio.listen(server); //WebSocket server Object associated
with the same port as HTTP
```

and in socket.js:

```
let SocketController ={
 // code to handle websocket communications
}
//expose the object to be imported
export default SocketController
```

The first event occurs when a client connects:

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

// code to deal with all other events of this connection

}
```

The callback function “function (socket)” receives the client socket as an input object. We can then use this object to respond to events sent from the client, or to emit events to the client. Receiving events from clients is dealt with by the “on” method of the socket object.

```
socket.on('newUser:username', function (data) {
 // callback function code to deal with the event
});
```

The first parameter is a string that identifies the event, the second is a callback function that receives as an input parameter a data object containing the received data. To send an event to a client we use the "emit" method.

```
socket.emit('init', {
  //JSON object to send data
});
```

The first parameter is again a description of the even and the second is an object containing the data to send.

Emit sends the event to client socket "socket" if we which to broadcast an event to every client connected we can use the "broadcast.emit" method.

```
socket.broadcast.emit('user:join', {
     //JSON object to send data
});
```

All these methods can be used inside the callback function "function (socket)" that is executed when a client connects and therefore treat the reception and sending of events to the specific client described by the socket object.

In this project you will need to send events received from a client to a different client. For example a message that arrived from a client and that the server will forward to the destination client. If you know the id of the socket that describes the connection with the destination client you can use it in the emit method to send an event to a different socket.

```
io.sockets.socket(SocketID).emit('send:message', {  // send to socket ID
                 //JSON object to send data
                });
```

As you can see in the example in this case we no longer use the socket object (since this would send the event to the same client) and instead we use the socket identified by "SocketID" in the list of active sockets (connections) in the web socket. The list of sockets is in "io.sockets" and we can obtain one of them using the function "io.sockets.socket(id)".

2.2.5 Data Base

We will use a database to store the information of the users of the chat. We will use a mongo db database http://www.mongodb.org/. You should download the software from http://www.mongodb.org/downloads and follow the installation instructions for you operating system in http://docs.mongodb.org/manual/. After installing the database software you need to star a database for that you should issue the command:

```
mongod --dbpath /pathtodatabe/database/
```

You can then use the mongodb console by issue the "mongo" command.

```
10-22-127-184:~ pedroamaral$ mongo
MongoDB shell version: 2.4.9
connecting to: test
Server has startup warnings:
```

```
    Thu Apr  3 14:47:20.598 [initandlisten]
    Thu Apr  3 14:47:20.598 [initandlisten] ** WARNING: soft rlimits too low.
Number of files is 256, should be at least 1000
    >
```

This will open a JavaScript shell on which you can issue JavaScript commands to manipulate the database. You can issue the command

```
    > show dbs
    local      0.078125GB
    >
```

to view a list of the running databased, then you can issue the command

```
    > use local
    switched to db local
```

to choose which database to manipulate. You can then use the same javascript commands you use in your code in the console to manipulate the database. For example the command

```
    db.users.find()
```

Will list all objects of the model “users” that exist in the database “db”.

In the app.js file you should add the following code to connect the app to the database.

```
    import mongoose from 'mongoose'; // Wrapper for interacting with MongoDB
    …
    mongoose.connect('mongodb://localhost:27017/ChatDB');
```

The first instruction imports “mongoose” that is associated with the mongoose Express model that manipulates mongodb databases. The second instruction connects to the database.

In order to make queries and insert records we must define a data model:

```
    import mongoose from 'mongoose';
    //create a schema for the User Object
    let UserSchema = new mongoose.Schema({
        name: String,
        email: String,
        username: String,
        password: String,
        islogged: Boolean, // indicates sign in status
        createdAt: { type: Date, 'default': Date.now } //stores date of record
creation
    });
    // Expose the model so that it can be imported and used in the controller
(to search, delete, etc)
    export default mongoose.model('user', UserSchema);
```

This creates a mongose.Schema object named “UserSchema” that holds the structure of a data entry. To maintain the code organised we can do this in a separate file and export the object.

When you need to access the database in another js file you can just import the object.

```
import usermodel from '../models/user.js';
```

In this case the “usermodel" object can then be used to perform operations in the database

To write an entry in the database you have to create an instance of the model:

```
var newUser = new usermodel({
                        name : “user name”,
                        email : “usermail@mail.com”,
                        username: “nickname”,
                       password: “password”,
                        newUser.createdAt = Date.now();
                      });
```

and then save it in the database.

```
newUser.save(function(err){
   if (err) {
             console.error("Error on saving new user");
             console.error(err); // log error to Terminal
             } else {
                     console.log("Created a new user!");
                   }
               });
```

The save function receives as an input parameter a callback function called upon completion.

To perform a query you use the following code that finds all users that have the field “islogged” with the value true. The result is returned in “onlineUsers” but it only contains the “username” field.

```
usermodel.find({islogged: true}, 'username', function(err, onlineUsers){
           if (err) {
             console.error(err);
             } else { // sucess send the array of online users
                 console.log(onlineUsers); //onlineUsers contains the result
              });        // end user model update
```

An alternative method finds only one value that as, in the example case, the username field equal to “usernametofind”.

```
usermodel.findOne({username:”usernametofind”}, function(err,ExistingUser)
                  {
                     if (err) {
                       console.error("ERROR: While inserting user");
                       console.error(err);
                     }
                     else {
                       console.log("User already exists");
                       console.log(res.data);
                     }
                  });
```

Another useful method is to update an entry of the database.

```
    usermodel.update({username:"usernametofind"},{$set:{islogged:
 false}},function(err, User){
                if (err) {
                  console.error(err);
                }
                if (User != null){ //user updated
                  console.log(User);
                }
          });
```

This searches for an entry with the username equal to "usernametofind" and then updates the "islogged" field to false. The "User" object contains the updated entry.

For a list of methods and further documentation please consult http://docs.mongodb.org/manual/.

As a final note the server code is written in JS6 and then translated to JavaScript by the babel model to run the server code you should run:

```
     npm run babel:w
```

In the terminal the executes babel and leaves running in the background (and translating the code whenever you change it) babel then outputs the translated code to the /lib directory to run the server you should then run:

```
    node lib/app
```

2.3 Client Side (Angular Js and socket.io)

The first step to use Angular for the client side scripting is actually done in the server, the page that is rendered and sent to client must contain a set of scripts with Angular code, this is done in the jade template so that the page returned to the client contains the appropriate "script" tags.

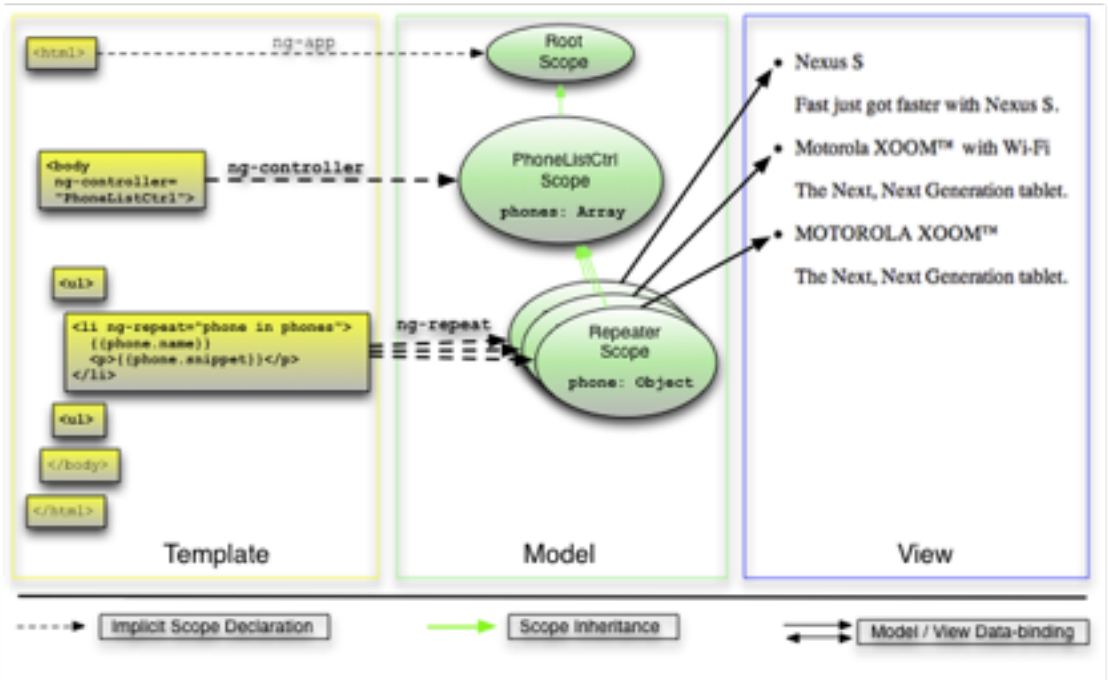
The Html that the server sends to the client will contain this script tags and the client will then request those JavaScritp files that it needs to use Angular and socket.io.

Two other things have also to be present in the html so that Angular works the first is a definition in the "HTML" tag of the Angular app with the word "ng-app"

```
    html(class="no-js", lang="en", ng-app="chatapp"
```

This means that the rendered HTML file will be controlled in Angular by a module called "chatapp" that module is defined in an JavaScript file that will be loaded by the client and therefore should also be included in the Jade template.

This file must be in the /public/js directory so that it might be sent to the client when the page loads.



2.3.1 Angular JS MVC

Angular uses a model view controller architecture http://en.wikipedia.org/wiki/Model%E2%80%93View%E2%80%93Controller. The following picture illustrates the concept.

The model consists in the application data and logic, the view is any output representation of information (in our case the several possible HTML files to display for the user) the controller glues the model and view by accepting inputs and issuing commands for the model or view.

In addition to dividing the application into three kinds of components, the Model–view–controller (MVC) design defines the interactions between them.

- A model notifies its associated view/views and controllers when there has been a change in its state. This notification allows views to update their presentation, and the controllers to change the available set of commands.
- A view is told by the controller all the information it needs for generating an output representation to the user. It can also provide generic mechanisms to inform the controller of user input.
- A controller can send commands to the model to update the model's state (e.g., editing a document). It can also send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document).

In Angular the view is the result of the model obtained via an HTML template. This means that whenever the model changes the view is updated. The following picture taken from the Angular js tutorial http://docs.angularjs.org/tutorial/ illustrates the concept.

The left side of the picture shows an Angular HTML template, Angular templates have a syntax similar to normal HTML enhanced with Angular directives and expressions. In the example in the body tag the “ng-controller" directive specifies that the controller for the that part of the DOM of the document is “PhoneListCtrl” this implicitly declares that everything in the body of the view belongs to the scope of that controller. This means that all Angular expressions and directives inside the body are accessible and controllable by “PhoneListCtrl”. The template has a simple list using the “ng-repeat” directive that iterates over an array of “phones” and lists each individual phone name and snippet. The controller can change the array (or any of its elements) and the view is automatically updated accordingly.

The concept of a scope in Angular is crucial. A scope can be seen as the glue which allows the template, model and controller to work together. Angular uses scopes, along with the information contained in the template, data model, and controller, to keep models and views separate, but in sync. Any changes made to the model are reflected in the view; any changes that occur in the view are reflected in the model.

To learn more about Angular scopes, see the angular scope documentation http://docs.angularjs.org/api/ng/type/$rootScope.Scope .

2.3.1.1 Angular routes

Lets see how we can built this in our project. The HTML returned from the server contains the "ng-app" directive meaning that this HTML will be associated with the Angular module "chatapp.js" it also contains the "ng-view" directive in a HTML "div" tag. This means that the part of the DOM in side the "div" will be a view controlled by the model defined in an Angular controller. If we want to have different views then we must set them up in the "chatapp" module.

```
    var chatapp = angular.module('rit2app', ['ngRoute','controllerschatapp']);
    //controllerschatapp.js containscontrollers code for the views
    /*
     *Callback for the config method of the module to configure the routes to
the different partial client side views
     * within the signin server side view
     */
    chatapp.config(['$routeProvider',
                    function($routeProvider) {
                         $routeProvider.
                         when('/register', {
                         templateUrl: 'partials/register.html', controller:
'registerCtrl'  //register view
                         }).
                           when('/signin', {
                           templateUrl: 'partials/signin.html',
controller:'signinCtrl'  //signin view
                           }).
                           when('/SeeYouHere', {
                             templateUrl: 'partials/listonline.html',
controller:'SeeYouHereCtrl'  //online users view
                             }).
                       // If invalid route, just redirect to the main signin
view
                          otherwise({ redirectTo: '/signin' });
              }]);
```

In this example we define the angular module and state that it uses the "ngRoute" model and the "controllerschatapp" module that will contain the code of the controllers for the different views. We then configure routes to different vies according to the url. In each case we specify the HTML template of the view and the respective controller. The final statement gives a last resort route (for any other URL) that points to the "signin" view.

2.3.1.2 Angular views

All partial HTML templates corresponding to the different views are stored in the /partials subdirectory of the public directory at the server the client will request those files to the server when angular wants to render them.

The following code is an example of an angular HTML template

```
    <div>
      <nav data-topbar class="top-bar">
           <ul class="title-area">
               <li class="name">
                     <h1><a href="/">RIT II {{view}}</a>
                     </h1>
               </li>
               <li class="toggle-topbar menu-icon">
                     <a href="#"><span>menu</span>
                     </a>
               </li>
           </ul><section class="top-bar-section">
           <ul class="right">
               <li>
                     <a href="#/register">Register</a>
               </li>
           </ul>
        </section>
    </nav>
      <div>
       <div class= "panel">
        <form>
         <div class= "row">
           <div class="medium-4 small-centered columns">
             <label>User Name
               <input ng-model="user.username" type="text" name="user"
placeholder="Username" />
             </label>
             <label>Password
               <input ng-model="user.password" type="password" name="pass"
placeholder="Password" />
             </label>
           </div>
         </div>
        </form>
        <div class= "row">
          <div class="medium-4 small-centered columns">
            <a class= "button [tiny small large]" ng-click="submit()" >sign
in</a>
            <a class= "button [tiny small large]" href="#/register">New?
Register Here</a>
          <div ng-show="isError" data-alert class="alert-box warning round">
            {{error}}
            <a href="#" class="close">&times;</a>
          </div>
          </div>
        </div>

       </div>
```

```
        </div>
    </div>
```

You can see that the syntax is equal to regular HTML with the addition of the angular directives and variables. The statement:

```
    <div ng-controller="signinCtrl">
```

Indicates that this part of the DOM (everything inside of this div) is controlled by the “signinctrl” controller.

```
    <h1><a href="/">RIT II {{view}}</a>
```

Indicates that a header will show a link to the URL “/“ showing the text RIT II followed by a string that will be defined in the controller using the variable {{view}}.

This view includes a form and in the input fields an angular directive is used to allow us the access in the controller to the value of the field

```
    <input ng-model="user.username" type="text" name="user"
 placeholder="Username" />
```

Other used directives are: the “ng-show” directive (controls if a part of the DOM appears or not according to the value of a boolean variable),

```
    <div ng-show="isError" data-alert class="alert-box warning round">
            {{error}}
            <a href="#" class="close">&times;</a>
          </div>
```

and the “ng-click” directive that associates a mouse click in an element of the view with a function in the controller.

```
    <a class= "button [tiny small large]" ng-click="submit()" >sign in</a>
```

2.3.1.3 Angular controllers

The final part is the definition of the “controllerschatapp” module that will contain the code of the controllers for the different views.

That module should be defined in a controllerschatapp.js file in the /public/js directory and should be included in a script tag in the initial page downloaded from the server.

In the file we start by defining the module

```
    var controllerschatapp = angular.module('controllerschatapp',[]);
```

Then each specific controller in this case we present the definition of the “signinCtrl” controller that is the controller of the above partial view.

```
      controllerschatapp.controller('signinCtrl', function ($scope, $http,
$window, $location, $socket) {
    $scope.view = 'Sign In';
    $scope.isError = false;
    $scope.submit = function () {
    /// code to execute when the submit button is clicked
    }
  });
```

The $scope word indicates that the following variable is one of the angular variables introduced in the partial HTML view template so for example “$scope.view” corresponds to the “{{view}}” variable in the HTML template that will be replaced with the ‘Sign In’ string in the view presented to the user.

“$scope.submit” refers to an “ng-click” directive value and therefore is a function. The callback of the controller functions receives a series of input parameters that start with an $ those are a series of angular services that can be used inside the function. You should add all the services that you will need.

2.3.1.4 Angular services

Angular services are objects that perform useful services they are good to organize and share code. In the example above $http, $window, $location, $socket are angular services. The first three are angular native and the last one is a service create by us to use the web socket.

The $http service allow us to make HTTP requests to the server. Consider the following example

```
          $http
            .post('/authenticate', $scope.user)
            .success(function (data, status, headers, config) {
               console.log("re-directing to chat view after successful
submit");
               $socket.connect();
               $location.path( "/chat" );
                });
            .error(function (data, status, headers, config) {
               $scope.isError = true;
               $scope.error = 'Error: Invalid user or password';
               });
            }
```

It uses the http service to issue a POST method HTTP request to the URL “/authenticate” and with the data contained in “$scope.user” (in the case of the our example view this would be the username in the form. The method as a success method and an error method both of which receive a callback function. The respective callbacks are then called according to the servers responses. In case of success we are using our own “$socket” service to connect to the web socket in the server and then we are using the factory service “$location$ to route to a different view controlled by a different controller. In case of error we set some scope variables that will put information about the error in the current view. You can find more information about angular services and their respective methods in http://docs.angularjs.org/guide.

2.3.1.4 Angular and socket.io

In order to use socket.io in the client side we have to add a two scripts in the initial HTML that is downloaded from the server. We did that by including the following lines in the Jade template that is rendered at the server in the initial request for our app first page.

```
        script(src="/socket.io/socket.io.js")
        script(src="js/angular/socket.js")
```

After the client downloads those scripts you know have access to an “io” object that represents the websocket. Since we will use the websocket at several parts of the code the best thing is to create a custom angular service that wraps the “io" object and provides us clean methods to operate the websocket. This can be done in the “rit2app.js” file after the definition of the module.

```
var chatapp = angular.module('rit2app', [‘ngRoute','controllerschatapp']);

chat2app.factory('$socket', function ($rootScope, $window) {
       var socket;
       return {
           connect: function(){
                 socket = io.connect('', {
                     query: 'token=' + $window.sessionStorage.token
             });
           },
           disconnect: function(){
                 socket.disconnect();
           },

          on: function (eventName, callback) {
           socket.on(eventName, function () {
             var args = arguments;
             $rootScope.$apply(function () {
               callback.apply(socket, args);
             });
           });
         },
         emit: function (eventName, data, callback) {
           socket.emit(eventName, data, function () {
             var args = arguments;
             $rootScope.$apply(function () {
               if (callback) {
                 callback.apply(socket, args);
               }
             });
           })
         }
       };
     });
```

The service warps the “connect” “disconnect" “on” and “emit” methods of the web socket. In the “connect” method the socket object receives the socket returned by the “io.connect” method provided by the web socket. In the other method that object is used to call the native methods. After the definition of the service you can use in the controllers the following methods.

```
$socket.connect();
```

Connects to the server.

```
$socket.emit('eventname', data);
```

Emits an event identified by “eventname” and containing the object “data”.

```
$socket.on('eventname', function (data) {
    // callback code
            });
```

Treats the reception of an event with name “eventname” and executes the callback function that receives as input the received object “data”.

```
$socket.disconnect();
```

Disconnects from the server.

The final service worth mentioning that you might need is the $window native service of angular the $window service which is a reference to the browser window object. It has a particular object called “sessionStorage” where you can create objects to store info that you want to persistent across different view and controllers. For example

```
$window.sessionStorage.username = “userxpto”;
```

stores the string “userxpto” in a username object inside sessionStorage.

2.4 Authentication (HTTP server and websocket)

In order to use the app the user has to authenticate itself both in the http server as well as in the web socket.

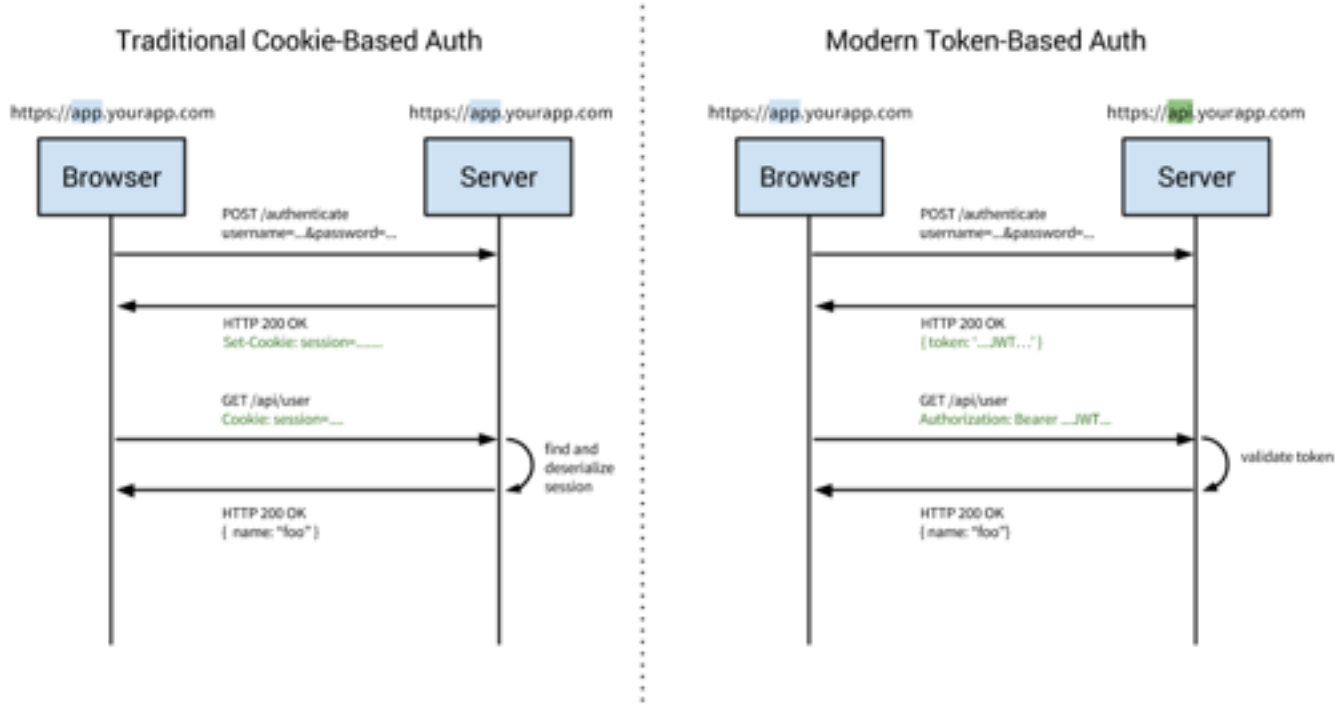
In the HTTP server the server will request that the authorisation header is filled to allow the response of requests. We will not use HTTPs in order to simplify the implementation but in a real production system that would be advisable, since without HTTPs the value in that header travels in clear and can be highjacked by an attacker. The following authentication scheme would be exactly the same in the HTTPS case with the only difference being that we would use an HTTPS server instead of the normal server.

2.4.1 Server side

There are basically two different ways of implementing server side authentication for apps:

The most adopted one, is Cookie-Based Authentication that uses server side cookies to authenticate the user on every request. A newer approach, Token-Based authentication, relies on a signed token that is sent to the server on each request. We will use this later approach.

The following diagram explains how both of these methods work.



As you can see in the cookie case the server verifies the username and password and with the OK responses it sends a session cookie that the client can then use in the following interactions. In the Token based case a toke is sent that is signed cryptographically, instead of the cookie the authorisation HTTP header is used by the client to send the signed token in the following interactions.

We will use two express modules called express-jet and jsonwebtoken so this lines must be present in the “package.json” file:

```
        "express-jwt": "x.x",
        "jsonwebtoken": "x.x",
```

In the app file we create two objects to represent those modules

```
      import expressJwt from 'express-jwt'; //for authentication based in tokens
      import socketioJwt from 'socketio-jwt'; //for token based authentication
    in the websocket
```

Then we create a secret that will be used to sign the token

```
      var secret = 'this is the secret secret secret 12356';
```

We can then configure express to protect access to a “/restricted” url.

```
      app.use('/restricted',expressJwt({secret:secret}));
```

In the code that handles that route we verify if the user and password are correct create the token sign it and send it like a JSON object in the HTTP response.

```
exports.ValidateUser = function (req,res) {
       //TODO validate req.body.username and req.body.password
       //if is invalid, return 401
       //If valid
       // We are sending a JSON object named in the authentication token
       //Get the JSON profile object from database for example
       var token = jwt.sign(profile, secret, { expiresInMinutes: 60*5 });
                        res.json({ token: token });
                    }
                 }
                );
            }else {
                res.send(401, 'Wrong user or password');
            }
          });
        };
```

and that is all that is needed for the http authentication server side.

For the web socket you need to set authentication for the “io” object

```
io.set('authorization', socketioJwt.authorize({
              secret: secret,
              handshake: true
            }));
```

Where “socketioJwt” and “secret” are the same objects as in the HTTP case.

2.4.1 Client side

At the client side in the code that handles the click in the username and password submit button you need to store the token if the response is successful a good place to so this is in “$window.SessionStorage”.

```
$http
      .post('/authenticate', $scope.user)
      .success(function (data, status, headers, config) {
        $window.sessionStorage.token = data.token;
        // do whatever is next
       })
      .error(function (data, status, headers, config) {
             // Erase the token if the user fails to log in
             delete $window.sessionStorage.token
             // Handle login errors here
          });
```

You also need to define a service and push it to the angularJs $http service so that in every subsequent request the token is sent in the authorisation header.

```
chatapp.factory('authInterceptor', function ($rootScope, $q, $window) {
       return {
         request: function (config) {
```

```
                config.headers = config.headers || {};
                if ($window.sessionStorage.token) {
                  config.headers.Authorization = 'Bearer ' +
    $window.sessionStorage.token;         }
                return config;
              },
              responseError: function (rejection) {
                if (rejection.status === 401) {
                  // handle the case where the user is not authenticated
                }
                return $q.reject(rejection);
              }
            };
          });

      chatapp.config(function ($httpProvider) {    //Adds the interceptor service
    to the $httProvider
            $httpProvider.interceptors.push('authInterceptor');
          });
```

For the we socket you need to add the authentication token in the “connect()” method in the socket service that we defined above.

```
      socket = io.connect('', {
                          query: 'token=' + $window.sessionStorage.token
                });
```

This will send the signed token in the initial query of the handshake to connect to the server.

3. Project

The project starts in the 14th of April and is due in the 22th of May, with a total of 6 lab classes in between. In the first class is mainly for setting up the environment and understating the tools. In this project there is room for improvements that each student might want to pursue in after the outlined goals are achieved. Has an example you might extend the chat concept with chat rooms where groups of students can be grouped in a conversation. You can also explore the possibilite to use REStfull APIs made available by services like google maps to enhance the chat experience in some way that you can think of. The following is a possible schedule for the implementation:

1. **in the end of the first class** You should have installed node.js, Express and all the needed modules for the project. You should have also installed mongodb and created a database. You should be able to start a simple express app and connect to the database, either using simply the command line or eclipse with the nodeclipse plug in. Finally you should have set up the boilerplate for the project and understand its structure and the skeleton code.
2. **in the end of the second class** You should have programmed the server side main file, setting up a route to deal with the initial browser url request. That route should render the html skeleton from the Jade template including all the necessary cliente side scripts; You should also create the model schema for the data to be stored in the database. You should start the client side programming of the sign in and register views, setting up the main angular modules and the controllers for those two views.

3. **<u>in the end of the third class</u>** The client side programming of the sign in and register view should be completed, you should program the authentication of the http part both in the server and client and the creation of users in the database in the server side.
4. **in the end of the forth class** You should have programmed the websocket connection between the client and the server with the respective authentication. You should start the client side chat view and respective controller.
5. **in the end of the fifth class** you should have finished the chat with websocket communication between pairs of users with an option to close a chat and another option to logout of the server.
6. **in the last class** You should use this class for the final tests and possible improvements. Some suggested improvements (NOT mandatory) are: creation of chat rooms where more than two users can chat simultaneously, use of third party REST API (like google maps) to enhance the experience (e.g. showing a map of the remote user location).

STUDENT POSTURE

Each group should take in consideration the following:

- Any changes in the HTML templates and in the UI views should be left to do in the end and according to the available time.
- Please code according to best practices (using correct indentation, comments and using meaningful variable names)
- The work should be distributed amongst all group members and all should have knowledge of all implementation options.