Departamento de Engenharia Eletrotécnica e de Computadores

Serviços e Aplicações em Redes

2024 / 2025

Mestrado Integrado em Engenharia Eletrotécnica
e de Computadores

4º ano
8º semestre

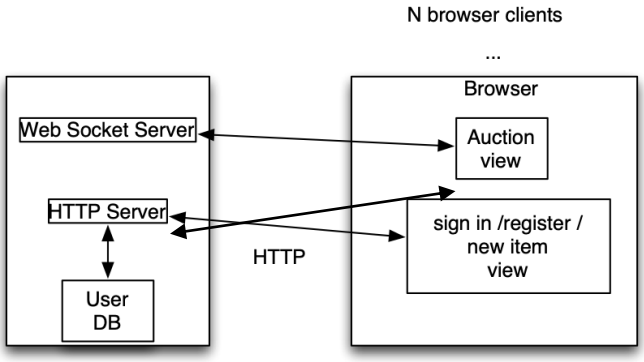
2nd Project:
WEB application: using a REST API and Websocket communication

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

1. Goals

Create a simple auction web application. The idea is to create an online Auction application. The application is built around a Web server that contains a database of users and items available for auction.

The front-end runs in a web browser. There is a first view where 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 auction view and a list of available items as well as the locations of the logged in users is displayed. The user can then click in one of the items and place a bid for that item, send a message to the owner of the item or if the item is owned by the user itself remove it from the database. The item list in this view should reflect the most current bid values as well as the actual time to the end of the auction. In the auction view the user can also click a button that re-directs to the new item view. The new item view contains a form to insert a new item, after the insertion the app returns to the auction view.



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

The server consists in a web server that can also start websocket connections. The web server serves HTTP requests from the client and serves static assets like the initial HTML file that bootstraps the client interface and the script files containing the code that will run in the client. The HTTP server also serves the requests for the RESTfull API.

The web socket server deals with the real time communication between the clients and the server (like bid actualisations and new item broadcasts).

The client consists in a dynamic webpage loaded in web browsers. The webpage has four views: the sign in view; the user registration view, the auction view and the new item 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 serves an index.html file that bootstraps the client-side JavaScript code.
3. The client receives the html file and performs GET requests to receive the java script files and other assets of the page (e.g. images, css files 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 REST API in the server. The server checks the values against the user database and returns a token object in Java Object Notation (JSON) format to the client.
6. The client checks if the response was successful and stores the token Object. It then issues a 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 client then renders the auction view and sends an HTTP request for the REST API to obtain the list of available items and can also send another request to obtain the list of logged users. These requests must contain the authentication token in the Authorization header to be accepted in the server. The server accesses the database and responds to the HTTP calls with JSON objects containing the list of the available items in auction or the list of logged in users.

9. Upon reception of that information the client runs some JavaScript code and renders an HTML auction view. The user may then click in an item and place a bid and visualize the locations of the logged in users.

10. The user can click a new item button, in that case JavaScript is run to render this new view. The user can fill the form and an HTTP post request is sent to the server. The server then uses the web socket to forward this new item to the other logged clients.

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

12. The clients can logout from the app, after this re-authentication is needed.

Both Server-side scripting and client-side scripting are implemented in TypeScript. You will also use a REST API to serve the sign in process, the request for the list of available items and the request for the list of logged in users and for the user and item creation and deletion. Real time communication for the control of the ongoing auctions is performed via web sockets.

In the server side upon reception of an HTTP GET request the server routes the request (i.e. it checks the file name in the GET or POST) and executes the corresponding API method. For all non-routed requests (i.e. the ones whose URI are not matched to an API method) the server acts as a normal file serving web server. Static assets (html, scripts, images etc..) are served from a specific directory (the server looks for the requested files in that path).

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 a HTTP request to the REST API or via events sent or received via the web socket connection.

In the course website you can find the link for the github repository of a boilerplate project for the implementation of the app. The boilerplate uses the Express framework for node.js in the server side and the Angular framework for the client side. Node and Express are JavaScript frameworks but are coded in TypeScript in the project. Angular is also coded in TypeScript all TypeScript code is then transpiled in to JavaScript before being run in node and being sent to the browser (the FrontEnd Angular code).
The boilerplate project contains all necessary backend and Front-End Angular code files that you will need to complete, as well as the needed developer container scripts to start two containers in docker desktop. One were the server will run and another for the MongoDB . The code was generated using Angular CLI (a powerful tool to scaffold Angular apps https://cli.angular.io/). Before building the client-side code a series of node modules must be installed, these dependencies are defined in the package.json files. You should start by:

1. Cloning the repository to your machine and enter the folder.
2. Install backend dependencies:

```
cd Backend
npm install
```

3. Install frontend dependencies:

```
cd ..
npm install
```

4. Build the front end code

```
Npmp run build
```

5. Start the backend server:

```
cd Backend
npm start
```

6. Start the development server:

```
cd Backend
npm run dev
```

7. Navigate to `https://localhost:3043` or `http://localhost:3000` in your browser

In the following sections we provide some background on JavaScript. TypeScript is a typed superset of JavaScript, meaning that it builds directly on top of JavaScript by adding new features-primarily static typing-while remaining fully compatible with all existing JavaScript code.

A detailed explanation of the project architecture can be found in the project folder in the filles called ARCHITECTURE.MD and STUDENT_GUIDE.md.

2. Background

2.1 JavaScript

Tools

Google Chrome JavaScript console (JSconsole)

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
```

Map

The *Map* object holds key-value pairs stored in a list, similarly to the *HashMaps* of Java. It is created using:

```
var listByKey = new Map();
```

New elements are added to the list associated with a key using:

```
listByKey.set(key, value);
```

Values can be searched by their keys,

```
let value= listByKey.get(key);
```

or iterated using a for loop

```
for (var ivalue of listByKey.values()) { // for all values
  console.log(‘value ‘, ivalue);  // print ivalue values
}
```

Elements can be deleted using `listByKey.delete(key)`.

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

An alternative way to deal with asynchronous method calls is to use promises. A **Promise** is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value. It's a way to handle asynchronous operations without blocking the rest of your code.

A Promise is in one of these states:

- **Pending**: The Promise's outcome hasn't yet been determined.
- **Fulfilled**: The operation completed successfully.
- **Rejected**: The operation failed.

An asynchronous function can return a promise object:

```
// define our function with the callback argument
function some_function(arg1, arg2) {
    // this generates a random number between
    // arg1 and arg2
  return new Promise((resolve,reject)=> {
     var my_number = Math.ceil(Math.random() * (arg1 - arg2) + arg2);
    // then we're done
    // pass our result
    resolve(my_number);
  }
}
// call the function
some_function(5, 15)
  .then(my_number =>{
    console.log("success: " + num);
  })
  .catch(error => {
    console.log("error! ");
  });
```

In this example, the some_function fucntion returns a promise object. The then method is used to specify what to do when the promise resolves (the function terminates sucessfuly), and the catch method handles any errors.

Promises are a powefull tool for handling asynchronous operations and keeping the code clean and organized, avoiding writing complex callback functions.

3. Project

The project has a total of 6 lab classes. The first class is for understanding the tools, environment, and the boilerplate project. 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 auction concept with a delete user option besides the mandatory delete item. You can also use the (already integrated in the project) google

maps REStfull APIs to enhance the auction 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 explored the provided source code. You should also learn how to compile and run the server using Vs Code and developer containers.

2. **Second class** You should have programmed the **server side** to store a new user and a new item in the database and the validation of the user login (`server/routes/api.js`)

3. **Third class** In the **server side**, handle the new user event in the web socket connection and the response to the get items HTTP API request and the get users API request. Implement a Log Out functionality, that updates the user info in the database and updates websocket connections info. Program the use of the web socket to inform clients of newly logged in users.

   The client side sends the following messages to the server side through the websocket interface:
   - `'newUser:username'` when the user signs in.
   - `'send:bid'` when the user bids for an offer;
   - the events `'connection'` and `'disconnect'` are received when a client connects or disconnects.

   The server side sends the following messages to the client side through the websocket interface:
   - `'new:user'` when a new user logged in;
   - `'items:update'` to send an update about the items information every second;
   - `'item:sold'` when an item is sold.

   These are the suggested minimal events, you can change them and or create new ones.

4. **Forth class** Program the regular updates (each second) of the items info (`backend/routes/api.js` and `backend/routes/socket.js`). Start programming the auction view interactions in the **client side**, program the events to place a bid. In the server-side program, the reception of a new bid and the sending off item sold events.

   The client-side code is fully implemented for the mandatory functionality except for the auction component, that implements the bidding actions.

5. **Fifth class** Implement the reception of the regular item updates in the client (`src/app/auction/auction.component.ts`) Program the buy now event logic in both the server and client sides.
6. **Sixt class** You should use this class for the final tests and possible improvements. One suggested improvement (NOT mandatory) is: creating an unregister user option. For this you might need to change other components in the client-side besides the auction component.

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.
- User interface improvements should be left for the end and should be
- 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.