ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

EMERGING MODELS FOR THE DEVELOPMENT OF SOCIAL MOBILE APPLICATIONS: PEOPLE AS A SERVICE, AND SOCIAL DEVICES. A PROOF OF CONCEPT.

Performed by
JOSÉ ANDRÉS CORDERO BENÍTEZ
Supervised by
CARLOS CANAL VELASCO
Department
LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

UNIVERSITY OF MÁLAGA
MÁLAGA, November 2014

Index

Introduction and goals

Nowadays smart devices have an excellent potential and connectivity, and can be used for multiple purposes: entertaining, socializing or sharing information. We have applications capable to do everything with the help of web services and cloud computing. However, we have currently some limitations.

These days everything is *something-as-a-service*. We are used to Software-as-a-Service and Platform-as-a-Service that have been well accepted by the industry. Now, when the usage of mobile devices have become customary for the majority of population, probably it is the moment of a new model. Mobile devices are becoming the most important piece in the communications, and they contain a lot of information about the owner. We could use this advantage to create a new type of communication, where devices interact with other devices as people would do.

The goal of this work is to learn about a new model, called People-as-a-Service [7][8], where mobile devices can deploy services to interact with other devices in their surroundings. In particular, this work studies the platform called *Social Devices*, developed as a research work by the University of Tampere in conjunction with Nokia [13]. This platform is currently in development and there is no documentation or further information available.

The main target of this work is to analyse the potential of the platform *Social Devices*, to study all the capabilities and create a base documentation about it that could be used for future works. I have created some case studies to study the potential and capabilities of the platform, which are completely documented to prepare the way for future projects. All the effort has been concentrated in researching and understanding the platform, documenting everything properly to facilitate the work for future projects using this platform and technology.

The platform is not finished and there is a long way to integrate everything, I have been collaborating with Niko Mäkitalo and his team in the University of Tampere improving some aspects of the platform and the Android client. The code is public to everyone and can be accessed in the next link:

https://github.com/nikkis

This work uses NodeJS[1] and Python[2] as basic technologies to develop new capabilities and functionality in the platform. The server service, called OrchestratorJS is currently developed in NodeJS, and the capabilities in Python. On the other hand, the client is focused in mobile devices, so we use Java as main technology to develop the client functionality for Android devices.

[1] http://nodejs.org/

[2] https://www.python.org/

Motivation

This project is aligned with the strategic objectives of the research groups SCENIC, from the University of Málaga, and QUERCUS, from Extremadura on Mobile Cloud Computing and related with the research projects *SeaClouds*[3] and SOFIA[4], in which is currently working the University of Málaga as many other universities across Europe. These research groups are focused in cloud technologies and mobile computing.

Cloud Computing is a new concept and there is a long future for it. It reduces time-to-market and provides on-demand scalability at a low cost to the business and a new concept is being developed: People-as-a-Service.

PeaaS is a new technology in development and there are only a few platforms that integrates the concept. The purpose of this project is to create a proof of concept of what People-as-a-Service can bring to us, focused in the platform Social Devices, currently in development as a PhD, by Niko Mäkitalo in the University of Tampere.

The target of this project is to do the proper investigation into the platform and elaborate a well know-how about it, exploring the possibilities and preparing all the documentation for further research and developments, as there is no documentation about it. The platform is currently in development and is not completely stable.

The concept of PeaaS will be also interesting for people, because this new technology will allow new ways of interacting between people and devices, providing new capabilities in how we use them. Combined with Internet of Things, people will be able to interact with the surroundings, the devices will know the context data of the user, and change its behaviour depending on it, making the work for the user without forcing him to send commands to the device.

Social Devices is a new platform that can be used for this purpose, it was introduced in 2011 as a joint work by Nokia Research Center, Aalto Univesity, and Tampere University of technology [13]. Smartphones have not only a lot of information about theis owners, but also capabilities that enable them to resemble humans (they can translate text into speech, for example).

[3] http://www.seaclouds-project.eu/

[4] http://sofia.lcc.uma.es/

Getting started with Social Devices

In this chapter we will see how to install and configure the OrchestratorJS Platform and all the needed software to get the platform prepared to use and develop. This guide explains how to install the platform and all the required software in Ubuntu. I have used a Virtual Machine to install everything, but it is valid for any version and computer.

Note: *It's very important to keep the same version of some of the dependencies, especially the npm modules, as the newer versions have changed some names and functions and they are not backward compatible.*

Dependencies

My test server runs on Ubuntu 14.04 LTS, however should be no problem in run another Linux Distribution or version. However this tutorial and all the dependencies included in the CD-ROM are specific for this version, and could not be valid in any other version or distribution.

- Ubuntu 14.04 LTS.
- NodeJS v0.10.25.
- Npm 1.3.10.
- MongoDB Shell version 2.4.9.
- OrchestratorJS Platform from my Github repository[5].
- Android Client for OrchestratorJS, which can be downloaded from my repository[6].

To test the applications we would need at least two Android devices, with Android 4.0 or higher, and Bluetooth capabilities.

Note: *Due the Platform is still under development, some applications could stop working if specifications change in the Github repository. If you want to test these applications properly I encourage you use my repository instead the official.*

---

[5] https://github.com/Jacb667/OrchestratorJS.git

[6] https://github.com/Jacb667/OrchestratorJSAndroid.git

Installation

Install Node.js

Probably you can install it from the repository of your Linux Distribution, but if it's not possible here you have one repository you might use. To add the repository to Ubuntu 14.04 you can run a terminal and write these commands:

```
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get update
```

Once the repository is successfully added, we can install Node.js:

```
sudo apt-get install python-software-properties python g++ make nodejs
```

If you have any problem to install Node.js or you want to install it in a different platform you can download and install from the Node.js website:

http://nodejs.org/download/

Install MongoDB

As in Node.js, probably you can install it automatically from your distribution's repository, if not, you can add this repository to Ubuntu and install it:

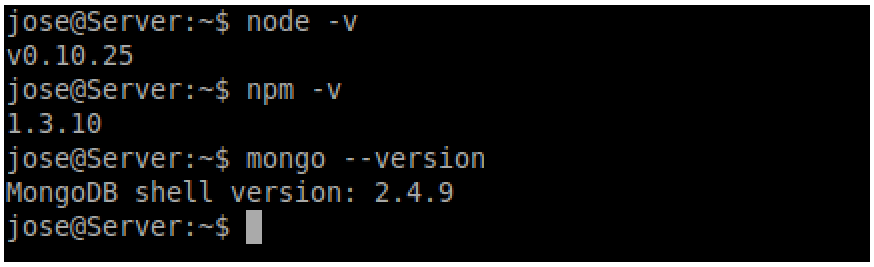
```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
echo  'deb  http://downloads-distro.mongodb.org/repo/ubuntu-upstart  dist
      10gen' | sudo tee /etc/apt/sources.list.d/mongodb.list
sudo apt-get update
sudo apt-get install mongodb-10gen
```

If you have any problem you can download and install it from the MongoDB website:

http://www.mongodb.org/downloads

Check if everything is correct

We can check if all the dependencies was correctly installed executing these simple commands in the terminal:



```
jose@Server:~$ node -v
v0.10.25
jose@Server:~$ npm -v
1.3.10
jose@Server:~$ mongo --version
MongoDB shell version: 2.4.9
jose@Server:~$
```

Figure 1. Terminal screen in Ubuntu showing the version of all the installed dependencies.

Note: *Usually npm will be installed with Node.js, if it was not installed automatically you will get an error executing the second command. In that case you can install npm manually in Ubuntu with the command:* sudo apt-get install npm

Installing the OrchestratorJS Platform

First of all we need the sources of the Platform. We can download them from the repository or simply take the version included in the CD-ROM, which is fully functional due the repository might change in future. If you decide to take the version included in the CD-ROM you can continue to next step.

Downloading sources

If you have the Git Tool installed you can clone the repository from Github with the command:

```
git clone https://github.com/Jacb667/OrchestratorJS.git
```

Or simply you can download the ZIP file prepared by Github which contains the last versión in the repository:

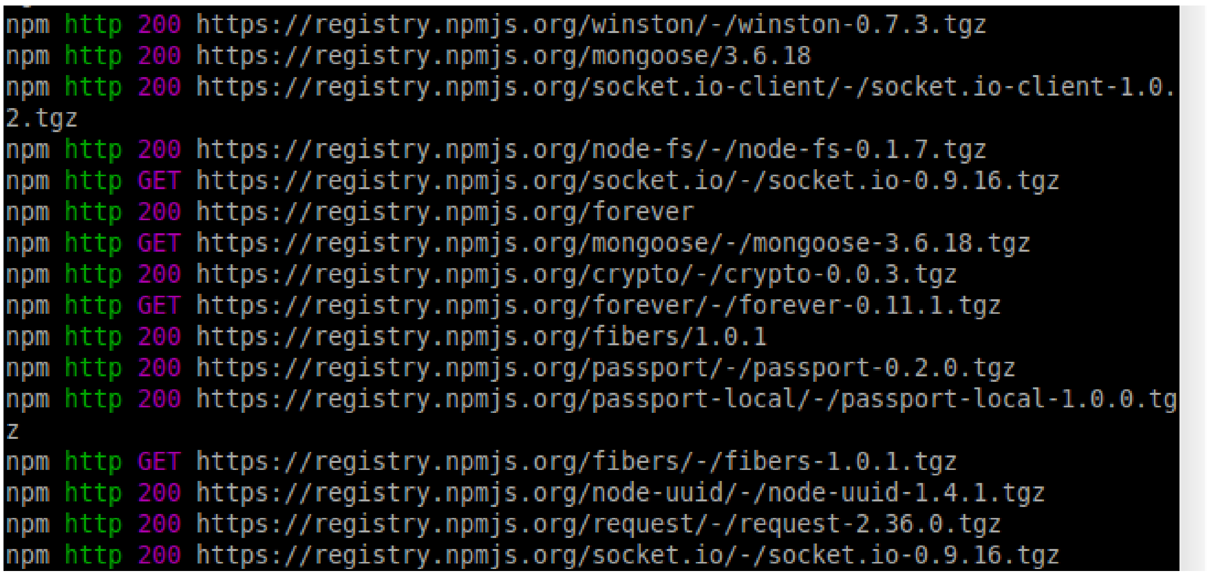
https://github.com/Jacb667/OrchestratorJS/archive/master.zip

Extracting and installing

Once you have the sources you can extract them in any directory. Then open a terminal in your Operating System and navigate to the directory where you copied the sources. To install the application you simply execute this command:

```
npm install
```

This command will automatically download and install all the required dependencies.



```
npm http 200 https://registry.npmjs.org/winston/-/winston-0.7.3.tgz
npm http 200 https://registry.npmjs.org/mongoose/3.6.18
npm http 200 https://registry.npmjs.org/socket.io-client/-/socket.io-client-1.0.
2.tgz
npm http 200 https://registry.npmjs.org/node-fs/-/node-fs-0.1.7.tgz
npm http GET https://registry.npmjs.org/socket.io/-/socket.io-0.9.16.tgz
npm http 200 https://registry.npmjs.org/forever
npm http GET https://registry.npmjs.org/mongoose/-/mongoose-3.6.18.tgz
npm http 200 https://registry.npmjs.org/crypto/-/crypto-0.0.3.tgz
npm http GET https://registry.npmjs.org/forever/-/forever-0.11.1.tgz
npm http 200 https://registry.npmjs.org/fibers/1.0.1
npm http 200 https://registry.npmjs.org/passport/-/passport-0.2.0.tgz
npm http 200 https://registry.npmjs.org/passport-local/-/passport-local-1.0.0.tg
z
npm http GET https://registry.npmjs.org/fibers/-/fibers-1.0.1.tgz
npm http 200 https://registry.npmjs.org/node-uuid/-/node-uuid-1.4.1.tgz
npm http 200 https://registry.npmjs.org/request/-/request-2.36.0.tgz
npm http 200 https://registry.npmjs.org/socket.io/-/socket.io-0.9.16.tgz
```

Figure 2. Console output after executing the *npm install* command.

Then we should install the forever library manually, because sometimes it is not installed. To install it manually simply execute this command:

```
sudo npm -g install forever
```

After executing this command I recommend to make sure that we have the required version of all the libraries installed, because we could get a newer library and sometimes they don't offer backward compatibility. We can check all the recommended versions in the Annex 1.

Configuration file

There is a config file in the *orchestratorjs* folder, called *config.json*. Most of the configuration parameters won't be changed, but we here we can enable or disable a debug mode, the database name in MongoDB (if the database doesn't exist in our MongoDB server it will be automatically created), the host and port to be used. Also we can add new device types.

Running the server

Now everything is installed and configured, the last step is running the server and hopefully everything will work well. To execute the server we need to open a Terminal and go to the directory *orchestratorjs*. In the directory we can execute the following command:

```
forever orchestrator.js
```

The server will take some seconds to start, now we can access to it from a browser to check it is running. Remember to put the configured port (9000 by default):

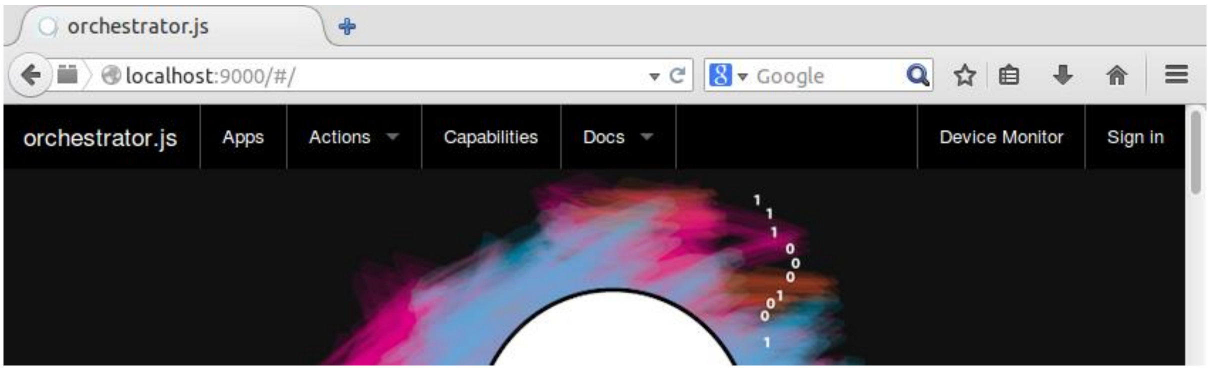


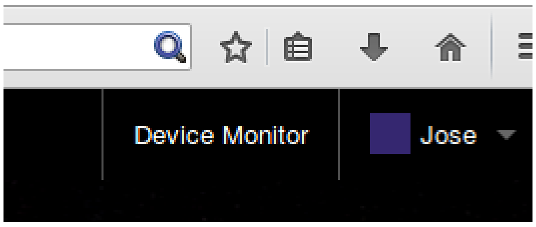
Figure 3. Welcome screen of OrchestratorJS.

We should access the welcome screen of OrchestratorJS.

Getting started

Creating a new user

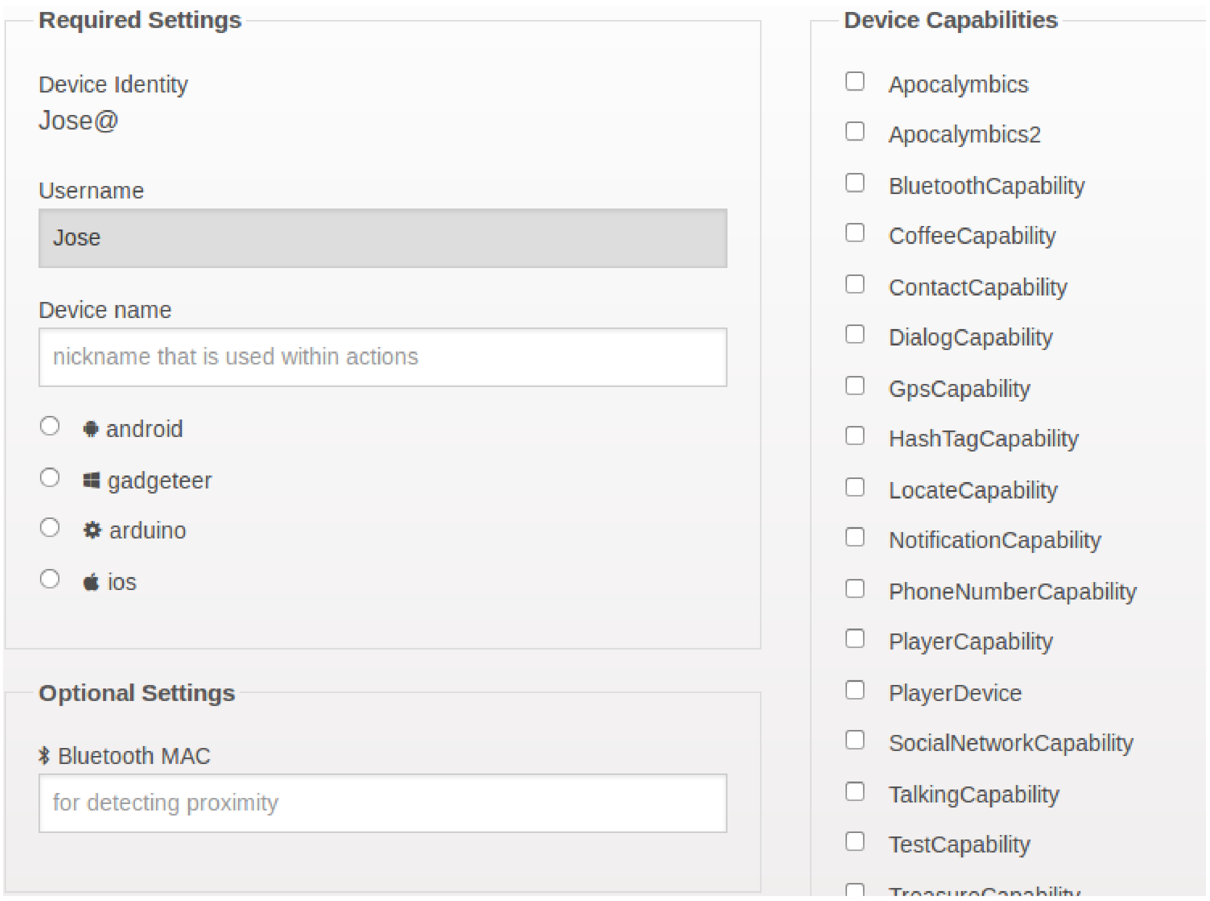
Now we have the OrchestratorJS server running and configured, let's start using it. The first step is register a new user in the platform. To do that we can go to the *Sign in* option in the main menu. A new screen will appear asking details to log in the application, to create a new user we click in the button at bottom.



The platform will ask for a username and password, after introducing them click the *Register* button. Your user is created successfully and now you can log in to the platform reintroducing the username and password. If everything went correct you should see now your name in the right corner instead of the *Sign in* button.

Registering a device

Now we should register a device to be able to use it from the platform. To do this we can go to our user → *Add Device*. A new view like in [Figure 4] will appear. In this view we have to specify a Device name, the type and the Bluetooth MAC Address.



Required Settings

Device Identity
Jose@

Username
Jose

Device name
nickname that is used within actions

- ( ) android
- ( ) gadgeteer
- ( ) arduino
- ( ) ios

Optional Settings

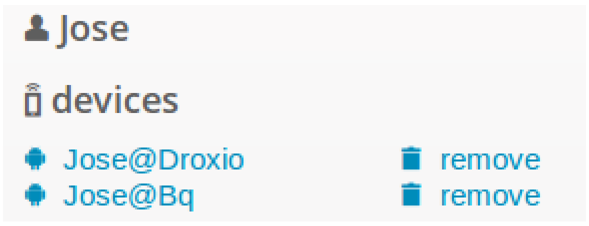
Bluetooth MAC
for detecting proximity

Device Capabilities

- [ ] Apocalymbics
- [ ] Apocalymbics2
- [ ] BluetoothCapability
- [ ] CoffeeCapability
- [ ] ContactCapability
- [ ] DialogCapability
- [ ] GpsCapability
- [ ] HashTagCapability
- [ ] LocateCapability
- [ ] NotificationCapability
- [ ] PhoneNumberCapability
- [ ] PlayerCapability
- [ ] PlayerDevice
- [ ] SocialNetworkCapability
- [ ] TalkingCapability
- [ ] TestCapability
- [ ] TreasureCapability

Figure 4. View to add a new device.

The *Device Identity* will be automatically generated from our username and the device name. This identity will be asked later when we install the Android client in our device.

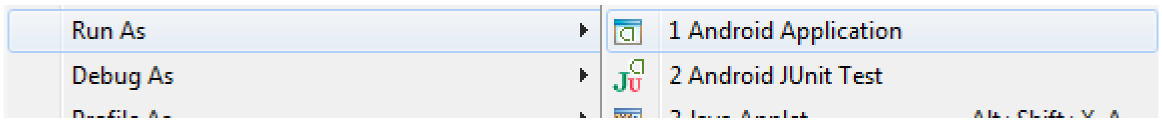


From this view we can enable or disable the desired capabilities directly from the server. We can do this now or we can return to this view in any moment from our user main screen.

The user screen can be accessed pressing in our name in the main menu (where was the *Sign in* button). In this view we will see all our registered devices, allowing us to modify their configuration or remove them.

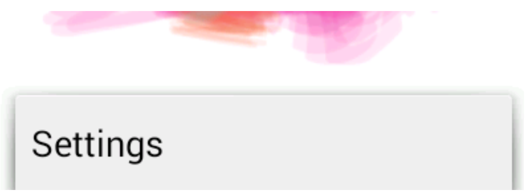
Installing the Android client

Everything is ready to install the android client. We can install the .APK file directly from the CD-ROM or import the project into the Eclipse AVD. Importing the project into eclipse will be necessary if you want to add your own changes to the client, like new capabilities.



To execute the project from Eclipse AVD it is only necessary to select the *Run As → Android Application*, like in any other android application. The application will be installed and executed in your device.

Note: *The Android Virtual Device cannot emulate Bluetooth capabilities, I recommend to test the platform on a real Android device.*

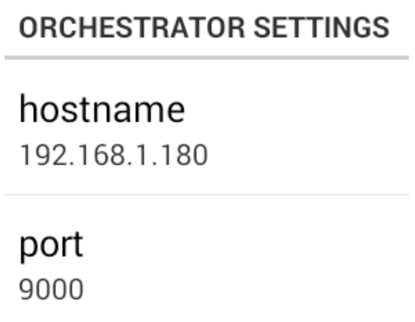


Once the application is running in our device, we won't be able to connect because we need to configure some parameters before. To open the application's settings view you have to press the button in your device to see the context menu, and then select the *Settings* option.

The first we should configure is the *Device Identity*, we should write here our device identity given by the website when we added the device. Usually this identity is formed in the way "*username@devicename*".

The device name must be written in the second field. In [Figure 45] the device name is set as "Emulator", and the device identity is "Jose@Emulator".

The second step, and probably the most important, is to configure the *Server Settings*. These advanced settings are the hostname and port of the OrchestratorJS server. If this is not well configured we won't be able to connect to our server neither see the available capabilities.

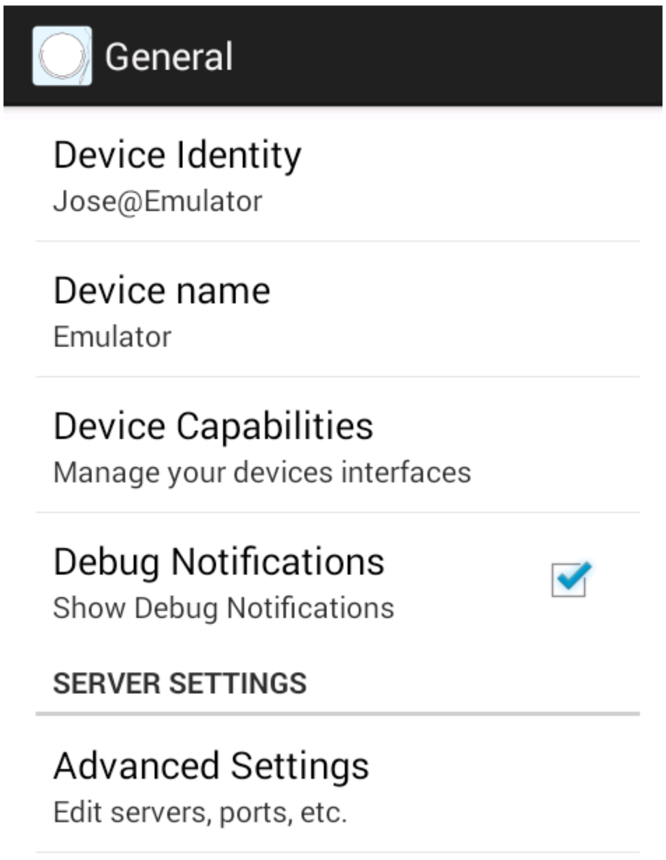

ORCHESTRATOR SETTINGS

hostname
192.168.1.180

port
9000

Below the *Orchestrator Settings* there are two input fields called *Proximity Settings*. These fields are not used in this moment and the can be ignored.

Note: *If Device Capabilities option shows an empty screen the problem is the client is not able to connect to the server and download the list of capabilities. If the server is correctly set up the main reason could be that the hostname and port are not correct.*


General

Device Identity
Jose@Emulator

Device name
Emulator

Device Capabilities
Manage your devices interfaces

Debug Notifications ✓
Show Debug Notifications

SERVER SETTINGS

Advanced Settings
Edit servers, ports, etc.

Figure 5. Settings screen of OrchestratorJS.

The *Device Capabilities* option will bring us to the view where we can enable or disable the desired capabilities. Take note that in this option we are disabling the capability on the device, not in the server.

It is possible to disable in the server-side and in the client-side. Only capabilities enabled in both sides will be executed. If the capability is disabled in server-side, the server will block immediately the actions using the capability, so the device won't receive any message.

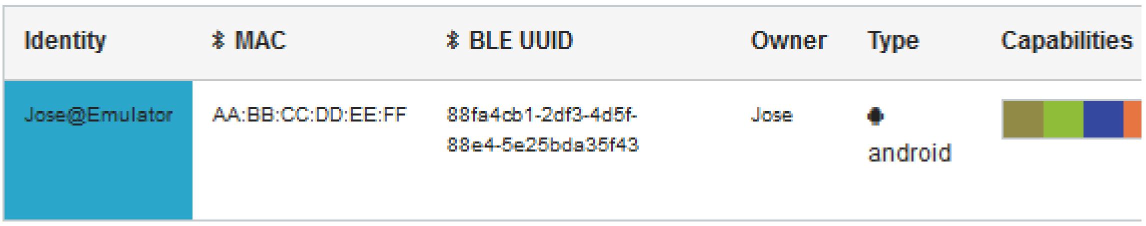
On the other hand, if we disable the capability on the device, but not on server-side, the server will send the actions, but the device won't execute them.

Debug Notifications is a useful option for developers, we can see small notifications showing any error occurred in the client. For example if the connection is lost or the delivery of context data fails for any reason.

Executing a first action

Everything is now configured and ready to user. In the Android client now we can close the settings menu and press the button *Connect* in the main screen. If everything is running and well configured we should see a message saying *Connected to Orchestrator.js*.

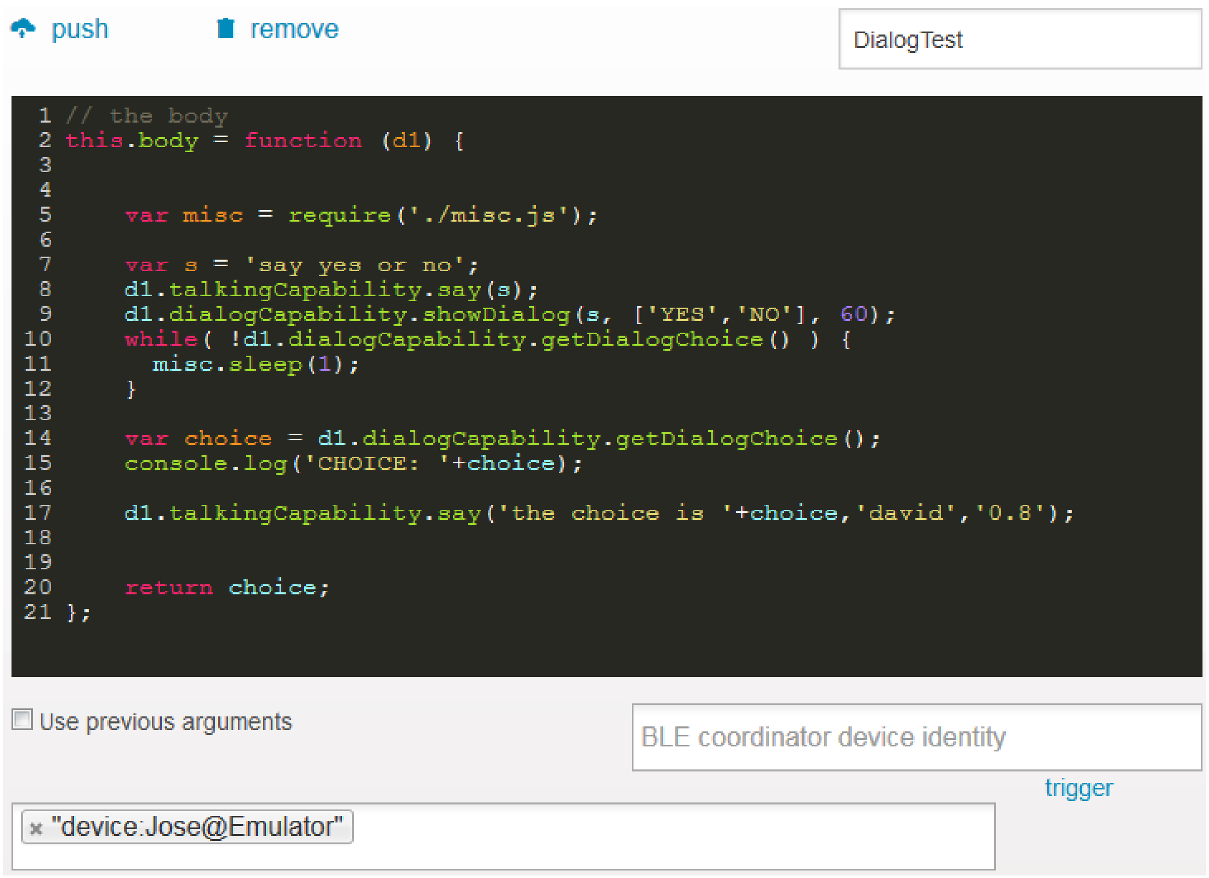
In the browser in the *Device Monitor* now the device has the background in a blue colour, which means the device is connected to the platform.



Identity	MAC	BLE UUID	Owner	Type	Capabilities
Jose@Emulator	AA:BB:CC:DD:EE:FF	88fa4cb1-2df3-4d5f-88e4-5e25bda35f43	Jose	android	🟫🟩🟦🟧

Figure 6. Device Monitor showing a connected device. The background is in blue.

Let's execute an action to see how it works. If they are not enabled, you should enable the capabilities *TalkingCapability* and *DialogCapability*, which are the capabilities used in this example.



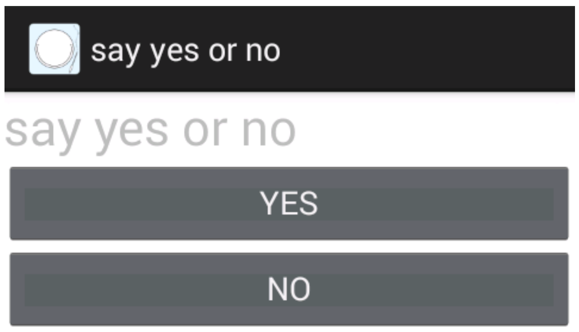
```
// the body
this.body = function (d1) {


    var misc = require('./misc.js');

    var s = 'say yes or no';
    d1.talkingCapability.say(s);
    d1.dialogCapability.showDialog(s, ['YES','NO'], 60);
    while( !d1.dialogCapability.getDialogChoice() ) {
      misc.sleep(1);
    }

    var choice = d1.dialogCapability.getDialogChoice();
    console.log('CHOICE: '+choice);

    d1.talkingCapability.say('the choice is '+choice,'david','0.8');


    return choice;
};
```

"device:Jose@Emulator"

Figure 7. DialogTest action implementation, being executed in the device *Jose@Emulator*.

When these capabilities are enabled, we can go to the menu *Actions → Definitions* and select the action called *DialogTest*. This action [Figure 7] will send a basic dialog to the device, asking to answer *YES* or *NO*. The question will be also said loudly by the device, using the Text-To-Speech Capability called *TalkingCapability*.

The action will then sleep for one second continuously while the user answers the question, once the user has answered, the action leaves the loop and reads the response, which will be output to the server console. Then, it calls the *TalkingCapability* again, which will say the chosen option to the user.



To execute the action we can go down to the arguments field and select our device in the list. This parameter will be sent to the action as the *d1* parameter of the function. Then, pressing the button *trigger* will send the action to the device.

The device will say the question using the *TalkingCapability* and show the question and the options to choose.

To understand how capabilities work, you can go now to the settings view in your device and disable the *TalkingCapability*. If you execute the same action now, it will show the question and ask the user for an option, but the device won't use the Text-To-Speech feature.

Starting the development

In this section I will explain the most important technical details that every developer should know about the platform. I think this information can be a great starting point for any developer interested in create new applications for the OrchestratorJS Platform.

***Note:** At the moment of this work there is no further information about how to develop applications for the platform, all of this information is the result of my investigation, and could vary in future versions.*

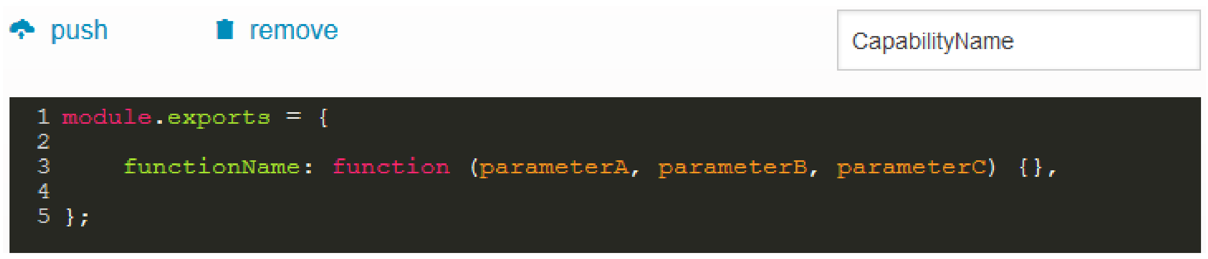
Capabilities

Capabilities are the main piece to develop when starting to implement a new functionality, as they are the only piece that is implemented in the client device, and thus, the only way to create new features.

We will use the Web IDE to create a new capability. To open the IDE, select the menu *Capabilities* in the main menu of the OrchestratorJS main page.

[Figure 8] shows the basic interface of a new capability called *CapabilityName* with a function *functionName*.

All the code inside the interface should be inside the JSON object *module.exports*. Inside the object we can add functions in the same way as we create JSON Objects, in the format *functionName* : *function (parameters…)*.



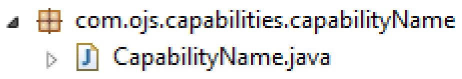
```
module.exports = {

    functionName: function (parameterA, parameterB, parameterC) {},

};
```

Figure 8. Basic code of a new functionality.

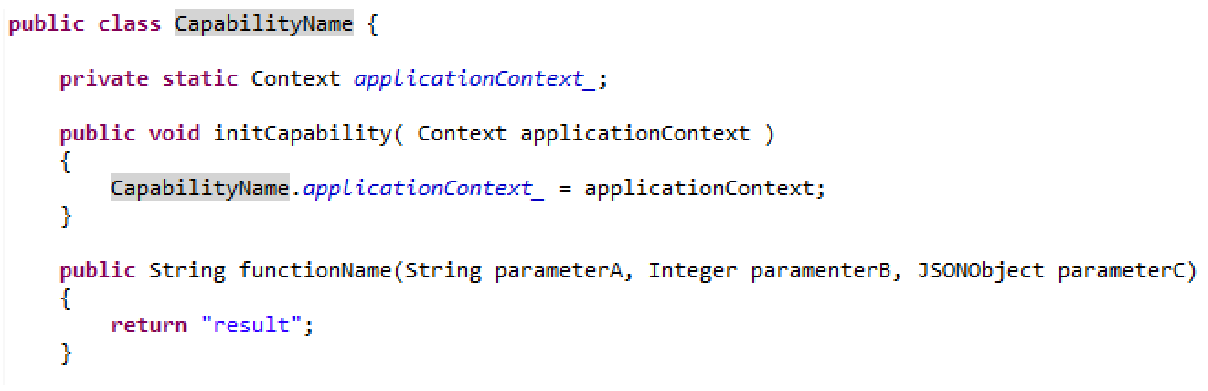
We can add as much functions as we want, all of these functions must be implemented in the client to avoid problems and runtime errors.

In this example I have created a function called *functionName* with 3 parameters: *parameterA*, *parameterB*, and *parameterC*. There is no need to specify the type of these parameters, the client will map them automatically through the name.

Let's see how this capability would be implemented in an Android client. First step will be create the right package to store the new capability. This package must be inside the package *com.ojs.capabilities* and the name of the package must be *capabilityName*, with the first letter in lowercase.



Inside the package we have to create the main class of the capability. The class has to be named exactly the same as the capability: *CapabilityName*.



```
public class CapabilityName {

    private static Context applicationContext_;

    public void initCapability( Context applicationContext )
    {
        CapabilityName.applicationContext_ = applicationContext;
    }

    public String functionName(String parameterA, Integer paramenterB, JSONObject parameterC)
    {
        return "result";
    }
```

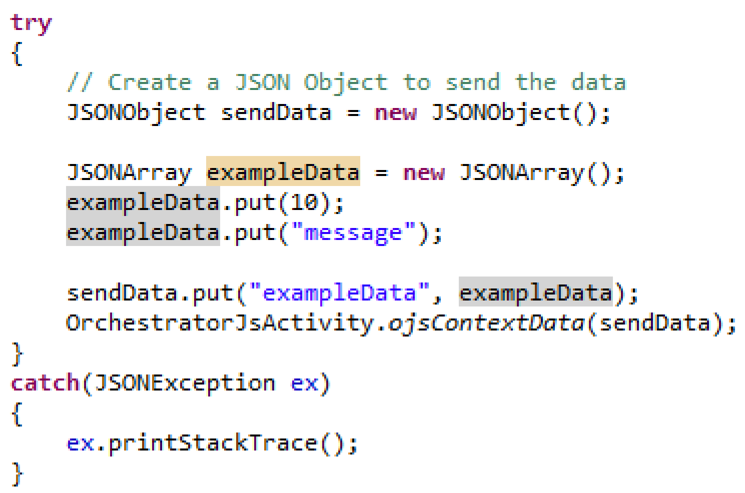
Figure 9. Implementation of the method *functionName* in *CapabilityName*.

Parameters of the method must be named as we defined in the interface, but they can be whatever type, including a JSONObject. Also we can specify a return type. The returned value will be taken by the server when executing the action, but the server won't wait for it.

We can specify anything inside the body of the function. For example creating a new Android Activity, calling a web service, asking an input from the user, etc.

The method *initCapability* is mandatory, we have to specify it in the main class of all the capabilities. This method will be executed when the client connects to the server and it allows to adjust any initialization settings the capability could need.

[Figure 10] shows how to send context data to the server. The context data must be a JSONObject, which can include any kind of data (JSONArrays, Strings, Integers, Booleans, Doubles, Floats, etc.). This data needs a name to differentiate it in the server-side.



```
try
{
    // Create a JSON Object to send the data
    JSONObject sendData = new JSONObject();

    JSONArray exampleData = new JSONArray();
    exampleData.put(10);
    exampleData.put("message");

    sendData.put("exampleData", exampleData);
    OrchestratorJsActivity.ojsContextData(sendData);
}
catch(JSONException ex)
{
    ex.printStackTrace();
}
```

Figure 10. Sending context data to the platform.

Actions

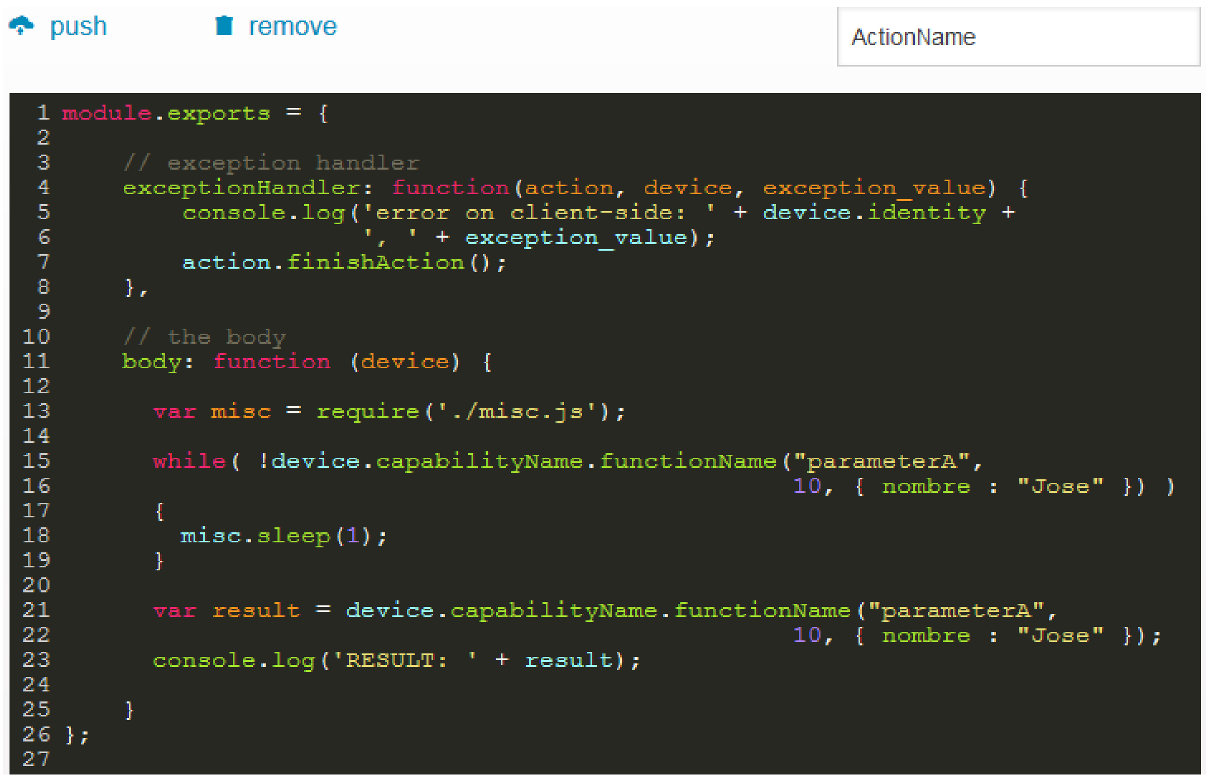
Let's create an action which uses the new capability. To create an action from the Web IDE we can go to the menu *Actions → Definitions*.

Actions have to be declared also inside the *module.exports* object, and they are really a new JSON Object. The body of the action must be always a function declared with the name *body*.

Additional functions can be declared, for example, an exception handler, which will be executed when the device throws an exception.

[Figure 11] shows a small example action which calls the method *functionName* from our capability. I included the library *misc.js*, which has a sleep method that will be useful as the return value from our capability method won't block the action waiting for the result. We need to poll the method until the value arrives.

I have implemented an *exceptionHandler* in this example, this method is optional and by default it will print the exception message in console and stop the action. We can specify here any behaviour to treat the exception.



```
module.exports = {

    // exception handler
    exceptionHandler: function(action, device, exception_value) {
        console.log('error on client-side: ' + device.identity +
                    ', ' + exception_value);
        action.finishAction();
    },

    // the body
    body: function (device) {

      var misc = require('./misc.js');

      while( !device.capabilityName.functionName("parameterA",
                                                 10, { nombre : "Jose" }) )
      {
        misc.sleep(1);
      }

      var result = device.capabilityName.functionName("parameterA",
                                                 10, { nombre : "Jose" });
      console.log('RESULT: ' + result);

    }
};

```

Figure 11. Action example using our new capability *CapabilityName*.

The *body* function has a parameter called *device*, this will be the device to execute the action. This parameter can be set in the arguments list when triggering the action.

Applications

Finally, we will implement a simple application which will call our new Action. Simply we have to go to the *Apps* menu and select the button *NEW APP* in the grid. This will open us the Web IDE with an example code of how to implement basic things.

An application will basically monitor context data received from the devices and trigger actions based on that data. We can also create threads (*Fibers*) to execute periodic tasks, like poll a website or service.

[Figure 13] shows the first part of a basic application. The first we should do is to include all the required modules from Node.js:

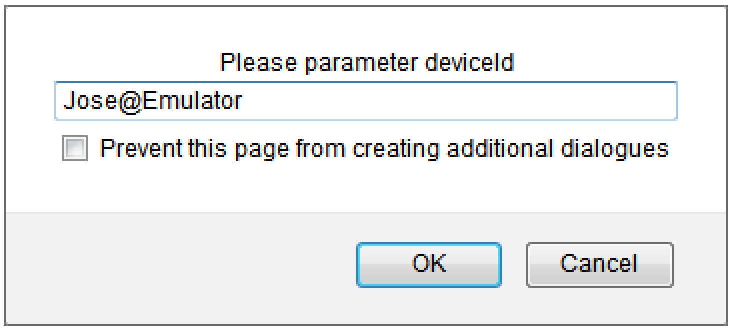
```
var httprequest  = require( '../../tools.js' ).httprequest;
var pubsub       = require( '../../tools.js' ).pubsub();
var tools        = require( '../../tools.js' );
var Fiber        = require( 'fibers' );
```

Then, the JSON Object *modules.exports* is declared, in the same way of actions and capabilities, we have to define the name of the object and the value. An application wil have basically two objects:

- **Settings:** this object will include a list of parameters required for the application. This parameters will be requested to the user when running the application [Figure 12].

- **Logic:** this object contains a function with the body of the application.

The settings defined can have a default value specified.



Please parameter deviceId

Jose@Emulator

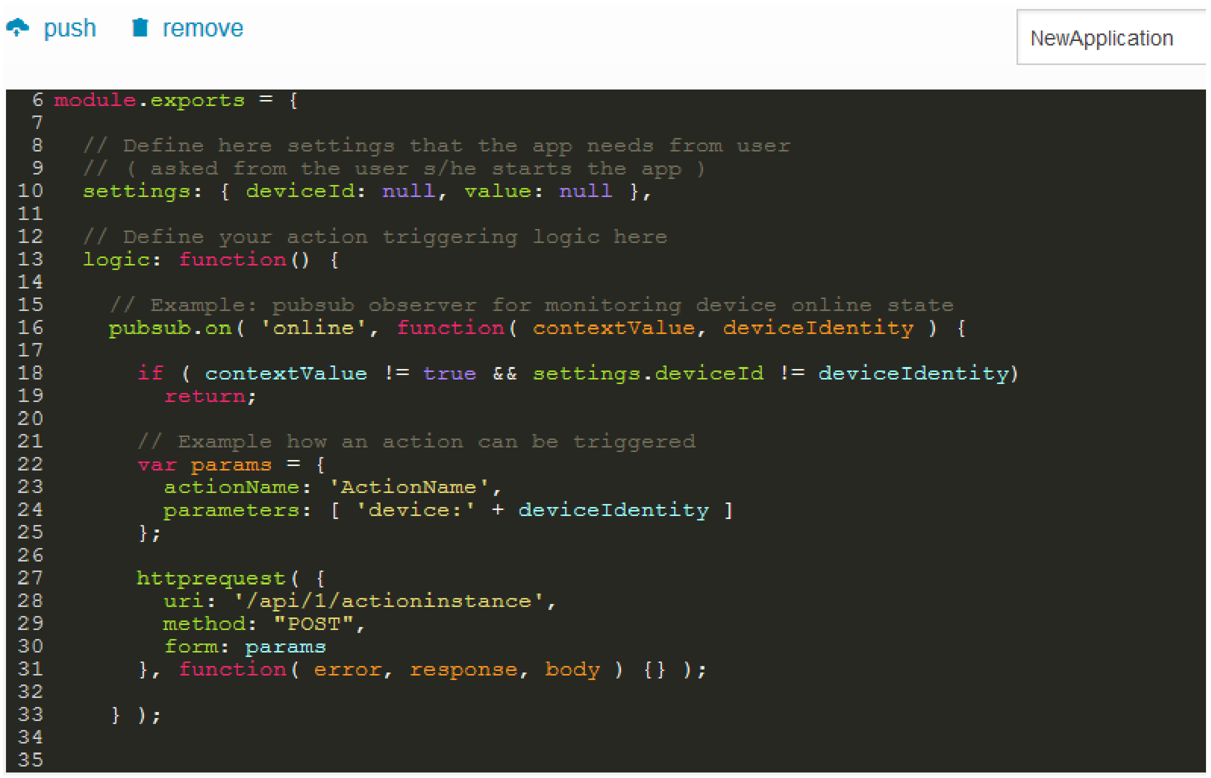
☐ Prevent this page from creating additional dialogues

Figure 12. When the user runs the application, it will be asked to introduce all the parameters defined in the *settings* object.

In the example in [Figure 13] I have defined an observer [Line 16] which is reading the *online* context data. This data is automatically sent to true when a new device is connected to the platform.

If the device is connected (if the contextValue is set to *true*), and the identity of the connected device is the identity the user has specified in the parameter *deviceId*, the program will continue, otherwise it will return without executing any instruction.

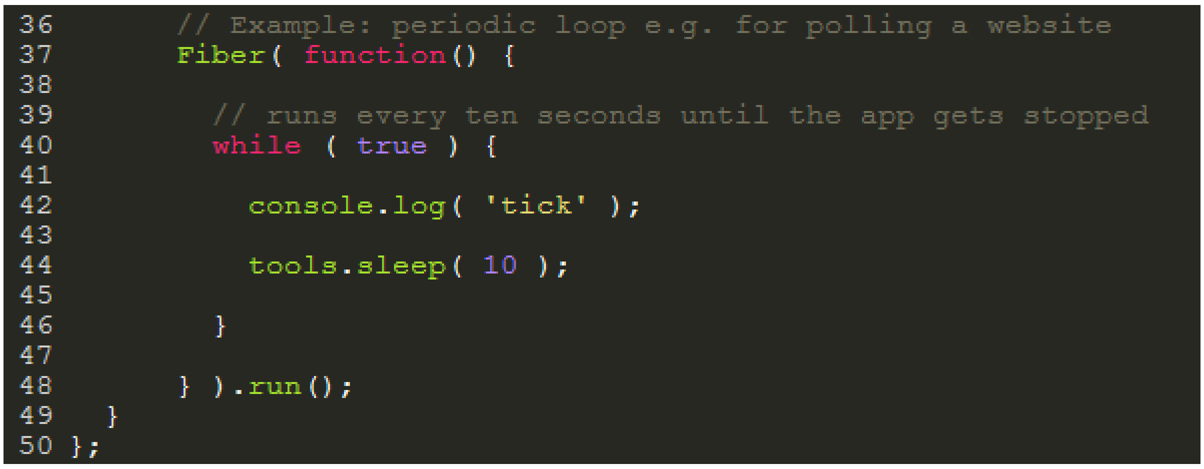
Then, the application will call the REST API using the *httprequest* module, executing the action *ActionName* on the device contained in the variable *deviceIdentity*.



```
 6 module.exports = {
 7
 8   // Define here settings that the app needs from user
 9   // ( asked from the user s/he starts the app )
10   settings: { deviceId: null, value: null },
11
12   // Define your action triggering logic here
13   logic: function() {
14
15     // Example: pubsub observer for monitoring device online state
16     pubsub.on( 'online', function( contextValue, deviceIdentity ) {
17
18       if ( contextValue != true && settings.deviceId != deviceIdentity)
19         return;
20
21       // Example how an action can be triggered
22       var params = {
23         actionName: 'ActionName',
24         parameters: [ 'device:' + deviceIdentity ]
25       };
26
27       httprequest( {
28         uri: '/api/1/actioninstance',
29         method: "POST",
30         form: params
31       }, function( error, response, body ) {} );
32
33     } );
34
35
```

Figure 13. Example application (part 1).

[Figure 14] shows the second part of the code. This part only creates a thread (*Fiber*) that simply prints the word *tick* in console every 10 seconds.



```
      // Example: periodic loop e.g. for polling a website
      Fiber( function() {

        // runs every ten seconds until the app gets stopped
        while ( true ) {

          console.log( 'tick' );

          tools.sleep( 10 );

        }

      } ).run();
  }
};
```

Figure 14. Simple example of a Fiber of the application (part 2).

Note: *Always remember to save your code before executing it. To save your code simply press the button 'Push'. If you change something in your code and you press the button 'Start' or 'Trigger' (for actions), it will execute the code saved in the server, not the code you are editing in your browser.*

Example 1

Application hiThere!

In this chapter I will introduce the first study application. The application *hiThere!* is the first of five applications to prove the potential of the platform. Also I will explain how the entire application, its actions, and its capabilities are developed, to help to understand the implementation process.

The goal of this application is to read the list of contacts from the device, and automatically send a notification to the user if one of his contacts is in the surroundings. This is achieved using a Bluetooth detection of nearby devices.

Each user will be asked to introduce his mobile phone number, this phone number will be mapped to the user's identity in the platform and, consequently, to his Bluetooth MAC address, which will be used to detect the proximity.

[Figure 15] shows how the application at high level. Now I will explain its details in deep, as actions, capabilities and context data.

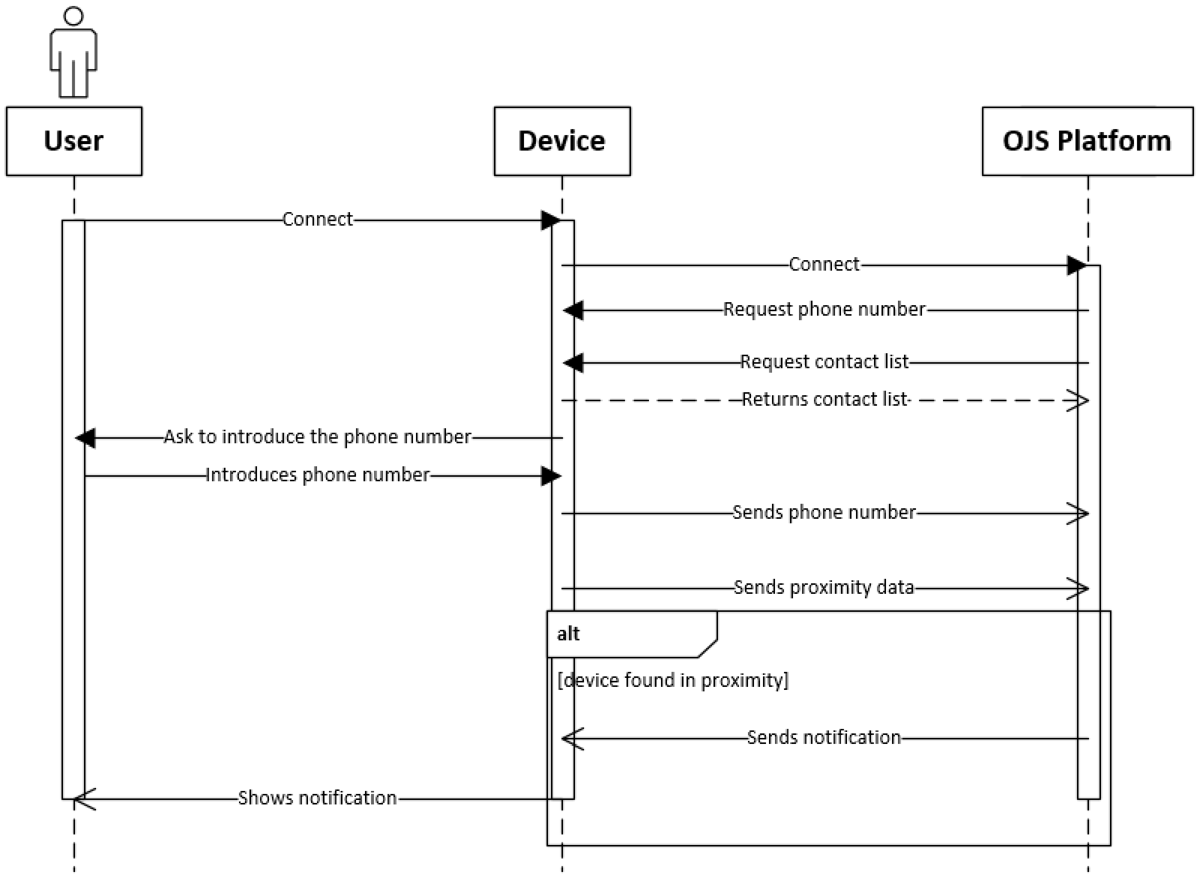


Figure 15. Activity diagram with the behaviour of the application *hiThere!*

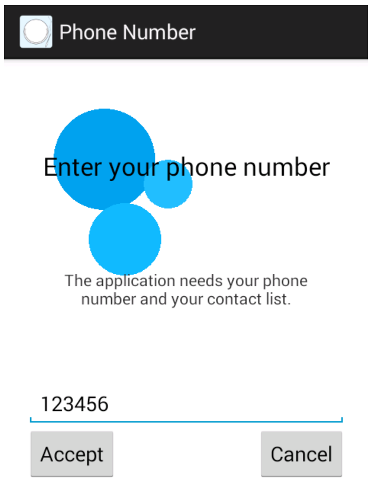
Implementation

I will explain now the most important things about the implementation. This application will use the actions: *PhoneDialog*, *GetContactList*, *SendNotification* and *UpdateProximity*.

The *PhoneDialog* action will call the method *askPhoneNumber* from the *phoneNumberCapability*.

```
public void askPhoneNumber( String message, Integer timeout ) throws Exception
{
    Log.d(OrchestratorJsActivity.TAG, "asking Phone Number!");

    // Create a new Intent to start the activity
    Intent i = new Intent(PhoneNumberCapability.applicationContext_, PhoneNumberCapabilityActivity.class);
    i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);

    // Send the arguments from the platform
    i.putExtra("phoneMessage", message);
    i.putExtra("timeout", timeout*1000);

    // Start the capability
    PhoneNumberCapability.applicationContext_.startActivity(i);
}
```

In the client this method simply starts a new Android's Activity, which will show a dialog to the user to input the mobile phone number. The platform sends two parameters: a *message* to show to the user, and a timeout, which indicates the number of seconds to automatically close the Activity if the user doesn't introduce the phone number.


Phone Number

Enter your phone number

The application needs your phone number and your contact list.

123456

The Activity will send the phone number introduced by the user in an asynchronous message, as a context data.

The code in [Figure 16] shows the method to send the context data to the platform.

```
try
{
    JSONObject sendPhone = new JSONObject();
    sendPhone.put("phoneNumber", phoneEdit.getText().toString());
    OrchestratorJsActivity.ojsContextData(sendPhone);

    SharedPreferences settings = getPreferences(0);
    SharedPreferences.Editor editor = settings.edit();
    editor.putString("phoneNumber", phoneEdit.getText().toString());

    editor.commit();
}
catch (JSONException e)
{
    e.printStackTrace();
}
finish();
```

Figure 16. Fragment of code that sends the *phoneNumber* introduced by the user as context data to the OrchestratorJS Platform.

```
// Devices without a valid phone number cannot participate in this app
pubsub.on( 'phoneNumber', function( contextValue, deviceIdentity )
{
      console.log(deviceIdentity + ": " + contextValue);
      phoneMap[deviceIdentity] = contextValue;
} );
```

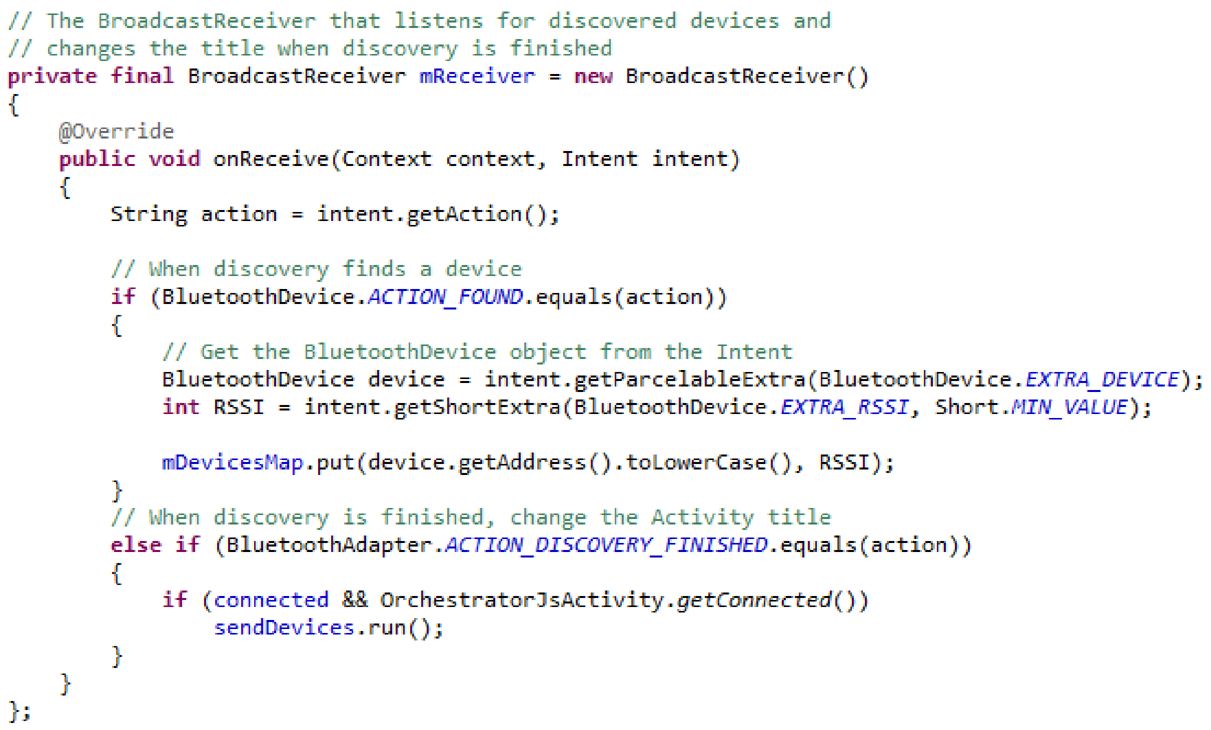
This is the code of the observer listening for the context data called *phoneNumber*, the application will receive the phone number introduced by the user and will store it in an object indexed with the device identity.

At the end of the application there is a Fiber that constantly executes updating the proximity of all connected devices. This will iterate through all the connected devices (devices which we have the phone number in the map) and request their Bluetooth proximity using the *UpdateProximity* action.

```
// Update Bluetooth visibility of all connected devices
Fiber( function()
{
   while ( true )
   {
      console.log( 'bluetooth tick' );

      // Iterate through all the devices and update the proximity.
      Object.keys(phoneMap).forEach(function(key)
      {
         console.log( "request proximity on " + key );

         var params = {
            actionName: 'UpdateProximity',
            parameters: [ 'device:' + key ]
          };

         httprequest( {
            uri: '/api/1/actioninstance',
            method: "POST",
            form: params
          }, function( error, response, body ) {} );

      });

      tools.sleep( 30 );
   }
} ).run();
```

BluetoothCapability is as I added Bluetooth support to the OrchestratorJS Platform. One of the lasts versions of the platform has added Bluetooth support too, but it is implemented as a service that runs constantly on the device. My capability is compatible with this method, but my capability allows the developer to ask for a Bluetooth update only when it is needed.

This capability has some initialization code in the Android device, that code is executed when the user connects to the platform. The most important piece of code in this capability is the Bluetooth discovery:



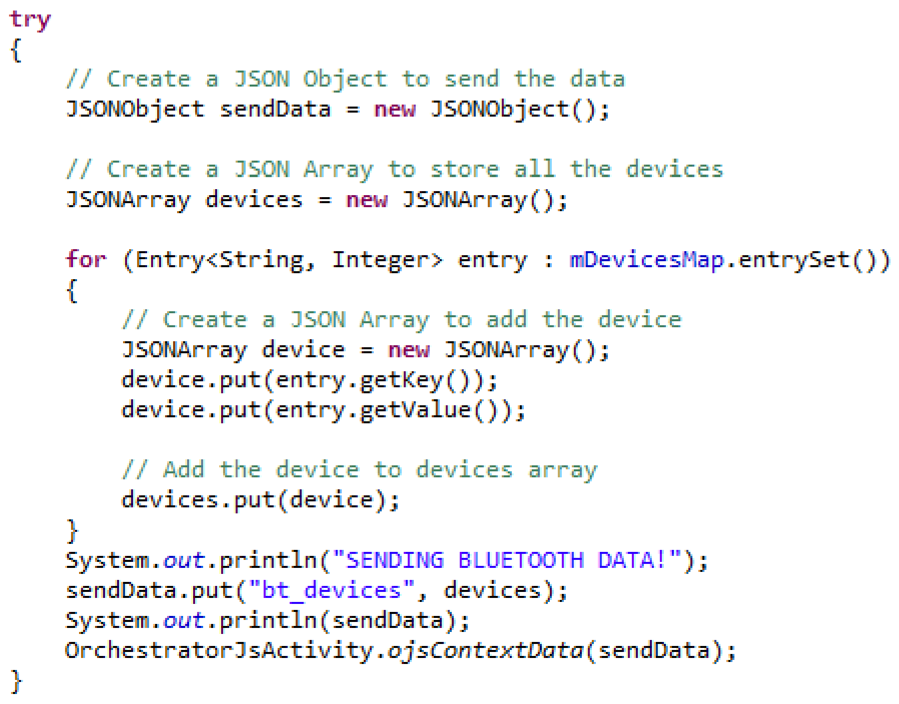
```
// The BroadcastReceiver that listens for discovered devices and
// changes the title when discovery is finished
private final BroadcastReceiver mReceiver = new BroadcastReceiver()
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        String action = intent.getAction();

        // When discovery finds a device
        if (BluetoothDevice.ACTION_FOUND.equals(action))
        {
            // Get the BluetoothDevice object from the Intent
            BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            int RSSI = intent.getShortExtra(BluetoothDevice.EXTRA_RSSI, Short.MIN_VALUE);

            mDevicesMap.put(device.getAddress().toLowerCase(), RSSI);
        }
        // When discovery is finished, change the Activity title
        else if (BluetoothAdapter.ACTION_DISCOVERY_FINISHED.equals(action))
        {
            if (connected && OrchestratorJsActivity.getConnected())
                sendDevices.run();
        }
    }
};
```

Figure 17. Piece of code that shows how the Bluetooth discovery is implemented.

This Receiver is executed for each discovered device in Android. What I do is storing all the discovered devices in the *mDevicesMap*, which stores the Bluetooth MAC address of the device and the RSSI, to be used as an approximation of the distance between both devices.

The capability has a timeout configured by the developer, after that timeout the client will send all the discovered devices. In the action *UpdateProximity* I configured this value as 10 seconds. That means that the device will send all the discovered devices after 10 seconds, but it will continue looking for devices and when it finishes will send all of them again.

The delivery is done in a similar way as the phone number, each device is a JSON Array with the Bluetooth MAC address and the RSSI. All the devices are stored in another JSON Array, and finally everything is wrapped into a JSON Object [Figure 18].



```
try
{
    // Create a JSON Object to send the data
    JSONObject sendData = new JSONObject();

    // Create a JSON Array to store all the devices
    JSONArray devices = new JSONArray();

    for (Entry<String, Integer> entry : mDevicesMap.entrySet())
    {
        // Create a JSON Array to add the device
        JSONArray device = new JSONArray();
        device.put(entry.getKey());
        device.put(entry.getValue());

        // Add the device to devices array
        devices.put(device);
    }
    System.out.println("SENDING BLUETOOTH DATA!");
    sendData.put("bt_devices", devices);
    System.out.println(sendData);
    OrchestratorJsActivity.ojsContextData(sendData);
}
```

Figure 18. Structure of the Bluetooth proximity message.

The OrchestratorJS Platform will now receive the object *bt_devices* internally (it is not parsed in my application). It will automatically translate the Bluetooth MAC addresses into Device Identities of the registered devices in the platform.

```
// Request data from proximity
pubsub.on( 'proximityDevices', function ( contextValue, deviceIdentity )
{
   for (var i = 0; i < contextValue.length;i++)
   {
      var device = contextValue[i][0];
      var phoneNum = phoneMap[deviceIdentity];

      // Check if the user has me in his contact list.
      if (contactList[device] != null)
      {
         for (var p = 0; p < contactList[device].length; p++)
         {
            // I have already sent a notification to this device.
            if (sendedMap[device] == deviceIdentity)
              continue;

            if (phoneNum == contactList[device][p])
            {
              sendedMap[device] = deviceIdentity;
              // Send notification to the device!
              // -- Rest of the code omitted for simplicity.
```

After parsing and translating the Bluetooth MAC addresses, it will broadcast the result in the context data *proximityDevices*. Now we can create an observer for this message in our application, which will read Device Identities instead of Bluetooth MAC addresses, what simplifies the process of searching nearby devices.

Now (see the simplified code in the previous page) I will check what of the proximity devices are in the contact list, the application will iterate through all the discovered devices checking if they are in the contact list and if I have already sent a notification. Please note that if the device is not connected to the application, its phone number will be null, so the notification will be never sent.

```
var request = require("request")
var url = "http://192.168.1.180:9000/api/1/devices"

request({
   url: url,
   json: true
   }, function (error, response, body)
      {
         if (!error && response.statusCode === 200)
         {
            for (var i = 0; i < body.devices.length; i++)
            {
               // This device
               if (body.devices[i]["identity"] == deviceIdentity)
               {
                  phoneMap[deviceIdentity] =
                          body.devices[i]["metadata"]["phoneNumber"];
               }
            }
         }

         // Request the phone number to the device if we don't have it
         if (phoneMap[deviceIdentity] == undefined ||
                  phoneMap[deviceIdentity] == null ||
                  phoneMap[deviceIdentity] == "")
         {
             // Call the PhoneDialog Action (omitted for simplicity)
         }
     });

     // Request the contact list to the device (GetContactList action)
```

The first part of this code will request the API with all the RAW JSON Data from the Platform (see [**Error! Reference source not found.**] in [Page **Error! Bookmark not defined.**]). Then, all that data is parsed to check if we have already stored the user's phone number in the platform. If we have the phone number I will read it instead of asking to the user to enter its number again. If the user changes his phone number he can delete it from the *Device Settings* in the Web Console, and the application will ask for the number again when he connects the device.

If the phone number was not found, the application will call the action *PhoneDialog* and the user will be able to enter it.

Finally, at the end of the code it requests the contact list to the device. This action will call the capability *ContactListCapability*, which will read the list of contacts from the device.

```
public void getContactList()
{
    ArrayList<String> phoneNumbers = new ArrayList<String>();
    ContentResolver cr = applicationContext_.getContentResolver();
    Cursor phones = cr.query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
            null,null,null, null);
    while (phones.moveToNext())
    {
      /*String name = phones.getString(phones.getColumnIndex(
        ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME));*/
      String phoneNumber = phones.getString(phones.getColumnIndex(
              ContactsContract.CommonDataKinds.Phone.NUMBER));
      phoneNumbers.add(phoneNumber);
    }

    phones.close();  // close cursor
```

This is how the list of contacts is read. They are stored in the *phoneNumbers* ArrayList, and then it is sent to the platform as a context data in a JSON Object.

Note: *In this application we use sensitive data from the user, as his phone number and contact list. Take note that hi can disable these capabilities in any moment, so the data won't be requested neither sent to the server.*

Conclusions

This application makes use of several capabilities to provide a new functionality to the user: the possibility to receive an alert when one of his contacts is in his surroundings.

Example 2

Application eGreetings

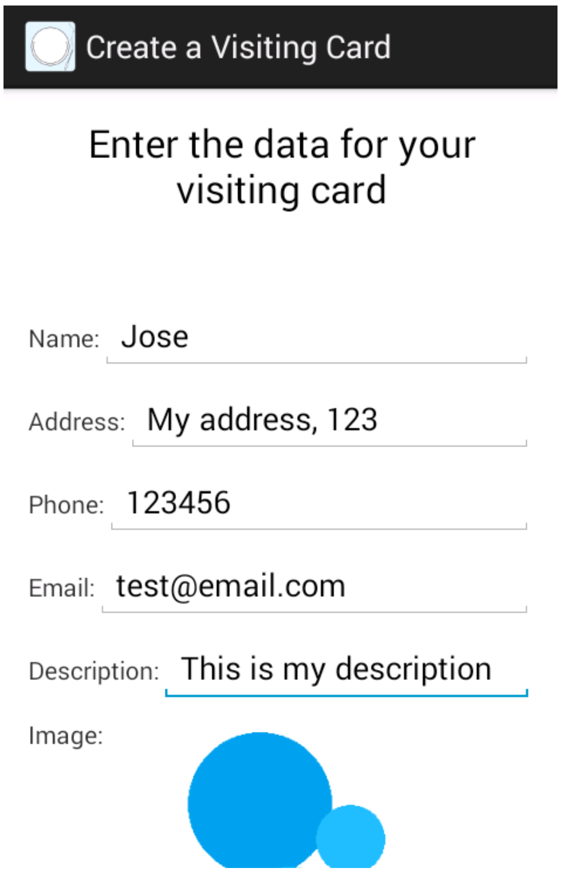
This is the second study application. The focus of this application is that when two people meet, their mobile device will automatically share their visiting cards each other. This interaction will be automatic and the user won't be required to do the interaction, he only needs to connect the device to the platform and set up the visiting card.

The behaviour of this application is simpler than the last one, however the difficult here is the devices have to share a bigger amount of data, which can include images.

Implementation

This application makes use of the actions: *RequestVisitingCard*, *ShowVisitingCard* and *UpdateProximity*:

- *RequestVisitingCard*: This is an action similar to *PhoneDialog* in the last one application. This action will call the method *requestVisitingCard* from the *visitingCardCapability*. The method simply creates a new activity where the user can introduce the data to create the visiting card.
- *ShowVisitingCard*: This action is which will send the visiting card of a user to another users in the proximity. This action has seven arguments, the first of them is the device identity to send the visiting card, and the rest of arguments are all the data of the visiting card. The last of these arguments corresponds to the image, which will be sent as a String in Base64 format [Figure 19].
- *UpdateProximity*: This is the same method as the first application and its behaviour is exactly the same.



Enter the data for your visiting card

Name: Jose

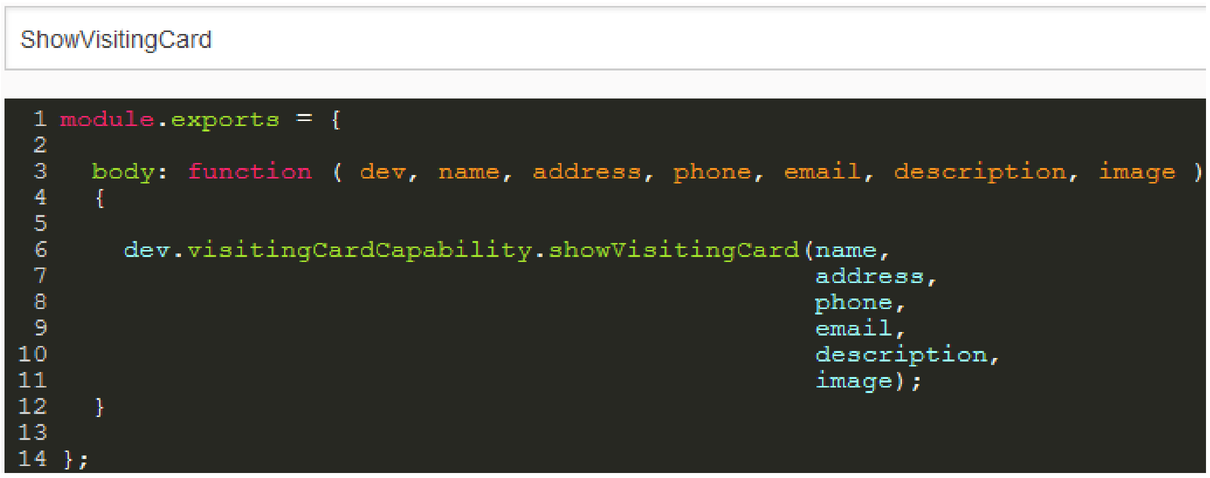
Address: My address, 123

Phone: 123456

Email: test@email.com

Description: This is my description

Image:


ShowVisitingCard

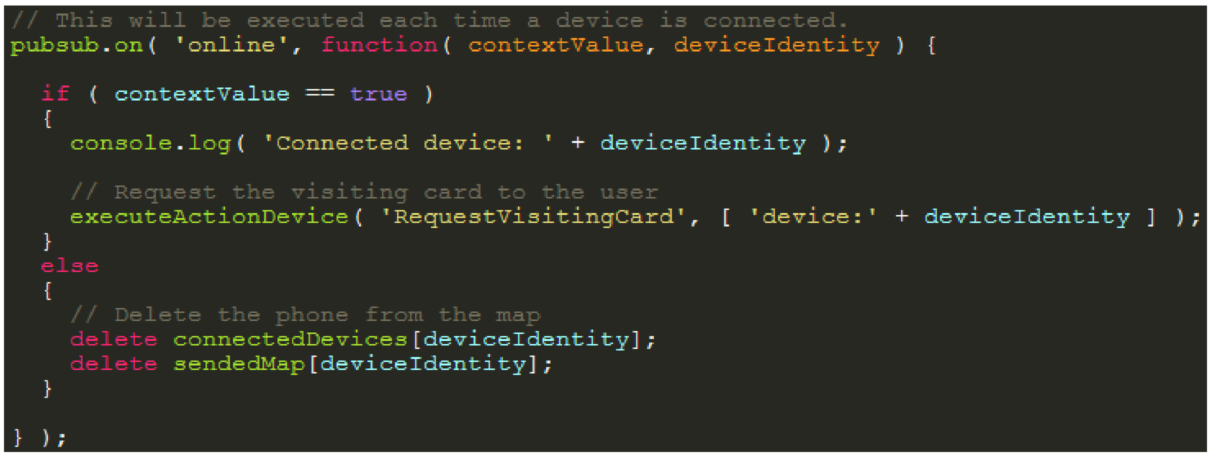
```
module.exports = {

  body: function ( dev, name, address, phone, email, description, image )
  {

    dev.visitingCardCapability.showVisitingCard(name,
                                                address,
                                                phone,
                                                email,
                                                description,
                                                image);
  }

};
```

Figure 19. ShowVisitingCard action implementation.

To simplify the code, I have created a new function called *executeActionDevice*, which will be present in all the applications. This method encapsulates the code to call the REST API to execute an action:

```
function executeActionDevice( actionName, parameters )
{
  var params = {
    actionName: actionName,
    parameters: parameters
  };

  httprequest( {
    uri: '/api/1/actioninstance',
    method: "POST",
    form: params
  }, function( error, response, body ) {} );
}
```

The first part of the application implements an observer as in the *hiThere!* application. This observer is executed each time a device is connected or disconnected from the platform. If the device is connection (the contextValue is set to *true*), the application will call the action *RequestVisitingCard* to ask the user create a new card.



```
// This will be executed each time a device is connected.
pubsub.on( 'online', function( contextValue, deviceIdentity ) {

  if ( contextValue == true )
  {
    console.log( 'Connected device: ' + deviceIdentity );

    // Request the visiting card to the user
    executeActionDevice( 'RequestVisitingCard', [ 'device:' + deviceIdentity ] );
  }
  else
  {
    // Delete the phone from the map
    delete connectedDevices[deviceIdentity];
    delete sendedMap[deviceIdentity];
  }

} );
```

Figure 20. Implementation of the first observer.

If the device is disconnecting from the platform, the application will delete all the stored data (the objects *sendedMap* and *connectedDevices*).

When the user creates a new visiting card, it is sent to the platform as context data called *visitingCard*. This data contains all the fields and the image in Base64 format:

```
private void sendVisitingCard()
{
	String name = tname.getText().toString();
	String address = taddress.getText().toString();
	String phone = tphone.getText().toString();
	String description = tdescription.getText().toString();
	String email = temail.getText().toString();

	byte[] b = {};
	Bitmap bm = BitmapFactory.decodeFile(selectedImagePath);
	if (bm == null)
		bm = ((BitmapDrawable)timage.getDrawable()).getBitmap();

	ByteArrayOutputStream baos = new ByteArrayOutputStream();
	bm.compress(Bitmap.CompressFormat.JPEG, 50, baos);
	b = baos.toByteArray();

	String image = Base64.encodeToString(b, Base64.DEFAULT);

	try
	{
		// Object to be sent
		JSONObject sendData = new JSONObject();

		// Visiting card
		JSONObject visitingCard = new JSONObject();

		visitingCard.put("name", name);
		visitingCard.put("address", address);
		visitingCard.put("phone", phone);
		visitingCard.put("email", email);
		visitingCard.put("description", description);
		visitingCard.put("image", image);

		sendData.put("visitingCard", visitingCard);
		OrchestratorJsActivity.ojsContextData(sendData);
	}
	catch (JSONException ex)
	{
		ex.printStackTrace();
		Toast.makeText(this, "Error!", Toast.LENGTH_SHORT).show();
	}

	Toast.makeText(this,     "Visiting     Card     created     succesfully!",
Toast.LENGTH_SHORT).show();
}
```

Then, the server will receive this JSON Object and store the visiting card in the *connectedDevices* map, with the device identity as key. This data is stored in the server to facilitate the share of the information between devices without having to request the information from the user each time. However the data is removed from the server when the user disconnects.

Finally, the last new part in this code is the *proximityDevices*. This code works in a similar way to the code in the last application, however in this case the visiting card will be sent to every user (connected to the platform) in proximity. Also I have added a configurable timeout in settings, set by default to 10 minutes. This means that the visiting card will be sent maximum once each 10 minutes.

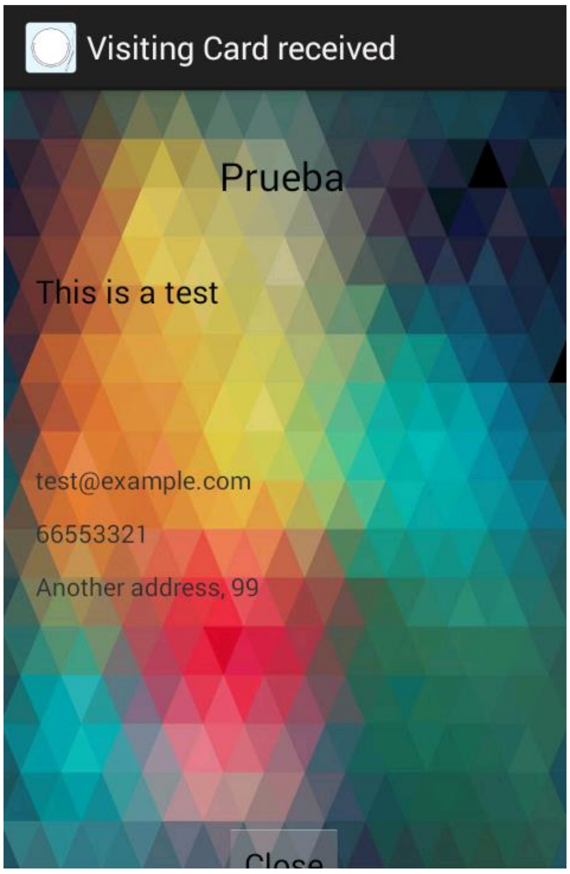
```
// Request data from proximity
pubsub.on( 'proximityDevices', function ( contextValue, deviceIdentity )
{
   if (deviceIdentity in connectedDevices)
   {
      // My visitingCard to send
      var visitingCard = connectedDevices[deviceIdentity];

      for (var i = 0; i < contextValue.length;i++)
      {
         var device = contextValue[i][0];

         if (device in connectedDevices)
         {
            var now = new Date().getTime();
            if (sendedMap[deviceIdentity] != null &&
                sendedMap[deviceIdentity][device] != null &&
                sendedMap[deviceIdentity][device] > now)
            {
                 console.log("I have already sent to this device!!");
            }
            else
            {
               // Don't send to this device in at least 10 minutes
               sendedMap[deviceIdentity][device] = new Date().getTime() +
                                                    settings.timeout;

               var parameters = [ 'device:' + device, visitingCard.name,
                           visitingCard.phone, visitingCard.address,
                           visitingCard.email, visitingCard.description,
                           visitingCard.image ]

               executeActionDevice( 'ShowVisitingCard', parameters );
```

Note that the code has been simplified, I have removed logs and debug information not needed in this example.

Actions can receive only primitive data (*Strings*, *Integers*, *Doubles*, *Floats*, and *Booleans*) as parameters, they don't allow JSON Objects as parameters, which is the reason I have to send all the fields in the visiting card one by one instead of sending the entire object.

When all the requirements are met the action *ShowVisitingCard* is executed in the target device, showing my visiting card to the other device.

Each device will send the visiting card based on his own proximity. This means that when two devices meet, maybe they don't interchange their visiting cards in the same moment. Some devices can take more time to discover nearby devices.



Prueba

This is a test

test@example.com

66553321

Another address, 99

[Figure 21] shows the code from *ShowVisitingCardActivity* which takes all the arguments, including the image, and shows the visiting card to the user.

```
String name = args.getStringExtra("name");
String phone = args.getStringExtra("phone");
String address = args.getStringExtra("address");
String description = args.getStringExtra("description");
String email = args.getStringExtra("email");
String image = args.getStringExtra("image");

tname.setText(name != null ? name : "");
tphone.setText(phone != null ? phone : "");
taddress.setText(address != null ? address : "");
tdescription.setText(description != null ? description : "");
temail.setText(email != null ? email : "");

if (image != null)
{
    try
    {
        byte[] decodedString = Base64.decode(image, Base64.DEFAULT);
        Bitmap decodedByte = BitmapFactory.decodeByteArray(decodedString, 0, decodedString.length);
        timage.setImageBitmap(decodedByte);
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}
```

Figure 21. Example of code to read all the arguments and show the visiting card to the user. The image is sent as a String codified in Base64. This string is decoded and showed in an ImageView.

Conclusions

This application can be very useful to businessmen who have a lot of meetings in a day. Their mobile devices can send the visiting card automatically to everybody in their surroundings. This can be a cheap way to do publicity.

For example, the application could run in a small device at the entrance of a shop, and it will send the visiting card to the people who pass in front of the shop. This system could be used to offer discount coupons to clients or to make small adverts.

Of course, some people can think this may be disturbing, however they can disable the capability in any moment and they will stop receiving visiting cards.

Example 3

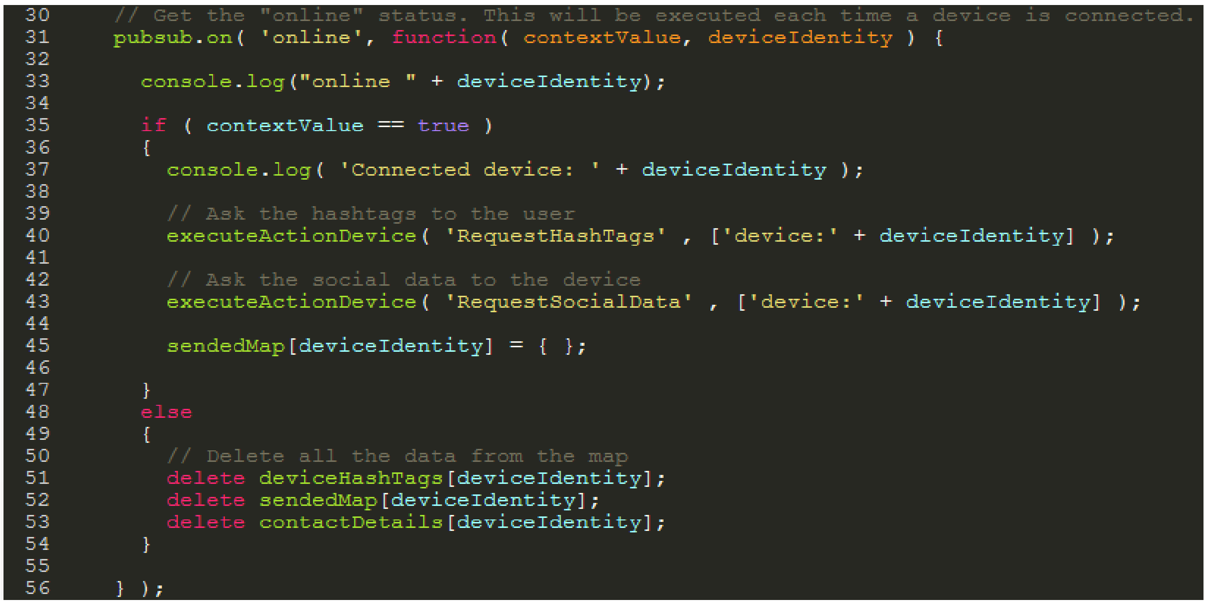
Application followMe

FollowMe is the third application. This basic application asks the user to enter their preferences in a hashtag way. Then, the application will get all the user's hashtags and match them based on the devices in your proximity.

With this application you can easily find people with your interests in your surroundings. When the devices find one or more common hashtags, they ask the users if they want to share their data with the other user. If both users answer "yes", their social network data will be shared each other.

Implementation

When a new device connects to the application, it automatically sends two actions. The first, *RequesthashTags*, is a view to request the user to introduce his interests. The second action, *RequestSocialData*, will request the social network data of the user.



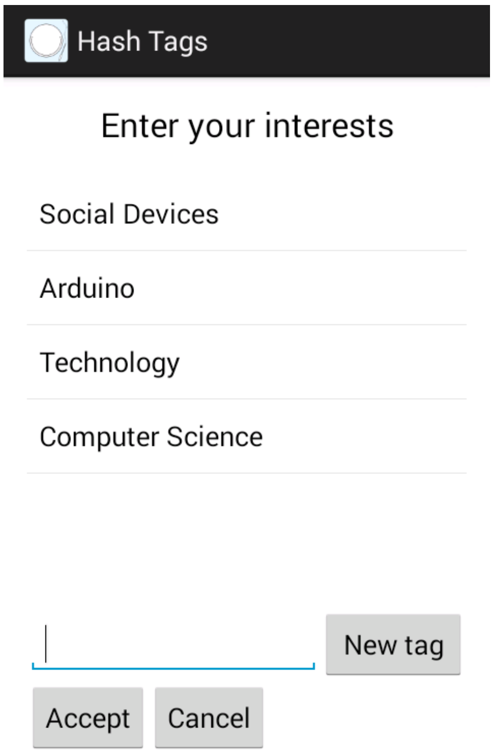
```
// Get the "online" status. This will be executed each time a device is connected.
pubsub.on( 'online', function( contextValue, deviceIdentity ) {

  console.log("online " + deviceIdentity);

  if ( contextValue == true )
  {
    console.log( 'Connected device: ' + deviceIdentity );

    // Ask the hashtags to the user
    executeActionDevice( 'RequestHashTags' , ['device:' + deviceIdentity] );

    // Ask the social data to the device
    executeActionDevice( 'RequestSocialData' , ['device:' + deviceIdentity] );

    sendedMap[deviceIdentity] = { };

  }
  else
  {
    // Delete all the data from the map
    delete deviceHashTags[deviceIdentity];
    delete sendedMap[deviceIdentity];
    delete contactDetails[deviceIdentity];
  }

} );
```

Figure 22. First part of the *followMe* code application with the login observer.

This data will be sent by the Android client, and receive by two different observers:

```
// Observer for hashTags
pubsub.on( 'hashTags', function( contextValue, deviceIdentity )
{
   deviceHashTags[deviceIdentity] = contextValue;
   console.log("hashTags " + deviceHashTags[deviceIdentity]);
} );
// Observer for socialData
```

```
pubsub.on( 'socialData', function( contextValue, deviceIdentity )
{
   contactDetails[deviceIdentity] = contextValue;
   console.log("socialData " + contactDetails[deviceIdentity]);
} );
```



The user has no limit in the number of hashtags he can use. To add a new hashtag he has to simply press the button *New tag*. To remove a hashtag he can swipe it from the list.

Used hashtags are also saved in the Android client, so the user doesn't have to specify them when he disconnects and connects again. They will be there and the user can send them back to the server of edit them before delivering.

Finally, this application also uses the *UpdateProximity* action, which updates the proximity of all connected devices each 20 seconds.

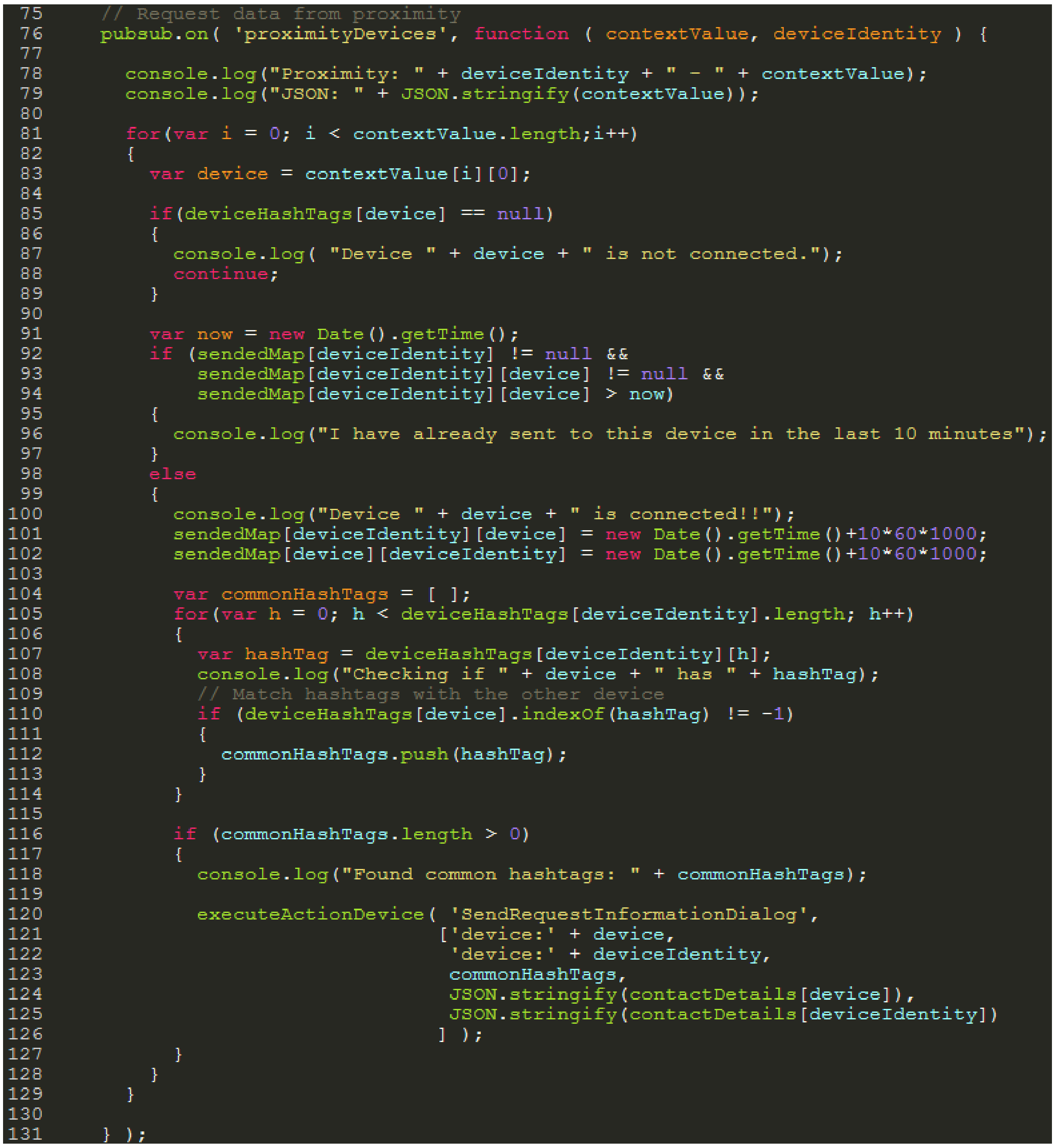
This is the code to retrieve the saved social networks from the Android device:

```
public void requestDetails() throws Exception
{
   Log.d(OrchestratorJsActivity.TAG, "requesting Social Network data!");

   try
   {
      JSONObject sendData = new JSONObject();
      JSONArray socialData = new JSONArray();

      Account[] accounts = AccountManager.get(applicationContext_).getAccounts();
      for (Account account : accounts)
      {
         if (account.type.equalsIgnoreCase("com.twitter.android.auth.login"))
         {
            JSONObject cuenta = new JSONObject();
            cuenta.put("twitter", account.name);
            socialData.put(cuenta);
         }
         else if (account.type.equalsIgnoreCase("com.facebook.auth.login"))
         {
            JSONObject cuenta = new JSONObject();
            cuenta.put("facebook", account.name);
            socialData.put(cuenta);
         }
     }
     sendData.put("socialData", socialData);
     OrchestratorJsActivity.ojsContextData(sendData);
```

This code will read your social network data from *Twitter* and *Facebook*. Support for more accounts can be provided, and an option to let the user which of them he prefers to share or not.

[Figure 23] shows the main implementation of the *proximityDevices* observer. This is the main code of the application. This observer will iterate all the nearby devices reported by your device, comparing their hashtags with yours to find potential matches.

It has a cool down timer like the last application to avoid sending the notification several times to the same user.



```
// Request data from proximity
pubsub.on( 'proximityDevices', function ( contextValue, deviceIdentity ) {

  console.log("Proximity: " + deviceIdentity + " - " + contextValue);
  console.log("JSON: " + JSON.stringify(contextValue));

  for(var i = 0; i < contextValue.length;i++)
  {
    var device = contextValue[i][0];

    if(deviceHashTags[device] == null)
    {
      console.log( "Device " + device + " is not connected.");
      continue;
    }

    var now = new Date().getTime();
    if (sendedMap[deviceIdentity] != null &&
        sendedMap[deviceIdentity][device] != null &&
        sendedMap[deviceIdentity][device] > now)
    {
      console.log("I have already sent to this device in the last 10 minutes");
    }
    else
    {
      console.log("Device " + device + " is connected!!");
      sendedMap[deviceIdentity][device] = new Date().getTime()+10*60*1000;
      sendedMap[device][deviceIdentity] = new Date().getTime()+10*60*1000;

      var commonHashTags = [ ];
      for(var h = 0; h < deviceHashTags[deviceIdentity].length; h++)
      {
        var hashTag = deviceHashTags[deviceIdentity][h];
        console.log("Checking if " + device + " has " + hashTag);
        // Match hashtags with the other device
        if (deviceHashTags[device].indexOf(hashTag) != -1)
        {
          commonHashTags.push(hashTag);
        }
      }

      if (commonHashTags.length > 0)
      {
        console.log("Found common hashtags: " + commonHashTags);

        executeActionDevice( 'SendRequestInformationDialog',
                            ['device:' + device,
                             'device:' + deviceIdentity,
                             commonHashTags,
                             JSON.stringify(contextDetails[device]),
                             JSON.stringify(contextDetails[deviceIdentity])
                            ] );
      }
    }
  }

} );
```

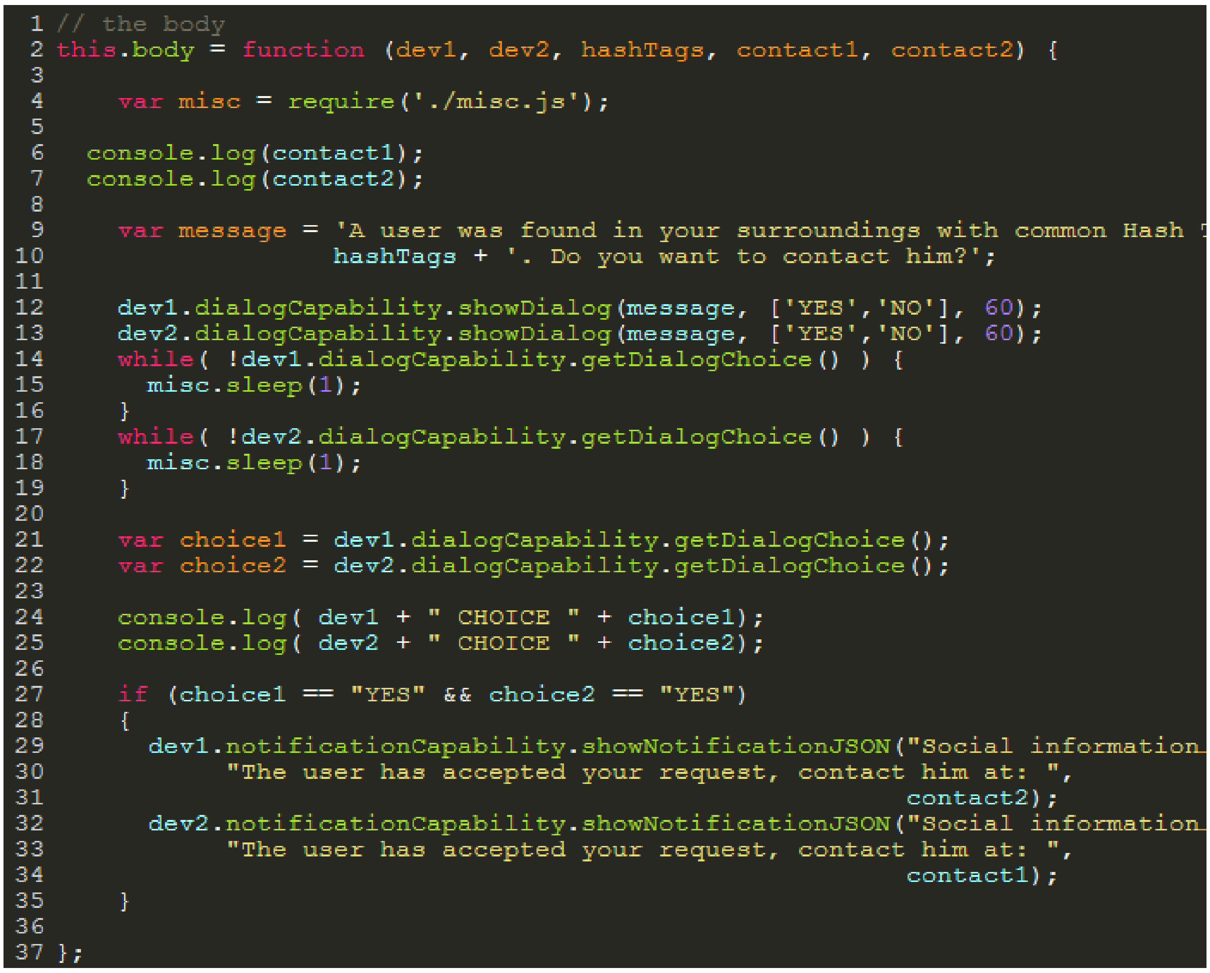
Figure 23. Main implementation of the *proximityDevices* observer.

However, this application doesn't user your proximity data to send your information (like the eGreetings app). In this case, the first device who finds the potential hashtag match will start the action. The timer will be set in both devices, to avoid to call the action again when the second device's proximity sensor finds the current device.

Once the application has found all the common hashtags, it will call the action *SendRequestInformationDialog* with 5 parameters: the device identity I have found, my own device identity, the list of common hashtags, the JSON Object converted to String (Stringified) with the contact details of the first device, and the contact details of myself.

[Figure 24] shows the implementation code of the action. This action will send a dialog to both devices asking if they want to share their contact details with each other. This is done using the capability *DialogCapability*, one of the original capabilities from the OrchestratorJS Platform. This capability will create a small dialog with the options *YES* and *NO*, with a timeout of 60 seconds.

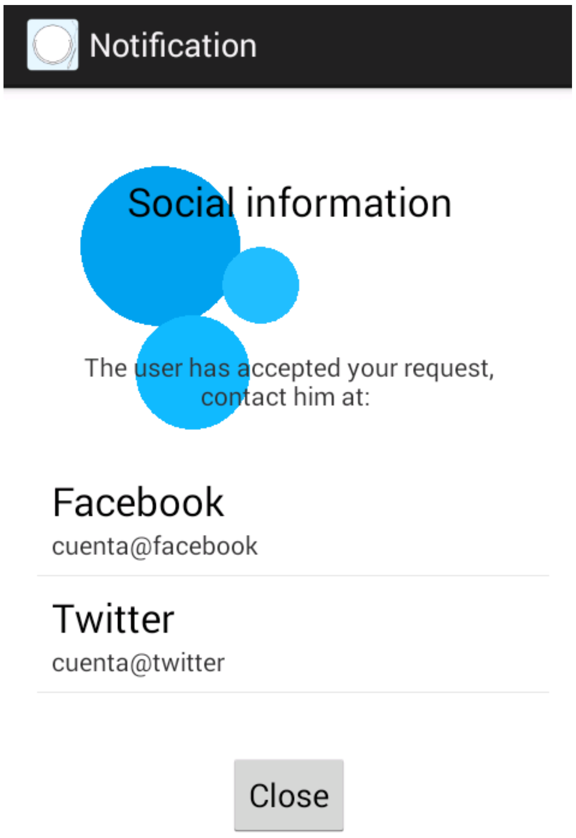
If both users answer *YES* to the sharing of contact details, both devices will share the information each other and the action will finish.



```
// the body
this.body = function (dev1, dev2, hashTags, contact1, contact2) {

    var misc = require('./misc.js');

  console.log(contact1);
  console.log(contact2);

    var message = 'A user was found in your surroundings with common Hash T
                   hashTags + '. Do you want to contact him?';

    dev1.dialogCapability.showDialog(message, ['YES','NO'], 60);
    dev2.dialogCapability.showDialog(message, ['YES','NO'], 60);
    while( !dev1.dialogCapability.getDialogChoice() ) {
      misc.sleep(1);
    }
    while( !dev2.dialogCapability.getDialogChoice() ) {
      misc.sleep(1);
    }

    var choice1 = dev1.dialogCapability.getDialogChoice();
    var choice2 = dev2.dialogCapability.getDialogChoice();

    console.log( dev1 + " CHOICE " + choice1);
    console.log( dev2 + " CHOICE " + choice2);

    if (choice1 == "YES" && choice2 == "YES")
    {
      dev1.notificationCapability.showNotificationJSON("Social information
           "The user has accepted your request, contact him at: ",
                                                        contact2);
      dev2.notificationCapability.showNotificationJSON("Social information
           "The user has accepted your request, contact him at: ",
                                                        contact1);
    }

};
```

Figure 24. Implementation of the action *SendRequestInformationDialog*.

As we have seen in previous examples, requesting data directly from a capability is not synchronized, what means that the action won't wait for the response. That means that the action has to poll it constantly until it receives the response. This is achieved using the *sleep* method from the *misc.js* library, which will sleep the running action for 1 second.


Notification

Social information

The user has accepted your request, contact him at:

Facebook
cuenta@facebook

Twitter
cuenta@twitter

Conclusions

This application can be useful to meet new people with same interests. Probably every day when you are going to the work, shopping, or home, you pass near people with your same interests.

This system can be very useful to help people in the street. If someone is looking for something he can configure an alert in the device, for example, he is looking for a room to rent in the neighborhood. Probably you are renting a room, so you can configure the advert in your device and when you are nearby in the street Social Devices will do the rest to join you.

Example 4

Application haveYouMet

Imagine you are in a big concentration of people and you are looking for somebody, which could be a friend or a new person you don't know. Social Devices can help you to find that person using the *haveYouMet* application.

This application is focused in meet people when you are near them but you don't know where. It uses GPS technology combined with Bluetooth to provide indications of proximity about the person you are trying to locate.

Implementation

This application requires the creation of a profile with the name and the photo of the user. This profile is requested when the user connects to the application [Figure 66].

When the user has sent the profile, he receives a list with all the users connected to the application. If he is looking for somebody he simply has to select that person in the list

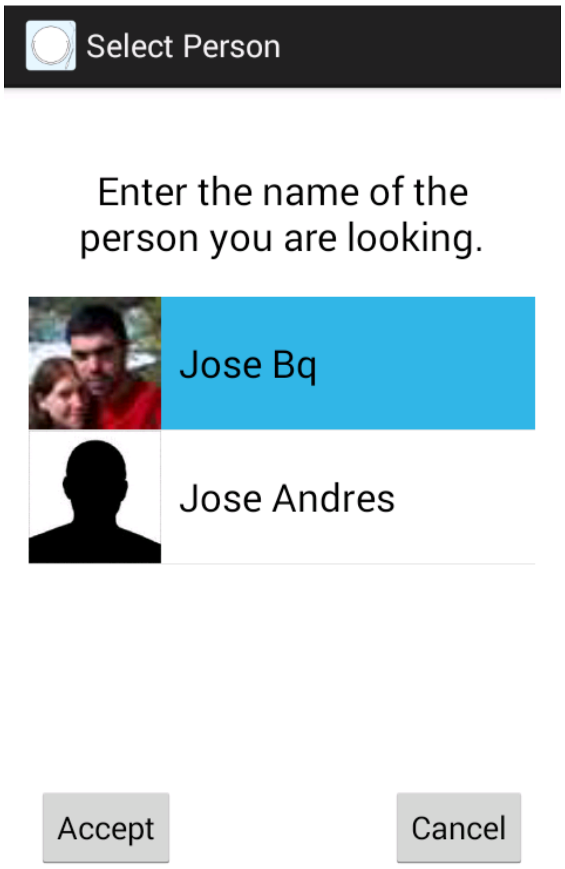


Figure 26. Creation of a new profile.

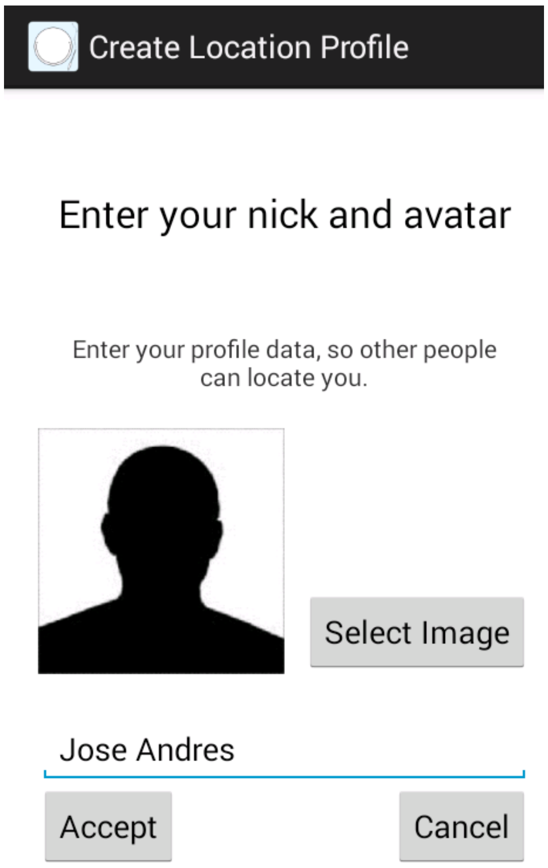
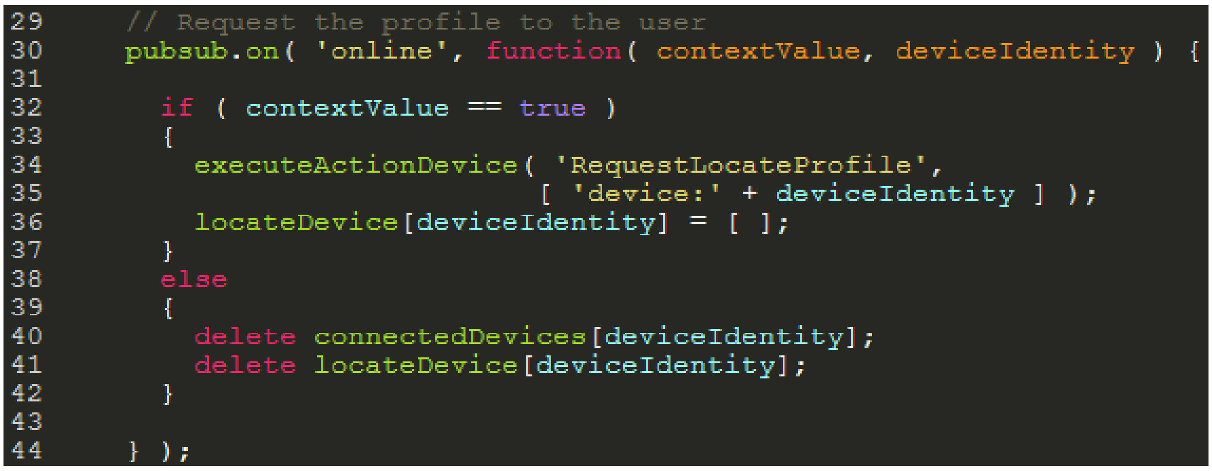

Create Location Profile
Enter your nick and avatar
Enter your profile data, so other people can locate you.
Select Image
Jose Andres
Accept
Cancel

Figure 25. View to select a person to locate.

[**Error! Reference source not found.**], and the application will help him to locate that person.



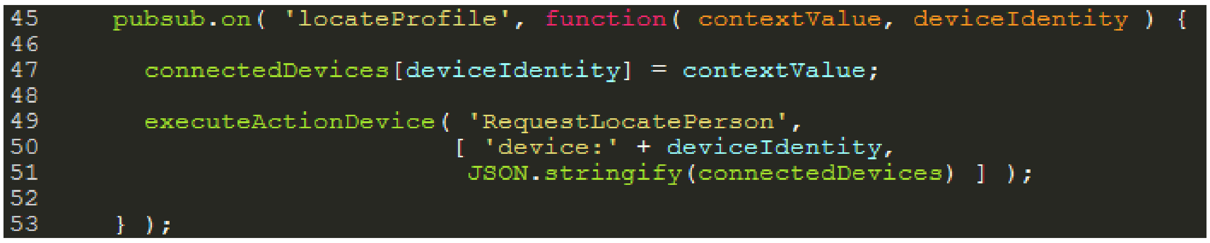
```
    // Request the profile to the user
    pubsub.on( 'online', function( contextValue, deviceIdentity ) {

      if ( contextValue == true )
      {
        executeActionDevice( 'RequestLocateProfile',
                             [ 'device:' + deviceIdentity ] );
        locateDevice[deviceIdentity] = [ ];
      }
      else
      {
        delete connectedDevices[deviceIdentity];
        delete locateDevice[deviceIdentity];
      }

    } );
```

Figure 27. Fragment of code that requests the profile to the connected device.

This application uses the actions:

- *RequestLocateProfile*: used to ask for the user profile to the device. It will show the Activity where the user can introduce his data [Figure 66].
- *RequestLocatePerson*: the device will show a view which contains information about all the connected people, where the user will be able to select a person from this list [**Error! Reference source not found.**].
- *RequestGpsPosition*: simply requests the GPS coordinates to the device, the response will be sent as asynchronous context data, which will be received by an observer in the application.
- *SetBluetoothTarget*: used to start locating a person. The device will receive information from the target device, as GPS coordinates and the Bluetooth MAC address of the target device.
- *SendNotification*: this is the simple notification message, it is used to notify the user if something was wrong trying to locate the device.
- *UpdateGPSLocation*: this action will send the current GPS coordinates of the device to all the devices trying to locate it.

The code executed when a new device connects to the application is shown in [Figure 27]. It simply requests the profile to the user. When the user disconnects all the information is removed from the application.

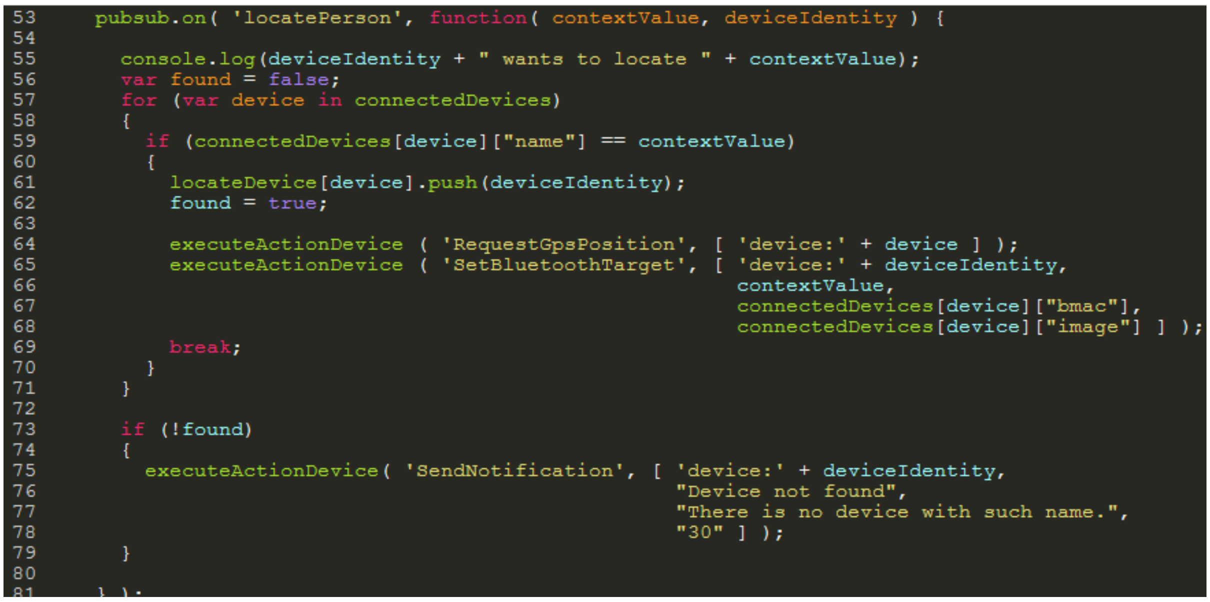


```
    pubsub.on( 'locateProfile', function( contextValue, deviceIdentity ) {

      connectedDevices[deviceIdentity] = contextValue;

      executeActionDevice( 'RequestLocatePerson',
                           [ 'device:' + deviceIdentity,
                            JSON.stringify(connectedDevices) ] );

    } );
```

Figure 28. Observer checking for the *locateProfile* context data.

The user's device will send the profile as context data called *locateProfile*. The observer in [Figure 28] will receive this data and store it. Then it will call the action *RequestLocatePerson*, sending a String with the content of the *connectedDevices* object, which contains all the information about the devices using the application.

The user can now select one of the connecter users from the list, and the server will receive the response as context data. If the user doesn't want to locate a person at this moment he can simply close or cancel the Activity.



```
    pubsub.on( 'locatePerson', function( contextValue, deviceIdentity ) {

      console.log(deviceIdentity + " wants to locate " + contextValue);
      var found = false;
      for (var device in connectedDevices)
      {
        if (connectedDevices[device]["name"] == contextValue)
        {
          locateDevice[device].push(deviceIdentity);
          found = true;

          executeActionDevice ( 'RequestGpsPosition', [ 'device:' + device ] );
          executeActionDevice ( 'SetBluetoothTarget', [ 'device:' + deviceIdentity,
                                                        contextValue,
                                                        connectedDevices[device]["bmac"],
                                                        connectedDevices[device]["image"] ] );

          break;
        }
      }

      if (!found)
      {
        executeActionDevice( 'SendNotification', [ 'device:' + deviceIdentity,
                                                   "Device not found",
                                                   "There is no device with such name.",
                                                   "30" ] );
      }

    } );
```

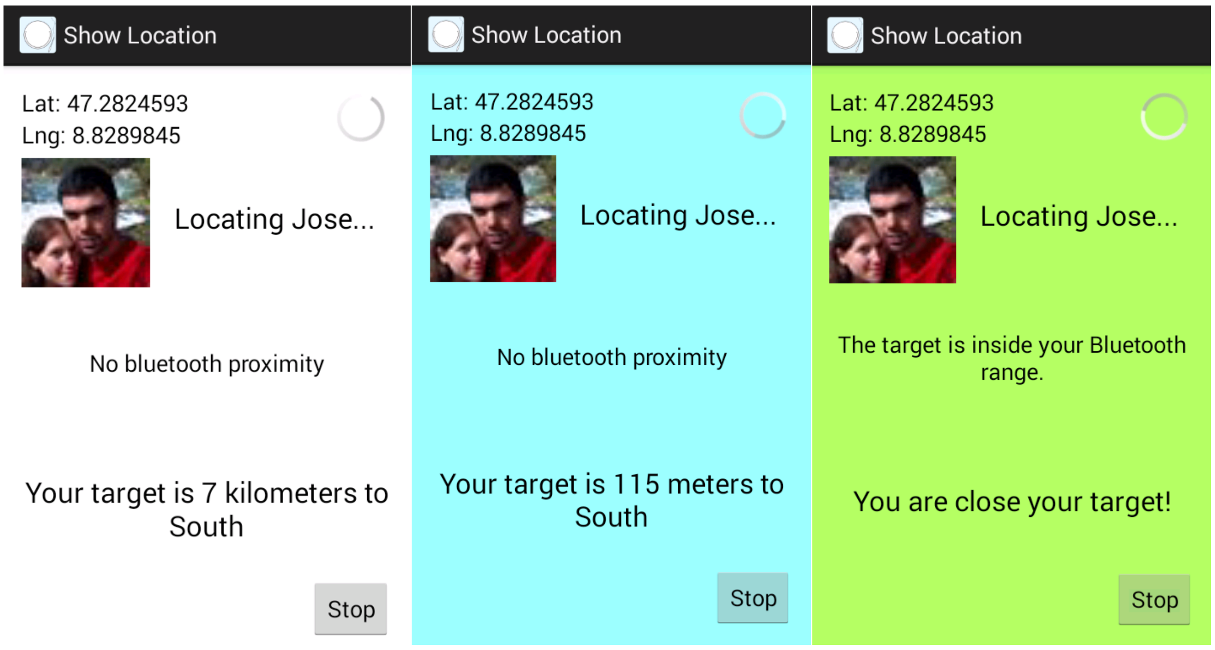
Figure 29. Code of the observer that receives the person you want to locate.

This figure shows the code of the observer, it looks for the user in the connected devices, because probably the user got disconnected while we were selecting him. If the user is connected, the application requests an update of his GPS Position, and send the information to our device.

The information received includes the name and the image of the user, and the Bluetooth MAC address of his device. This MAC address will be user to locate the device in out proximity.

Our device will now open a new Activity with some information of the user and the location data. Each minute we will receive the GPS coordinates of our target, and our device will calculate the distance to our target using our GPS position. Our device will also try to locate the device by the Bluetooth MAC address. The discovering of devices will be done each minute, however when we are getting closer conform the GPS distance, the Bluetooth discovering will update faster to try to locate the device.

The *locateDevice* will store all the devices currently trying to locate me. The object is indexed by the identity device who is trying to be located, and the values are all the devices trying to locate me.


Show Location
Lat: 47.2824593
Lng: 8.8289845
Locating Jose...
No bluetooth proximity
Your target is 7 kilometers to South
Stop

Show Location
Lat: 47.2824593
Lng: 8.8289845
Locating Jose...
No bluetooth proximity
Your target is 115 meters to South
Stop

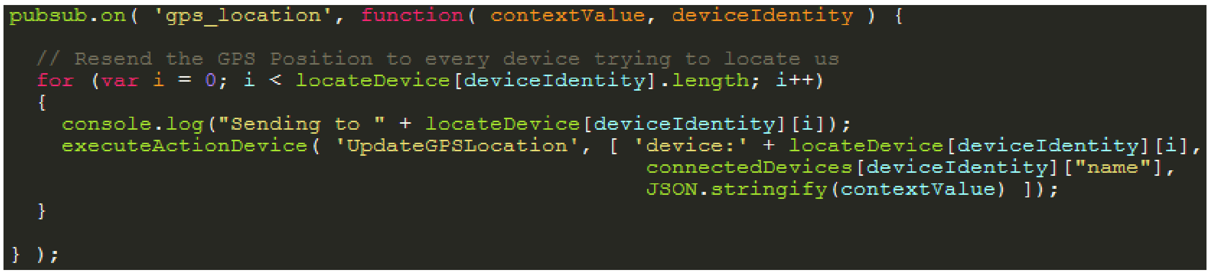
Show Location
Lat: 47.2824593
Lng: 8.8289845
Locating Jose...
The target is inside your Bluetooth range.
You are close your target!
Stop

Figure 30. Show location view in three different stages.

[Figure 30] shows three different stages from the application in the client. The first shows the screen when the target is very far (more than 200 meters). The second, when the target is closer, at this moment the background will change to blue, and the Bluetooth proximity check will be faster to try to locate the target device. Finally, when the device has been found by Bluetooth the background colour changes according the Bluetooth signal strength. It has 4 different states, from orange to green.

The code of this activity is bigger, so I won't comment it here. However it is not difficult to understand. The activity used is *MapLocateCapabilityActivity.java*.

Finally, the last interesting part of this application. There is a *Fiber* at the end of the script that requests the GPS position of all the connected devices. That request will provoke the client send a context data called *gps_location*. [Figure 31] shows the code of the observer. This observer will take the *locateDevice* object and send my GPS position to all the devices trying to locate me.



```
pubsub.on( 'gps_location', function( contextValue, deviceIdentity ) {

  // Resend the GPS Position to every device trying to locate us
  for (var i = 0; i < locateDevice[deviceIdentity].length; i++)
  {
    console.log("Sending to " + locateDevice[deviceIdentity][i]);
    executeActionDevice( 'UpdateGPSLocation', [ 'device:' + locateDevice[deviceIdentity][i],
                                                connectedDevices[deviceIdentity]["name"],
                                                JSON.stringify(contextValue) ]);
  }

} );
```

Figure 31. Observer for the *gps_location* context data.

Conclusions

This application can be useful in meetings and conferences with a lot of people, also it can be used to locate a new person you have meet for Internet for example, and you don't know him.

The application can be extended with extra functionality like google maps to see the exact place of your target, however this could violate the privacy of the user.

GPS location is useless inside a building, however this application combines it with Bluetooth proximity to give a more accurate distance of your target.

Example 5

Application treasureHunt

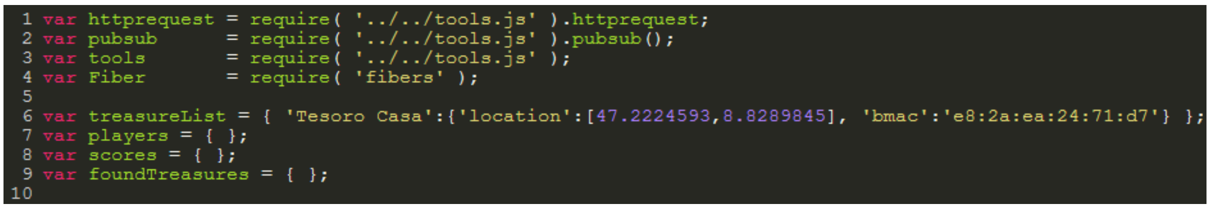
The last application is a small treasure hunting game where the users can play and compete each other. This application also combines GPS location with Bluetooth proximity to know where you are and what do you have in your surroundings.

Each treasure can have some information like a name, a description or hint, the GPS location, a score, etc. There is also a scoreboard with the scores of the players.

Implementation

Treasures are static devices, they cannot move and they can be implemented in any device with Bluetooth capabilities. They won't need to execute the client neither Internet connection. For example, a treasure could be a small Arduino or similar device with Bluetooth capabilities.

They are set in the first lines of the application:



```
var httprequest = require( '../../tools.js' ).httprequest;
var pubsub      = require( '../../tools.js' ).pubsub();
var tools       = require( '../../tools.js' );
var Fiber       = require( 'fibers' );

var treasureList = { 'Tesoro Casa':{'location':[47.2224593,8.8289845], 'bmac':'e8:2a:ea:24:71:d7'} };
var players = { };
var scores = { };
var foundTreasures = { };

```

This application makes use of the following capabilities:

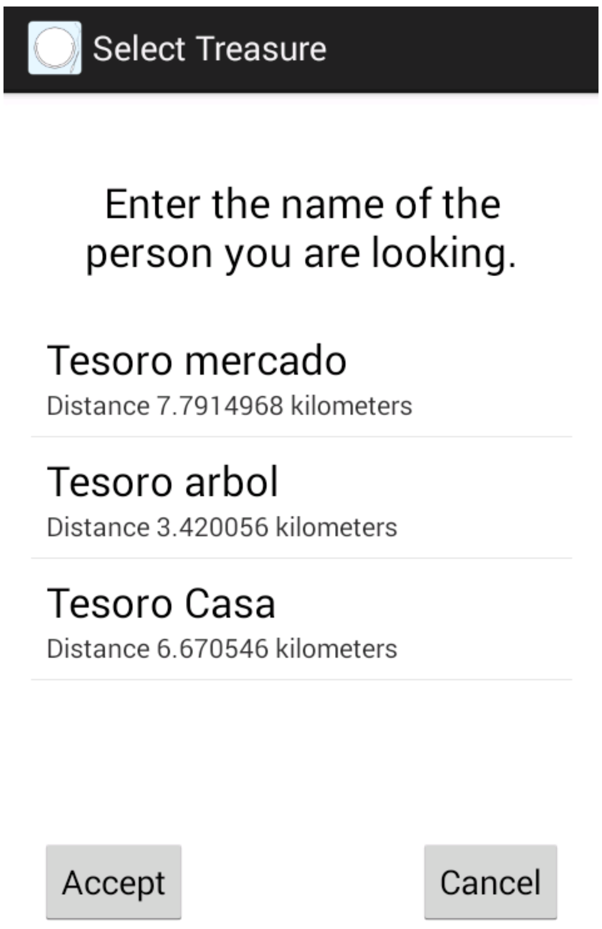
- *RequestPlayerProfile*: This is very similar to the used in the last application, it requests data from the user, like the username and the avatar that will be shown to another players. This action will call the capability *locateCapability*.

- *RequestLocateTreasure*: Very similar to the *RequestLocatePerson* of the last application, but this will show the list of all nearby treasures, so we can pick one to start the hunting.

- *SendPlayerList*: as the name says, this will be the action responsible for sending the scoreboard to all the players. The scoreboard is sent when a new user connects to the application, and each time a user finds a treasure.

- *SetBluetoothTreasure*: similar to *SetBluetoothTarget*, this action will send special data like position and Bluetooth MAC address of the target treasure.

- *SendNotification*: basic notification to inform the user when something was wrong. For example, when he has already found a treasure.

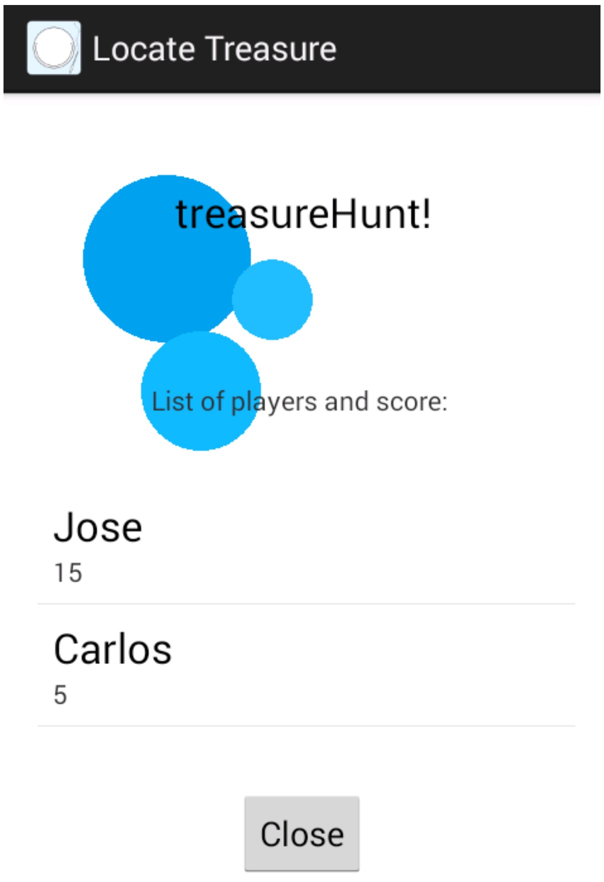

Select Treasure

Enter the name of the person you are looking.

Tesoro mercado
Distance 7.7914968 kilometers

Tesoro arbol
Distance 3.420056 kilometers

Tesoro Casa
Distance 6.670546 kilometers


Locate Treasure

treasureHunt!

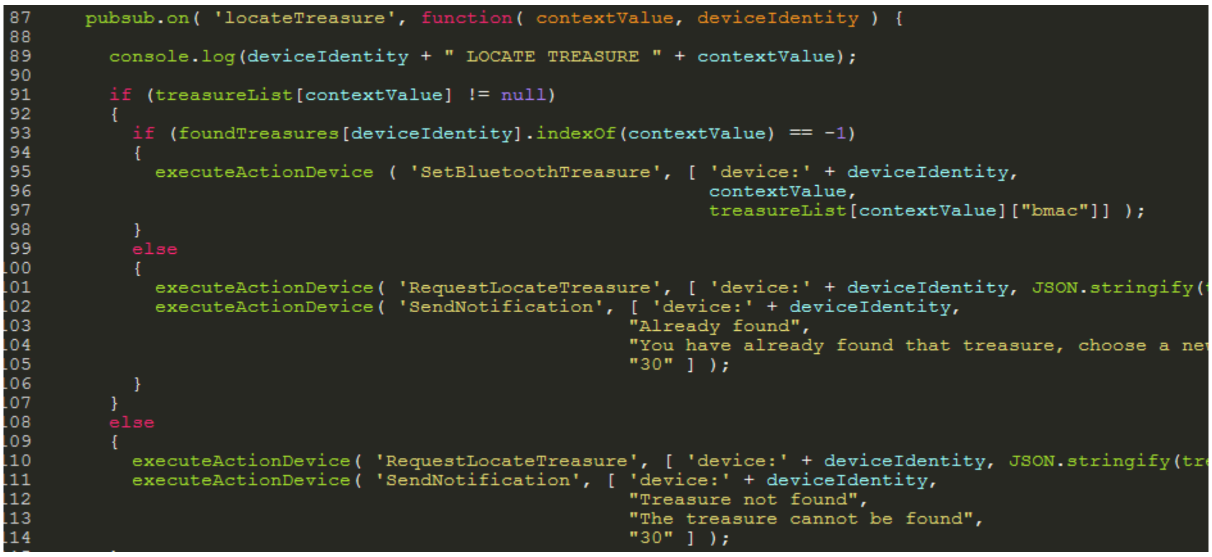
List of players and score:

Jose
15

Carlos
5

Figure 32. Example of views. The first shows all the nearby treasures, the user should select one of them to start the location. Each treasure shows the distance to the current position of the user. The second figure is an example of the scoreboard.

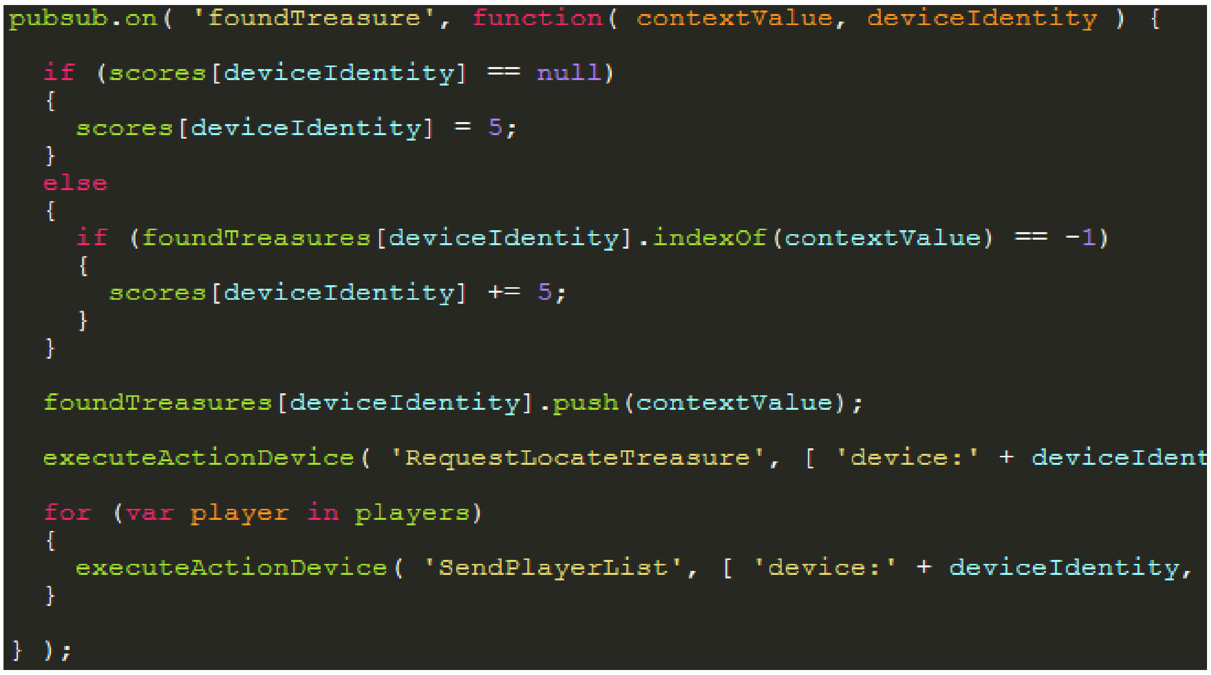
The code of the observer *locateTreasure* is similar to the *locatePerson* in the last application, however now we have to check if the user has already found the selected treasure:



```
pubsub.on( 'locateTreasure', function( contextValue, deviceIdentity ) {

  console.log(deviceIdentity + " LOCATE TREASURE " + contextValue);

  if (treasureList[contextValue] != null)
  {
    if (foundTreasures[deviceIdentity].indexOf(contextValue) == -1)
    {
      executeActionDevice ( 'SetBluetoothTreasure', [ 'device:' + deviceIdentity,
                                                      contextValue,
                                                      treasureList[contextValue]["bmac"]] );
    }
    else
    {
      executeActionDevice( 'RequestLocateTreasure', [ 'device:' + deviceIdentity, JSON.stringify(
      executeActionDevice( 'SendNotification', [ 'device:' + deviceIdentity,
                                               "Already found",
                                               "You have already found that treasure, choose a ne
                                               "30" ] );
    }
  }
  else
  {
    executeActionDevice( 'RequestLocateTreasure', [ 'device:' + deviceIdentity, JSON.stringify(tr
    executeActionDevice( 'SendNotification', [ 'device:' + deviceIdentity,
                                               "Treasure not found",
                                               "The treasure cannot be found",
                                               "30" ] );
```

Finally, the *MapTreasureCapabilityActivity* will do the same as *MapLocateCapabilityActivity*, it will check periodically the location of the treasure, giving hints of the orientation and distance. When the treasure is inside the Bluetooth range, the screen will change its color according to the distance.

When you are closer enough the treasure (about 2-3 meters), you will get your score, and the possibility to select another treasure to hunt. The scoreboard will be sent to all the players to know you have found a new treasure.

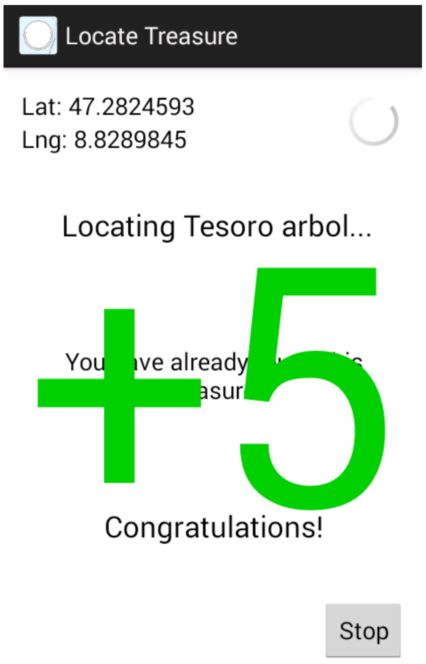


```
pubsub.on( 'foundTreasure', function( contextValue, deviceIdentity ) {

  if (scores[deviceIdentity] == null)
  {
    scores[deviceIdentity] = 5;
  }
  else
  {
    if (foundTreasures[deviceIdentity].indexOf(contextValue) == -1)
    {
      scores[deviceIdentity] += 5;
    }
  }

  foundTreasures[deviceIdentity].push(contextValue);

  executeActionDevice( 'RequestLocateTreasure', [ 'device:' + deviceIdent

  for (var player in players)
  {
    executeActionDevice( 'SendPlayerList', [ 'device:' + deviceIdentity,
  }

} );
```

Figure 33. Code of the observer *foundTreasure*.

[Figure 33] shows the code of the *foundTreasure* observer. This observer will give you the corresponding points (5 in this case, but they can be configured). Then, it will ask you to select a new treasure to hunt, and send the scoreboard to all the connecter players.

You will see a small animation [see figure at right] giving you the score. This treasure will be automatically disabled for you, so you won't get your points again.



Lat: 47.2824593
Lng: 8.8289845

Locating Tesoro arbol...

Congratulations!

Conclusions and future work

Social Devices is aimed to increase, facilitate and enrich social interactions between people in various kinds of co-located and face-to-face situations. In this work I have presented the latest platform implementing this concept: OrchestratorJS. This platform allows to create and manage all the devices, capabilities, actions and apps through the website without the need of external tools.

Furthermore we have seen how the concept can be integrated with such technologies like *Internet of Things* and *pervasive computing*, which are not the main target of the platform, but has a big potential with them.

To facilitate the development, the platform is based in the *Action-Oriented Programming Model*, which offers abstractions for developers that help them to structure pervasive applications. The development of applications is very flexible and modular because the main piece are *Actions*, a small piece of code which can create a small interaction between some devices. Actions are reusable and easily integrated in any application.

In this work I have presented several prototypes of thirds using the People-as-a-Service concept or related with it. To finish this work, I have developed five study cases which show the potential of the platform. These applications are only small examples, with an important didactical objective, because at the moment of this work there was no documentation about how to start the development in the platform.

In the future, the platform could have new features and capabilities, recently the Bluetooth support has been added, and probably GPS support will be a feature soon. Probably in the videogames field this kind of platforms can have an important future, there are two example videogames where users can start the game and invite people in their surroundings to start playing. With the different frameworks and the REST API, the technology can also be extended to many different types of devices.

Furthermore the new Bluetooth 4.0 Low Energy can be a perfect solution to avoid the current Bluetooth problems, which make the devices to waste huge amounts of energy when finding or communicating with nearby devices. Without Bluetooth Low Energy won't be possible the deployment of a technology like Social Devices.

References

[1] Aaltonen, T., Myllärniemi, V., Raatikainen, M., Mäkitalo, N., & Pääkko, J. (2013). An Action-Oriented Programming Model for Pervasive Computing in a Device Cloud. *20th Asia-Pacific Software Engineering Conference*, 467-475.

[2] Abowd, G., Brumitt, B., & Shafer, S. (2001). Ubicomp 2001: Ubiquitous Computing. *International Conference Atlanta.* Georgia: Springer.

[3] Cui, Y., & Honkala, M. (2013). A Novel mobile Device User Interface with Integrated Social Networking Services. *Int'l J. Human-Computer Studies, 71*(9), 919-932.

[4] Edwards, W., Newman, M., Seciivy, J., & Smith, T. (2005). Bringing network effects to pervasive spaces. *Pervasive Computing*, 15-17.

[5] *Event-driven programming*. (2014). Retrieved from Princeton University: http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Event-driven_programming.html

[6] Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems, 29*, 1645-1660.

[7] Guillén, J., Miranda, J., Berrocal, J., García-Alonso, J., Murillo, J. M., & Canal, C. (September de 2013). Architecting Infrastructures for Cloud-Enabled. *Advances in Service-Oriented and Cloud Computing*, 277-287.

[8] Guillén, J., Miranda, J., Berrocal, J., García-Alonso, J., Murillo, J. M., & Canal, C. (17 de March de 2014). People as a Service: A Mobile-centric Model for Providing Collective Sociological Profiles. *Software, IEEE, 31*(2), 48-53.

[9] Hodges, S., Villar, N., Scott, J., & Schmidt, A. (2012). A New Era for Ubicomp Development. *Pervasive Computing*, 5-9.

[10] Jansen, M. (2012). About using mobile devices as cloud service providers. *CLOSER'12*, (págs. 147-152).

[11] Klein, A., Mannweiler, C., Schneider, J., & Schotten, H. D. (2010). Access schemes for mobile cloud computing. *First International Workshop on Mobile Cloud Computing at MDM'10*, 387-392.

[12] Mäkitalo, N. (2014). Building and programming ubiquitous social devices. *Proceedings of the 12th ACM international symposium on Mobility management and wireless access* , 99-108.

[13] Mäkitalo, N. (2014). *Collaborative Co-Located Interactions in a Mobile Cloud*. Retrieved from Social Devices: http://socialdevices.github.io/

[14] Mäkitalo, N., Aaltonen, T., & Mikkonen, T. (2013). First Hand Developer Experiences of Social Devices. *Advances in Service-Oriented and Cloud Computing*, 233-243.

[15] Mäkitalo, N., Pääkkö, J., Raatikainen, M., Myllärniemi, V., Aaltonen, T., Leppänen, T., . . . Mikkonen, T. (2012). Social devices: collaborative co-located interactions in a mobile cloud. *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, Article No. 10.

[16] Miranda, J., Mäkitalo, N., García-Alonso, J., Berrocal, J., Mikkonen T, Canal, C., & Murillo, J. M. (2014). From the Internet of Things to the Internet of People. *IEEE Internet Computing*, (Submitted).

[17] Oulasvirta, A., Rattenbury, T., Lingyi, M., & Raita, E. (2012). Habits make smartphone use more pervasive. *Personal and Ubiquitous Computing, 16*(1), 105-114.

[18] Satyanarayanan, N. (2001). Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 10-17.

[19] Tampere University of Technology. (2014). *Co-Located User Interaction with Social Mobile Devices*. Retrieved from CoSMo: http://www.cs.tut.fi/ihte/projects/CoSMo/

[20] University of Tampere. (2014). *Social Devices*. Retrieved from http://social.cs.tut.fi