Project: Light Up my Pebble

Authors: Anja Cacciatori, Nina

Tutor: Iacopo Carreras, Daniele Miorandi

Sommario

Abstract

This document explains how the project “Light Up my Pebble” was done and it focus on the implementation of a Pebble application and on how the APIs of the Philips Hue lights works.

The explanation starts from of what the smartwatch Pebble is and how to implement an application for it. After that the reader could find what the Philips Hue lights are and how they can be used in an application. The main core of the treatise is on how the HTTP queries are called from an application that wants to control the lights. At the end the user can find an overview on the code of the Project “Light Up my Pebble”.

Until now the project “Light Up my Pebble” was just mentioned a few times. In the following part of this chapter we briefly explain what the project is about.

Light up my Pebble is a smartwatch application that should be able to control the Philips Hue lights from the activities which are recorder from his accelerometer. An example of what the result should be is for example that the light should turn on or off if the user moves his arm.

The outcomes of the Project that can be found by running the code is a menu with many elements. At each element corresponds a different action that allows the user to control the light just by moving his wrist. The choice of just using the shake of the wrist is that Pebble is not having a gyroscope. This absence is a big problem if someone would like to implement some other gestures because, even if the accelerator itself has the x, y and z axis, it is not possible to control the actual orientation of the arm.

The result application is controlling the light in two methods. The first one is by controlling each single light, the second method is on controlling a group of lights. For each method the menu is giving the possibility to control the choice of the color, brightness and saturation, the possibility of selecting a light or a group, to switch them on and off and to impose the color loop. The last two elements of the application menu is implementing some settings. The first setting is the creation of an authorized user and the second setting is the automatic discovery of the IP address.

Another kind of settings used in the Pebble application can be found on the phone application. This settings are to be enable when the Watch application is created and consists of a button called “SETTINGS” present on the button of the related application in the “my Pebble” view. “Light Up my Pebble” uses this possibility in order to find the IP. This IP can be discovered in an automatic way as the user can find in the menu, or manually by insert the IP address.

About Pebble

What is Pebble

Pebble is a smartwatch that can be connected with the phone using Bluetooth. It provides pre-installed applications which are used in order to listen music, get notification as text messages, e-mail, incoming calls and sending some small and preconfigured messages directly from it.
Pebble provides an application for iOS and one for Android devices which are used to install and control the applications that are used and installed on the smartwatch. In this way a developer can create an application which is can be used in relationship with iOS and Android devices. The smartwatch itself can be mainly used just in relation with the smartphone. If those two devices are not paired the smartwatch can be used just in order to discovery what time it is.
The smartwatch can be customized by installing other application, called Watchapps and the Watchfaces, which are present in Pebble's market.
Another interesting feature that Pebble provides is that everyone can create his own application. This can be done online on the framework they provide and by having a knowledge on C or JavaScript languages.
The following sub-chapters are explaining how to connect Pebble with an Android device, how to create an smartwatch application and some of the main functions that are provided from the Pebble team in order to create an application with JavaScript

Pairing with the android phone and install an application from the Pebble market

In order to use the Pebble application the user should connect it to the Phone. On the Google Play store is present an application called Pebble (available at the following link https://play.google.com/store/apps/details?id=com.getpebble.android&hl=it). After the installation, follow the instruction on the screen. The user probably has to do an upgrade of the software, enable the Bluetooth, pairing the devices, and connect the phone to his Pebble account, which will be useful if he use CloudPebble to create a WatchApp.

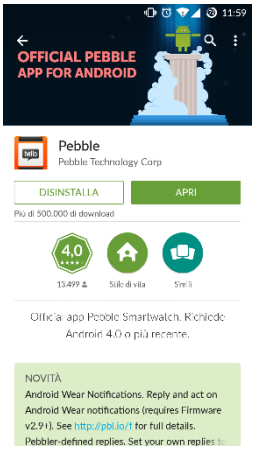

Fig. 1 Screenshot of the Pebble application present on the Google Play Store

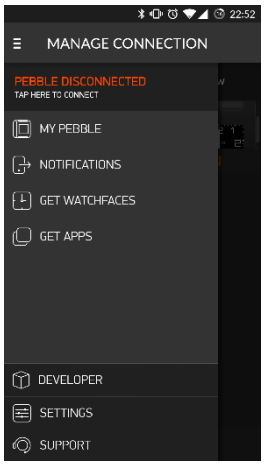

Fig. 2 Menu of the Android Pebble application

Once the user is done Pebble should be connected to the phone. The user can now open the menu on the smartphone application and click on "get watchfaces" or "get apps". Once he gets in one of them, he can choose the app he wants and just click on "add" (it can be found it on the right side on the same line of the app name). The chosen application should get on the Pebble automatically and it can see it in "my Pebble" in the locker where it can be uploaded to the smartwatch. If the smartwatch is connected on the phone, the user has just to upload the application on the smartwatch.

The installed application will be shown in the watch just on the bottom of the menu.

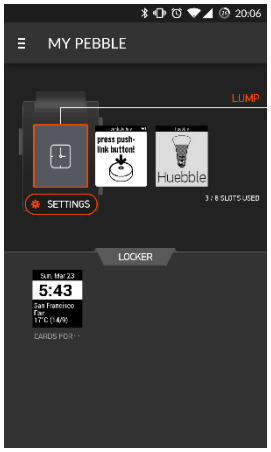

Fig. 3 Applications on the Smartwatch and the one that has to be uploaded

Enable developer options

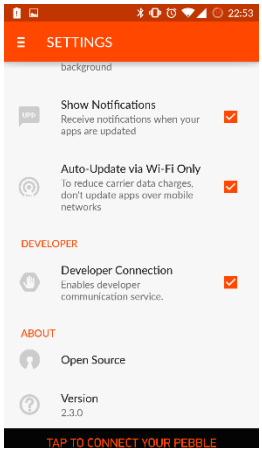

Fig. 4 The view of the Settings present on the Pebble application on Android

If a developer wants to create his own application for the Pebble application he has to create an account on CloudPebble. After he creates his application he has to enable the installation of the application on the phone. In order to do that he has to open the menu, go on "settings" end enable the voice "Developer Connection". After that a new voice, called "Developer", will appear in the menu.

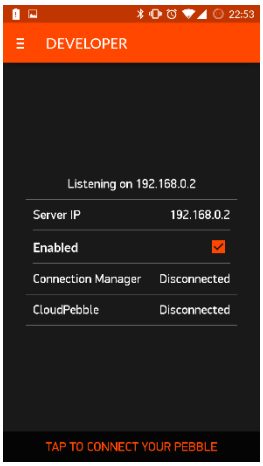

Fig. 5 the "developer" screen

When the developer will install the application on the smartwatch he has to go on the menu and open the "Developer" page. This page displayed will be the one in figure 5. He should be sure that the phone is listening on the right IP address and eventually updating it by deselecting "enabled" and selecting it again.

When the developed application is being installing on the phone the voice "CloudPebble" will show "connect" instead of "disconnected".

Application Developing

An application can be developed using C or using JavaScript. As environment it is possible to create the application offline or online. In order to create the application offline, the developer can download the SDKs and use them in the environment where he is developing. Instead of downloading the application the developer can use the online environment present on the link https://cloudpebble.net/ide/ . The project "Light Up my Pebble" was done using that environment, so the following treatise is based on the usage of this one. The functions, which are going to be explained in the following pages will be the one used in the project.

"Light Up my Pebble" consists in a Pebble application which has to be connected to the Philips Hue lights through HTTP requests. In order to be able to make those requests it was necessary to use the provided function called "ajax". The lights should change their state each time an activity is recorded from the smartwatch. Those activities are recorded by an accelerator which functions are explained soon.

In order to create a user interface, CloudPebble provides an object called Window. This is composed from three element which are a Card, a Menu and a Window. In the project the Menu and the Window are used. Let's first see how to start to develop an application and then focus on the functions itself.

How to develop the application by using CloudPebble

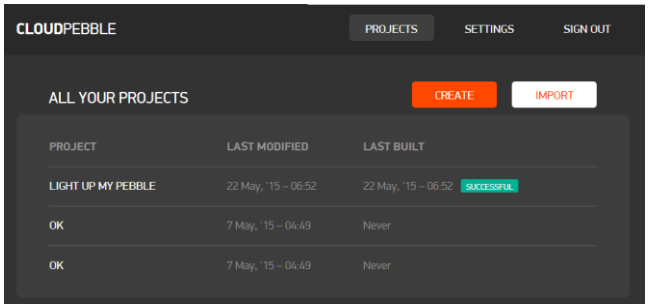

Fig. 6 Cloudpebble Project view

The user should open the CloudPebble from https://cloudpebble.net/, then make the login or create an account. When he is done he has to click on "Projects" on the top and then create a new project selecting the voice "Create". A small window is opening. The developer should insert the project name, as for example "Light Up my Pebble" and selecting the type/language he want to use from the field "Project type". If he choose to use as programming language "C" he probably has more elements to select. In order to do a project in JavaScript he has to select the project type "Pebble.js(beta)" and then click on "create".

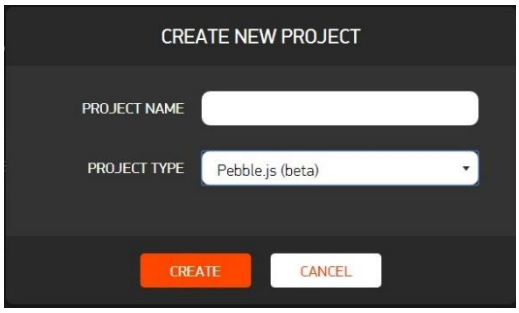

Fig. 7 PopUp window that has to fill in order to create a new project

Once that is done the developer should be in the project environment.

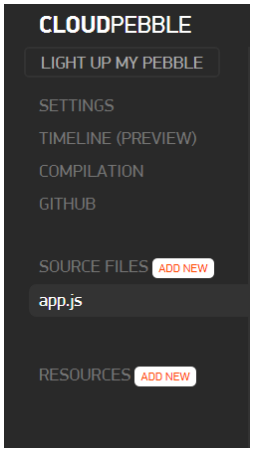

Fig. 8 The menu on the right side window

On the left side of the window that is now shown, a menu can be shown. In this menu, as shown in figure 8, in order to start to code the application, the developer should select the voice "app.js". On the center of the window the space where the user can code is shown.

In order to develop the application the following code should be written:

```
var UI = require('ui');
```

That lines allows to use the Pebble functions.

In order to install and run the application, to view the logs and some others information the user should click on the voice "compilation" that can be found in the menu.

Some other information can be found on the following webpages.

- Tutorial: http://developer.getpebble.com/getting-started/pebble-js-tutorial/part1/
- Documentation: http://developer.getpebble.com/docs/
- Using JavaScript on Pebble: http://developer.getpebble.com/guides/js-apps/

Ajax call

In order to do a simple HTTP call, the function which has to be used is the “ajax” provided by the Pebble team. An Ajax call has to be include in the project and this is done with the following line of code:

```
var ajax = require('ajax');
```

After the request to the library it is possible to make an HTTP query. The function “ajax” has three field. The first one contains the options, the second part is a function which takes in input the values returned from the HTTP request and the third one is managing the error that is returned if the HTTP query fails.

```
ajax(
        {
        url: 'http://api.theysaidso.com/qod.json',
        type: 'json'
        },
        function(data, status, request) {
        console.log('Quote of the day is: ' + data.contents.quote);
        },
        function(error, status, request) {
        console.log('The ajax request failed: ' + error);
        }
);
```

The options that can used are many. In the project the following ones were used:

- **url :** it is the URL to which the query has to be done.
- **method:** it represents what the developer would like to use. As default it is set to get, but it allows to use even post, put, delete and other one, if they are supported.
- **type** : it is the type of data we would get to have back from the HTTP query. An example is the “json” type.
- **data :** it allows to send some information to the URL. An example could be sending some data in order to impose it to the lights.

For more information on Ajax the reader should have a look at the following webpage: http://developer.getpebble.com/docs/pebblejs/#ajax .

Accelerator

The accelerator, which function is called Accel, allows to get the events recorded from the accelerator. As the function ajax, it has to be request from the library before it can be used. This request is done with the following code:

```
var Accel = require('ui/accel');
```

In order to use the accelerator it has to be initialized. This can be done by writing:

```
Accel.init();
```

After the initialization it is possible to call different functions as the configuration and the recording of data from the application.

The accelerator can be configured with the command

```
Accel.config(accelData);
```

but the developer team suggest to use the preconfigured events and to not modify the values of the accelerator. (More info on: http://developer.getpebble.com/docs/pebblejs/#accel-config-accelconfig)
The accelerator has another function which is called Accel.peek() and allows to get peeks on the current value of the accelerometer.
The accelerator has a function which permits to recording data. It has two types of data that can be taken as function input:

- Tap events. They are triggered when the user flicks his wrist or tap on the Pebble. Those kinds of data are battery efficient because they are generated directly on the accelerator.
  An example is the following code:

  ```
  Accel.on('tap', function(e) {
  console.log('Tap event on axis: ' + e.axis + ' and direction: ' + e.direction);
  });
  ```

  In the example we can see the fields e.axis and e.direction which are present in the data called by Accel.on();

- Continuously data flows. They are used to send continuously data from the Pebble accelerometer. The accelerator collects samples at a given frequency (from 10 to 100 Hz), save them in an array and pass them to the event manager. It will drain fast the battery because it continuously transmits data to the processor and through the Bluetooth to the phone.
  An example of continuously data flows is the following one:

  ```
  Accel.on('data', function(e) {
  console.log('Just received ' + e.samples + ' from the accelerometer.');
  });
  ```

  Those kind of data has different ways used to save the data. Those ways are the *samples, accel, accels* and each of them has different kinds of options.

It would be nice to use the accelerator in order to create some gestures. Unfortunatly the Pebble smartwatch has no gyroscope so it is not possible to create them and the only one that the developer team has provide is the shake of the wrist.

More information on the accel.on() function for tap and data callback can be found at the following link: http://developer.getpebble.com/docs/pebblejs/#accel-on-39-tap-39-callback .

Configuration

The Pebble smartwatch can be used by pressing 4 buttons or by moving the arm, but what if the developer would like to ask the user of his application to insert something like a small text? The Pebble team provide a solution to that. In fact it is possible to insert some settings in the smartwatch application using the phone. In Pebble's phone application, the application installed on the smartwatch can be found in the "my Pebble" page selected from the menu. Under the developed smartwatch applications it is possible to find a "Settings" button which allows to show a small configuration page on the phone. This configuration page is an HTML webpage that can be used to edit some settings or doing some web authentication.

Fig. 9 Smartwatch application with the settings button that is shown on the smartwatch

In order to display the gear icon in the smartphone application, the developer has to check "configurable" under the settings tab of the project in the CloudPebble environment (figure 10).

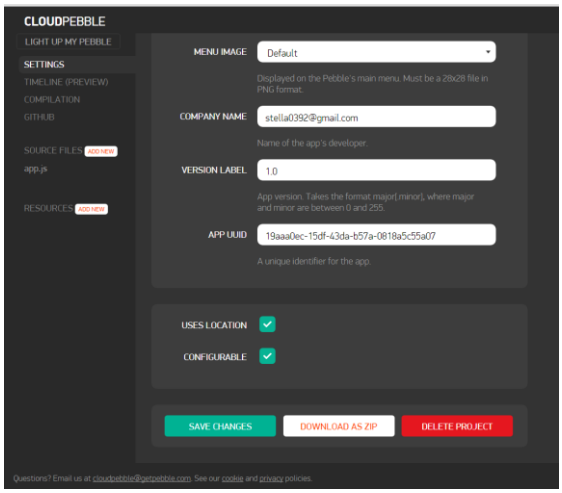

Fig. 10 CloudPebble's settings where configurable is enabled

The configuration page has to be called in the code. This is done in the following code.

```
Pebble.addEventListener('showConfiguration', function(e) {
        // Show config page
          Pebble.openURL('https://my-website.com/config-page.html');
});
```

In the previous code it is possible to notice that the function Pebble.addEventListener('showConfiguration', function(e){ … }); is waiting for an event which is the pressing of the application settings button on the Phone. Another important thing is that showConfiguration must always implement a Pebble.openURL(); in order to show the configuration page. If that last function is not implemented an error will be shown.

From the function Pebble.openURL(); it is possible to see that the configuration page is a small webpage containing all the settings that the developer would like to ask to the user.
When the user is done with the configuration window, the configurations, has to be received form the Pebble. The HTML code present on the webpage has to include a JavaScript code. This code has to call a special URL that is *pebblejs://close*. Any data added in the anchor of this URL, will be passed to the Pebble Application. Pebble developer recommends to use json to pass data.
More information on that topic can be found on:

- http://developer.getpebble.com/guides/js-apps/pebblekit-js/app-configuration/

Some examples are on:

- https://github.com/pebble-hacks/js-configure-demo/blob/master/src/js/pebble-js-app.js
- https://github.com/pyro2927/Pebble-Quick-Config
- https://github.com/BalestraPatrick/PebbleConfigurationExample

Window and Menu

Window is a basic build block for a Pebble.js application. It is divided in three types which are:

- Menu that allows to create some selectable elements
- Card that displays a title, a subtitle, a banner image and text on a screen. The position of the elements are fixed and cannot be changed.
- Window: It is more flexilble and enables to use some elements as circle, rectangular, text, timetext or image. The size of the screen is variable and the elements can be animated.

Those kind of windows has to be declared. This is done in the following ways. For the window it is

```
var wind = new UI.Window();
```

for the card it will be

```
var card = new UI.Card();
```

and for the menu it is

```
var menu =new UI.Menu();
```

Note that a menu is an array of elements.

The Window, Card and Menu are objects with some functions. Those functions are the following ones:

- wind.on(); it allows to create a menu, or a window or card, where we define the variables and what we think each part should do
- wind.add(Element); it adds for example a text on the window stack
- wind.show(); it shows the window or menu or card
- wind.hide(); or wind.remove(); it hides a window and if it is not use it removes it from the stack

Notice that instead of “wind” it is possible to call a card or a menu.
For more information on windows please check: http://developer.getpebble.com/docs/pebblejs/#window
For the card: http://developer.getpebble.com/docs/pebblejs/#card
And for the menu: http://developer.getpebble.com/docs/pebblejs/#menu

About Philips Hue

Philips Hue is the brand of smart lights which are connected to the wlan. Those lights are able to change color, saturation and brightness when they are connected to the bridge. The bridge controls the single lights or a group of them. Some feature that the lights are offering are the possibility of use some scenes, imposing an effect or an alarm. Scenes are lights that should be able to recreate an atmosphere as a romantic one, while the only possible effect is the color loop.

Philips Hue lights can be completely programmed and it is possible to use a SDK or to use API's.

In the project "Light Up mu Pebble", the APIs were used. Those APIs allows the developer to use those lights by sending some HTTP queries.

A good emulator for the Hue lights can be found here: http://steveyo.github.io/Hue-Emulator/.

This emulator is working like the real lights expect for some bugs. One bug is when a group is switched down and the user wants to make a GET query in order to get the state of the lights present in the group. The answer of the GET request is that the lights are o, this means it is not really possible to switch a group of lights off, even if the image is saying that they are actually off. Another thing that is not implemented is the possibility to set the color loop on the "effect". Another problem was the discovery of the IP Address since the lights are on the emulator.

Getting started

Installation of the kit Hue and control that everything works

After the user has bought his kit hue, he has to install it. After that he should be sure that the bridge is connected to his network and that it works properly. In order to start to use the lights the user has to install the hue application on the smartphone and test if he is able to control the lights. Please notice that the user and the bridge has to be connected on the same network.
The Hue application for android provide by Philips can be found on the Google Play Store or by clicking on the following link: https://play.google.com/store/apps/details?id=com.philips.lighting.hue&hl=it .

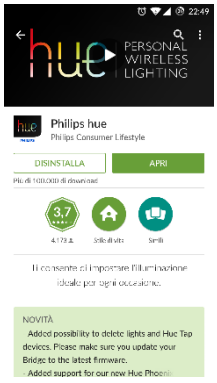

Figura 11 Philips Hue application present on the Google Play Store

How to discover the IP address

When a developer creates an application he could include the Hue SDKs to his project or using HTTP queries. If he choose this last method he has to discover the IP address of the bridge. This can be done in several ways.

1) Use a UPnP discovery app
2) Using Philips broker server discovery process by visiting www.meethue.com/api/nupnp
3) Log into your wireless router and search for the IP
4) Using the Hue App as descripted in the references on the webpage www.developers.meethue.com

In the project “Light Up my Pebble” the IP address is discovered by using principally method 2. The other possibility is to ask the user to insert the IP address, but the way to search the IP address is left to the user itself.
Ones the IP Address is found, it is possible to connect to the test application by visiting the following address:

http://<bridge ip address>/debug/clip.html

where the <bridge ip address” is the address that has been found as descripted before.
For more information on how to get the IP address and the methods, please have a look to the references at http://www.developers.meethue.com/documentation/getting-started/ .

Create a new developer

Once the IP address of the bridge is found, the developer would be able to send a get query to it. In order to do that he should go on the HTTP address http://<bridge ip address>/debug/clip.html and trying insert the in the window the URL

/api/newdeveloper

Since “newdeveloper” is not an authorized user an error will be shown (see figure 11).

What can be done is the creation of an authorized user with the name that the user would like to use.

To create the user called “newdeveloper”, which should be able to control the lights, it is possible to use an http request to the address:

http://<bridge ip address>/api

by using the POST method and insert as body message:

{"devicetype":"test user","username":"newdeveloper"}

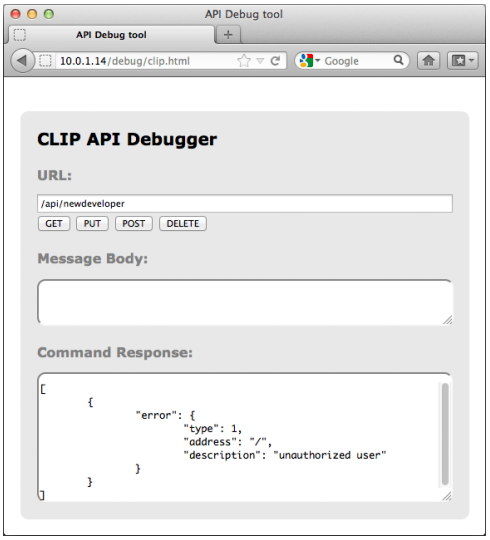

Fig. 12 Unauthorized user error

When the first post is done the user should get back an error message which says that the button on the bridge is not pressed. The user should press the button on the bridge and then use again the same post command as before. This is a security test in order to prove that the user has a physical access to the bridge. Once the developer gets an authorized user he can start to send HTTP queries to the bridge.

Some core concepts

There are different kind of resources to interact with by using the HTTP request.

1) /lights resource which contains all the lights
2) /groups resource which contains all the groups
3) /config resource which contains all the configuration items
4) /schedules which contains all he schedules
5) /scenes which contains all the scenes

6) /sensor which contains all the sensors
7) /rules which contains all the rules

It is possible to query each resource available on the bridge by doing a GET request. For example by doing a GET query to

http://<bridge ip address>/api/newdeveloper/lights/1

it is possible to get the state of the light number 1.

The user can control the lights, groups and so on by doing PUT querying. For example in order to switch the light 1 on, it is possible to do a PUT query to

http://<bridge ip address>/api/newdeveloper/lights/1/state

And set the body

{“on”:true}

Some more complicate things to do is changing the color. This can be done in two methods. The more complicate one consists to impose the value of the “xy” field by looking on the chromaticity diagram showed in figure 12.

A much easier way which is to set a value for the “hue” field. This field is just a number from 0 to 65535 and for each value a given color will be assigned. The color specification will be seen when it comes to the Lights and group API.

Some more informations about core concepts of those lights can be found on the webpage recalled in this link: http://www.developers.meethue.com/documentation/core-concepts

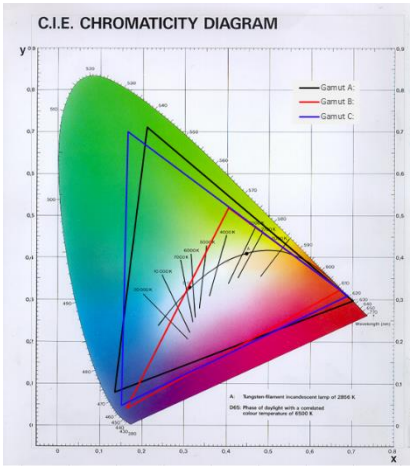

Fig. 13 C.I.E. chromaticity diagram

Philips hue API

As we said before there are two ways to using the lights. One is by using tools and SDKs another is by using the API. Since the project was done by using the APIs related to the lights and to the groups, so what will be found from here on will be just related to those to kind of things. For the other types of resources please refer to http://www.developers.meethue.com/philips-hue-api . Note that in order to read the APIs a login is request.

Lights API

The treatise is starting from the Lights APIs. This APIs are principally HTTPs queries to the light bridge and that allows the user to control the lights. Please note that that the URLs, if the user is not using the debugging tool provided from the developer team, the user should add http://<bridge ip address> to the beginning of the URL that is mentioned each time.

The first thing that a developer would be able to do, is getting all the lights connected to the bridge. This can be done by using a GET query to the following address:

/api/<username>/lights

The response is a json type and is something like the following one.


```
{
  "1": {
    "state": {
      "on": true,
      "bri": 144,
      "hue": 13088,
      "sat": 212,
      "xy": [0.5128,0.4147],
      "ct": 467,
      "alert": "none",
      "effect": "none",
      "colormode": "xy",
      "reachable": true
    },
    "type": "Extended color light",
    "name": "Hue Lamp 1",
    "modelid": "LCT001",
    "swversion": "66009461",
    "pointsymbol": {
      "1": "none",
      "2": "none",
      "3": "none",
      "4": "none",
      "5": "none",
      "6": "none",
      "7": "none",
      "8": "none"
    }
  },
"2": {
    "state": {
      "on": false,
      "bri": 0,
      "hue": 0,
      "sat": 0,
      "xy": [0,0],
      "ct": 0,
      "alert": "none",
      "effect": "none",
      "colormode": "hs",
      "reachable": true
    },
"type": "Extended color light",
    "name": "Hue Lamp 2",
    "modelid": "LCT001",
    "swversion": "66009461",
    "pointsymbol": {
      "1": "none",
      "2": "none",
      "3": "none",
      "4": "none",
      "5": "none",
      "6": "none",
      "7": "none",
      "8": "none"
    }
  }
}
```


What we get are two lights (1 and 2) which have fields called state, type, name and so on. The state is somehow an array which contains the following objects:

- "on" that indicates if the light is on or off. It can be set with Boolean values that are true and false.
- "bri" shows the brightness of the light. It assumes values from 1 to 254, where 1 is not indicating that the light is off.
- "hue" is a value of the color. This can assume values from 0 to 65535. Both 0 and 65535 are red, 25500 is green and 46920 is blue.
- "sat" indicates the saturation of the colors. It goes from 0, which represents the white color, to 254 which is the one with the most statured (colored).
- "xy" represents the color from the CIE color space and it is an array of two values given from the space mentioned before.
- "ct" is a mixed color temperature.
- "alert" is an alert which temporary change the value of the light.
- "effect" allows to create a color loop. The only values that can be used are "colorloop" or "none".

- “colormode”.
- “reachable” is indicating if the light can be reached in the network.

In order to change the values present in the state it is possible to use a PUT query to the URL

```
/api/<username>/lights/<id>/state
```

where <id> is the number of the light that the developer would like to change. The body of that query request can be for example

```
{
“on”:true,
“bri”:199,
“sat”:50
}
```

Where the values that are set are the one that were explained before. If everything goes well the user should get a success message back.

It is even possible to update the number of lights or to delate a light from the bridge. More information about how to do that can be found here: http://www.developers.meethue.com/documentation/lights-api .

Group API

In the previous sub-chapter the treatment was about the single lights, but sometimes the user likes to control more than a single light at one time. For example he could like to switch all the lights on or off. This is a possibility that is provided from the developer. Philips Hue uses groups which are composed from two or more lights, which can be controlled simultaneously.

The control of groups of lights can be done by adding the following line to the IP address.

```
/api/<username>/groups
```

If a GET query to this address is done, the response should be similar to following example.


```
{
  "1": {
    "name": "Bedroom",
    "lights": [
      "1",
      "2"
    ],
    "type": "LightGroup",
    "action": {
      "on": true,
      "bri": 254,
      "hue": 10000,
      "sat": 254,
      "effect": "none",
      "xy": [
  "2": {
"name": "Living Room",
"lights": [
  "3",
  "4",
  "5"
],
"type": "LightGroup",
"action": {
  "on": true,
  "bri": 153,
  "hue": 4345,
  "sat": 254,
  "effect": "none",
  "xy": [
```

```
            0.5,
            0.5
          ],
          "ct": 250,
          "alert": "select",
          "colormode": "ct"
        }
      },
```

```
            0.5,
            0.5
          ],
          "ct": 250,
          "alert": "select",
          "colormode": "ct"
        }
      }
    }
```

This case is the most complete one, because the response could have just the name of the groups and the lights contained in each group. As in the lights API it is possible to find a group named "1" which has the values "type" and "action". Note that the values of action are the same as the "state" field that present in the lights response.
The field "type" can assume 4 kinds of values:

1) "0" that is a special group containing all the lights and is not returned when a get query is called
2) "Luminaire" that is a lighting installation of default grouping of hue light and cannot be created manually
3) "Lightsource" that is a sub group of the previous case.
4) "LightGroup" that is a group of lights that can be controlled together.

It is possible to create a group by using a POST query to /api/<username>/groups. The body should be similar to the following code:

```
{
    lights":[
    "1",
    "2"
    ],
    "name":"bedroom"
}
```

If everything was fine a success response should be returned.
Another thing that can be modify are the group attributes by using a PUT query to

```
/api/<username>/groups/<id>
```

This can be done by sending to the bridge the name or the lights that should be modify.
In order to modify the state of the lights a PUT query could be done to the URL:

```
/api/<username>/groups/<id>/action
```

In both cases the field <id> of the URL is the number of the group that has to be changed. As in the light API it is possible to change the following values of the state:

1) "on" which indicates that the state of the light and the values are true if the light should be on and false if the lights should be off.
2) "bri" is the brightness value and is a scale from 0 to 255

3) “sat” is the saturation and has values from 0 to 255 as in the light APIs
4) “hue” and “xy” that changes the value of the color. “hue” is using values from 0 to 65535 while “xy” is using the coordinates of a color in the CIE color space.
5) “ct” is the mired color temperature of the light
6) “alert” is a temporary change to the bulb’s state
7) “effect” as “none” or “colorloop”
8) “transitiontime”
9) “bri_inc”
10) “sat_inc”
11) “hue_inc”
12) “ct_inc”
13) “xy_inc”
14) ”scene”

For more reference to those values and to the groups API, please refer to the following webpage: http://www.developers.meethue.com/documentation/groups-api .

Cloud controlling of the light

The lights can be controlled even from the cloud. The possibility provided from the Philips Hue team can be found on the http://www.meethue.com/ webpage. It is necessary to create an account and after it, it is possible to connect the bridge to the account and controlling the lights from the webpage itself. The Philips Hue team provides even some API, but the developer should require them to the team by compiling the form on the webpage present on the webpage http://www.developers.meethue.com/content/remote-api .

Another way to control the lights from a cloud is to using the unofficial APIs that can be found on the following links:
http://philips-hue-remote-api.readthedocs.org/en/latest/
https://github.com/jarvisinc/PhilipsHueRemoteAPI

About Light up my Pebble

In the project “Light Up my Pebble” it was required to control the Philips Hue lights from the Pebble application. The lights has to be modified if the user, that has the smartwatch on his wrist, is for example moving his arm. In order to do that, the movement has to be recorded from the smartwatch by using the accelerator.
In order to do the smartwatch application the first thing to do is creating an account in order to having access to CloudPebble. After that it is possible to create an application on that webpage. Since the project is written by using JavaScript, the project type has to be Pebble.js.
It was necessary to create a small webpage that is used as configuration page.
In the following treatise it is possible to find in the first part the CloudPebble code and in the second one the HTML of the configuration page.

CloudPebble code

The application code starts from the variable initialization.

```
var UI = require('ui');
var Accel = require('ui/accel');
var Vector2 = require('vector2');
var ajax= require('ajax');

var spashText = new UI.Text({
      position: new Vector2(0, 0),
      size: new Vector2(144, 168),
      text:'Recording data from accelerometer...',
      font:'GOTHIC_28_BOLD',
     color:'black',
     textOverflow:'wrap',
     textAlign:'center',
     backgroundColor:'white'
});

var spashWind = new UI.Window();
var wind = new UI.Window();
var getUrl='http://';
var putUrl='http://';
var getGroupUrl;
var putGroupUrl;
var ipAddr;
var lightNum;
var developer;
var groupNum;

var errorajax = new UI.Text({
      position: new Vector2(0, 0),
      size: new Vector2(144, 168),
     text:'Sorry!Probably you set the wrong ip address',
     font:'GOTHIC_28_BOLD',
     color:'black',
     textOverflow:'wrap',
      textAlign:'center',
      backgroundColor:'white'
});
var ip;
var ipA;

var errorLight;

var menu = new UI.Menu({
    sections: [{
        title: 'Select what you want to do',
        items: [{
             title:'Select # of Light' //case 0
            }, {
             title: 'Switching Light on or off', //case 1
             subtitle: 'Is my light on?',
            }, {
            title: 'Change color of light' //case 2
            },{
             title: 'Change light saturation' //case 3
            },{
            title: 'Change light brightness' //case 4
            },{
             title: 'Select Group' //case 5
            },{
            title: 'Switch Group on/off' //case 6
            }, {
            title: 'Change color of group' //case 7
            },{
            title: 'Change group saturation' //case 8
           },{
           title: 'Change group brightness' //case 9
           },{
          title: 'One-Time Configuration', //case 10
          subtitle:'If the brigde has more memory then me!'
          },{
          title: 'Ip address discovery' //case 11
         }]
    }]
});
```

From line 1 to 4 some request where made in order to be able to use the functions implemented by the Pebble team. At line 1 it is possible to find the mandatory code explained in the chapter which was speaking about how develop a Pebble application.
Line 6 and 28 are defining some text which will be shown by the Pebble. In order to be shown it has to be added in a window and the window is the element that has to be shown. The windows are defined in line 17 and 18. Since it the actions that have to be done from the smartwatch are many, a menu is defined in lines from 43 to 74. This menu contains a menu title and the items and allows to switch between its elements by pressing a button.
The other things are variable used in the code. By having a look at the terms from line 19 to 26. In those lines some dynamic url are defined. The creation of dynamic URLs is useful because the IP address should be automatically discovered, the user can be changed and the lights or groups has to be selected by the user.

Going ahead a the setting page was implemented in order to allow users to insert manually the IP address. This is done in the following code.

```
1.  /*
2.  *Settings on phone
3.  *
4.  */
5.
6.
7.  //ricerca automatica ip address
8.
9.  var initialized = false;
10. var options = {};
11.
12. Pebble.addEventListener("ready", function() {
13. console.log("ready called!");
14. initialized = true;
15. });
16.
17. Pebble.addEventListener("showConfiguration", function() {
18. console.log("showing configuration");
19. Pebble.openURL('http://www.lump.pe.hu');
20. });
21.
22. Pebble.addEventListener("webviewclosed", function(e) {
23. //Pebble.sendAppMessage(options);
24. console.log("configuration closed");
25. // webview closed
26. //Using primitive JSON validity and non-empty check
27. if (e.response.charAt(0) == "{" && e.response.slice(-1) == "}" && e.response.length > 5) {
28. options = JSON.parse(decodeURIComponent(e.response));
29. ipA = encodeURIComponent(options.ipA);
30. console.log("Options = " + JSON.stringify(options));
31. ipAddr=JSON.stringify(options.ip);
32. console.log(ip);
33. } else {
34. console.log("Cancelled");
35. }
36. });
37.
38. /*
39. *Close settings
40. *
41. */
```

This code is explained in the chapter where the treatment was about the configuration. In any case this part is waiting that the user is pressing the “settings button on the Pebble application at the phone, then it waits that the opened webpage is closed and it manages the returned values.

The initialization of the URL is done in a function. This function has as body the following lines of code.

```
getUrl='http://'+ipAddr+'/api/'+developer+'/lights/'+lightNum+'/';
putUrl='http://'+ipAddr+'/api/'+developer+'/lights/'+lightNum+'/state/';
getGroupUrl='http://'+ipAddr+'/api/'+developer+'/groups/'+groupNum+'/';
putGroupUrl='http://'+ipAddr+'/api/'+developer+'/groups/'+groupNum+'/action/';
```

It is possible to see that the variables for the URLs are four. This is done because the IP addresses are different for the GET and the PUT queries and even for the single lights or for the groups.

After this initialization the menu is going to be created. In order to do that it is possible to use the following function:

```
menu.on('select', function(e){
//code
}
```

In this code a switch is present. This allows the user to choose the item and define what it is doing. Since the part of the group and the single lights are similar, the code explanation will be just about the lights part.

The first case, that can be found is the selection of a light.

```
Accel.init();
spashWind.add(spashText);
spashWind.show();
var ln=0; //number of lights found from ajax

ajax({
        url: 'http://'+ipAddr+'/api/'+developer+'/lights',
        method:'get',
        type:'json'
        },
        function(data){
        var key=[];
        var lights=data;

        errorLight=data[0];
        console.log(errorLight);
        if(errorLight!==undefined){
            key=Object.keys(errorLight);
            console.log(key[0]);
        }else{
             key[0]="success";
        }
        if(key[0]==="error"){
          console.log(errorLight.error.description);

          var errore = new UI.Text({
               position: new Vector2(0, 0),
               size: new Vector2(144, 168),
               text: errorLight.error.description,
               font:'GOTHIC_28_BOLD',
              color:'black',
              textOverflow:'wrap',
              textAlign:'center',
              backgroundColor:'white'
              });
         spashWind.hide();
         wind.hide();
         wind.add(errore);
                if (obj.hasOwnProperty(key)) size++;
               }
               Return size;
     };//how to find lenght????
     ln=Object.size(lights);
   console.log(ln);
  }
},

function(error) {
        spashWind.hide();
       wind.hide();
       wind.add(errorajax);
       wind.show();
       console.log('The ajax request failed: ' + error);
 });
 var k=1;
 if(errorLight===undefined){
    Accel.on('tap',function(e){
        var text = new UI.Text({
              position: new Vector2(0, 0),
              size: new Vector2(144, 168),
              text:'You choose the Light '+k,
              font:'GOTHIC_28_BOLD',
              color:'black',
              textOverflow:'wrap',
              textAlign:'center',
              backgroundColor:'white'
         });
       spashWind.hide();
       wind.hide();
       wind.add(text);
       wind.show();
       lightNum=k;
        if(k<ln){k=k+1;
       }else{
       k=1;
       }
```

```
39.         wind.show();
40.     }else{
41.         Object.size = function(obj) {
42.             var size = 0, key;
43.             for (key in obj) {
82.     getUrl='http://'+ipAddr+'/api/'+developer+'/lights/'+lightNum+'/'
        ;
83.     putUrl='http://'+ipAddr+'/api/'+developer+'/lights/'+lightNum+'/
        state/';
84.     });
85.     }
```

At line 1 the accelerator is initialized and then used at line 62. After that a text is shown in the window (line 2 and line 3) and starting from line 6 to line 59 the ajax function is used in order to get the lights. In all the ajax functions some kind of controls are defined. In function(data) it is possible to find a control which will show to the user if some errors occurs. A typical error is given from an unauthorized user. Another kind of error is in the function(error), which will return an error if the HTTP request is failing.
If the HTTP query is going well, at line 41 a function which is getting an object in input and returns the size of it, is defined. At line 48 the length of the lights is returned by using the previous function. If no error is occurring the accelerator is started at line 62. A variable k counts the number of times the user shakes his arm and it will define the number of the light that is selected (line 77). At lines from 78 to 81 the value of the counter of k is increasing. Note that the maximum value of k is the same as the number of lights connected to the bridge and if k is greater than that number it set to 1 again.
At lines 82 and 83 the value of the light that is chosen in imposed the URLs.

After the selection of the light the code explains howt to switch the light on and off. Since the errors are the same we just use the same code as in the previous case and it will not be shown here. The code for the light is the following one.

```
1.  Accel.init();
2.  spashWind.add(spashText);
3.  spashWind.show();
4.  var status;
5.  ajax({
6.        url: getUrl,
7.        method: 'get',
8.        type: 'json',
9.  },
10. function(data){
11.     var key=[];
12.     errorLight=data[0];
13.     console.log(errorLight);
14.     if(…){
15.         //error controll
16.     }else{
17.          status=data.state.on;
18.     }
19. },
20. function(error) {
21.       //error
22. }); //chiude ajax
23.
24. if(errorLight===undefined){
25.
26. Accel.on('tap', function(e){
27.       console.log(status);
28.      if(status===false){
29.      var text = new UI.Text({
30.          position: new Vector2(0, 0),
31.          size: new Vector2(144, 168),
32.          text:'Your light is ON!',
33.          font:'GOTHIC_28_BOLD',
34.          color:'black',
35.          textOverflow:'wrap',
36.          textAlign:'center',
37.          backgroundColor:'white'
38.          });
39.     ajax({
40.          url: putUrl,
54.          },
55.          function(data){
56.              status=data.state.on;
57.          },
58.          function(error) {
59.              //error code
60.          }); //chiude ajax
61.     },
62.     function(error) {
63.          //error code
64.     });
65.      }else{
66.      var newText = new UI.Text({
67.          position: new Vector2(0, 0),
68.          size: new Vector2(144, 168),
69.          text:'Your light is OFF!',
70.          font:'GOTHIC_28_BOLD',
71.          color:'black',
72.          textOverflow:'wrap',
73.          textAlign:'center',
74.          backgroundColor:'white'
75.          });
76.      ajax({
77.          url: putUrl,
78.          method: 'put',
79.          type: 'json',
80.          data: {"on":false}
81.          },
82.          function(data) {
83.              spashWind.hide();
84.              wind.hide();
85.              wind.add(newText);
86.              wind.show();
87.              ajax({
88.                  url: getUrl,
89.                  method: 'get',
90.                  type: 'json',
91.              },
92.              function(data){
93.                  status=data.state.on;
```

```
41.             method: 'put',
42.             type: 'json',
43.             data: {"on":true}
44.             },
45.             function(data) {
46.                 spashWind.hide();
47.                 wind.hide();
48.                 wind.add(text);
49.                 wind.show();
50.                 ajax({
51.                     url: getUrl,
52.                     method: 'get',
53.                     type: 'json',
94.                         },
95.                         function(error) {
96.                             //error code
97.                         }); //chiude ajax
98.                     },
99.                     function(error) {
100.                //error code
101.            });
102.    }
103. }); //chiude accel.on
104. }
```

In this case the accelerator (line 26) is called in the first ajax function which gives us the value of the light state (line 17). Depending on the light state a PUT query is setting the value on to true or false. This is done with an if-else statement where if the light is off it is set to true and otherwise (lines 39 to 49 and lines 76 to 86). In the function(data) of those two PUT query a GET request is asking the system which is the new state (we use the get request as in line 50 and 87) and update the actual value state(line 56 and line 93).

The cases of changing color, saturation and brightness can be discussed all together, because the principle is the same in each of them and the only things that really changes are the values they have. Once again the errors control code is not reported here because the code is the same as in the previous cases.

```
1.  spashWind.add(spashText);
2.  spashWind.show();
3.  Accel.init();
4.
5.  var text = new UI.Text({
6.      position: new Vector2(0, 0),
7.      size: new Vector2(144, 168),
8.      text:'You changed the color, is it the one you wanted?',
9.      font:'GOTHIC_28_BOLD',
10.     color:'black',
11.     textOverflow:'wrap',
12.     textAlign:'center',
13.     backgroundColor:'white'
14.     });
15.
16. var i;
17. ajax({
18.       url: getUrl,
19.       type: 'json'
20. },
21. function(data) {
22.       var key=[];
23.
24.     errorLight=data[0];
25.     console.log(errorLight);
26.     if(…){
27.         //error code
28.     }else{
29.         i=data.state.hue;
30.     }
31. },
32. function(error) {
33.     //error code
34. });
35. if(errorLight===undefined){
36.   Accel.on('tap', function(e){
37.     ajax({
38.         url: putUrl,
39.         method: 'put',
40.         type: 'json',
41.         data: {"hue":i}
42.     },
43.     function(data) {
44.         spashWind.hide();
```

```
45.             wind.hide();
46.             wind.add(text);
47.             wind.show();
48.             if(i<52850){i=i+12150;}
49.             else{i=0;}
50.         },
51.         function(error) {
52.             //error code
53.         });
54. });
```

In order to change the color we recall the ajax function with a GET method (line 17) to get the actual state of the light. This state is set in a value(line 29). After then the accelerator is called (line 36) and it makes an PUT query (line 37) in order to impose the value of i and improve i by adding 12150 at each wrist shake until it reach the maximum value. If it reach this value it will just set i to zero. The values of the color goes from 0 to 65535.
For saturation and brightness the code does not change much. The main important is that for line 29 the values returned from the GET query are saved in a variable by calling i=data.state.sat or i=data.state.bri respectively for saturation and brightness.For both cases the if-else code of line 48 is if(i<254){i=i+50;} while the else is the same for saturation and i=1*;* for brightness. Another thing that changes is at line 41 where the data that is sent with the PUT query is imposed to {“bri”:i} for brightness and {“sat”:i} for saturation.

Another element of the switch statement is that is possible to impose to the lights the possibility to cycle through all the colors. The code for this is the following one.

```
1.  Accel.init();
2.  spashWind.add(spashText);
3.  spashWind.show();
4.  wind.hide();
5.  var loop;
6.  var textLoop = new UI.Text({//text
7.  });
8.  var textNone = new UI.Text({ //text
9.  });
10. ajax({
11.    url: getUrl,
12.    type: 'json'
13.    },
14.    function(data) {
15.    //errorcode
16.    If(…){}else{
17.      loop=data.state.effect;
18.      }
19.    },function(error) {
20.        spashWind.hide();
21.        wind.hide();
22.        wind.add(errorajax);
23.        wind.show();
24.        console.log('The ajax request failed: ' + error);
25.     });
26. if(errorLight===undefined){
27. Accel.on('tap', function(e){
28. if (loop==="none"){
29. ajax({
30.      url: putUrl,
31.      method: 'put',
32.    type: 'json',
33.    data: {"effect":"colorloop"}
34.   },function(data) {
35.       spashWind.hide();
36.       wind.hide();
37.       wind.add(textLoop);
38.       wind.show();
39.       ajax({
40.           url: getUrl,
41.           type: 'json'
42.           },function(data) {
43.              loop=data.state.effect;
```

```
            },function(error) {
            spashWind.hide();
            wind.hide();
            wind.add(errorajax);
            wind.show();
            console.log('The ajax request failed: ' + error);
            });
}, function(error) {
            });
}else{
ajax({
            url: putUrl,
            method: 'put',
            type: 'json',
            data: {"effect":"none"}
            },
            function(data) {
            spashWind.hide();
            wind.hide();
            wind.add(textNone);
            wind.show();
            ajax({
                  url: getUrl,
                  type: 'json'
            },function(data) {
                loop=data.state.effect;
            }, function(error) {
             });
            }, function(error) {});
}
});
}
```

Even this code has the same principle as the code written in order to switch the lights on and off, but the change is that instead of the true and false, for the value event, the strings “colorloop” or “none” are imposed depending on the values of the variable loop. Even in this case a GET query is updating the value of loop.

In the menu it is possible to find two configuration elements. The first one is for configuring an authorized user and the second one is for automatically discovers the IP address.
For the configuration of the user following code is implemented.

```
Accel.init();
var text2 = new UI.Text({
      position: new Vector2(0, 0),
      size: new Vector2(144, 168),
      text:'Some starting info: Shake Pebble once, then go on the Bridge, press the Button on top of it and then shake once again. Be fast!',
      font:'GOTHIC_18_BOLD',
      color:'black',
      textOverflow:'wrap',
      textAlign:'center',
      backgroundColor:'white'
      });
spashWind.hide();
wind.hide();
wind.add(text2);
wind.show();
Accel.on('tap', function(e) {
    ajax({
            url: 'http://'+ipAddr+'/api/',
            method: 'post',
            type: 'json',
            data: {"devicetype":"Pebble","username":"pebble"}
            },
            function(data) {
                  var key=[];
                  errorLight=data[0];
                        backgroundColor:'white'
                        });
                  spashWind.hide();
                  wind.hide();
                  wind.remove(text2);
                  wind.add(errore);
                  wind.show();
            }else{
                  if(key[0]==='success'){
                        var text = new UI.Text({
                              position: new Vector2(0, 0),
                              size: new Vector2(144, 168),
                              text:'You are done, bravo!',
                              font:'GOTHIC_28_BOLD',
                              color:'black',
                              textOverflow:'wrap',
                              textAlign:'center',
                              backgroundColor:'white'
                         });
                        spashWind.hide();
                        wind.hide();
                        wind.remove(text2);
                        wind.add(text);
                        wind.show();
                        developer='pebble';
                        }
                  }
      },
```

```
                key=Object.keys(errorLight);
                console.log(key[0]);
                if(key[0]==='error'){
                    console.log(errorLight.error.description);
                    var errore = new UI.Text({
                        position: new Vector2(0, 0),
                        size: new Vector2(144, 168),
                        text:  errorLight.error.description + '! Press the Bridge!',
                        font:'GOTHIC_28_BOLD',
                        color:'black',
                        textOverflow:'wrap',
                        textAlign:'center',
        function(error) {
            spashWind.hide();
            wind.hide();
            wind.add(errorajax);
            wind.show();
            console.log('The ajax request failed: ' + error);
        });
});
```

In this code the first thing that is implemented is the a window with a user guide, then by using the acceleration two POST queries with the same code are done to the bridge. After the first request, the bridge responds with an error and the error description is shown on the display of Pebble. The error will say that the user has to press the bridge button. After that he should shake his arm again and he should get a success answer from the bridge.

For the automatic discovery of the IP address the method used is one of those proposed from the Philips hue, and actually the NUPnP.

```
1.  ajax({
2.      url: 'https://www.meethue.com/api/nupnp',
3.      method: 'get',
4.      type: 'json'
5.      },
6.      function(data) {
7.          ip=data[0].internalipaddress;
8.          var text4 = new UI.Text({
9.              position: new Vector2(0, 0),
10.             size: new Vector2(144, 168),
11.             text:'The ip addr is!'+ip,
12.             font:'GOTHIC_28_BOLD',
13.             color:'black',
14.             textOverflow:'wrap',
15.             textAlign:'center',
16.             backgroundColor:'white'
17.             });
18.         spashWind.hide();
19.         wind.hide();
20.         wind.add(text4);
21.         wind.show();
22.         ipAddr=ip;
23.         },
24.         function(error) {
25.         console.log('The ajax request failed: ' + error);
26.         spashWind.hide();
27.         wind.hide();
28.         wind.add(errorajax);
29.         wind.show();
30. });
31. console.log(ip);
```

For doing the automatic discovery of the code uses again an ajax request. This request will be on a website which is https://www.meethue.com/api/nupnp which return the IP address which can be seen at the line 7.

After the implementation of the menu, it is displayed it with the following code:

```
1.  menu.show();
2.  spashWind.hide();
```

Configuration page code

We said before that in order to create a configuration page on the phone, it is necessary to create a webpage. This webpage contains two header showing if the user wants to insert manually the IP address, or if the IP address has to be automatically searched.

The HTML code is the following one.

```
<!DOCTYPE html>
<html>
  <head>
                        <title>Configurable</title>
                        <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta charset="utf-8">
          <link rel="stylesheet" href="http://code.jquery.com/mobile/1.3.2/jquery.mobile-1.3.2.min.css" />
                        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
    <script src="https://code.jquery.com/jquery-1.10.2.js"></script>
          </head>
          <body>
    <h1>Configuration Pebble Application</h1>
                        <div id="menu">
                                    <h1>Manual insertion of the IP Address</h1>
      <button type="button" id="o1">show</button>
      <button type="button" id="o1a">hide</button>
          <div style="display:none" id="opt1">
                              <p> Please insert the Ip Address here. Thanks! </p>
                                    <p><label for="ip">Bridge IP Address:</label>
                                    <textarea cols="40" rows="1" name="ip" id="ip"></textarea>
                                                        </p>
                        <button type="submit"  id="b-cancel">Cancel</button>
                        <button type="submit"  id="b-submit">Submit</button>
      </div>

      <h1>Automatical find IP Address</h1>
      <button type="button" id="o2">set</button>
    </div>
  <script>

  </script>
 </body>
</html>
```

In the HTML it is possible to see that the manual insertion on the code has two buttons, show and hides, while the automatical research has just one button which is set.
In the script part it is possible to find some jQuery and JavaScript code which is the following one

```
  function saveOptions() {
    var options = {
                                    'ip':$("#ip").val()
                        };
    //Add all textual values
    return options;
  }
  $(document).ready(function(){
                        $( "#o1" ).click(function() {
      $( "#opt1" ).show( "fast" );


      $("#b-cancel").click(function() {
          console.log("Cancel");
          document.location = "pebblejs://close";
```

```
    });
    $("#b-submit").click(function() {
        console.log("carlo");
        console.log("Submit");
        var location = "pebblejs://close#" + encodeURIComponent(JSON.stringify(saveOptions()));
        console.log("Warping to: " + location);
        console.log(location);
        document.location = location;
    });
  });
  $("#o1a").click(function(){
    $("#opt1").hide("fast");
  });

  $( "#o2" ).click(function() {
    $( "#opt2" ).show( "fast" );
      var request = $.ajax({
        url: "https://www.meethue.com/api/nupnp",
        method: "GET",
        dataType: "json"
        });
        request.done(function( msg ) {
          var ipAddress=msg[0].internalipaddress;
        //$( "#log" ).html( msg );
var options = {ip:ipAddress};
          var location = "pebblejs://close#" + encodeURIComponent(JSON.stringify(options));
          console.log("Warping to: " + location);
          console.log(location);
          document.location = location;
          $("#text1").append("<p>The ip address is: </p>"+ipAddress);
        });
        request.fail(function( jqXHR, textStatus ) {
        alert( "Request failed: " + textStatus );
      });
  });
  $("#o2a").click(function(){
    $("#opt2").hide("fast");
  });
  //Set form values to whatever is passed in.
  var obj = jQuery.parseJSON(decodeURIComponent(window.location.search.substring(1)));
  for(key in obj) {
   $("#"+[key]).val(obj[key]);
   $("#"+[key]).val(obj[key]).slider("refresh");
  }
                    });
```

This code shows some parts of the HTML code after the user clicks on the button show. An insertion field is going to be shown and even other two buttons. In the insertion code the user should insert the IP address and after that he should press on the button submit. The insertion will be saved in a json file and send to the Pebble smartwatch.
If the user decides to ask the system to search the IP address, he should just press on set. When this button is pressed, the system is sending a request to https://www.meethue.com/api/nupnp using the ajax function provided from the jQuery team. The answer will be converted to a json object and then sent do the smartwatch.