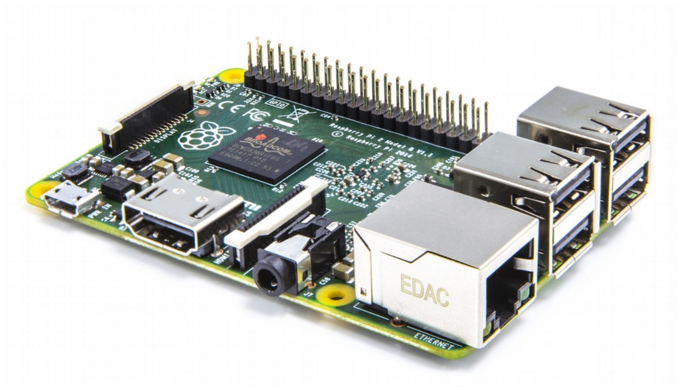
Velisphere SDK Tutorial #2

The entire code for this tutorial can be found on
https://github.com/thomeudt/VelisphereClientSDK/tree/master/DemoRaspberryPiSensor

How to build a multi-sensor device using a RaspberryPI and Phidgets



In this tutorial, we will create a multi-sensor device that connects to Velisphere allowing to track and analyze sensor data. In a later tutorial, we will expand on this setup by adding a relay than can be controlled via Velisphere rules. But let's concentrate on building the sensor device first.

I. What you need

1. Skills

   You should be reasonably familiar with the JAVA programming language and the Eclipse IDE.

   You will find this tutorial easier to follow if you have already managed to get a Raspberry Pi setup, including installing a Linux operating system on the SD card. Some basic Linux skills are helpful, too.

2. Hardware

   1x Raspberry Pi, while either version 1 or version 2 should work, we have tested this tutorial with version 2 *only*

   The RaspberryPi needs to have internet connectivity!

   1x Phidgets Interface Kit 8/8/8
   1x Phidgets Light Sensor
   1x Phidgets Touch Sensor
   1x Phidgets Pressure Sensor
   1x Phidgets Rotary Dial
   1x Powered USB 2.0 hub (sufficient to provide power for both the Raspberry Pi and the Interface Kit. The USB ports on the Raspberry Pi do not provide sufficient power for the interface kit!

3x USB cables, one each connecting the Raspberry PI and the interface kit with the USB hub for power supply, and one connecting the interface kit with the Raspberry Pi for connectivity.

Please refer to the manuals provided for the Raspberry Pi, your USB hub, and the interface kit to determine which type of USB cable will be required.

3. Software

On your Developer Machine (PC, Mac, etc.):

- Eclipse Standard IDE for you to build the JAVA client application that will then be deployed to the Raspberry Pi. The tutorial has been written for Eclipse "Luna" (Version 4.4), but other recent releases such as Mars or Kepler will also work. There might be slight differences to the UI though, therefore the tutorial might not be 100% accurate for these releases.

- Eclipse requires a supported Java Runtime Environment (JRE). Please refer to the Eclipse documentation for further details.

On the Raspberry Pi:

- Your Raspberry Pi should be running a Linux distribution. I am using ArchLinux as it currently is the only distribution that supports Docker on Raspberry Pi and I need this for other purposes.

  If you prefer Raspbian or Ubuntu, stick with those, they should work equally well for the purposes of this tutorial.

- A working Java Runtime Environment, Oracle Java and OpenJDK will both do the job. Java 7 is tested, newer versions should also work.

  If not already installed, on Ubuntu/Debian systems, you may install OpenJDK using the following command:

  ```
  sudo apt-get install openjdk-8-jre
  ```

  For other distributions, please refer to the respective documentation.

II. Physical Setup

Getting the Raspberry Pi to work with the Phidgets Interface Kit and the connected sensors, a bit of pre-work is required. I recommend that you follow the free tutorial provided here:

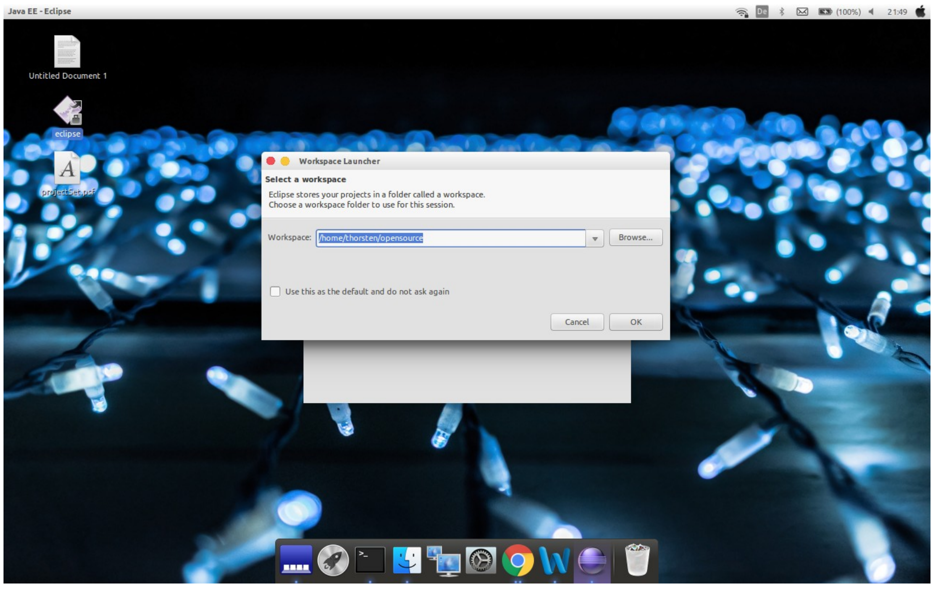
http://www.instructables.com/id/Getting-Started-with-Phidgets-on-the-Raspberry-Pi/?ALLSTEPS

Make sure you connect the four sensors to the analog ports on your Phidgets Interface Kit, as we will be using these four ports later in the tutorial.
The assignments used in this tutorial are as follows:

Rotary Dial:	Port 1
Pressur Sensor:	Port 5
Light:	Port 6
Touch:	Port 7

III. Writing the Client Application in Java

1. On your developer machine, launch the Eclipse IDE. When asked, you can choose either to create a new workspace for your project, or use an existing one.



Select a workspace

Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: /home/thorsten/opensource

☐ Use this as the default and do not ask again

2. When Eclipse is started up, create a new project

   FILE > NEW > JAVA PROJECT

   In the dialog box, enter

   `PiDemo`

   in the **PROJECT NAME** field, leave all other options in their default settings and hit **FINISH**. The new project PiDemo should now show up in the package explorer on the left side of your Eclipse window.

3. We now need to create the default Velisphere configuration file, which we will later use to store information required to authenticate your RaspberryPi in the Velisphere network as well as to validate messages your RaspberryPi will receive from the network.

   To do so, right-click on the newly create “PiDemo” project and select NEW > FILE in the context menu.

4. In the new file dialog, please enter the following file name in the field “File name:”

   ```
   velisphere_config.xml
   ```

   and hit the FINISH button.

5. An empty file `velisphere_config.xml` will be created and opened on the right hand side of the screen.

6. Copy the following XML snippet into the newly created file

   ```
   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
   <!DOCTYPE properties SYSTEM
   "http://java.sun.com/dtd/properties.dtd">
   <properties>
   <comment>This file contains Velisphere authentication
   information. Do not overwrite!</comment>
   <entry key="Secret Key"></entry>
   <entry key="Endpoint ID"></entry>
   <entry key="Endpoint Class ID">33492b88-3626-416c-
   86c3-90221a849da2</entry>
   <entry key="isProvisioned">false</entry>
   </properties>
   ```

   and hit the SAVE (floppy disk) button in Eclipse.

7. Let's have a look at this file. In essence, it is an XML file that specifies four different keys:

   Secret Key

   The secret key that your Raspberry Pi will use for checking the authenticity of an incoming message (i.e. to verify whether the message has truly been sent by the Velisphere system) and which will also be used to allow the Velisphere system to check whether it was truly your device that sent an outgoing message.

   The principle used by Velisphere is based on the HMAC (Keyed hash message

authentication code) concept:
https://en.wikipedia.org/wiki/Hash-based_message_authentication_code

The secret key will be provided by the Velisphere server during the provisioning process and automatically be stored in the configuration file.

Endpoint ID

The endpoint ID is a unique identifier that identifies your Raspberry Pi, it will also be assigned by the Velisphere server during provisioning.

Endpoint Class ID

The endpoint class ID defines which kind of endpoint your RaspberryPI is, so that the system has a clear description of the capabilities of this device.

Think of it as follows (Germans like car analogies):

Endpoint Class > “Raspberry Pi with Phidgets”
> Car Analogy: “2016 Volkswagen Passat Sedan SE”

Endpoint > “Thorsten's Raspberry Pi with Phidgets”
> Car Analogy: “VIN WVWZZZZZxyz123456”

Within the Velisphere System, multiple endpoints of the same endpoint class can exist. Any given endpoint is, in other words, the implementation of an endpoint class. The endpoint class describes the characteristics, the “properties” of and endpoint.

For this reason, the endpoint class ID needs to be provided to Velisphere when sending the provisioning request and therefore needs to be provided via the XML file *before* sending the provisioning request.

The endpoint class for this tutorial is

```
33492b88-3626-416c-86c3-90221a849da2
```

IsProvisioned Flag

Before provisioning a device for the first time (or re-provisioning a device that was previously provisioned), the isProvisioned-Flag needs to be set to “false”. We will later read this flag to determine on startup whether to send a provisioning request or continue with regular boot.

8. Next, we need to download the Velisphere Software Development Kit JAR file and add it to the JAVA build bath of our newly created repository.

   To download the file, go to

   https://github.com/thomeudt/VelisphereClientSDK/releases

   Under the "Downloads" section, you will find a file called **VelisphereMilkSDK.jar** (In case you wonder about the dairy content – all Velisphere components are codenamed around products you need to produce chai late, and milk certainly qualifies for that…)

9. Switch back to Eclipse and make a right click on you newly created project "PiDemo", select **Build Path >** and **Configure Build Path…** .

10. Select the **Libraries** tab and click on **Add external JARs…** on the right side of the dialog box

11. In the JAR selection dialog box, navigate to the location in which you stored the downloaded VelisphereMilkSDK.jar. Select this file and click **OK**. The JAR selection dialog closes, then hit **OK** again to close the Java Build Path dialog.

    We can now use routines from the SDK JAR file to build our client, which saves a lot of time.

12. In order to be able to read data from sensors connected to the Phidgets interface kit, we also need to add the Phidgets Java library file to our build path. Download the respective file **phidget21.jar** from the phidget web site and add it in the same was as you added **VelisphereMilkSDK.jar** (steps 9-11)

13. Velisphere provides you with functionality to track location data of your devices and perform further analysis on this data. As the Raspberry Pi does not include a GPS receiver for precise location data, we will approximate the device position based on its IP address, using GeoLite provided by MaxMind.

    The required library geoip-api-1.2.15.jar can be downloaded from www.maxmind.com, and needs to be added to the build path following again steps 9-11.

14. Next, we will create a new Java package in which we will store the code that we are about to create. For the purpose of this tutorial, we are using the following package name, but you are certainly free to use whatever name you like (we suggest you stick to Java naming conventions):

```
com.velisphere.demo.rpiSensors
```

15. Right click again on the "PiDemo" project and select New > and Package.

16. In the New Java Package dialog box, leave the default setting for the source folder (should be PiDemo/src) and enter the package name as above:

    ```
    com.velisphere.demo.rpiSensors
    ```

17. Within the newly created package, which will show up in the Eclipse Package Explorer after expanding the src folder, we need to create four Java classes that contain all of the code we need.

    Three classes are the basic requirements for any Velisphere client implementation, but are always to be adapted specifically for the endpoint we are developing for.

    In our case these are:

    - PiSensorsMain.java
      Main class that will handle startup:

    - PiEventListener.java
      An implementation of the `com.velisphere.milk.interfaces.EventListener` which is used to respond to incoming messages

    - PreDeployment.java
      A class to trigger initial deployment of the device:

    In addition, we will use two classes that provide additional functionality:

    - PhidgetMonitoringEngine.java
      A class that will listen to sensor value changes on the Interface kit and sends them, via AMQP, to the Velisphere message broker.

    In order to create these classes, please repeat the following steps for each class, using the respective class name:

    - Right click on the package com.velisphere.demo.rpiSensors you created in step 16, and select NEW > CLASS
    - In the New Java Class dialog box, enter the name of the class you are creating (see above), without the ".java" extension. Leave all other options in their default settings.
    - Hit the FINISH button to create the class

18. After having created all four empty classes, it is time to create the code. We will start with the most simple class, the pre-deployment class, PreDeployment. In Eclipse file explorer, double click on the class, which will open the (mostly) empty class:

```
package com.velisphere.demo.rpiSensors;
public class PreDeployment {
}
```

19. Add the following code to the PreDeployment class:

```
package com.velisphere.demo.rpiSensors;

import com.velisphere.milk.restClient.ProvisioningClient;

public class PreDeployment {

   public static void initiateDeployment()
   {
         System.setProperty("jsse.enableSNIExtension", "false");
         String provisioningID = ProvisioningClient.macProvisioning();
         System.out.println("[IN] Provisioning request successfully
         submitted.");
         System.out.println("[IN] Go to www.velisphere.com, log in with
         your user name and start the provisioning wizard to
         complete.");
         System.out.println("[IN] Your Device ID is: " +
         provisioningID);

   }

}
```

This class calls the `ProvisioningClient.`*`macProvisioning`* static method that comes with the Velisphere Client SDK. This class files a provisioning request with Velisphere, using the MAC address of the network interface as the unique identifier.

There are other methods available for filing provisioning requests using the `ProvisioningClient` class, we will discuss these in later tutorials.

Calling the MAC provisioning method returns a string value that is the MAC address of the device, we print them to the console so that the user can use this identifier to finalize the provisioning, using the provisioning wizard provided by velisphere.com

Hit the “save” button in Eclipse to save your code.

20. Next, we will create the code for the PiEventListener class. It will have to look as follows:

```
package com.velisphere.demo.rpiSensors;

import java.util.HashMap;

import com.velisphere.milk.amqpClient.AmqpClient;
import com.velisphere.milk.configuration.ConfigData;
import com.velisphere.milk.interfaces.EventListener;

public class PiEventListener implements EventListener {

   @Override
   public void requestIsAlive(AmqpClient amqpClient) {

         System.out.println(" [IN] IsAlive Requested...");

         HashMap<String, String> messageHash = new HashMap<String,
String>();
         messageHash.put("setState", "REACHABLE");

         try {
               amqpClient.sendHashTable(messageHash, ConfigData.epid,
"CTL");
         } catch (Exception e) {
               // TODO Do something meaningful for error handling
               e.printStackTrace();
         }

   }

   @Override
   public void requestAllProperties(AmqpClient amqpClient) {

         System.out
                     .println(" [IN] AllProperties requested and
discarded as not supported by PiSensor");

   }

   @Override
   public void newInboundMessage(AmqpClient amqpClient, String message)
{

         System.out.println(" [IN] New Inbound Message, but no action
implemented yet by PiSensor");

   }

}
```

As already mentioned, this class implements the EventListener interface of the SDK, which takes care of responding to incoming messages.

The interface describes three methods we need to implement:

requestIsAlive
This method is triggered when the client receives a request asking if it is stil "alive", similar to the PING command you might be familiar with. We respond with setting the state of the endpoint as "reachable"

requestAllProperties
This method is triggered when the client receives a request asking it to submit the current values of all sensors. We disregard this for the time being and leave the implementation for the next tutorial.

NewInboundMessage
This method is called when a new inbound message (other than the two above) is received. It provides the JSON message as a parameter so that the endpoint can be programmed to respond to it. We disregard this for the time being and leave the implementation for the next tutorial.

Please note the `@Override` annotation for each of these method, indicating that the method implements the abstract method inherited from the EventListener interface.

21. Next, we will create the code for the PhidgetMonitoringEngine class. This is the class that reads data from the sensors connected to the interface kit and submits data to Velisphere on change of a sensor value.

    This is the most complex class of our project and requires the import of several classes from the phidget and velisphere SDK libraries, as well as several other libraries provided with the JRE.

```
package com.velisphere.demo.rpiSensors;

import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.HashMap;
import com.maxmind.geoip.Location;
import com.maxmind.geoip.LookupService;
import com.maxmind.geoip.regionName;
import com.phidgets.InterfaceKitPhidget;
import com.phidgets.Phidget;
import com.phidgets.PhidgetException;
import com.phidgets.event.AttachEvent;
import com.phidgets.event.AttachListener;
import com.phidgets.event.DetachEvent;
import com.phidgets.event.DetachListener;
import com.phidgets.event.ErrorEvent;
import com.phidgets.event.ErrorListener;
import com.phidgets.event.InputChangeEvent;
```

```java
import com.phidgets.event.InputChangeListener;
import com.phidgets.event.OutputChangeEvent;
import com.phidgets.event.OutputChangeListener;
import com.phidgets.event.SensorChangeEvent;
import com.phidgets.event.SensorChangeListener;
import com.velisphere.milk.amqpClient.AmqpClient;
import com.velisphere.milk.configuration.ConfigData;

public class PhidgetMonitoringEngine {

   private static final String touchSensorID = "f3b93c91-dee0-4609-
ac1c-100dd00e060a";
   private static final String lightSensorID = "67f6edfa-3413-41a7-
ab16-e9ed72136696";
   private static final String pressureSensorID = "72d3bbef-1f8e-4d8c-
a838-ea119ef98592";
   private static final String dialKnobID = "ac4bd814-3a65-42f8-b5ad-
75e71e63640b";
   private static final String locationID = "24c1ea6d-c2c5-420d-9743-
f718e221e07d";

   private AmqpClient amqpClient;

   public PhidgetMonitoringEngine(AmqpClient amqpClient)
   {
         this.amqpClient = amqpClient;
   }


   public void startMonitoring() throws PhidgetException, IOException
   {

         System.out.println(Phidget.getLibraryVersion());
         final InterfaceKitPhidget ik = new InterfaceKitPhidget();
         ik.addAttachListener(new AttachListener() {
               public void attached(AttachEvent ae) {
                     System.out.println("attachment of " + ae);
               }
         });
         ik.addDetachListener(new DetachListener() {
               public void detached(DetachEvent ae) {
                     System.out.println("detachment of " + ae);
               }
         });
         ik.addErrorListener(new ErrorListener() {
               public void error(ErrorEvent ee) {
                     System.out.println("error event for " + ee);
               }
         });
         ik.addInputChangeListener(new InputChangeListener() {
               public void inputChanged(InputChangeEvent oe) {
                     System.out.println(oe);
               }
         });
         ik.addOutputChangeListener(new OutputChangeListener() {
               public void outputChanged(OutputChangeEvent oe) {
                     System.out.println(oe);
```

```
                                            }
                });
                ik.addSensorChangeListener(new SensorChangeListener() {
                        public void sensorChanged(SensorChangeEvent se) {
                                System.out.println(se);
                                System.out.println("[IN] Value change triggered on
Sensor "+se.getIndex() +". New Value:" + se.getValue());


                                try {
                                        // send to controller

                                        HashMap<String, String> messageHash = new
HashMap<String, String>();

                                        // Sensor Data

                                        messageHash.put(touchSensorID,
String.valueOf(ik.getSensorValue(6)));
                                        messageHash.put(lightSensorID,
String.valueOf(ik.getSensorValue(7)));
                                        messageHash.put(pressureSensorID,
String.valueOf(ik.getSensorValue(5)));
                                        messageHash.put(dialKnobID,
String.valueOf(ik.getSensorValue(1)));

                                        // Geo Location

                                        File dbfile = new File("GeoLiteCity.dat");
                                        LookupService lookupService = new
LookupService(dbfile, LookupService.GEOIP_MEMORY_CACHE);

                                        URL whatismyip = new
URL("http://checkip.amazonaws.com");
                                        BufferedReader in = new BufferedReader(new
InputStreamReader(
                                                        whatismyip.openStream()));

                                        String ip = in.readLine(); //you get the IP
as a String

                                        Location location =
lookupService.getLocation(ip);

                                        // Populate region. Note that regionName is
a MaxMind class, not an instance variable
                                        if (location != null) {
                                            location.region =
regionName.regionNameByCode(location.countryCode, location.region);
                                        }

                                        messageHash.put(locationID, "{"     +
String.valueOf(location.latitude) + "}" +
                                                    "[" +
String.valueOf(location.longitude) + "]");
```

```
                            // Send out
                            
                            System.out.println("Message Hash Sent to 
Controller: " + messageHash);
                            
                            amqpClient.sendHashTable(messageHash, 
ConfigData.epid, "REG");
                            
                            
                      } catch (Exception e) {
                            // TODO Do something meaningful for error 
handling
                            e.printStackTrace();
                      }
                }
         });
         ik.openAny();
         System.out.println("waiting for InterfaceKit attachment...");
         ik.waitForAttachment();
         System.out.println(ik.getDeviceName());
         System.in.read();
         ik.close();
         System.out.println(" ok");
         
   }
}
```

22. Let's look at some aspects in more detail.

    At the beginning of this class, we are defining several constants that contain the property id of the sensors of our device. Every sensor and every actor of each endpoint class carry their own ID.

```
private static final String touchSensorID = "f3b93c91-dee0-4609-ac1c-
100dd00e060a";
private static final String lightSensorID = "67f6edfa-3413-41a7-ab16-
e9ed72136696";
private static final String pressureSensorID = "72d3bbef-1f8e-4d8c-
a838-ea119ef98592";
private static final String dialKnobID = "ac4bd814-3a65-42f8-b5ad-
75e71e63640b";
private static final String locationID = "24c1ea6d-c2c5-420d-9743-
f718e221e07d";
```

    The id codes can be found in the endpoint class catalog available on velisphere.com.

    We then declare an instance variable for the amqpClient class imported from the velisphereSDK.

```
private AmqpClient amqpClient;
```

    This variable is declared in the following constructor by injecting it as a dependency for later use.

```
public PhidgetMonitoringEngine(AmqpClient amqpClient)
   {
         this.amqpClient = amqpClient;
   }
```

The following code is largely specific to ready sensor values from the phidget interface kit, which we will not elaborate on in this tutorial.

Where things become interesting again is the code which packages our sensor data up in the standard Velisphere JSON format and sends it off to the message broker.

The good thing Is, you don't have to worry about most of this, as we can use a single method of the amqpClient to take care of this. In our case, it's the following line:

```
amqpClient.sendHashTable(messageHash, ConfigData.epid, "REG");
```

`messageHash` in this case is a variable of the type HashMap that we have declared earlier:

```
HashMap<String, String> messageHash = new HashMap<String, String>();
```

A HashMap is a Java structure that takes stores key/value pairs. This is exactly what we need, as we need to store a sensor value (the value) that is assigned to a property (the key) of our endpoint.

To add these keys and values to the HashMap, we use the following line:

```
messageHash.put(touchSensorID, String.valueOf(ik.getSensorValue(6)));
messageHash.put(lightSensorID, String.valueOf(ik.getSensorValue(7)));
messageHash.put(pressureSensorID,
String.valueOf(ik.getSensorValue(5)));
messageHash.put(dialKnobID, String.valueOf(ik.getSensorValue(1)));
```

What follows are several lines of code to obtain the geolocation from GeoLite. Specific to VeliSphere is again this line:

```
messageHash.put(locationID, "{" + String.valueOf(location.latitude) +
"}" +
                                        "[" +
String.valueOf(location.longitude) + "]");
```

Here we are putting the geolocation obtained from GeoLite into our HashMap. Please note that the format for submitting geolocations to Velisphere is a string structured as follows:

```
{latitude in degree}[longitude in degree]
```

When we have added the value of the four sensors and the geolocation from GeoLite, we are ready to send our key/value hashmap to Velisphere, using the following call already mentioned above:

```
amqpClient.sendHashTable(messageHash, ConfigData.epid, "REG");
```

The more generic way to describe this is

```
amqpClient.sendHashTable(HashMap, EndpointID, MessageType);
```

The EndpointID is loaded from the velisphere_config.xml file introduced in step 4, and can be accessed using the static variable ConfigData.epid provided by the SDK.

The message type we are using is “REG”, for regular message.

That's it – what follows is just a bunch of error handling code which you probably want to turn into somehting more meaningful later on.

Let's finally build our main class PiSensorsMain.

```
package com.velisphere.demo.rpiSensors;

import java.io.IOException;
import com.phidgets.PhidgetException;
import com.velisphere.milk.amqpClient.AmqpClient;
import com.velisphere.milk.configuration.ConfigFileAccess;
import com.velisphere.milk.configuration.Provisioner;

public class PiSensorsMain {

  public static void main(String[] args) throws IOException {

        System.out
                    .println("
-----------------------------------------------------------------------
-------------------------");
        System.out
                    .println(" VELISPHERE DEMO: RaspberryPI Sensor
Device using Phidgets");
        System.out.println(" (C) 2015 Thorsten Meudt");
        System.out
                    .println(" Licensed under the GPLv2 license,
http://www.gnu.org/licenses/old-licenses/gpl-2.0.de.html");
        System.out
```

```
                        .println(" Learn more about the VeliSphere IoT
System at www.connectedthingslab.com");
                System.out
                        .println("
----------------------------------------------------------------------
------------------------");
                System.out.println(" ");

                // Load Configuration Data

                ConfigFileAccess.loadParamChangesAsXML();

                // Check if device is already deployed. If not, trigger pre-
deployment
                // cycle

                System.out
                        .println(" [IN] Calling provisioner to determine
if device is flagged as provisioned.");

                if (Provisioner.isProvisioned() == false)
                        PreDeployment.initiateDeployment();
                else
                        regularStartup();

        }

        public static void regularStartup() {

                // Activate Event Responders

                PiEventListener eventListener = new PiEventListener();

                // Start Server and activate listener

                AmqpClient amqpClient = new AmqpClient(eventListener);
                amqpClient.startClient();
                PhidgetMonitoringEngine engine = new
                        PhidgetMonitoringEngine(amqpClient);
                try {
                        engine.startMonitoring();
                } catch (PhidgetException e) {
                        // TODO Do something meaningful for error handling
                        e.printStackTrace();
                } catch (IOException e) {
                        // TODO Do something meaningful for error handling
                        e.printStackTrace();
                }

        }

}
```

This class consists of two static methods, **main** which is called when the Java executable is started, and **regularStartup**, which is called by main.

Looking at the **main** method first, the first step is to load the configuration data from the velisphere_config.xml file.

```
ConfigFileAccess.loadParamChangesAsXML();
```

Next, we do a simple if/then/else to determine whether the device needs to be provisioned first, or whether this step has already been completed and regular startup can commence:

```
   if (Provisioner.isProvisioned() == false)
         PreDeployment.initiateDeployment();
   else
         regularStartup();
```

That's all.

Now look at the **regularStartup** method:

First we instantiate an **EventListener** from (using the class in the SDK).

```
PiEventListener eventListener = new PiEventListener();
```

Next we instantiate an amqpClient in which we inject our eventListener and start the amqpClient.

```
AmqpClient amqpClient = new AmqpClient(eventListener);
amqpClient.startClient();
```

We finally start the **PhidgetMonitoringEngine** we built in step 21 by calling

```
PhidgetMonitoringEngine engine = new
                     PhidgetMonitoringEngine(amqpClient);
engine.startMonitoring();
```

It is important that you do this after creating the amqpClient, as you need to inject the amqpClient as a dependency.

23. Our code is complete! We now need to create a **runnable configuration** so that we can run it as a stand alone application on our raspberry pi.

    To do so, open the **RUN** menu in the Eclipse main menu bar and select **RUN CONFIGURATIONS…**

24. In the following dialog, click select **Java Application** in the list on the left, make a

right click on it and click **NEW** in the context menu.

25. A new run configuration is created. On the right side of the dialog, we need to give a name to our configuration, we call it **PiDemoConfig**. Leave all other values at their defaults and hit **APPLY** and then **CLOSE.**

26. All that is left to do is to export everything into an executable JAR file. To do this, make a final right click on your project **PiDemo** and click **EXPORT.**

27. Expand the Java folder and select **Runnable JAR file** and hit the **NEXT** button
28. In the "Runnable JAR file export" dialog, select the runnable configuration we just created ("PiDemoConfig – PiDemo") in the "Launch Configurations" dropdown list.

29. Select a path to which you want to export your file in "Export Destination". On my computer, I chose "/home/thorsten/Documents/tutorials/PiDemo.jar", but you are free to chose any path you like. However you need to be able to locate this file later on on your computer.

30. Make sure you select "Extract required libraries into generated JAR" under "Library handling".

31. Hit **FINISH**

32. If a warning about referenced libraries shows up, confirm with **OK.**

33. Download the GeoLite city database binary from here

    http://dev.maxmind.com/geoip/legacy/geolite/

    and extract the contained GeoLiteCity.dat file to a temporary location.

34. Now, copy the following files into a directory on your RaspberryPI:

    - the created JAR file
    - velisphere_config.xml
    - GeoLiteCity.dat

    You probably will use a tool like WinSCP when using a Windows machine, but I assume you are familiar with how to get a file copied to the RaspberryPi.

IV. Starting the client

Now it's time to start your client for the first time and provision it with Velisphere.

1. Make a ssh connection to your RaspberryPi. How you do this depends on the OS you are using on your developer machine (for instance using PUTTY if you're on windows, or the SSH command on Linux or MacOS).

2. Navigate to the directory where you stored your files in step 33 and enter the following command to start your client:

   ```
   java -jar ./PiDemo.jar
   ```

   Note that if you used a different file name in step III/29, you will have to use this filename here instead of PiDemo.jar

3. If everything goes well (it is important that your RaspberryPi has a working internet connection), the client will execute the PreDeployment routine and return a DeviceID – the MAC address of your Pi.

4. Go to www.velisphere.com, log in with your user account and click on "Deployment Wizard" on the right side of the screen.

5. Enter the DeviceID returned in step 2 and the captcha word and hit the SEARCH button

6. Velisphere should find your device in an unassigned state. In this screen, click on CLAIM OWNERSHIP to complete the provisioning process and add the newly created device to your inventory by giving it a name, like "My First Velisphere Client".

7. Go back to your SSH connection to the RaspberryPi and start our client:

   ```
   java -jar ./PiDemo.jar
   ```

   As the device is now fully provisioned, the client will connect with the message broker and start sending sensor values to VeliSphere. You can follow this on your screen as the client currently is fairly verbose, providing a lot of debug information.

Congratulations!