Smart Systems Technology

Exercise 1: The Cloud-connected potentiometer

Written by Marien Wolthuis. Last edit: 27/01/16

Before we can start making complex applications, it's important to familiarise ourselves with the platform(s) in use in this course. For this purpose, you will connect a simple sensor – a potentiometer – to the ThingWorx platform.

Prerequisite skills
This exercise assumes you have the following skills before starting:

- Experience in programming an Arduino:
  - Knowledge of the data types: int, double, float, String, char, char[]
  - Using and defining functions and their return variables
  - Using software developed by others
  - Troubleshooting the code that you're using
  - Installing libraries
- Basic knowledge of HTML
- Basic electronics skills:
  - The ability to read electronic diagrams
  - The ability to build circuits from diagrams
  - The ability to calculate resistor values appropriate for these circuits

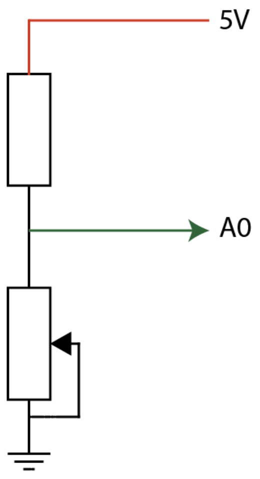
Before you start
It is highly recommended to read through and follow this guide from Sparkfun on how to use the ESP8266 WiFi shield: **https://learn.sparkfun.com/tutorials/esp8266-wifi-shield-hookup-guide**
Apart from the AT commands, knowledge of what is in this guide is presumed for the rest of the course. We also assume you imported the libraries mentioned in this guide.
The ESP8266 shield cannot connect to the Eduroam network due to the security settings of this network. It is therefore recommended to either use a home network (with WPA2-like security) or to set up an ad-hoc network from your computer or phone. During the workshops we will set up a suitable network for you.

1.1 Hardware set-up

For this exercise it is needed to have an Arduino (R3 preferred) and a Sparkfun ESP8266 WiFi Shield. The shield should be plugged in the Arduino, and allows the normal use of all the Arduino's pins except 0,1, 8 and 9.

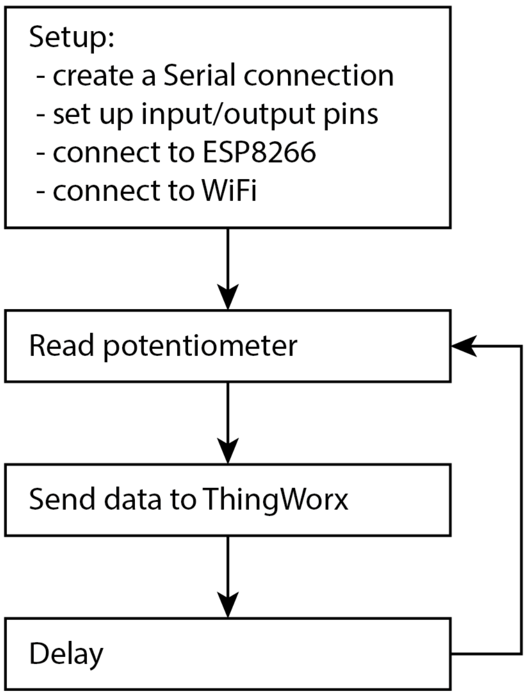
Because the shield occasionally uses more power than can be provided from the USB connection to your computer, it is highly recommended to power the Arduino with an adapter or battery connected to its power connector. This can be any voltage between 7 – 12V, but it is recommended to use 9 Volt since this is both available on most power supplies and as a battery.

The sensing electronics for this exercise are quite simple, as can be seen below:



1.2 Arduino Software design

In essence, the program that will be running on the Arduino is quite straightforward, as can be seen from the flowchart below:



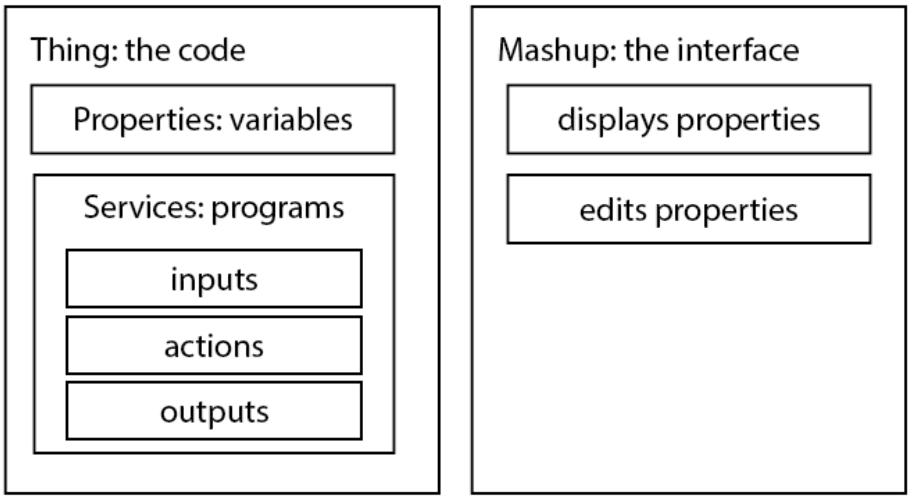
The most difficult tasks in this program, connecting to the ESP8266 shield and to the WiFi network, will be handled by functions provided for you by Sparkfun that have been adapted for this course. These set everything up and send data to the server when asked, and are based upon functions provided by Sparkfun in their library and examples.
Those tasks will be explained, but you are not expected to have a detailed knowledge of their inner workings. How to send data to ThingWorx, however, is something you are expected to understand thoroughly after this exercise.

Side-note

As you might notice, you will see what looks like a function call to "F(String)" - this is actually a barely documented "macro", that tells the Arduino to not load entire Strings into its working memory at once (since we have very little of that), but byte-by-byte. Long story short: It's good practise to use it, because it saves you some memory space for variables – especially since the libraries needed for the shield use a large portion of the available memory already.

1.3 ThingWorx basics

ThingWorx is an Internet platform for creating Internet of Things (IoT) applications. It works with “Things”, which function as the code that processes and holds information and “Mashups” which function as a graphical interface to the Things. A basic schematic of this system can be seen below:

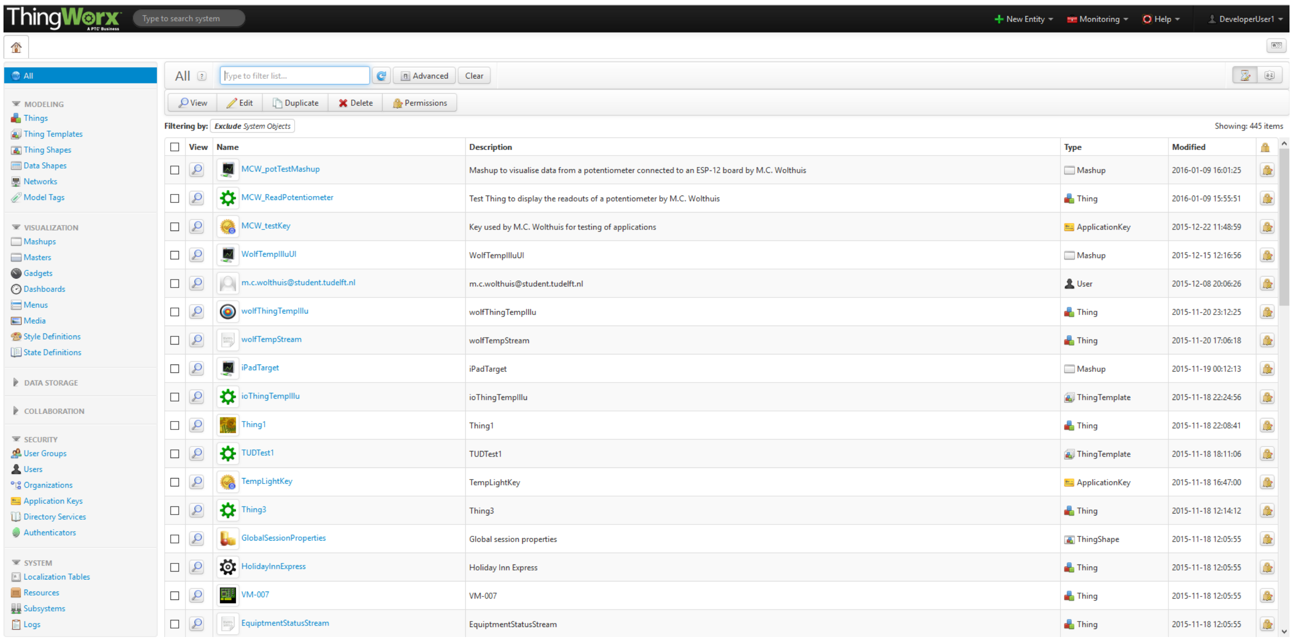


What you can see here is that Things contain both “Properties”, which function as variables like you'd find in Arduino code, and Services, programs that do something with these properties. The Services in turn contain inputs, outputs and actions; just like Arduino sketches contain inputs and outputs (digital or analog pins), and code to react to the inputs (for instance a button) by manipulating the outputs (like a LED) based on those.
Mashups are in essence Web pages that can display the properties visually and can contain elements where users can edit these properties.

In this exercise we will use a service to change the value of a property based on the data we receive from the Arduino, and display this property in a mashup. In order to do this, we first need to set up the Thing, since we need the names of the thing, the properties and services involved to be able to send data to it.

1.4 Creating our ThingWorx program

First, go to http://tudelft.cloud.thingworx.com/Thingworx/Composer/ and log in with the credentials you have received for this course. After logging in, you will see a screen that looks like the screenshot below:



Filtering by: *Exclude* System Objects

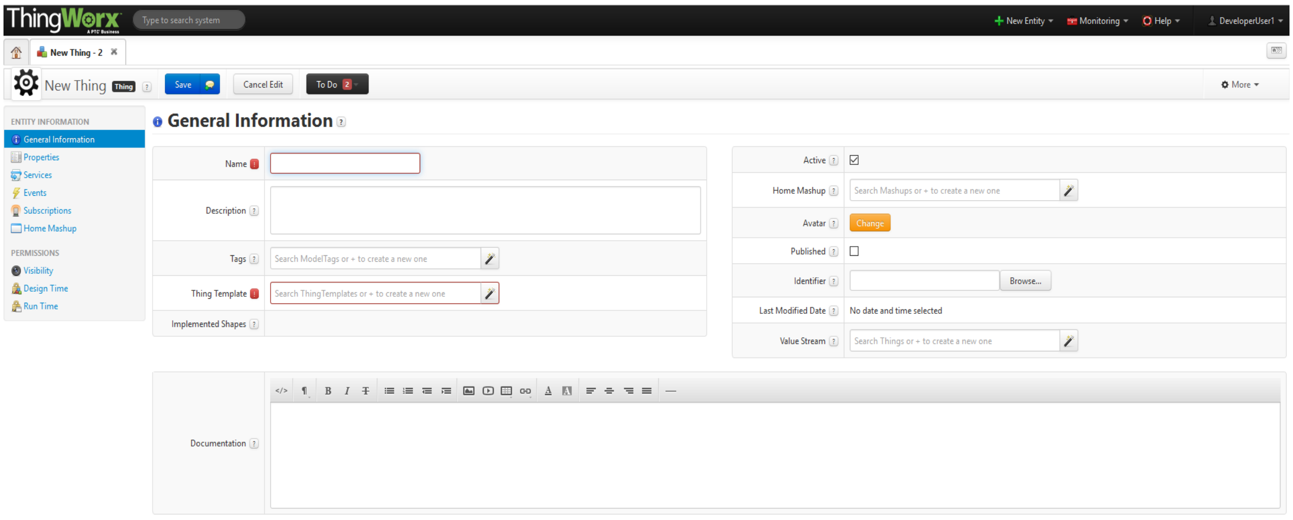
Showing: 445 items

View	Name	Description	Type	Modified
	MCW_potTestMashup	Mashup to visualise data from a potentiometer connected to an ESP-12 board by M.C. Wolthuis	Mashup	2016-01-09 16:01:25
	MCW_ReadPotentiometer	Test Thing to display the readouts of a potentiometer by M.C. Wolthuis	Thing	2016-01-09 15:55:51
	MCW_testKey	Key used by M.C. Wolthuis for testing of applications	ApplicationKey	2015-12-22 11:48:59
	WolfTemplIluUI	WolfTemplIluUI	Mashup	2015-12-15 12:16:56
	m.c.wolthuis@student.tudelft.nl	m.c.wolthuis@student.tudelft.nl	User	2015-12-08 20:06:26
	wolfThingTemplIlu	wolfThingTemplIlu	Thing	2015-11-20 23:12:25
	wolfTempStream	wolfTempStream	Thing	2015-11-20 17:06:18
	iPadTarget	iPadTarget	Mashup	2015-11-19 00:12:13
	ioThingTemplIlu	ioThingTemplIlu	ThingTemplate	2015-11-18 22:24:56
	Thing1	Thing1	Thing	2015-11-18 22:08:41
	TUDTest1	TUDTest1	ThingTemplate	2015-11-18 18:11:06
	TempLightKey	TempLightKey	ApplicationKey	2015-11-18 16:47:00
	Thing3	Thing3	Thing	2015-11-18 12:14:12
	GlobalSessionProperties	Global session properties	ThingShape	2015-11-18 12:05:55
	HolidayInnExpress	Holiday Inn Express	Thing	2015-11-18 12:05:55
	VM-007	VM-007	Thing	2015-11-18 12:05:55
	EquiptmentStatusStream	EquiptmentStatusStream	Thing	2015-11-18 12:05:55

This may look confusing at first, but after this exercise you will be able to make sense of it quickly. On the left side of the screen you see the “Modeling” and “Visualization” categories. These are the ones most important in using ThingWorx, since this is where we can make and display Things and Mashups. The centre of the screen shows a list of the created elements in the workspace. In the case of the above screenshot, the filter is set to “All”, as indicated by the blue bar on the left. By clicking other items on the left menu we can filter out the separate types. When you mouse over one of the types on the left, you will see a green “+” icon. By clicking this, you create a new item of that type.

Creating a new Thing

Click on the “+” icon that appears next to “Thing” when you mouse over it in the left menu. This will bring you to the following screen:



In order to create a new Thing, we need to enter at least a name and select a template. Templates are default blueprints for your Things, a feature that is very useful if you have to create multiple similar Things.

For the name it is recommended to use a name that is both descriptive and unique. In the examples for this exercise we will use a prefix in the name: "SST_". When you create your own Things, you can use your initials instead. For instance, "Yu Song" would have the prefix "YS_" for all his created Things and Mashups.
The "Description" field is optional, but it's good practise to always provide one so other people (or you, after taking a break from a project) can see at a glance what the purpose of the Thing is.
This also goes for the "Tag" field; using a unique tag for your project can help you search for all Mashups and Things that relate to it easily. To create a Tag, select the Tag field, click on the wand icon and click "+ Term". Under "New Term:", enter the name of your new Tag. To use an already existing Tag, start typing its name and select it or click on the wand icon and select it from the list.

Bearing the above in mind, create a Thing with the following settings:

Name:	SST_ReadPotentiometer
Description:	Thing to display the readouts of a potentiometer connected to an Arduino.
Tags:	SST_exercise1
Thing Template:	GenericThing

Creating a Property
To create a property, select "Properties" from your Thing's left menu. Then click "+ Add My Property" and enter the following settings for it:

Name:	potMeterValue
Description:	The value of the potentiometer connected to the Arduino.
Base Type:	INTEGER

Then click "Done" if you're finished or "Done and Add" if you need to create more Properties.

Creating a Service
Now that we have a Property to save our potentiometer's value in, we also need to have something to receive the data that we will send from the Arduino and save it to the Property. This is done by creating a new Service. Services are programmed in a language called "JavaScript", which is one of the most important programming languages for Web sites. It is similar to what you know from Arduino, but for the purpose of this assignment you will not need to write it yourself. The JavaScript is what dictates the actions of the Service on its inputs and outputs.

To create a Service, select "Services" from your Thing's left menu, and click "+ Add My Service" and enter the following settings:

Name:	SST_setPotValue
Description:	A service to save the value the Arduino sends to the potMeterValue property.
Script:	`me.potMeterValue = parseFloat(potMeterValue)`

Then go to the Input/Output tab, and click "+ Input". Create an input with the following settings:

Name:	potMeterValue
Description:	The value received from the Arduino.
Base Type:	STRING

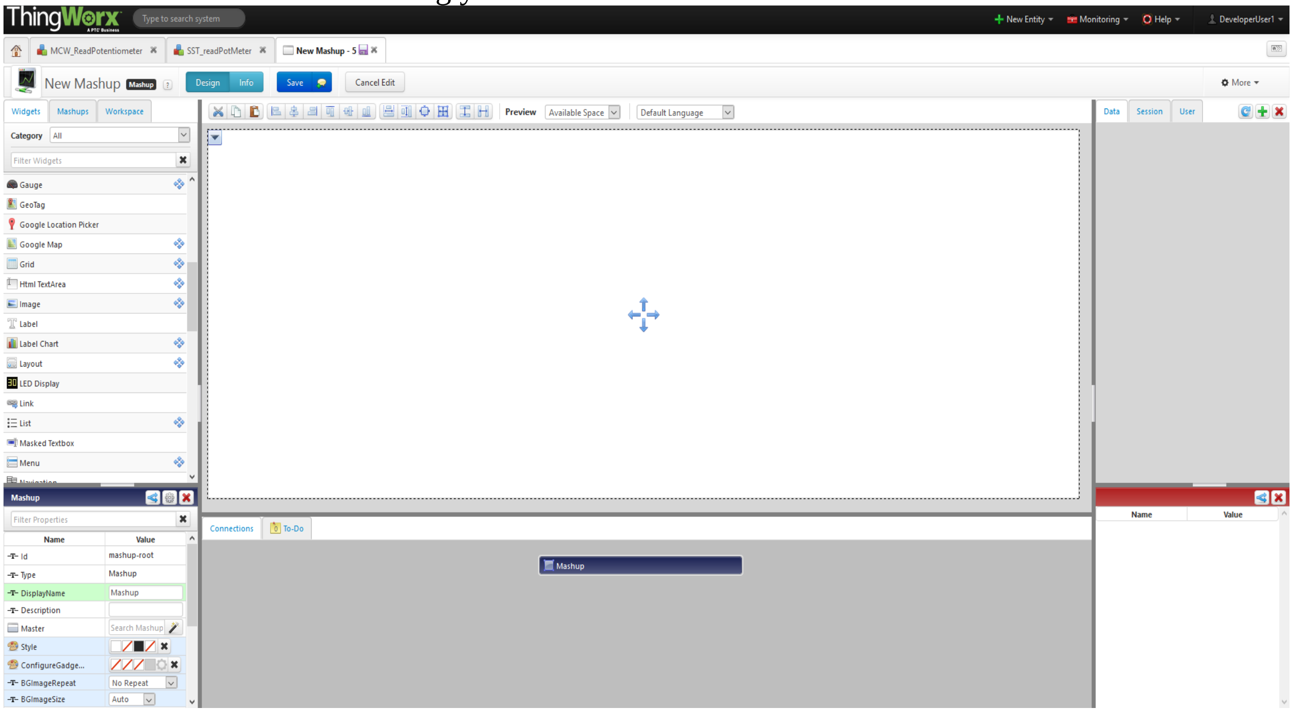
And click "Done" on the pop-up screen, and then "Done" on the bottom right.

It is important to use the same name as the Property you created earlier. While it normally not a good idea to recycle variable names, it will not work if you choose a different one!
The base type of this input may be confusing at first, but it should be a String because of the way the data is sent by the Arduino. It is sent via a HTTP "`POST`" message, which will convert it to a String to send it. This is then compensated by calling "`parseFloat()`" on the String we received, converting it to a number again.

We are now done with creating the Thing, so click the blue "Save" button on the horizontal menu next to "New Thing". Then go back to the overview page by clicking the house icon on left-most tab.

Creating the Mashup
Creating a Mashup is similar to creating a Thing. Click on the green "+" icon when you mouse over the "Mashups" button on the left menu to bring up the "New Mashup" screen. In this screen you can select a few options, but we only need to use the two that are selected by default:

Mashup Type:	Page
Layout Options:	Responsive

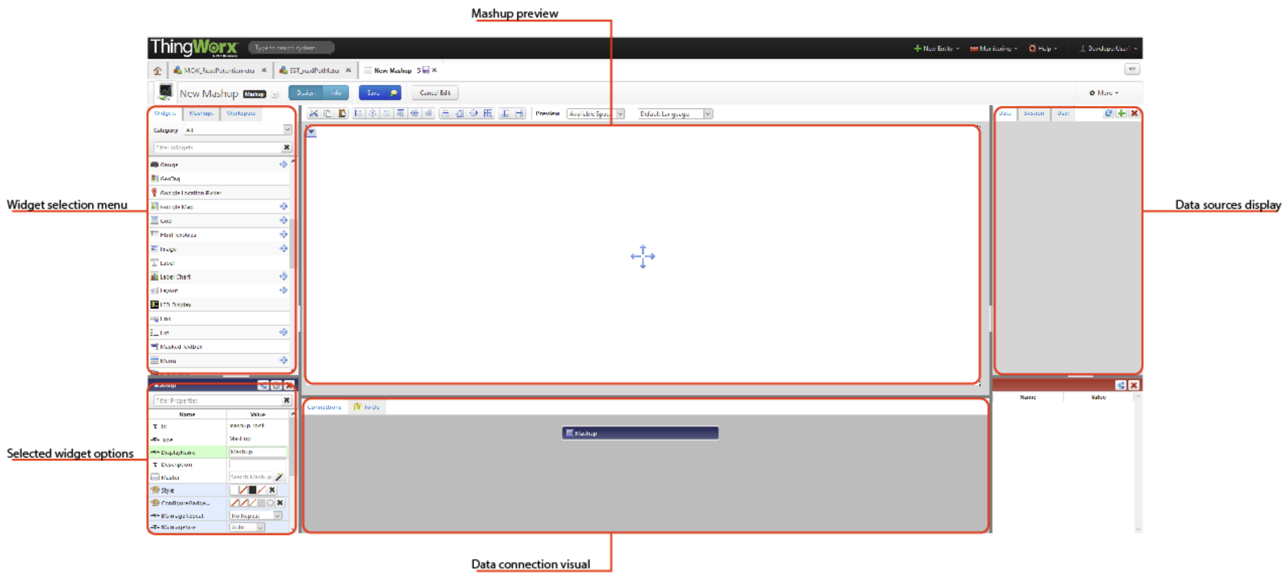
Then click "Done". This will bring you to the screen below:


We'll explain the use of the different parts of this screen in a moment. First, we'll give it a name, description, and tag again. Click on the bright blue "Info" button on the top of the screen. On the info page, enter the following:

Name:	SST_showPotValue
Description:	Displays the value of potMeterValue on a gauge.
Tags:	SST_exercise1

And click "Save". This will take you out of the edit mode, so click on the orange "Edit" button when the Mashup has been saved.

The Mashup screen has several interesting parts. It has a list of widgets you can select from (top left), the Mashup preview screen (centre), the data sources for the Mashup(right), the options associated with the currently selected widget (bottom left), and a visualisation of the data of the currently selected widget (bottom). How they work will become clear when we create our visualisation of the potentiometer.



First, in order to be able to put multiple display widgets on the page, we will need a “panel”. You can find this widget by typing “panel” in the search box in the widget selection menu. When you've found it, click and drag it to the preview area. You will notice the blue arrows in the preview area have disappeared and that the widget options menu now says “Panel-1” or something similar and displays the options for the panel.

Next, we want one of those nifty little gauges to display the potentiometer's value. Add one by searching for the “gauge” widget in the same way you did with the panel and dragging it to a spot on the page you like. You can resize it if you like by selecting one of the black squares and dragging it.

If we press “view Mashup” now, you will see a gauge where you placed it, but without a needle. This is because the gauge doesn't have any data to display yet. To fix this, go back to the Mashup edit window. In the data sources display area, click on the green “+” icon. This allows us to add a data source to the Mashup. In the search area, type “ReadPotentiometer” to find the Thing we created earlier, and click its name. We can now select from a list of services. The advantage of that we created our Thing based on a template is that we have a lot of default services to use without having to go through the effort of programming them. Type “GetProperties” in the search field and select the service by clicking on the blue arrow. It is now listed on the field to the right. Tick the “Mashup Loaded?” box to have this service be loaded when the Mashup is started, then click “Done”.

You will now see that the data sources display area has two new items: a black bar with the name of our Thing and a red one that says “GetProperties”. If you click the “+” next to GetProperties, you can see all the properties we can access in our Mashup. Click and drag the “potMeterValue” property onto the gauge to link them. We want to link it to the “data” property of the gauge, so select that option. The gauge can now display the potentiometer’s value as soon as we feed it to ThingWorx.

If you select the gauge and look at its options, you can see that the MinValue and MaxValue are set to 0 and 100. We know that a value read by the Arduino can range between 0 and 1023, so we will have to adjust the MaxValue to accommodate this. Type the number in the corresponding field, and then press enter to apply the change.

To see the exact number the Arduino sends to ThingWorx, also add a LED display in the same way you added the gauge. Since we already linked the getProperties service to our Mashup, we only have to link the potMeterValue to the LED display again.

By now, we're almost done creating our ThingWorx Mashup. While the “GetProperties” service is loaded when we view the Mashup, it does not update while the page is open. We can change this by adding an “Auto Refresh” widget to the Mashup. Find it in the widget selection menu and add it. If you mouse over the top left corner of this widget, you will notice the word “Refresh” and an arrow. Drag this over to the data sources menu and drop it onto the red “GetProperties” bar. You will notice in the data connections visualisation that the GetProperties service is now activated by both loading the Mashup and by the Auto Refresh widget. You can adjust the settings of the Auto Refresh so it refreshes about as often as you send data instead of every 30 seconds to get a smoother value display. Also, if you don't like the look of the Auto Refresh on your page, simply untick the “ShowControls” box.

In summary, we have now created:

Thing	SST_readPotentiometer
Property	potMeterValue
Service	SST_setPotValue
Mashup	SST_showPotValue

Always keep a list like this when working with ThingWorx, since we will need these names in the following step to be able to send data from the Arduino to ThingWorx. After creating both the Mashup and the Thing, we can also add the Mashup as the 'default Mashup' to the Thing by editing the settings of the Thing. This makes it easier to find the corresponding Mashup, since it will now be accessible under “default Mashup” in the sidebar of the Thing.

1.5 The Arduino code

Now that we know where to send our data, we can get our Arduino code up and running. A large part of the code we use here is written for you by either SparkFun or the SST staff to connect the shield to the Arduino and to ThingWorx. While most of the functions of this code should be clear from the provided comments, they will be briefly explained here as well.

Sending a message

One last topic to address before we start with the writing and examining of Arduino code is the message that will be sent to ThingWorx. This message is what is called an "HTTP POST request". HTTP requests are all messages that are sent as an initiation of Internet communication between a client and a server. For instance, if you tell your browser to show you "www.google.com", the browser will send an HTTP GET request to the server to ask it to send back the HTML code that shows you the page.
POST messages are a method to send data to a server. The request we are sending to ThingWorx must have a specific format for ThingWorx to accept it. This format can be described as below:

Please note that for legibility some lines are split; the ones starting with an indent are continuations of the previous line.

```
POST /ThingWorx/Things/thing_name/Services/
      service_name?appKey=appKey&method=post&x-thingworx-session=true
      <&property1_name=property1_value&property2_name=property2_value> HTTP/1.1
Host: thingworx_server
Content-type: text/html
```

All text in **bold** here is dependent on your ThingWorx application. The appKey is a way to bypass the normal username/password login to ThingWorx, and is specifically generated for you by the staff. For our example application, the request would then be as follows (excluding the app key, which you should have received from the staff):

```
POST /ThingWorx/Things/SST_readPotentiometer/Services/
      SST_setPotValue?appKey=appKey&method=post&x-thingworx-session=true
      <&potMeterValue=500> HTTP/1.1
Host: tudelft.cloud.thingworx.com
Content-type: text/html
```

Assuming that the value we want to send is "500".

On the next pages we will give you the entire code needed for the example project. There are some things that need to be changed in order for it to work for your Thing and network; simply search for "TODO:" in the code and you will find this. It is up to you to understand the code and to be able to use it for your assignment.

Part 1: including libraries and defining constants
A major issue that can be encountered by the Arduino when working on larger projects is its low amount of memory. To combat this, we will use a 'trick' in the C language: we can use `#define` to create things that work like variables, but are modified by the compiler to be 'hard-coded' into the instructions it creates for the Arduino. The practical upshot of this is that we can use certain words like we would use normal variables, but they won't take up space in the memory used for variables at all!

This part of the code then looks like this:

```
#include <SoftwareSerial.h>
#include <SparkFunESP8266WiFi.h>

#define mySSID "TODO:YOUR_NETWORK_NAME_HERE"
#define myPSK  "TODO:YOUR_PASSOWRD_HERE"

#define destServer "tudelft.cloud.thingworx.com"

//ThingWorx App key which replaces login credentials
#define appKey "TODO:YOUR_APPKEY_HERE"

// ThingWorx Thing name for which you want to set properties values
// TODO: change this to your Thing's name
#define thingName "SST_ReadPotentiometer"

// ThingWorx service that will set values for the properties you need
// TODO: change this to your Service's name
#define serviceName "SST_setPotValue"

// Name of the variable used in Thingworx
// TODO: change this to your Property's name
#define propertyName "potMeterValue"

//Interval of time at which you want the properties values to be sent to TWX
server [ms]
#define timeBetweenMessages 500

// TODO: change this to the pin you are using
#define potPin A0
```

Since we can only use the `#define` trick for variables that can not be changed, this is all that we can use it for. Still, that's quite some memory saved!

Side-note
One thing to remember is that often the things for which `#define` has been used are written in capital letters. So, if you are reading someone else's code and it uses a variable name that is not defined in the file itself and in capitals, it's probably created with a `#define` in some included library.

Part 2: Variables and setup()
The next bit should be quite familiar. The `setup()` is mainly comprised of functions that have been supplied by SparkFun. Don't worry if you don't understand exactly how they work; focus instead on what they do that you need.

```
String httpRequest;
bool displayResponse = true; // this controls whether we see the HTTP response
bool displayRequest = true;  // this controls whether we print the request we
                             // send

// Initialised to -1 to be able to see whether we read any values from the
// sensor
int potValue = -1;


void setup(){
  Serial.begin(9600);

  // initializeESP8266() verifies communication with the WiFi shield,
  // and sets it up.
  initializeESP8266();

  // connectESP8266() connects to the defined WiFi network.
  connectESP8266();

  // displayConnectInfo prints the Shield's local IP and the network it's
  // connected to.
  displayConnectInfo();

  pinMode(potPin, INPUT);
}
```

Part 3: Constructing our POST message

The next part is the `loop()`. As you can see from the flowchart in 1.2, we need to both read the sensor and send data in the loop. To be able to send it, this is also where we will construct our POST request.

```
void loop()
{

      potValue = analogRead(potPin);
      Serial.print(F("potValue = "));
      Serial.println(potValue);

      // now make a request to post this to the server.
      httpRequest = "POST /Thingworx/Things/";
      httpRequest+= thingName;
      httpRequest+= "/Services/";
      httpRequest+= serviceName;
      httpRequest+= "?appKey=";
      httpRequest+= appKey;
      httpRequest+= "&method=post&x-thingworx-session=true";
      httpRequest+= "<&";
      httpRequest+= propertyName;
      httpRequest+= "=";
      httpRequest+= potValue;
      httpRequest+= "> HTTP/1.1\r\n";
      httpRequest+= "Host: ";
      httpRequest+= destServer;
      httpRequest+= "\r\n";
      httpRequest+= "Content-Type: text/html\r\n\r\n";

      if(displayRequest) {
            Serial.println(F("Sending request:\n"));
            Serial.println(httpRequest);
      }else{
            Serial.println(F("Sending data to ThingWorx."));
      }

      sendRequest();

      delay(timeBetweenMessages);
}
```

As you can see, we create the `String httpRequest`, which we will send to the server, in multiple steps. The += operator is very useful here, since it can easily add things to the existing `String`. A simple explanation of it is that “`i+=1`” is the exact same instruction as “`i=i+1`”, but shorter. Especially with longer variable names using += is more readable and prevents typing errors.

Side-note

While this may seem like a very verbose and long method of constructing a `String`, using shorter versions caused the code to have memory issues and magically restart the Arduino. These memory issues are difficult to deal with since they do not generate an error message, but do cause weird behaviour. Keep this in mind if your code ever does something unexplainable and unexpected!

A thing to note is that in the Arduino code you will see “`\r\n`” at some points in the `httpRequest String`. These are special characters; `\r` denotes a “carriage return”, meaning it tells the program reading the String that this is the end of this line of text, and \n denotes a “new line”, meaning the text will be on a new line. The latter is easily understood by comparing the following commands on your Arduino:

```
Serial.print(“The text after this ”);
Serial.println(“will be on the same line. ”);

Serial.println(“The text after this ”);
Serial.println(“will be on a new line.”);

Serial.print(“The text after this \n”);
Serial.println(“will also be on a new line.”);
```

```
                    –------- Serial output below--------
```

```
The text after this will be on the same line.
The text after this
will be on a new line.
The text after this
will also be on a new line.
```

According to the HTML standard for POST requests, we need to end each line with both a carriage return and a newline character. This is why we use “`\r\n`” after certain parts of the request. The last line is an empty message 'body', which is why we use it twice.

Side-note
Technically, the POST request we are sending here is not a POST request at all. This is a quirk in the ThingWorx API; they ask us to send a POST message but the one we're sending is actually structured like a GET request, as you can see from the examples at http://www.w3schools.com/tags/ref_httpmethods.asp.

Part 4: Sending the request
One of the last calls in the `loop()` is to a function called "`sendRequest()`". This is a function that has been adapted from the original SparkFun code for the purpose of this assignment. What it does is connect to the destination server we have defined earlier using `#define`, and attempts to send the contents of the `httpRequest String` constructed in the `loop()`. It can also display the response the server gives to this request if we set `displayResponse` to `true`. This will display the HTTP error code generated in response to our request. Confusingly enough, an error code of "200" means that there was no error and the message was received correctly.

The code for this function can be found below:

```
// Sends a HTTP request based on the contents of a variable called httpRequest.
// Originally written by SparkFun, adapted by Adrie Kooijman and Marien
// Wolthuis
void sendRequest()
{
  ESP8266Client client;
  // ESP8266Client connect([server], [port], [keepAlive]) is used to connect to
  // a server (const char * or IPAddress) on a specified port, and keep the
  // connection alive for the specified amount of time.
  // Returns: 1 on success, 2 on already connected, negative on fail
  // (-1=TIMEOUT, -2=UNKNOWN,-3=FAIL).
  int retVal = client.connect(destServer, 80, 100);
  if (retVal <= 0)
  {
    Serial.print(F("Failed to connect to server."));
    Serial.print(F("retVal: "));
    Serial.println(retVal);
    return;
  }

  // Send data to a connected client connection.
  client.print(httpRequest);

  // clear httpRequest to free up some memory
  httpRequest = "";

  // available() will return the number of characters currently in the receive
  // buffer.
  while (client.available()){
    char rep = client.read();
    if(displayResponse){
      // This outputs the entire response to Serial and saves none of it
      Serial.print(rep); // read() gets the next character
    }
  }

  // connected() is a boolean return value; 1 if the connection is active, 0 if
  // it's closed.
  if (client.connected()){
    client.stop(); // stop() closes a TCP connection.
  }
}
```

What you see happening on the first line of function is that something called "client" is created. What this means is that in the library files from SparkFun, a new *data type* has been created called "`ESP8266Client`". Just like Arduino has things like `String`, `int`, `bool`, etcetera, new types can be defined by skilled programmers using "classes". What we need to take away from this here is that this "client" can hold a different type of information and contains functions to deal with data.
For instance, it is useful to know that we can send our HTTP message by using client.print(), but

exactly how this works is not important to us now.

After we have sent our request, we do not need the contents of the httpRequest String any more until the next loop, where we will re-create it anyway. To save some memory for later use, it's therefore wise to make it as small as possible by deleting its contents.

What you can see near the end of the function is that we use client.read() to get the response from the server. We do not save the reply, since the entire reply would be a rather large thing to save (remember the memory issues we can run in to?). Because client.read() gives us a single character at a time (and automatically moves to the next one the next time we call it), we can print it character-by-character in the Serial Monitor instead.

Side-note
The use of `ESP8266Client` here is an example of object-oriented programming. SparkFun has written a 'class', which is a programming construct that can contain both information stored in it as well as functions to act on that or other information. A basic example of this is `Serial.print();` we use the function `print()` from the class `Serial` to do something with the information we provide it. The difference between `ESP8266Client` and `Serial` is that we can create an instance of the `ESP8266Client` class as has been done here to be able to access information from it, as you will see happening later in the code where instructions like `client.connected()` are used to ask the client whether there still is a connection with the server.

Part 5: Helper functions

The functions below are, as said, written by SparkFun and needed to connect the ESP8266 shield to the Arduino. It is wise to read these in order to understand the messages they may generate, but you do not need to thoroughly understand their inner workings.

```
// ---- Helper functions for the ESP, written by Sparkfun -------

void initializeESP8266()
{
  // esp8266.begin() verifies that the ESP8266 is operational and sets it up
  // for the rest of the sketch.
  // It returns either true or false -- indicating whether communication was
  // successul or not.
  int test = esp8266.begin();
  if (test != true)
  {
    Serial.println(F("Error talking to ESP8266."));
      // Inserted by Marien Wolthuis; easiest way to fix connection issues
      Serial.println(F("This is usually fixed by resetting the ESP8266 (ground
the RST pin for a second), then resetting the Arduino."));
    errorLoop(test);
  }
  Serial.println(F("ESP8266 Shield Present"));

}

void connectESP8266()
{
  // The ESP8266 can be set to one of three modes:
  //  1 - ESP8266_MODE_STA - Station only
  //  2 - ESP8266_MODE_AP - Access point only
  //  3 - ESP8266_MODE_STAAP - Station/AP combo
  // Use esp8266.getMode() to check which mode it's in:
  int retVal = esp8266.getMode();
  if (retVal != ESP8266_MODE_STA)
  { // If it's not in station mode.
    // Use esp8266.setMode([mode]) to set it to a specified
    // mode.
    retVal = esp8266.setMode(ESP8266_MODE_STA);
    if (retVal < 0)
    {
      Serial.println(F("Error setting mode."));
      errorLoop(retVal);
    }
  }
  Serial.println(F("Mode set to station"));

  // esp8266.status() indicates the ESP8266's WiFi connect status.
  // A return value of 1 indicates the device is already connected.
  // 0 indicates disconnected. (Negative values equate to communication
  // errors.)
  retVal = esp8266.status();
  if (retVal <= 0)
  {
    Serial.print(F("Connecting to "));
    Serial.println(mySSID);
    // esp8266.connect([ssid], [psk]) connects the ESP8266
    // to a network.
    // On success the connect function returns a value >0
    // On fail, the function will either return:
    //  -1: TIMEOUT - The library has a set 30s timeout
    //  -3: FAIL - Couldn't connect to network.
    retVal = esp8266.connect(mySSID, myPSK);
    if (retVal < 0)
    {
      Serial.println(F("Error connecting"));
      errorLoop(retVal);
    }
  } else {
    Serial.print(F("Already "));
  }
}

void displayConnectInfo()
{
  char connectedSSID[24];
  memset(connectedSSID, 0, 24);
  // esp8266.getAP() can be used to check which AP the ESP8266 is connected to.
  // It returns an error code.
  // The connected AP is returned by reference as a parameter.
  int retVal = esp8266.getAP(connectedSSID);
  if (retVal > 0)
  {
    Serial.print(F("Connected to: "));
    Serial.println(connectedSSID);
  }

  // esp8266.localIP returns an IPAddress variable with the ESP8266's current
  // local IP address.
  IPAddress myIP = esp8266.localIP();
  Serial.print(F("My IP: ")); Serial.println(myIP);
}

// errorLoop prints an error code, then loops forever.
void errorLoop(int error)
{
  Serial.print(F("Error: ")); Serial.println(error);
  Serial.println(F("Looping forever."));
  for (;;)
    ;
}
```

The `errorLoop()` function may cause some confusion: what this basically does is tell you what went wrong and then stop the execution of any other code by going in to an infinite `for()` loop. While this is not a very nice way to deal with problems, it can prevent things from escalating.

1.6 Online sources

For this exercise there are several online sources available:

SparkFun introduction to the ESP8266 shield:
Start with this to get to know the shield that is used in this course.
https://learn.sparkfun.com/tutorials/esp8266-wifi-shield-hookup-guide

Complete code for this assignment:
This online source can contain an updated version of the code used here.
https://github.com/dotCID/SST