Smart Systems Technology

Exercise 2: Light from the Cloud

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

To be able to develop a truly 'connected' application, data communication needs to go both ways. In this exercise we will build on the skills and knowledge acquired in exercise 1. We will be creating a Thing and a Mashup in the ThingWorx environment and have them control three LEDs that are connected to an Arduino. One of these will be blinking at an interval defined by the user of the Mashup, one will be turned on or off in the Mashup, and one will be used as an indicator that data was received.

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

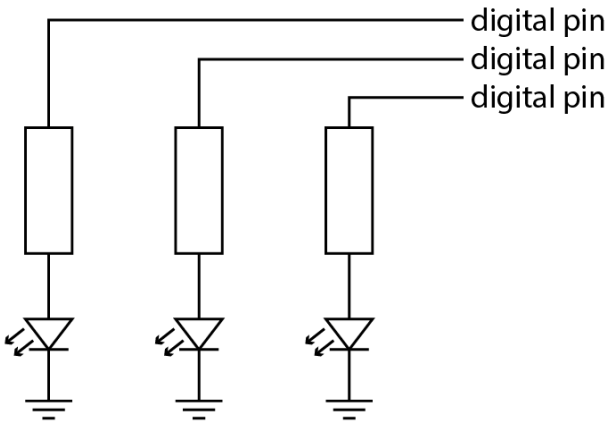
Before you start

This exercise continues where exercise 1 left off, and since a lot of the code for this assignment is the exact same as the code used in exercise 1, not all of it will be shared or discussed in this document. We therefore advise you to have exercise 1 in front of you as well, or at least the resulting Arduino code.

2.1 Hardware set-up

Like exercise 1 we will be using an Arduino Uno (preferably R3), an ESP8266 WiFi shield from Sparkfun, a power supply for those, and some simple electronics.

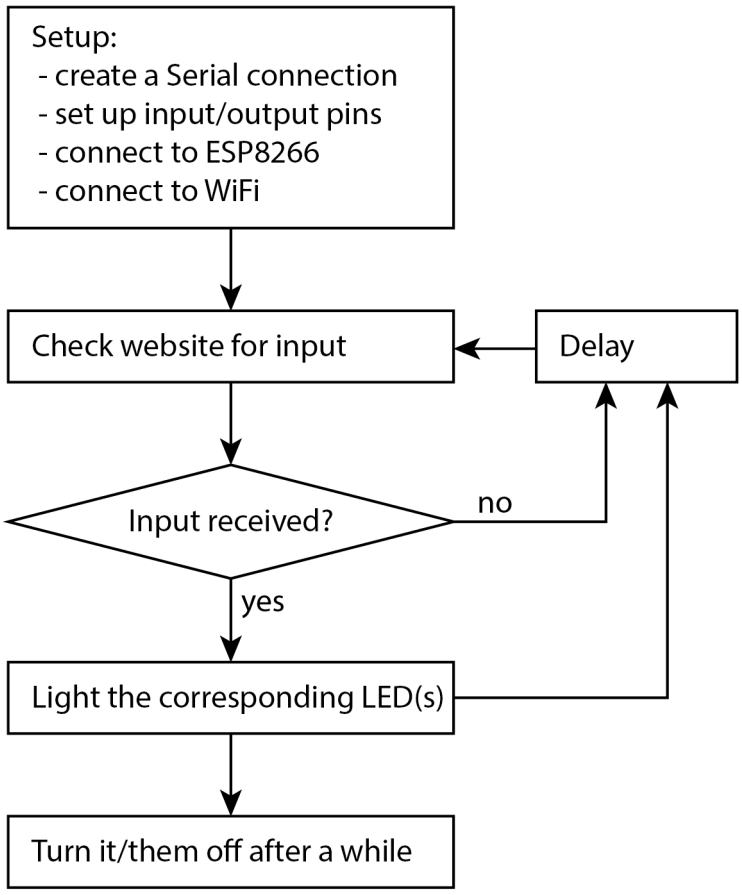
The electronics are briefly described in the diagram below. Please note that the resistor values may differ from another if you use differently coloured LEDs.



These LEDs are to be connected to separate digital pins. The specific pins only matters for their assignment in the code, as long as they are not pins 0,1, 8 or 9 since these are in use for the WiFi shield.

2.2 Arduino software

The program that we're running on the Arduino is similar to the one in the previous exercise, except we will get our data from the Internet rather than from a potentiometer. We will also use a slightly more sophisticated way of timing how long the blinking LED is on than you might be used to, in order to miss as little input as possible. The same method of timing is used to send requests to the server at regular intervals.



In order to get input from ThingWorx, we are going to ask it for a specific Web page of which we can control the contents based on the Properties in the related Thing. This Web page is then used by the Arduino to react to the user input from the Mashup we will create.

The most difficult part of the software we will need for this exercise is the Arduino code that interprets the response we receive from ThingWorx and saves it to variables the Arduino can work with. The hardest parts of this have already been written for you, but it is important to understand how the work on a general level since you have to adapt them slightly for your own project (and variable names).

Side-note

As you may have thought, it is also possible to do this by navigating to a certain page (let's say /arduino/set/blue_led/on) that is hosted on the Arduino. This, while a more elegant approach in concept, poses a problem that is sometimes hard to overcome (securely): the Arduino's IP address on the WiFi network should be reachable from the Internet to be able to access that page from ThingWorx.

2.3 ThingWorx set-up

For the name of your ThingWorx creations 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.

With that in mind, we can start by creating our Thing and Mashup. Just like in exercise 1, please navigate to http://tudelft.cloud.thingworx.com/Thingworx/Composer/ and log in with the credentials you have been given for this course.

Then, create a Thing by clicking the green "+" icon next to "Things" in the left menu and use the following settings:

Name:	SST_LedController
Description:	A Thing to control the behavior and status of several LEDs connected to an Arduino
Tags:	SST_exercise2
Thing Template:	GenericThing

Creating the Properties to hold data
Like was done in exercise 1, we will need some Properties to hold our data. Because we will be controlling 3 LEDs, we need three of them this time.

First, create the Property that will tell the Arduino to switch the status of the LED when it receives data:

Name:	messageLEDStatus
Description:	The status (on/off) of an LED connected to the Arduino that switches state when data is received.
Base Type:	BOOLEAN

Instead of clicking "Done" on the bottom right, it is useful to use "Done and Add" instead, since we need to add multiple Properties.
Next, add the Property that holds the delay at which we blink the second LED:

Name:	blinkDelay
Description:	The delay between switching an LED connected to the Arduino on and off.
Base Type:	INTEGER
Min Value:	5
Has Default Value?	[X]
Default Value:	250

In order to prevent unexpected behavior, we give the delay a minimum and default value.

Lastly, create the Property that holds the desired state of the third LED:

Name:	controlledLEDStatus
Description:	The status (on/off) of an LED connected to the Arduino that is controlled from the Mashup.
Base Type:	BOOLEAN

Then click “Done”.

Creating a Service to serve us data

Now that we have our Thing and Properties set up, we can create a Service to give us the data. Services can have an “output”: effectively this means that we can save all our calculations to a single variable, and this variable is accessible on its own Web page. Since we want three variables from ThingWorx and want to send as little requests to it as possible to save on calculations and memory in the Arduino, we will be using something called “JSON” to send all three variables at once.

JSON: a way to send data between applications in a readable way

JSON is an abbreviation for JavaScript Object Notation. It is in essence an array of labels and data points, which allows us to transfer these data points between ThingWorx and the Arduino all at once. Wikipedia summarises it as follows:

> *JSON is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is the primary data format used for asynchronous browser/server communication, largely replacing XML.*

Both of these are a rather cryptic description of a very simple concept. Since seeing it in its actual 'code' format is much clearer, here is an example of what it looks like for a string of text, a number, and a boolean value:

```
{
  'someText' : 'This is some text we want to send',
  'aNumber' : 6,
  'aBoolean' : true
}
```

The labels can be anything that is useful to us, and differs per application. An example specific to this exercise can be as follows:

```
{
  'messageLEDStatus' : true,
  'blinkDelay' : 232,
  'controlledLEDStatus' : false
}
```

We will be creating this in the JavaScript that is executed every time the Service we will create next is asked to serve data to the Arduino, just like we used JavaScript in exercise 1 to save the value we sent to ThingWorx every time it was received.

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

Name:	SST_getData
Description:	A service to send the Properties and their values from ThingWorx to an Arduino.

Under “Inputs/Outputs” select a data type for the “result”. If we do not select a data type, ThingWorx will not output any result! The data type we are using for this exercise is “STRING”. Since this Thing has no input, we are not selecting that. If it did have an input, it is considered 'good practise' to us a separate service called “set***” for this, with *** being the variable(s) you are setting the values of. For instance, in this case that would be “SST_setData”.

Please enter the following code in the Script field:

```
// We want the message LED to alternate between on and off every time we send
// some data, and since this script is executed every time data is
// requested, we can use this to invert the value of our Boolean every time
// this runs. For this we use the logical operator ! (not).
me.messageLEDStatus = !me.messageLEDStatus;

// Then we can construct our data string. For human readability this is split
// over several lines; for the JavaScript code this makes no difference.
result = "{ 'messageLEDStatus' : " + me.messageLEDStatus    + ", "
       +"'blinkDelay' : "          + me.blinkDelay          + ", "
       +"'controlledLEDStatus' : "+ me.controlledLEDStatus + " }";
```

After this, please click the “save” button on the top bar.

Creating the Mashup that will serve as an interface
In exercise 1 we used a Mashup to display data. In this exercise we will use it for that purpose, but also to set the value of the variables, creating a graphical user interface (GUI).

Once again, create a Mashup by clicking the green “+” icon next to “Mashup” in the left menu. In the pop-up menu, select the two settings that are selected by default:

Mashup Type:	Page
Layout Options:	Responsive

Then click “Done”.

Side-note
You may notice that rather than using “JSON” as the type we have opted for “STRING”. This is done on purpose, since we will not be using a 'proper' JSON interpreter in the Arduino code to save on memory use. Ordinarily we would use a feature-rich interpreter that is capable of converting JSON to an Array with named indices, but this proved to be too much for the Arduino to handle. A disadvantage of using a simpler implementation is that we cannot deal with a changing order of label/data pairs, which proved to sometimes happen when using JSON as output type. By using JSON notation and String as an output variable, we can control both the syntax and the order of label/data pairs much more easily, solving the problem.

Next, click on the bright blue “Info” button on the top of the screen. On the info page, enter the following:

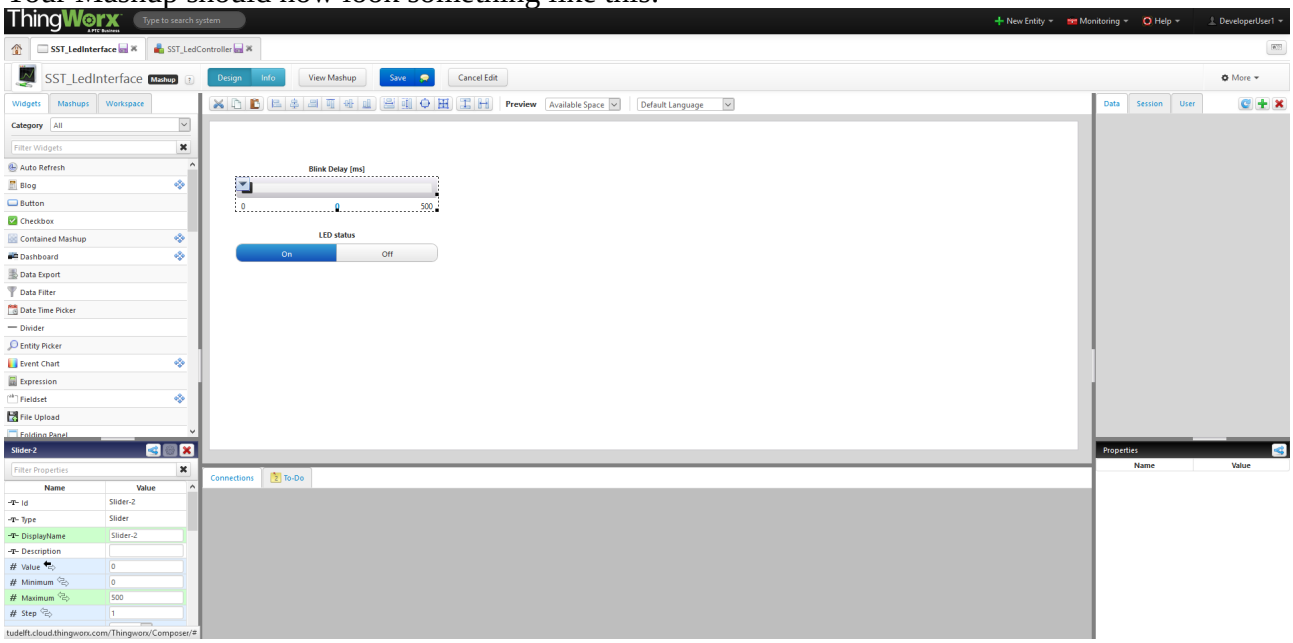
Name:	SST_LedInterface
Description:	A Mashup to display and edit the settings of two controllable LEDs connected to an Arduino.
Tags:	SST_exercise2

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.

Like was done in exercise 1, first add a “Panel” to the Mashup by typing “panel” in the Widget Selection menu and dragging it onto the page. This allows us to put multiple Widgets in the Mashup.
Next, add a “slider” and set its maximum value to 1000 by changing the “Maximum” value in the Widget Options menu, and its minimum value to 5. Sliders do not have any descriptive text by default, so to make the interface more clear we should add a “label”. You can do this the same way as the other Widgets. After adding it, you can edit the text by selecting it (click on it so it has a dashed border) and adding some text in the “Text” field in the Widget Options menu. You can for instance call it “Blink Delay [ms]”. Do not put the label over the slider Widget; if you do this you cannot click the slider any more since the label is on top of it and will be clicked instead.

For the LED whose state (on/off) is controlled by the Mashup, add a “Radio Button”. You will notice that unlike the other Widgets, when you've dropped it in the Mashup it will not show. This is because for Radio Buttons we will need to select what Button States are available to it in the Widget Options menu. To do this, click the wand icon in the Widget Options menu and find “On – Off Radio Button”. It will now show up properly in the Preview window, and its buttons will be labelled “On” and “Off”. Add a label to the Radio Button as well to make your Mashup more understandable.

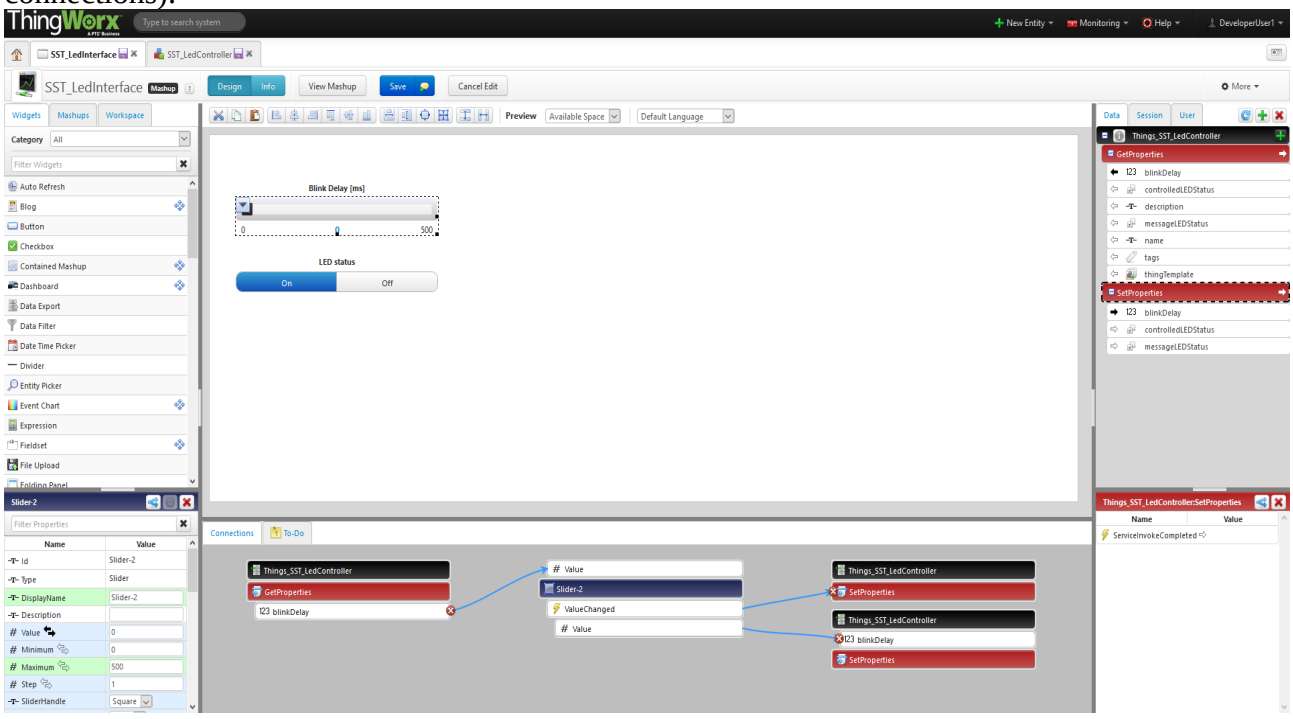
Your Mashup should now look something like this:



Adding Data
To link the slider and the buttons to our Properties, we will need to add a Data entity, like was done in exercise 1. In the “Data” tab on the top right, click the green “+” icon and search for “LedController”. After you've clicked that, we will need to add the same service that was used in exercise 1, GetProperties. But, since we also want to be able to control and thus change the values or our Properties, we will also need its counterpart: SetProperties. Search for both of them and click the blue arrows next to the names. Finally, only click the tick box next to “GetProperties” under “Mashup Loaded?”. We do not tick the one next to SetProperties to prevent it from setting all values to their defaults when the Mashup is loaded. Then click “Done”.

Data Bindings
Just like the gauge in exercise 1, we want the slider to display the current value of a Property. To do this, expand the “GetProperties” menu on the top right by clicking the “+” icon on it. Then drag the “blinkDelay” property to the slider and bind it to the Value of the slider.
If we were to open the Mashup now, we would be able to see the current value of blinkDelay on the slider, but despite being able to change the slider's position we would not be able to change it. To fix this, expand the “SetProperties” menu, mouseover the top left corner of the slider and drag the “Value” option over to the “blinkDelay” under SetProperties.
This will link the values to one another, but will not tell ThingWorx to update them just yet. To make sure they are updated whenever we change the value, mousover the top left corner of the slider Widget again, and drag “ValueChanged” over to the red “SetProperties” bar and drop it there. This will cause the SetProperties Service to be run whenever we change the value of our slider.

If you now click on the slider Widget, your Mashup should now look like this (note the data connections):



As you can see, the value received from GetProperties is now send to the slider, and the value of the slider is, whenever it is changed, sent to the SetProperties function.

Side-note
You may notice that unlike in exercise 1, we do not need an auto-refresh Widget. This is because the value of blinkDelay is not changing by any means other than our Mashup, and we do not need to load it every so often with GetProperties since every time it is changed this is already visible.

We can now test our slider by clicking “Save”, then “View Mashup”. If we then switch back to our ThingWorx window and open our Thing's Property list, we can see what happens when we move the slider. First, click the blue “refresh” button next to “Value” on the Properties page, then change the value of the slider. When you switch back to the Properties page and click the refresh button again, you will see that the value of the Property has changed accordingly.

Now repeat the data binding steps you used for the radio button, but bind “controlledLEDStatus” to “SelectedValue” and vice versa. SelectedValue is the button that is selected, since the “value” in this case is either “true” or “false”, which corresponds to the two states of the radio button.

Now that we have a Mashup that can change the value of our Properties according to what a user does in its interface, we can have our Arduino react to this.
Before we do that, let's summarise our ThingWorx settings:

Thing	SST_LedController
Property	messageLEDStatus
Property	blinkDelay
Property	controlledLEDStatus
Service	SST_getData
Mashup	SST_LedInterface

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.

2.4 The Arduino code

Since the Arduino code is largely the same as it was in exercise 1, it will not be discussed here in its entirety. The parts that differ will be discussed; the others are considered to be known to you.

Part 1: including libraries and defining constants

Apart from the libraries included in exercise 1, the file “string.h” is also included. This is because it contains some functions that will be useful to us later in the code. Remember how in exercise 1 we used character arrays for the reply we got from ThingWorx? In this exercise that is also the case, but unlike the precious one, we will need to filter out some parts of it that are useful. To do this we will need some functions that are included in the 'string' library.

There are also some differences in the defined variables: our Thing name has changed, as has our Service. Notice that the properties we use are not defined in the code: this is because, unlike in exercise 1, we do not need to send those names to ThingWorx.

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


#define mySSID "TODO:YOUR_SSID_HERE"
#define myPSK  "TODO:YOUR_PASSWORD_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_LedController"

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

// Interval of time at which you want the properties values to be gotten from
TWX server [ms]
#define timeBetweenMessages 2500

// The maximum length we expect for a replied JSON string.
#define REPLY_MAX_LENGTH 150

// TODO: change this to the pins you are using.
#define messagePin 4
#define blinkPin 5
#define controlledPin 6
```

Part 2: Variables and setup()

In the part below you will see some familiar and some unfamiliar variables. There is two variables that are used to keep track of when things happened last, so we can avoid using `delay()` to execute certain tasks at regular intervals since `delay()` stops all execution of code; when we use that function the Arduino will literally be doing nothing! A more detailed explanation of this will be in the next section, where we use it.

Next to Booleans that control whether we see the HTTP request and response, there is now also one that controls whether we see the JSON string that was found in the response. We also use Boolean values to remember whether our LEDs should be on or off.

```
String httpRequest;
unsigned long send_time;        // this holds the timestamp of our last request
bool displayRequest = false;  // this controls whether we print the request we
send
bool displayResponse = false; // this controls whether we see the HTTP response
bool displayJSON = true;        // this controls whether we print the JSON that
was found

// The variables related to the LEDs
int blinkDelay;
unsigned long blink_on_time;  // this holds the timestamp of when the blinking
LED was last turned on
bool messageLEDStatus, controlledLEDStatus, blinkStatus;

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(messagePin, OUTPUT);
  pinMode(blinkPin, OUTPUT);
  pinMode(controlledPin, OUTPUT);
}
```

Part 3: Constructing our POST message and processing the results

The `loop()` is where we see a major difference between exercise 1 and this one. If we save the last time we sent an HTTP request to the ThingWorx server in the `send_time` variable, we can use it to check if enough time has passed since we last did that. This, in contrast to the `delay()` function, allows us to run some other part of the loop while not enough time has passed.

We can easily calculate whether it's time to send a new request with an if statement, as can be seen from the code snippet below:

```
if( (currentTime – lastSendTime) > desiredDelay){
    // execute some code here
    lastSendTime = currentTime;
}
```

This only works if this is executed every so often of course, which is the case when we use it in the Arduino's `loop()` function. Since we can use `millis()` to get the current time from the Arduino, the `if()` statement that is seen below works like we want it to:

```
if((millis() - send_time) > timeBetweenMessages){
    // execute some code here
    send_time = millis();
}
```

Which is what we can see in the first part of the `loop()` of our Arduino code below.

```
void loop()
{
      // we use this if statement to not overflow the server with requests
      if((millis() - send_time) > timeBetweenMessages){
            // To get data from the server, we use a GET request.
            httpRequest = "GET /Thingworx/Things/";
            httpRequest+= thingName;
            httpRequest+= "/Services/";
            httpRequest+= serviceName;
            httpRequest+= "/result";
            httpRequest+= "?appKey=";
            httpRequest+= appKey;
            httpRequest+= "&method=post&x-thingworx-session=true";
            httpRequest+= " HTTP/1.1\r\n";
            httpRequest+= "Host: ";
            httpRequest+= destServer;
            httpRequest+= "\r\n";
            httpRequest+= "Content-Type: text/html\r\n\r\n";
```

Since this code might create a lot more Serial output than the code in the previous exercise, some extra empty lines are used for readability. This has been done by adding the newline character `\n` to the `Serial.println()` commands below:

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

As you can see, the `displayRequest` variable is used to control whether we see the request in the Serial Monitor.

Now that we have our request taken care of, we can send it, save the response to a character array, and save the time at which we did this:

```
            // We are using character arrays to use less memory than Strings
            char *response = sendRequest();
            send_time = millis();
```

At this point in the `loop()` we have the response we need from the server, and can try to use it. One thing to note here is that Internet communication is not always flawless; something might have gone wrong in our device, the connection towards the server, on the server, or on the way back. This means that sometimes we get an empty response from the server or one that is not of use to us; luckily we can check if this is the case.
The file `string.h` we included earlier contains a function `strlen()`. This can tell us what the length of the character array we give it is. For instance, an array with the characters “a”, “b” and “c” will have a length of 3. This means that if our response is less than 1 character long, we will know beforehand that processing it further is useless.
If you need to make sure that data is received often enough, it is also wise to make sure the program sends and/or receives data more often than strictly needed. For instance, we have measured a success rate of about 75% with this setup, so in order to get an update about every 5 seconds we get our data from the server every 2.5 seconds.
Once we have verified that the response we received is probably valid, we can use the function `saveToVars()` to save the contents of the JSON string to our variables. How this is done will be discussed in part 6.
Now that we have our variables updated with data from the server, we can use them. For your convenience we have created a function `ledBlink()` to handle the timing of our blinking light. The other two LEDs are controlled directly from the Booleans we received from ThingWorx.

```
            // process the data:
            if(strlen(response) > 1){ // Sometimes data doesn't come through
                                      // properly. This checks whether we got
                                      // any
                  if(displayJSON){
                        Serial.print(F("  Received the following JSON: \n  "));
                        Serial.println(response);
                  }
                  saveToVars(response);
            }else{
                  Serial.println(F("  No data received."));
            }
      }

      // You will notice how the blinking stops at certain intervals. This is
the duration of the HTTP request to the server and getting data back from it.
      ledBlink(blinkDelay);
      digitalWrite(controlledPin, controlledLEDStatus);
      digitalWrite(messagePin, messageLEDStatus);
}
```

With all of these parts together, we now have our `loop()` done.

Part 4: Blinking the LED

To make the loop() more readable, a separate function for the blinking of the LED is used. Using functions rather than writing code in the loop() directly makes it easier to see what the sequence in which things are executed is, and makes it easier to read a program when you've not seen it for a while.

The timing principle we used to send HTTP requests is also used here, so the code below should be understandable to you.

```
// Blinks an LED at certain intervals
// This method avoids using delay() since that stops all other execution
// Written by Marien Wolthuis
void ledBlink(int interval){
      // (de)activate red LED based on how long it's been on
      if(!blinkStatus){
            // if it's been off long enough, turn it on
            if((millis() - blink_on_time) > (interval * 2)){
                  blink_on_time = millis();
                  blinkStatus = HIGH;
            }
      }else{
            // if it's been on long enough, turn it off
            if((millis()-blink_on_time) > interval){
                  blinkStatus = LOW;
            }
      }
      digitalWrite(blinkPin, blinkStatus);
}
```

Part 5: Sending the request
What you will notice is that, in contrast to the previous exercise, the `sendRequest()` function returns a data type you might have not seen before: `char*` . This is what is know as a *character array.* A `char` variable is a numeric representation of a character, and an array is a construct that can hold multiple variables, and allows access to them by their index number.

An example of a `String` array can be as follows:

```
String myArray[3] = {"first", "second", "third"};
Serial.println(myArray[0]);
Serial.println(myArray[1]);
Serial.println(myArray[2]);
```

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

```
first
second
third
```

Note that the “index number” with which we can access the parts of the array starts at *0*, not at *1*!

An example of a `char*` array can be as follows:

```
char myCharacterArray[27] = "This is an example sentence";
Serial.println(myCharacterArray);
Serial.println(myCharacterArray[0]);

Serial.print(myCharacterArray[5]);
Serial.println(myCharacterArray[6]);
```

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

```
This is an example sentence
T
is
```

We use this data type here because, when compared to a `String`, the character array uses a lot less memory. This is because the `String` class has a lot of built-in functions that can operate on the `String` and are loaded into memory when we use a `String` variable, while the character array is a lot simpler and does not have those features.

Also included in this more advanced version of `sendRequest()` is a *switch statement*. This functions as a sort of '`if()`' statement, but with better readability when a lot of options are to be examined. It was added to help you understand what might have gone wrong if there was an error code.

An example of a switch statement can be as follows:

```
int someInteger = 5;

// Do something with the integer so it's value can change in between

switch(someInteger){
  case 1:
    Serial.println("The value was 1");
    break;
  case 5:
    Serial.println("The value was 5");
    break;
  default:
    Serial.println("This is the output if none were true");
}
```

After each `case` we need a to give the `break` command, otherwise all cases below the (first) one that was `true` will be executed. This can be helpful in some cases, but often is not. The `default` case is optional, but can be very useful if there might be some case that wasn't covered in the switch statement.
In the code below we can only receive values of `-1`, `-2` or `-3`, so we do not need the `default`.

```
// Sends a HTTP request based on the contents of a variable called httpRequest.
// Originally written by SparkFun, adapted by Adrie Kooijman and Marien
// Wolthuis
char* 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, timeBetweenMessages);
  if (retVal <= 0)
  {
    Serial.print(F("  Failed to connect to server."));
    Serial.print(F(" retVal: "));
    Serial.println(retVal);
    switch(retVal){
      case -1:
        Serial.println(F("  The connection timed out."));
        break;
      case -2:
        Serial.println(F("  An unknown error occurred."));
        break;
      case -3:
        Serial.println(F("  The connection failed."));
        break;
    }
    return "";
  }
```

The next part should be familiar:

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

  // Clear the httpRequest to free up some memory
  httpRequest = "";
```

Side-note:
Switch statements can be very useful especially if there is a lot of options to deal with. They do have one problem: In Arduino they can only deal with numbers or characters! A switch statement with a `String` as `case` for instance will not execute. If you decide to use one, remember that `"a"` *is a* `String`*, and* `'a'` *is a* `char`*. So,* `[case "a": ]` *will not work, but* `[case 'a': ]` *will.*

After we've sent the request, it is now time to handle the response. As you've seen in the previous exercise, we can retrieve it character-by-character from the client, which is great because it involves little memory use and can be done quite quickly. Seeing as the response will be a Web page including all sorts of HTML, the best way to find our JSON string in it is to search for its start character: "{". Anything after this is useful to us, so we save it, until we encounter the closing brace "}". Anything after that we don't need.

```
  // available() will return the number of characters currently in the receive
  // buffer.
  // The code below filters out all HTML and only saves what might be a JSON
  // string
  // This cannot deal with nested JSON arrays!
  char reply[REPLY_MAX_LENGTH] = "";
  int i = 0;
  while (client.available()){
    char rep = client.read(); // read() gets the next character
    if(displayResponse){
      Serial.print(rep);        // This outputs the entire response to Serial
    }
    if(rep == '{'){                 // JSON strings always start with this
      while (client.available() && rep != '}'){
        reply[i] = rep;
        rep = client.read();
        i++;
      }
      reply[i] = rep; // to get the closing bracket too
    }
  }

  // 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.
  }

  return reply;
}
```

We use the index `i` here because of the way we need to write to a character array: we can only write to a specific slot that is available in it. To do this, we need something along the lines of `array[0]` for the first slot, `array[1]` for the second, etcetera. It then is more sensible to put this in a loop (a while loop in this case) and increment the value of `i` every time we do so.

Part 6: Saving the JSON's values
Next, we will need a function to save the parts we need from the JSON to their corresponding variables. To do this, we will use two functions that are available in the C language upon which the Arduino language is built: `strtok()` and `atoi()`.

In short, this is how `strtok()` ("split string by tokens") works:

```
strtok(char* array, char* deliminators)
INPUT: char* array       : the character string that is to be split into pieces
       char* deliminators: the characters that define the start and end of
                           these pieces

OUTPUT: char* firstPart: the first part that has been split off the string

To receive the rest of the parts, call the function like so:
strtok(NULL, char* deliminators)
```

A practical example would be:

```
char input[27] = "This is/an;example sentence";
char* output;
char deliminators[3] = " /;";

output = strtok(input, deliminators);
Serial.println(output);
output = strtok(NULL, deliminators);
Serial.println(output);
output = strtok(NULL, deliminators);
Serial.println(output);
output = strtok(NULL, deliminators);
Serial.println(output);
output = strtok(NULL, deliminators);
Serial.println(output);
```

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

```
This
is
an
example
sentence
```

The other function, `atoi()` ("array to integer") works a lot simpler; it converts a character array to an integer. This is needed because of how characters are saved in memory; the explanation is rather lengthy and too complicated for this course. Suffice to say, without using `atoi()` you might get unexpected results when saving a character array like "`520`" to an integer directly!

These two functions lie at the heart of the function we need to save the data we got from ThingWorx to variables. Please read the function carefully, as you may need to edit it for another project to save different variables instead! You can find it on the next page.

Side-note:
As you may notice while reading the code of this function, it will only work for this specific purpose and needs the data to always be in the exact same order. This is not generally a good idea when writing software; if there is some unexpected change in the input the output becomes unpredictable! Nevertheless, in this case it is necessary because of the memory constraints we face. By making sure that the data is always in the same order (hence the use of a String in ThingWorx) we try to prevent unexpected things from happening.

```
// This function saves the variables that are contained in a JSON-like
// character array to separate variables.
// Please note that this can not deal with variable names or values that
// contain a space " " character due to this being a custom implementation of a
// JSON interpreter that is purely aimed at conserving as much memory as
// possible and to serve only purposes like the one in the exercise at hand.
// Written by Marien Wolthuis
void saveToVars(char *input){
      // Deliminators are characters that are used to separate parts of a
      // phrase or expression
      #define DELIMINATORS "{ ':,}"
      char * pch;

      // We expect this to be our first variable name: messageLEDStatus
      pch = strtok(input,DELIMINATORS);
      // Then this is the value associated with it:
      pch = strtok(NULL, DELIMINATORS);

      if(pch[0] == 't'){   // Checks whether the first character in the array
                           // is a 't' for 'true'
            messageLEDStatus = true;
      }else if(pch[0] == 'f'){
            messageLEDStatus = false;
      }

      // blinkDelay name:
      pch = strtok(NULL, DELIMINATORS);
      // blinkDelay value:
      pch = strtok(NULL, DELIMINATORS);
      // strtok() gives us a character array instead of a numeric value, atoi()
      // converts this back to a proper integer.
      blinkDelay = atoi(pch);

      // controlledLEDStatus name:
      pch = strtok(NULL, DELIMINATORS);
      // controlledLEDStatus value:
      pch = strtok(NULL, DELIMINATORS);

      if(pch[0] == 't'){
            controlledLEDStatus = true;
      }else if(pch[0] == 'f'){
            controlledLEDStatus = false;
      }
}
```

So, after executing this function we will have saved the values for messageLEDStatus, blinkDelay and controlledLEDStatus to their respective variables.

Part 7: Helper functions

The functions below are 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 (;;)
    ;
}
```

2.5 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