Polytechnic University of Timişoara

Documentation

Topic: Embedded application on Arduino with web service
Course: Syncretic project 1A

Student: TONU Alexandru
Mentor: BOGDAN Razvan

Timişoara 2020

Table of contents

Introduction **2**

Analysis of the current state in the field of the problem **3**

Solution architecture (design / development methodology) **3**

Diagrams: 6

Setup diagram 6

Logs diagram 7

UML diagram 7

Implementation **8**

Application testing **18**

Test scenarios 18

API protection: 18

Account protection: 20

Static analysis: 21

Unit tests: 22

Metrics used in application testing 22

Conclusions **23**

Bibliography **24**

Introduction

As a theme for this project, I've chosen to build a service that would help automate the care of different crops in greenhouse conditions. The solution must be convenient to use, following the ideology of plug and play. Furthermore it must be possible to dynamically change the parameters related to the grown culture without interfering with the code or hardware. Also it should provide the facility to monitor all real-time environmental parameters.

To simplify use, a centralized data storage system should be provided. Its purpose will be to provide users with the service of storing logs and greenhouse configuration, so that they do not have the need to own their own local storage. Storing

data in clouds will also bring the benefit of accessing that information from anywhere in the world.

In order to achieve those goals, an account registration, login and administration service will also be required.

Analysis of the current state in the field of the problem

Of course, I'm not the first to try to create an autonomous greenhouse. Moreover, functional solutions have already been created. So what is so special about my attempt and why should the potential client take it into account?

First of all - consistency. This implementation does not end with automated greenhouse control. It proposes the dynamic adaptation of all parameters used by the hardware, real-time monitoring, data storage in the cloud and access to the account from anywhere and any device with internet connection.

The second strength is simplicity in use. Once connected, the hardware runs freely without the need for maintenance, even without internet access.

And the last but not the least - the financial aspect, of course. The necessary equipment has a very low cost and its installation does not require engineering knowledge. So the customer will be able to receive a kit with great functionality and flexibility at an attractive price, which will make the product competitive on the market.

Solution architecture (design / development methodology)

I’ve chosen to use Arduino Uno for hardware because of its simplicity in programming and use together with a ridiculously low price. Another mandatory criteria was ability to plug-and play - a feature that has made Arduino so popular in comparison with solutions like RaspberryPi.

Here are Arduino Uno specs:

- Microcontroller: ATmega328
- Operating Voltage: 5V
- Input Voltage (recommended): 7-12V
- Input Voltage (limits): 6-20V
- Digital I/O Pins: 14 (of which 6 provide PWM output)
- Analog Input Pins: 6
- DC Current per I/O Pin: 40 mA
- DC Current for 3.3V Pin: 50 mA
- Flash Memory: 32 KB of which 0.5 KB used by bootloader
- SRAM: 2 KB (ATmega328)

- EEPROM: 1 KB (ATmega328)
- Clock Speed: 16 MHz

For the purpose of this project it is way too much more than satisfying.
Hardware should do the following:

Collect data about temperature, brightness and humidity.
Temperature is regulated using a thermometer, fan and a heater which I don't have, thus I have not included one in the scheme, but we'll talk about it later.
Humidity is regulated using a pump, which I don't have either, neither I have a humidity sensor, we'll talk about this later too.
Brightness is regulated using LEDs and a photoresistor sensor.

Software should do the following:

To show off, I used three methods to regulate things:

- For humidity (which is the simplest one) a if-else statement: when humidity is below a given point - turn the pump on.
- For temperature a regulator with hysteresis to avoid continuously turning fan on and off (like refrigerator works - cooling a few degrees under the setted point, then rest a little)
- For brightness (and here is the hardest one) a PID controller.

Also Arduino should continuously communicate using serial port to log data and receive configuration values.

As a bridge between Arduino that runs on C/C++ and WEB server that runs on Linux, which is written in C, I've chosen Python. The reason why I've made this choice is the following: Python libraries are written also in C, it means that serial communication using USB technology is executed natively which increases speed, but most importantly - ensures stable working.

In addition Python has simple syntax and huge community and documentation, hence entry threshold is quite low and learning time is short as well.

The purpose of Python code is to create a configuration message using data received from the server and send it to Arduino via USB, after that it has to collect log messages and send them back to the server.

Since it is a simple app, I decided not to use a framework for WEB like Angular or React, but go instead with Vanilla JavaScript, which by the way, has become incredibly powerful nowadays.

JavaScript has to rule the client side. To show logs and preferences in real-time, to manage them and a lot of stuff with server requests.

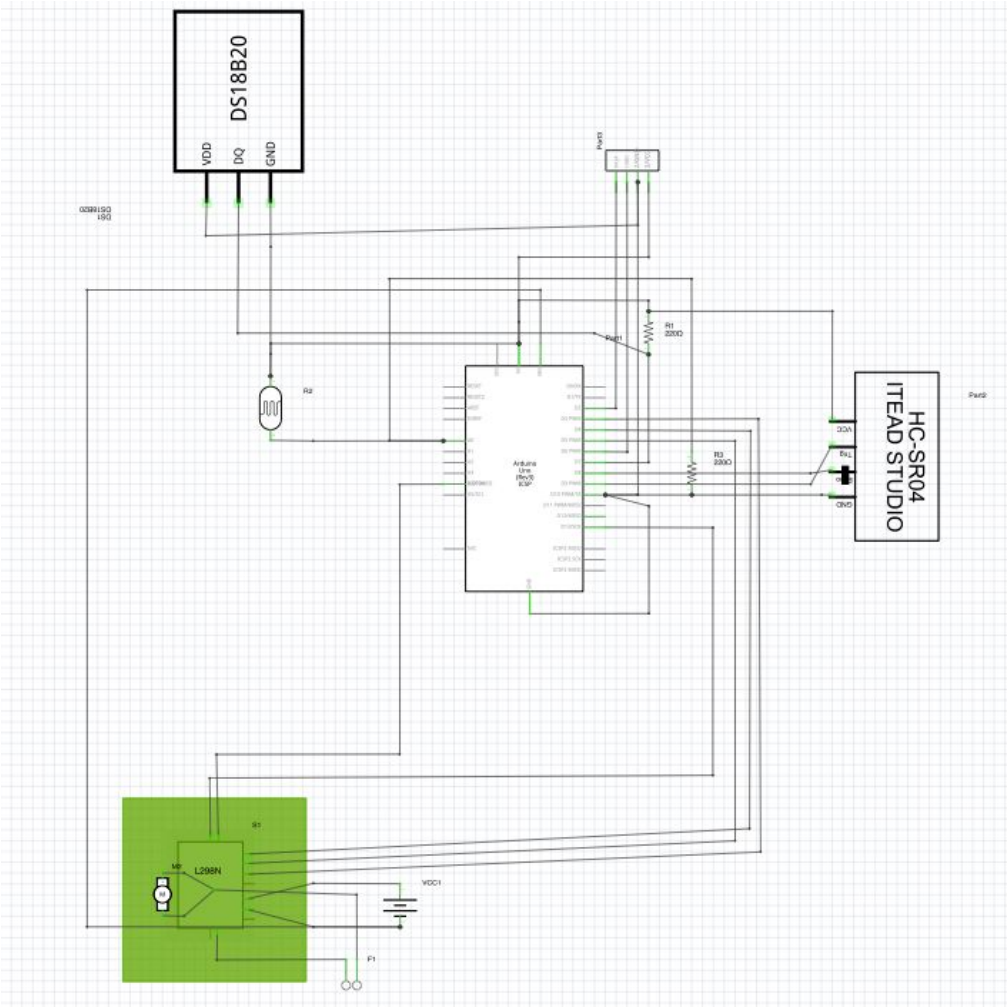
But I didn't use JS for the server side, and I regret it, and I knew that I would regret it from the very beginning, yet I did it for a simple reason: I had already setted everything up on my server to work with PHP from the box, so God forgive me for I have sinned.

Backend has a lot of work to do with database management for which MariaDB is used. PHP will serve requests like to insert a new log (very often), delete one, get them all, login, logout, register, change preferences.

To sum up solution architecture:

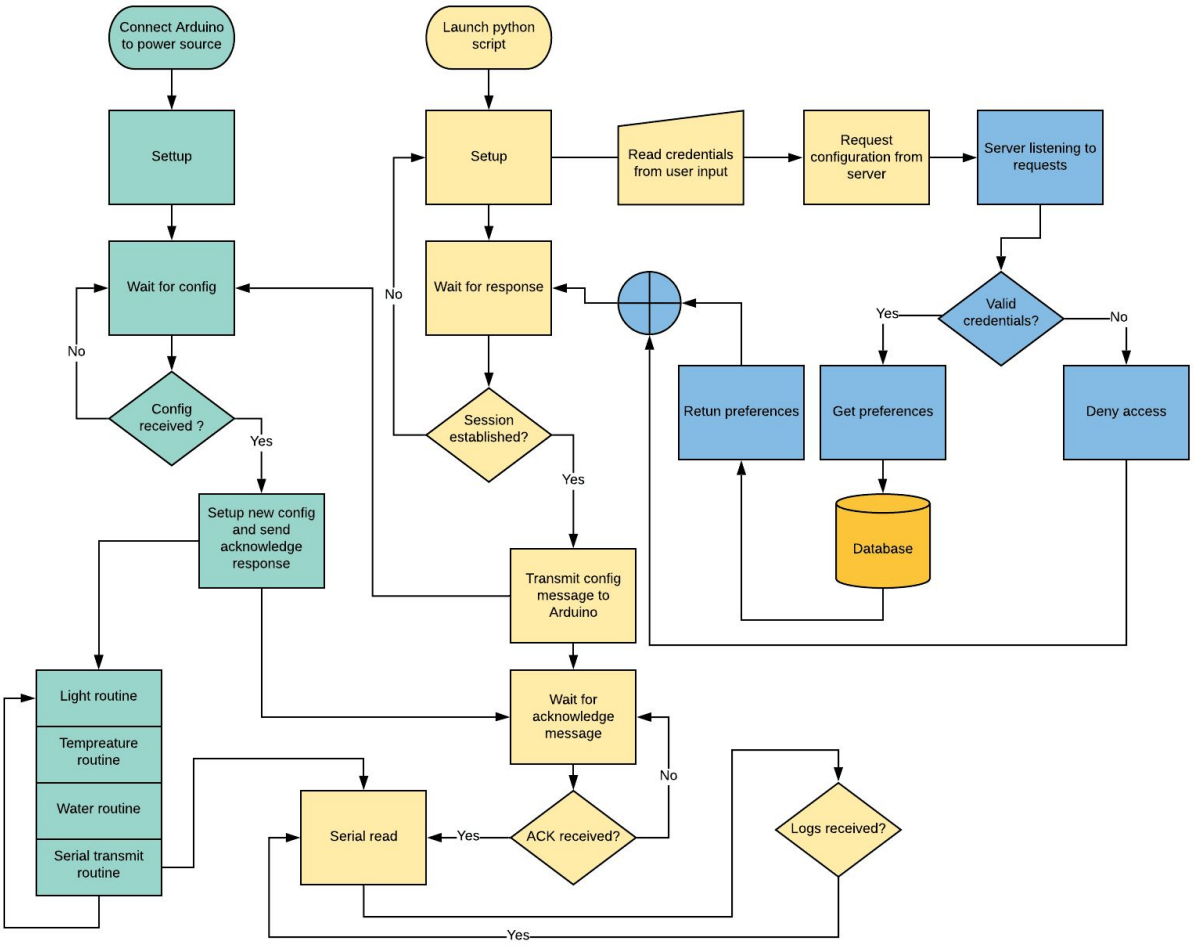
- Hardware: Arduino Uno on ATmega328
- Embedded systems: C/C++ on Arduino, Python on PC
- Frontend: JavaScript
- Backend: PHP
- Database: MariaDB

Schematic design:

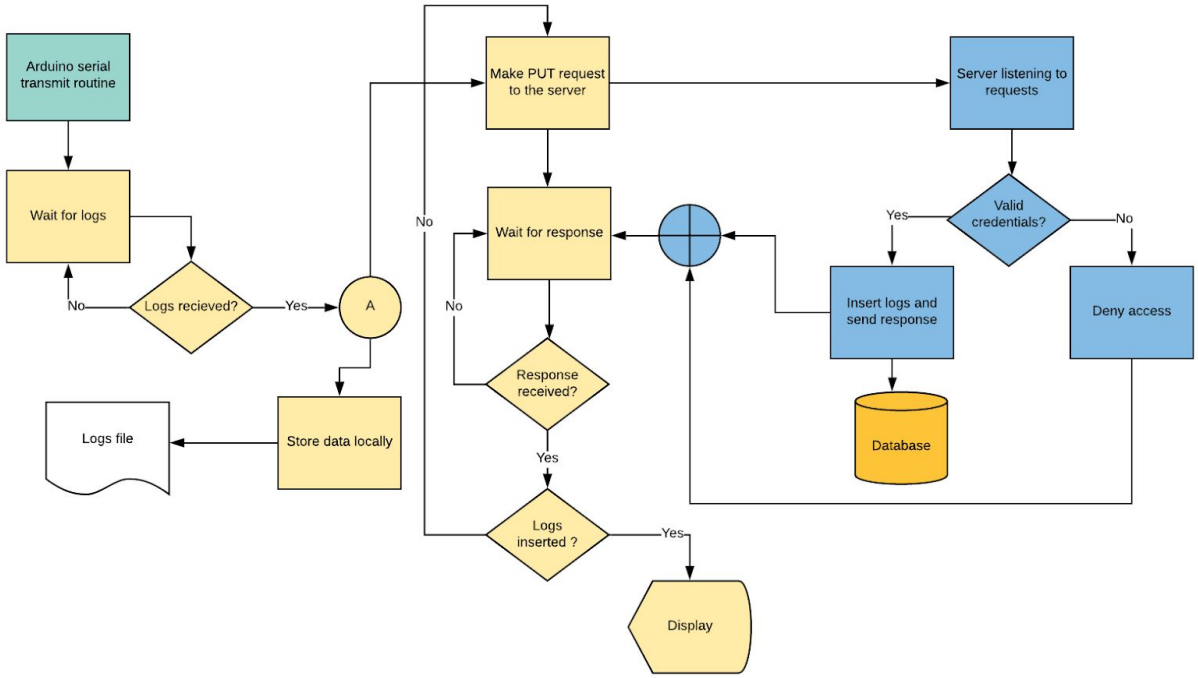


Diagrams:

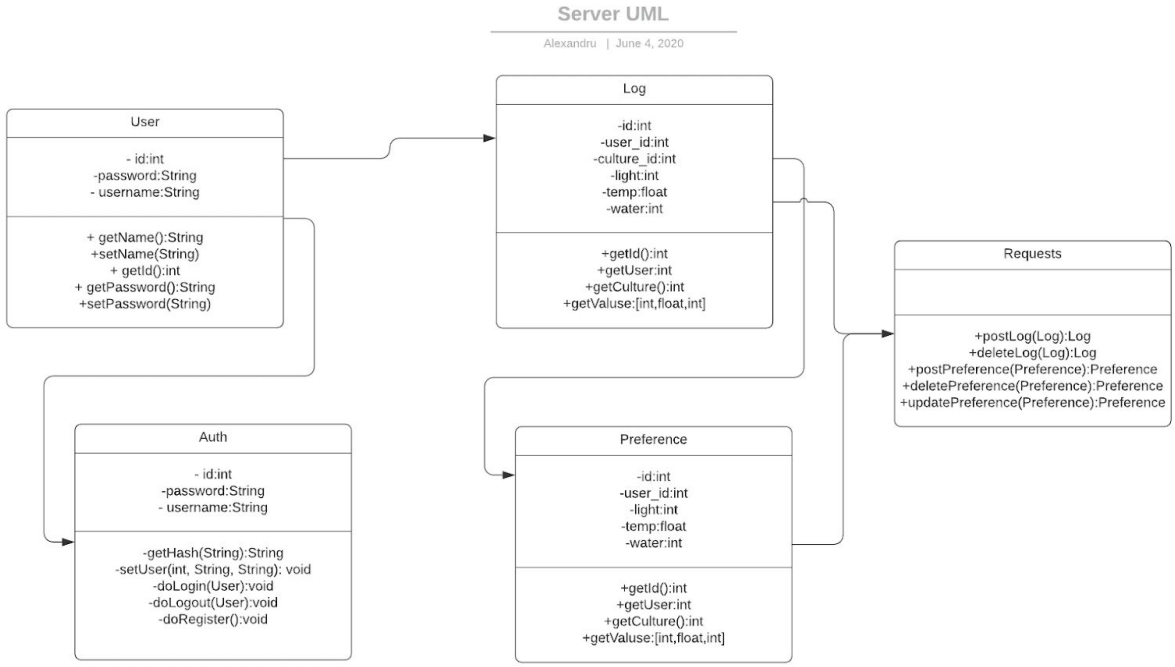
Setup diagram



Logs diagram

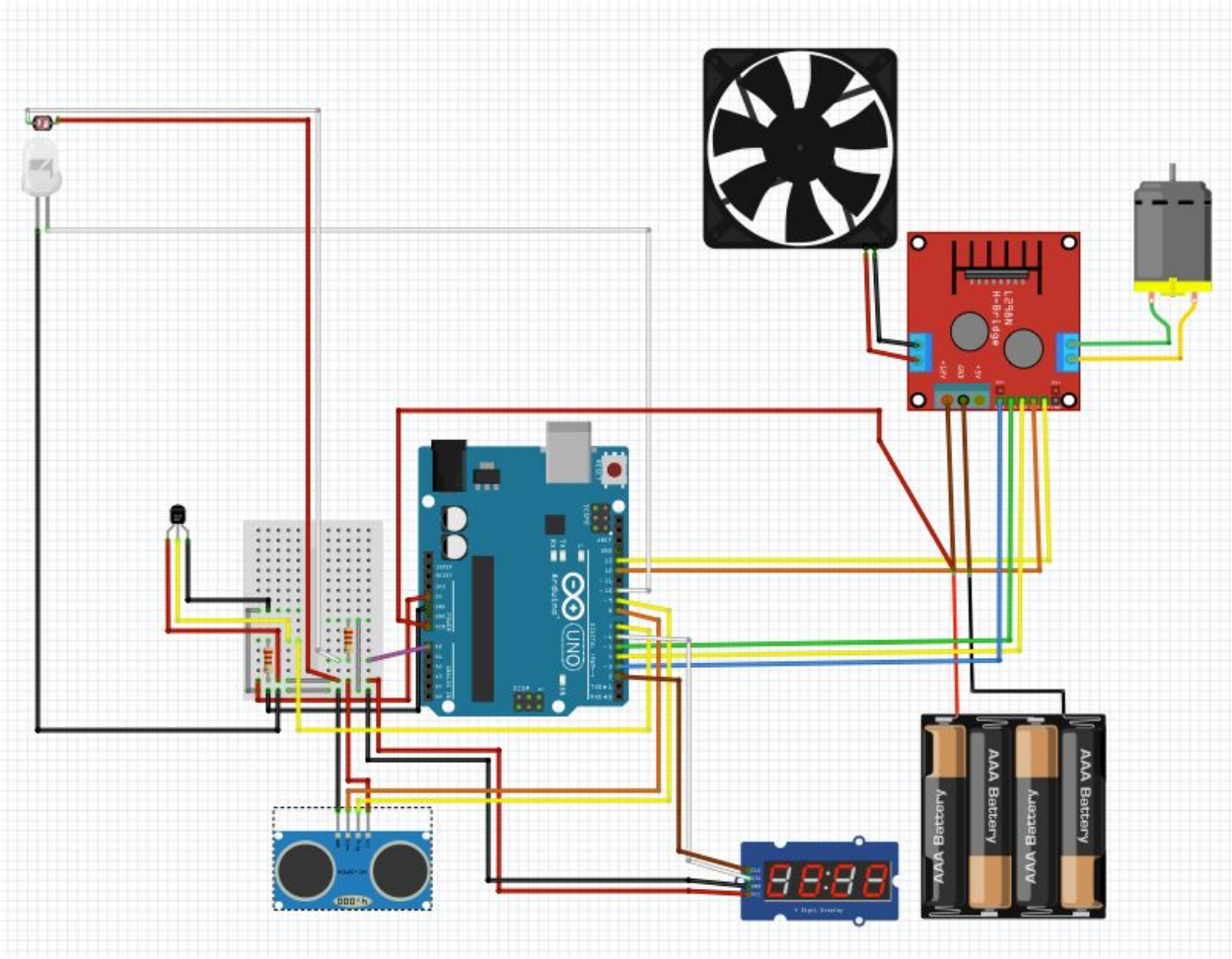


UML diagram



Implementation

Arduino scheme:



It is designed in Fritzing tool, and can also be seen on Fritzing site

When you connect the Arduino to the power source it boots and executes the following code:

```
// required libraries
#include <OneWire.h>
#include <DallasTemperature.h>
#include <Arduino.h>
#include "TM1637Display.h"
#include <PID_v1.h>
```

```
// define pinout macros
#define ONE_WIRE_BUS 7 // temperature sensor data
#define CLK 2 // 4 rang 7 segment display clock
#define DIO 6 // 4 rang 7 segment display data I/O
#define pResistor 11 // photoresistor digital input for controller using mapping
#define pResistorAnalog A0 // photoresistor analog input for PID controller
#define lightOut 10 // LED
#define trigPin 9 // ultrasonic sensor trigger
#define echoPin 8 // ultrasonic sensor echo
#define enA 3 // enable fan DC motor
#define in1 5 // L298N driver in1
#define in2 4 // L298N driver in2
#define in3 12 // L298N driver in3
#define in4 13 // L298N driver in4
```

Here required libraries are imported and pinout macros are setted.

Declaring globals:

```
double lightSet;   // will be the desired value for brightness
double lightValue; // photoresistor value
double lightPWM;   // PWM output to LED

int waterSet,   // will be the desired value for humidity
   waterValue; // humidity value (not really)

float temperatureSet, // will be the desired value for temperature
   temperatureValue; // temperature sensor input

char data;                 // message received from the serial
double buff[3] = {0};      // splitted message
int loaded = 0, index = 0; // serial receive state flags
```

Next step is to declare utilities:

```
// setup PID parameters
```

```
double Kp = 0, Ki = 3, Kd = 0;
PID lightPID(&lightValue, &lightPWM, &lightSet, Kp, Ki, Kd, DIRECT);

// temperature sensor utilities
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);

// 4 rang 7 segment utility
TM1637Display display(CLK, DIO);

// degree Celsius symbol
const uint8_t degreeCelsius[] = {
   SEG_G,                          // -
   SEG_G,                          // -
   SEG_A | SEG_B | SEG_F | SEG_G, // o
   SEG_A | SEG_D | SEG_E | SEG_F  // C
};
```

Since communication between Arduino and Python is accomplished using USB, we'll need a helper function to extract data in a convenient way:

```

// store received characters from the serial in a buffer array
void recieveSerial()
{
 data = Serial.read(); // read 1 byte
 switch (data) // check what kind of byte has arrived
 {
 case 'f': // emitor finished serial transmit
   loaded = 1; // set loaded flag true
   break;
 case 'e': // [e]nd of a value
   index = (index + 1) % 3; // select index from buffer array
   break;
 default: // a integer byte arrived, push it at the end
   buff[i] *= 10;
   buff[i] += (data - 48);
   break;
 }
```

```
}
```

Now, each time something is written in the read descriptor, update configuration:

```
// update configuration
void updateConfig()
{
 while (!loaded) // while serial receiving is not finished
   if (Serial.available() > 0)
     recieveSerial(); // call helper

 // update configuration variables
 lightSet = buff[0];
 temperatureSet = buff[1];
 waterSet = buff[2];

 // Acknowledge emitor that data was successfully received
 Serial.print("Ready\n");
}
```

Finally, we are ready to start doing stuff, so setup function is called:

```
void setup()
{
 // set pinout
 pinMode(pResistor, INPUT);
 pinMode(lightOut, OUTPUT);
 pinMode(trigPin, OUTPUT);
 pinMode(echoPin, INPUT);
 pinMode(enA, OUTPUT);
 pinMode(in1, OUTPUT);
 pinMode(in2, OUTPUT);
 pinMode(in3, OUTPUT);
 pinMode(in4, OUTPUT);

 // setup DC Driver
 digitalWrite(in1, HIGH);
 digitalWrite(in2, LOW);
 digitalWrite(in3, LOW);
```

```
 digitalWrite(in4, LOW);

 // setup display
 display.setBrightness(0x20);
 display.clear();

 // open read descriptor at 115200 baud rate
 Serial.begin(115200);

 // call function to update configuration variables
 updateConfig();

 // start temperature sensor
 sensors.begin();

 // display on display degree symbol
 display.setSegments(degreeCelsius);

 //Turn the PID controller on
 lightPID.SetMode(AUTOMATIC);
  //Adjust PID values
 lightPID.SetTunings(Kp, Ki, Kd);
}
```

As I said, I don't have a humidity sensor, instead I used a ultrasonic proximity sensor to simulate it. Also the role of the pump will be played by a DC motor. So everything except the configuration part is applicable:

```
// humidity (not really) handler
void ultrasonicRoutine()
{
 // config
 long duration;
 pinMode(trigPin, OUTPUT);
 digitalWrite(trigPin, LOW);
 delayMicroseconds(2);
 digitalWrite(trigPin, HIGH);
 delayMicroseconds(10);
 digitalWrite(trigPin, LOW);
```

```
 pinMode(echoPin, INPUT);
 duration = pulseIn(echoPin, HIGH);
 waterValue = duration * 0.34 / 2;

  // turn off pump if humidity is higher than setted and on otherwise
 if (waterValue > waterSet)
   digitalWrite(in4, HIGH); // pump off
 else
   digitalWrite(in4, LOW); // pump on
}
```

Temperature is regulated by the following logic:

```
// temperature handler
int pwmOutput = 0; // by default cooler is turned off
void coolerRoutine()
{
 // get data from the L298N sensor
 sensors.requestTemperatures();
 temperatureValue = sensors.getTempCByIndex(0);

 // apply hysteresis logic
 if (temperatureValue > (temperatureSet + 3))
   pwmOutput = 255;
 if (temperatureValue <= temperatureSet)
   pwmOutput = 0;

 // Send PWM signal to L298N Enable pin and display temperature value
 analogWrite(enA, pwmOutput);
 display.showNumberDec((int)temperatureValue, false, 2, 0);
}
```

I said that a heater can be used, and here is how to do that without changing a line in code:

Since the heater is only working when the fan is not, you can connect the heater's Vcc to GND, but ground with the same pin as enA. Genius.

And last and most difficult is brightness controller:

I wrote two implementations: one using PID and one using mapping.

Map controller is a brilliant and elegant solution to keep brightness still under environmental influence. But it can not be set dynamically to a certain value which is mandatory for my purpose. Instead, LEDs will glow full power in complete darkness and stay dim in brightest light.

The right way to brighten greenhouse at a specified level is to use a PID controller. If you are not familiar with this technology, I strongly recommend you to watch a few tutorials or read an article.

```
// brightness handler
void lightRoutine()
{
 // Controller implemented using map function
 // lightValue = map(analogRead(pResistorAnalog), 0, 1023, lightSet, 0);
 // analogWrite(lightOut, lightValue);

 // map light value to discret range
 lightValue = map(analogRead(A0), 0, 1024, 0, 255);
 // PID calculation
 lightPID.Compute();
 // Write the lightPWM as calculated by the PID function
 analogWrite(10, lightPWM); // LED is set to digital pwm pin 10
}
```

Don’t forget to send logs to Python as well:

```
// serial transmission routine
void transmitSerial()
{
 // print a value followed by delimiter
 Serial.print((int)lightValue);
 Serial.print("_");
 Serial.print(temperatureValue);
 Serial.print("_");
 Serial.print(waterValue);
 Serial.print("\n");
}
```

And last thing to do on Arduino is calling each routine in the infinite-loop function:

```
void loop()
{
 // execute each routine
 coolerRoutine();
 ultrasonicRoutine();
 lightRoutine();

 // uncomment for Arduino IDE's serial monitor
 //  if(Serial.available() > 0) {
   transmitSerial();
 //  }
 updateConfig();
}
```

Level up: Python

Version of the interpreter that I am using is 3.7.3 and required libraries are:

```
import requests
import serial
import time
import threading
import json
from datetime import datetime
import tkinter as tk
```

Global setup:

```

# global config
API = "https://tonu.rocks/school/GreenHouse/api/"
preferences_endpoint = API + "preferences"
sensor = "DH11"
serial_port = '/dev/cu.wchusbserialfa130' # (1)
baud_rate = 115200
today = datetime.now()
today = today.strftime("%b %d, %Y")
log_file = "Logs/log_" + today + ".txt" # store logs locally as well
serial_connection = serial.Serial(serial_port, baud_rate, timeout=0.5)
time.sleep(1)  # give the connection a second to settle)
data = []
```

```
# user input variables
session_username = ''
session_password = ''

```

Please notice that you have to setup serial port at `# (1)` by yourself for your machine. And the API link will be different of course. Also you can notice that logs will be stored on the local machine too, you can remove this feature if you want.

Will set default values and configuration message

```
# default preferences
light = "200"
temperature = "25"
water = "40"
culture_id = "1"
config_msg = light + "e" + temperature + "e" + water + "ef"
```

To make things easier, a GUI will be draw to get username and password

```
# get values from inputs
def show_entry_fields():
   global session_username, session_password
   session_username = username.get()
   session_password = password.get()
```

Then a new session starts with credentials provided at the input

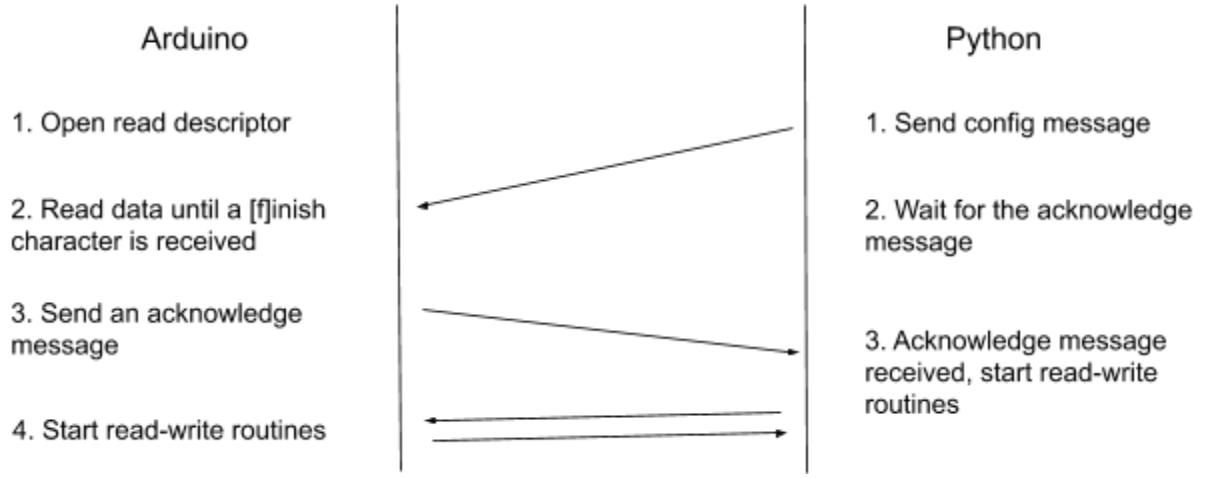
```
# start a new session
session = requests.Session()
login_data = {'username': session_username, 'password': session_password}
login_response = session.post(API, json.dumps(login_data))
```

When script receives an ok response from the server, it updates configuration message with the received preferences and starts communication with Arduino.

```
# update config from preferences
updatePreferences()

# write configuration to serial
serial_connection.write(config_msg.encode())
```

The communication looks as something like this:



So Python waits for the Acknowledge message

```
# wait until Arduino is ready to communicate
state_msg = serial_connection.readline().decode("utf-8").strip()
if(state_msg == "Ready"):
   print("Connection settled")
   readRoutine = setInterval(3, handleLogs)
```

When connection is established, read and write routines starts with the delay of 1 second between to avoid concurrent threads

```
# update config message
   time.sleep(1)
   writeRoutine = setInterval(9, updatePreferences)

   # listen on serial to data from Arduino
   while 1:
       serial_connection.write(config_msg.encode())
       line = serial_connection.readline()
```

In the read routine, Python receives a line ending with [\n] character.
As one may remember, Arduino delimited log values with the underscore character. So the read handler extracts light, temperature and water values by splitting the message.

```
      # split message on logs' delimiter
          log = logs.split("_")
          # create string from log object
          log_object = json.dumps({
              "cultureId": culture_id,
```

```
            "light": log[0],
            "temperature": log[1],
            "water": log[2]
        })
```

The newly created object is send to the server with a PUT request

```
        # make PUT request and store response
        response = session.put(url=API + "logs/",
                               data=log_object,
                               headers=headers)
```

The whole project can be found on the link at the bibliography section.

Application testing

Test scenarios

API protection:

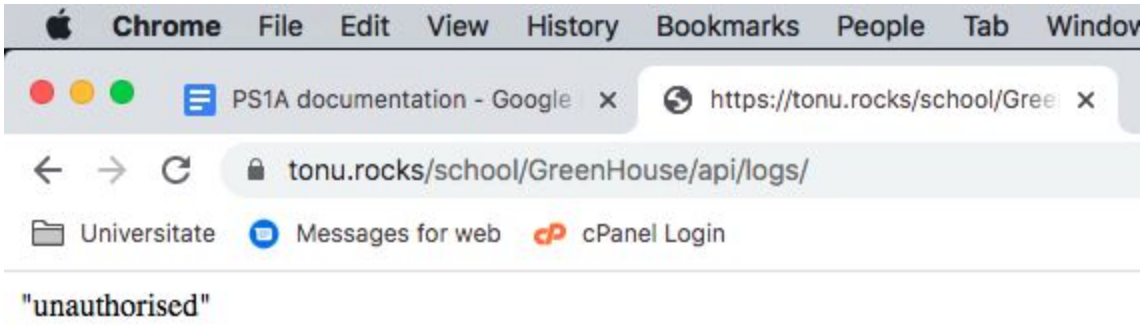
Unauthorised access to API must be forbidden, for this purpose a session is used to authenticate the user and application.

Scenario:

Test unauthorised request:

Expected: catch authorisation error and throw an exception

Result:



Testcase passed.

Scenario:

Try to login with invalid credentials

Expected: permission denied

Result:

Test1: nonexistent user

```
MacBookAir:PS1 alexandru$ python3 log.py
Username: someuser
Password: somepasword
"account with this username does not exist"
```

Test2: wrong password

```
MacBookAir:PS1 alexandru$ python3 log.py
Username: tonualexandru
Password: WRONGPASS
"incorrect password"
```

Test3: empty fields

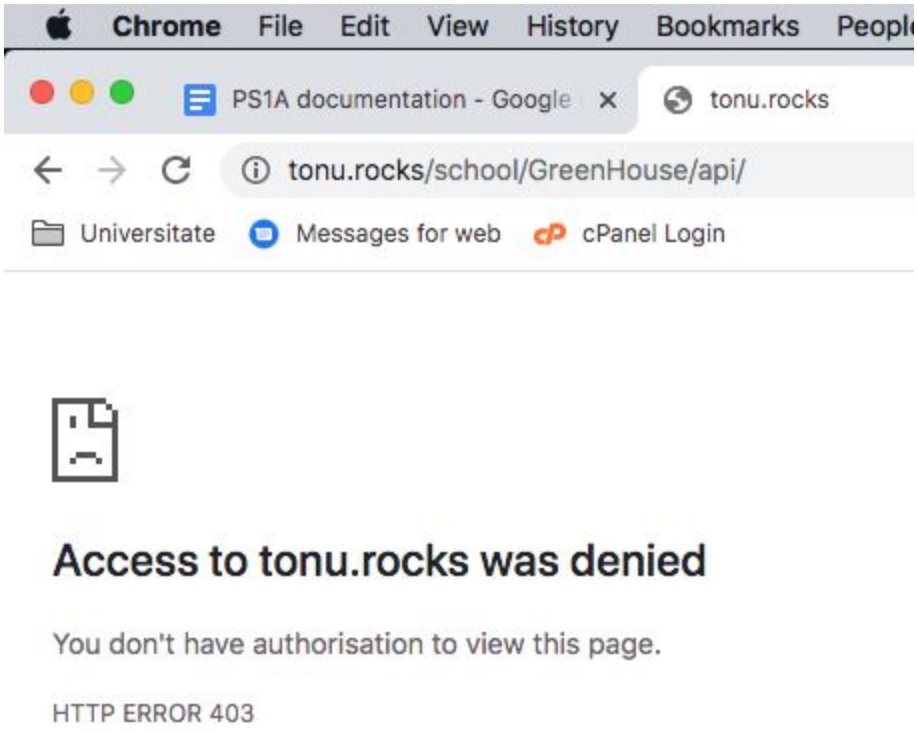
```
MacBookAir:PS1 alexandru$ python3 log.py
Username:
Password:
"insert username""insert password""account with this username does not exist"
```

Testcase passed

Scenario:

Try to access a forbidden route:

Result:



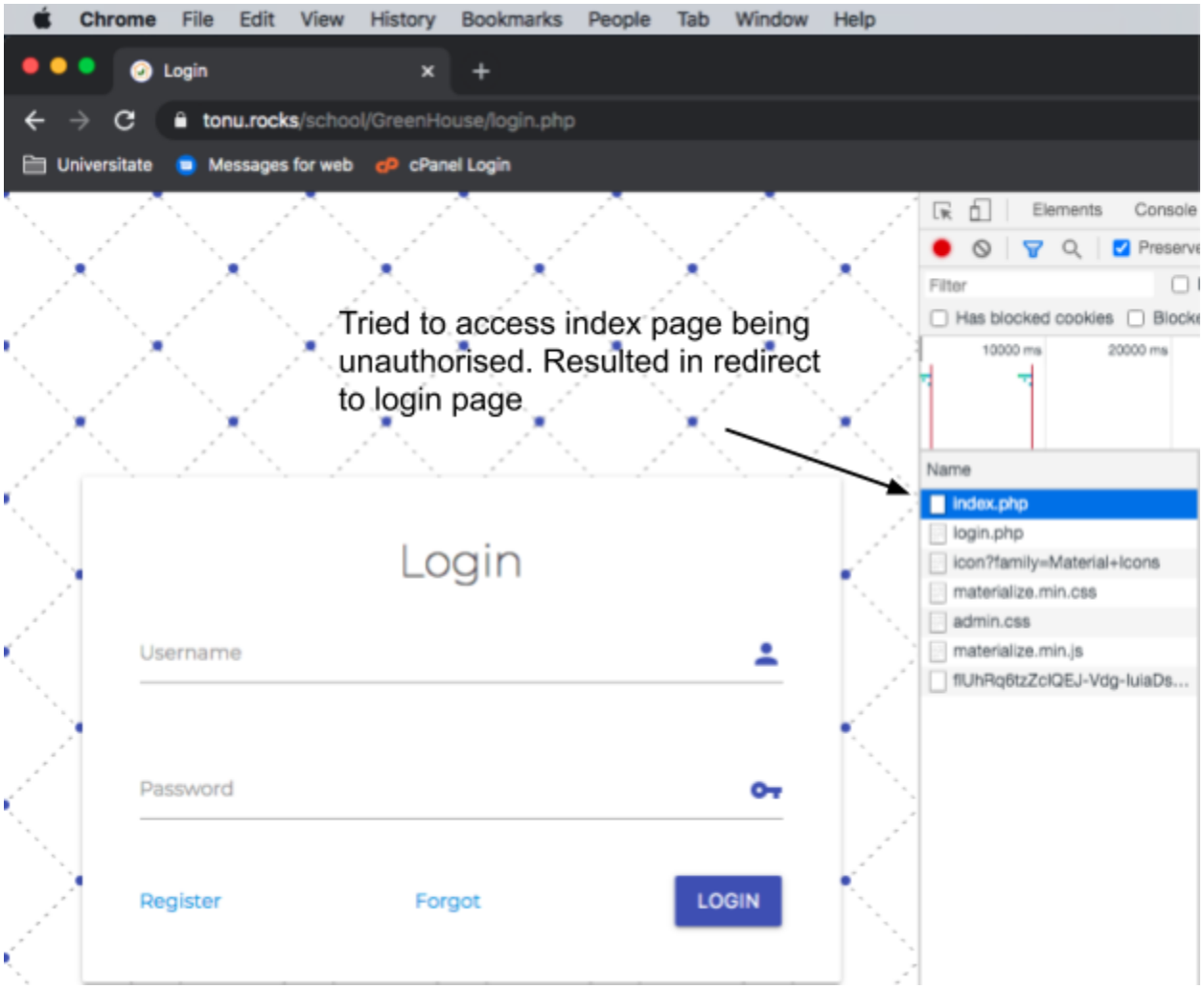
Testcase passed

Account protection:

Scenario:

Try to enter account without being logged in
Expected: redirect to login page

Result:



Testcase passed

Static analysis:

Static analysis helps identify errors while typing, prevent using deprecated functionalities and guide you to use a proper design pattern for the project you're working on.

C/C++ IntelliSense for VS Code
Python extension for VS Code
ESLinter for VS Code
PHP IntelliSense for VS Code

Unit tests:

In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Wikipedia

Technology that I used to run unit tests is called Jasmine. And it is being used like that:

```
describe("A suite is just a function", function () {
   var a;
   it("and so is a spec", function () {
       a = true;
       expect(a).toBe(true);
   });
});
```

Asserting code with snippets like this ensures you that code will work fine after every change you made, thus less bugs and more consistent code is delivered.
Another aspect of testing is code coverage. This practice assures us that all the code has been tested, and there are no redundant sections. For Javascript a great tool is Karma.js

Metrics used in application testing

1. Retention Rate:

Interaction with service is based on a website with JavaScript on the client side. In order to retain a lower percentage of customers JS code should run on a bigger amount of browsers. To achieve that I compile modern JS 2020 code to ES5, an ECMAScript standard from 2009. Which according to kangax is covering over 99% of browsers.

2. Churn Rate:

1 - 0.99 = 0.01

To get better metrics on my website I use Google Analytics:

```
    <title>Greenhouse - Home</title>


    <!-- Google Analytics -->
    <script>
       (function(i, s, o, g, r, a, m) {
           i['GoogleAnalyticsObject'] = r;
           i[r] = i[r] || function() {
               (i[r].q = i[r].q || []).push(arguments)
           }, i[r].l = 1 * new Date();
           a = s.createElement(o),
               m = s.getElementsByTagName(o)[0];
           a.async = 1;
           a.src = g;
           m.parentNode.insertBefore(a, m)
       })(window, document, 'script', 'https://www.google-analytics.com/analytics.js', 'ga');

       ga('create', 'RA-12264-3', 'auto');
       ga('send', 'pageview');
    </script>
    <!-- End Google Analytics -->
</head>
```

Code coverage is also a metric. It shows us how well tested an application is and because of thi, this one should be also taken into account.

Conclusions

The evolution of IoT is becoming more pronounced. New and new areas of our daily lives are automated, thus increasing the quality of life. But before diving into development one should consider making a deep analysis on the subject, to make an investigation about other solutions that may have been implemented and decide if his approach is better in some way than others.

Having this in mind, one can begin to develop the architecture of the system. Diagrams like Use-case, Flow-chart and UML will solve a lot of time by minimizing the possibility of writing a malfunctioning system, thus it avoids a lot of refactoring in the future.

When the architecture is built, everything is hooked-up, and a clear vision on how the app should work is settled, work on implementation can start. The built system must be consistent and easy to maintain. In order to ensure the quality of the written code, during the development you also need to write and test that would cover the whole system, both at the functional level and at the level of business logic. Adherence to metrics recommendations is a considerable plus.

When you are done, don't forget to document your code, someone will have to work on it after you and a good documentation in pairs with well commented code is the key to a great maintenance.

Bibliography

Source code:
http://github.com/tonualexandru/univ/blob/PS1/
by Tonu Alexandru (author of this paper)
at Jun 4, 2020

Arduino PID Controller:
https://playground.arduino.cc/Code/PIDLibrary/
by Brett Beauregard
contact: br3ttb@gmail.com

Arduino 7 segment display:
https://github.com/sparkfun/SevSeg
by Dean Reading and Nathan Seidle
in 2012, latest commit on 14 Feb 2017

Arduino thermometer:
https://create.arduino.cc/projecthub/TheGadgetBoy/ds18b20-digital-temperature-sensor-and-arduino-9cc806#:~:text=DS18B20%20is%201%2DWire%20digital,used%20on%20one%20data%20bus.
by Konstantin Dimitrov
on November 22, 2016

Python serial read/write:
https://pyserial.readthedocs.io/en/latest/pyserial_api.html
by Chris Liechti
in 2017

Python threads:
https://stackoverflow.com/questions/2697039/python-equivalent-of-setinterval
by doom
on Feb 9 '18 at 15:44

Python tkinter:
https://www.python-course.eu/tkinter_entry_widgets.php
by Bernd Klein

Python sessions:

https://stackoverflow.com/questions/12737740/python-requests-and-persistent-sessions
by Anuj Gupta on Oct 5 '12 at 0:24
and ohndodo on Nov 10 '19 at 21:04

JavaScript 2020 release notes:

https://www.youtube.com/watch?v=7TpAN4FISeI&t=2648s
by Владилен Минин
on May 29, 2020

Jasmine documentation:

https://jasmine.github.io/tutorials/your_first_suite

Karma code coverage:
http://codereform.com/blog/post/unit-test-code-with-jasmine-and-code-coverage-with-karma-coverage-using-istanbul/
by George Dyrrachitis
In 2018

Metrics documentation:

https://buildfire.com/mobile-app-metrics-you-should-track/
by Ian Blair

https://www.braze.com/blog/essential-mobile-app-metrics-formulas/
by MARY KEARL
at Mar 9, 2016