Automatic Gardening System

Kristmund Ryggstein and Hergeir Winther Lognberg

University of the Faroe Islands

December 29, 2018

ii

Abstract

We were asked to do an IT project using an Arduino Uno. We chose to to do an automated gardening system with it, because that is relevant for some projects that are currently in progress on Sandoy, where we are from. Our ardunio project entails an automated system that waters, measures moisture, heat and light exposure. That is done by customizing an Arduino Uno which has a moisture-, heat and light sensor that does all our measurements. All the the data is then collected by the Arduino, and then sent to a Web Service via. a esp-05 wifi module. The Web Service receives the data, and presents it to the user. The user is registered with the Web Service, and can track his plant data through his registered user account, where he also can customize water input for the plant/s.

ii

Contents

Foreword

The foreword often consists of the author's personal opinion, the history of how the work emerged, how the work is structured, how the reader can choose to read it etc. It might also contain dedications, acknowledgements and similar things.

Acknowledgement

We want to thank The University of The Faroe Islands for providing us with the materials provided to us from the beginning, and to thank them for providing us with the funds to procure new materials. We also want to thank Benadikt Joensen for guiding and teaching us on the more electrical side of things in our project.

Chapter 1

Introduction

The purpose of this report is to document our project and the process of developing it.

We have chosen to work with automatic a gardening system because it has some tasks, what we believe, are tedious and easy to forget. There has been an increasing interest on Sandoy in trying to grow more vegetables on the islands, and to sell them. That grown interest is in part thanks to initiatives such as "Veltan" and "Eplafestivalurin". Trying to be self-sufficient with regards to food produce is, according to us, a good idea, and we believe our modest electronic gardening project fits well with that philosophy, and will be interesting for those, who want to try to be more self-sufficient.

Chapter 2

Setup

The Arduino Uno, TMP36GZ heat sensor, resistors and the cables which we use in this project, was provided by the one responsible by the course. At the start we used the FC28 moisture sensor, but it uses DC current which means that it corrodes quickly because of electrolysis in the soil which alters the soil composition and could potentially damage the plant. This would also eventually lead to tainted readings. We augmented it to use an AC pulse that can be seen in 2.2.1, but we decided to opt for Capacitive Soil Moisture Sensor V1.2 that is Corrosion Resistant. The FC28 sensor can be seen in A.1 and the Capacitive sensor can be seen in figure **??**.

2.1 Arduino

Arduino is an open-source electronics platforms which is based on user-friendly software and hardware components[2]. It can read outputs from these components e.g. readings from sensors, and it can give input to these sensors.

We worked on an Arduino Uno. It is a microcontroller board. It has 14 digital i/o pins. they can be located on the board right next to the text *DIGITAL*. Six of those pins can be used as Power Width Modulation, shortened as PWM, outputs. Those pins are the ones who have a next to their ascribed numbers. It has six analog pins. It then has five different kinds of power pins. We only use the GND and the 5V and 3.3V pins. The 5V and 3.3 pins output a 5V and 3.3V from the regulator on the board which can be used to power different hardware components. The 3.3V pins has a maximum current draw of 50mA. GND stands for ground, and serves as the common return path for current from the different hardware components in our Arduino system.

2.2   Moisture Sensors

2.2.1   FC28

Our intention was to use the Arduino to induce an alternating current through
the sensor. To simplify things, we circumvented the circuit of FC-28 such
that we connected directly to the pins on the sensor. To alternate the cur-
rent, we first emit 5V from pin 12 and set pin 13 to low (0V) for 1 ms then
we take a reading through the analog pin A0. Then we flip the pin 12 and
13 (5V from pin 13) for the same amount of time. This should somewhat
remedy the problems of electrolysis by shifting the ions back and forth be-
tween the moisture sensor pins, instead of just letting them congest at one
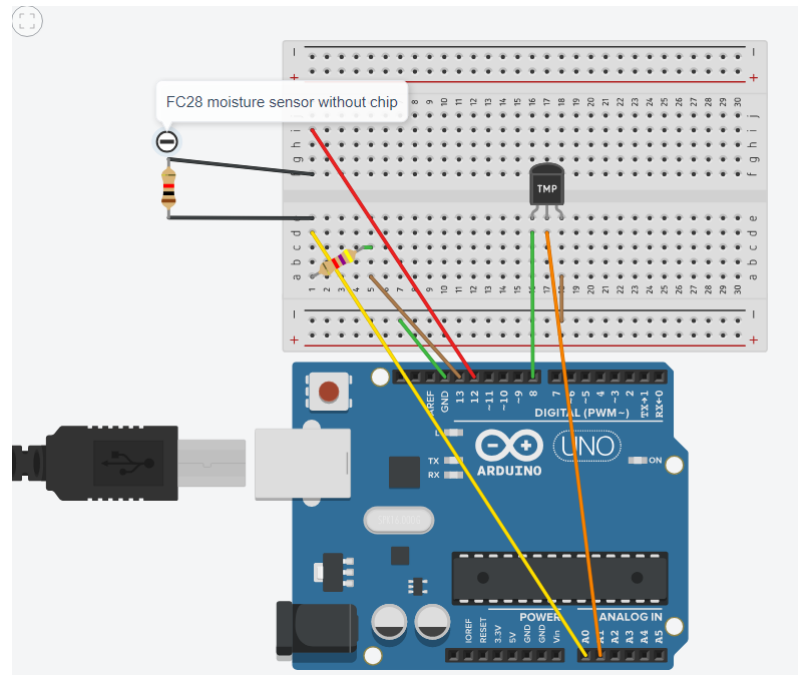pin. The chip had to be removed so that it could use AC. The setup can be
seen below.

Figure 2.1:

The resistor in the circuit helps increase the amount of current entering
the analog sensor. Without it about half the current would go in the pin 13
when the measurement occurs. We are therefor able to steer the range of
current entering the analog input. The left left leg is connected to 8 digital
pin, the middle leg is connected to A1 pin, and the third leg is connected
to -18 on the breadboard which is then connected to the ground pin on the

Arduino.

2.2.2 Capacitive Soil Moisture Sensor v1.2

We looked at reviews, and this one seemed to be the most corrosion resistant of the ones we looked at. There is no metal facing the surface, so it has a higher lifetime than the other sensor. It measures moisture, or rather the ions in the soil, by reading the capacitance between two plates in the sensor. It has lesser range than the other one, as it measures air values as 555 and completely submerges as 280. We thought the minimized range wouldn't worsen our readings, and the benefit from increased lifetime far outweighed it.

```
1 uint8_t CapacitiveMoistureSensor::readPercent(uint16_t
      val)
  {
3     /* 555 is the value for moisture in the air
    // 288 is the value for when the sensor is
        submerged in water
5    then sett 555 to be 0%, 288 to be 100%, and map
         the percent between them */

7   val = constrain(val,280,555);
    double valPercent = map (val, 555, 280, 0, 100);
9

    return valPercent;
11 }
```

readPercent takes the read value from the analog read, and converts it to a percent that is mapped between air- and water value. Line 7 sets a constrain on val variable which says that it can't be more than 55 and not less than 280. Line 8 creates a new variable, which is val parameter mapped as a percentage between 555 and 280 where 555 is 0% and 280 is 100%.

```
  uint8_t CapacitiveMoistureSensor::read()
2 {
    digitalWrite(dPin,HIGH);
4   delay(1000);
      uint16_t val = analogRead(aPin);
6

      return readPercent(val);
8 }
```

This method reads the output from the capacitive sensor. digitalWrite(dPin, HIGH) starts the sensor. dPin is the the digital pin in which the sensor is connected to the Arduino. delay(1000) pauses the code at one second, then it initalized the val varaiable through analogRad(aPin), where aPin is the analog pin which the sensor is connected. Then the method calls and the above readPercent(val) method, which converts the value to a percentage, and the read() method returns that percentage to where the method was called from i.e. the main Arduino program.

2.3 Temperature Sensor

2.3.1 TMP Sensor
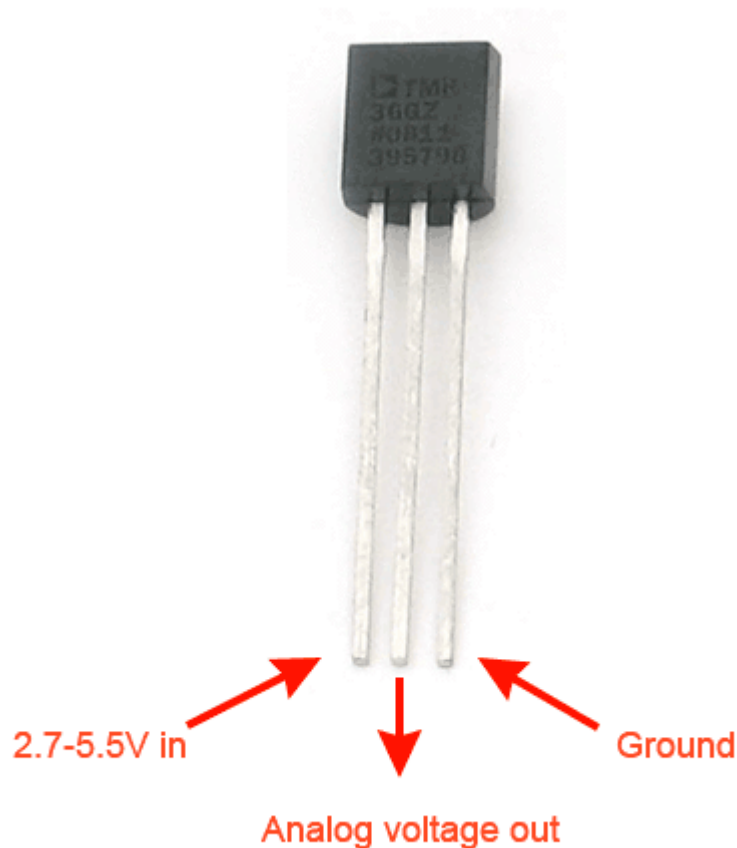
We used the TMP36 temperature sensor, as it was supplied to use by the course administrators and that it suits our purpose. The TMP36 is fairly simple. It can measure degrees from -40°C to °125C, where it has a precision[3] of ±2°C. It works as a diode so when it changes temperature, it then its voltage changes accordingly. The sensor measures the small change and outputs an analog output based on it, so its possible to calculate the temperature according to that output. The sensor's collector pin is attached to the 8 digital pin, the gate to the analog input A1 and the third leg is We use a readCelcius() method from the Temperature Class, and it can be seen below.

```
const int8_t TemperatureSensor::readCelsius()
{
    digitalWrite(triggerPin, HIGH);
```

```
4      delayMicroseconds(200);
       uint16_t reading = analogRead(readPin);
6      digitalWrite(triggerPin, LOW);
       // converting that reading to voltage, for 3.3v
          arduino use 3.3
8      double voltage = reading * 5.0; // * 0.0009775171;
       voltage /= 1024.0;
10     return round((voltage - 0.5) * 100);
   }
```

digitalWrite(triggerPin, HIGH) sends a pulse to the sensor which starts it, then delayMicroseconds() delays the program, so that the sensor waits while the sensor is activated. Then reading = analogRead(readPind) reads the output from the sensor. The output from it, is converted to voltage by multiplying it with five. That is then divided by 1024 to find the percentage of the analog read, because the Arduino outputs between 0 and 1023, where 0 is not voltage and 1023 is 5V. Five is then subtracted from that, and then that result is multiplied with 100 to convert it from mV to degrees. Five is subtracted from the reading because we want it so that 0V corresponds to -50°C.

2.4   Light Sensor

2.4.1   LDR Resistor
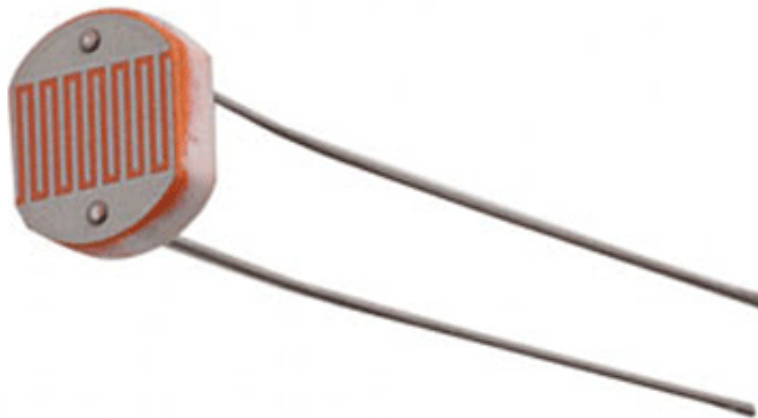
We use a LDR Resistor to detects the level of light in the room. The resistor changes its resistance based on how much light is in the room. If the room is unlit then the resistor will have a resistance of a few ohms, but will decrease if the room gets lighter. It's not ideal for our purpose because it's

not very sensitive, and that it needed to be calibrated. We chose it because it was available and that we wanted to focus on the webserver and other areas of the project.

2.5   Water Pump

Our water pump is said to have a flow rate of 80-120L H. We haven't tested that, but our tests deem that it is strong enough for our purpose- maybe even too strong. It uses a DC current 2.5-6V. The rest of the product description is in A. Its tube was bought in the local Pet store, and was glued on the facing left in the picture.

2.6   Wifi Module
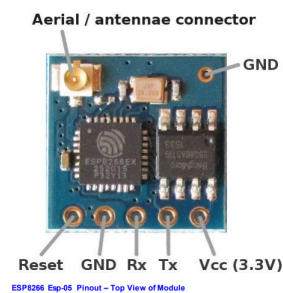
The ESP8266-05 wifi module was the hardest to work with. We initially bought an ESP8266-01A.2, but that was even tougher to work with, because it had 8 pins, and the distance between each pin on the module was so small that we couldn't properly solder each part without connecting it to the next one. We then bought the 05 version which was much easier to solder. A picture is attached of the module in A.2. We don't use the antennae nor the upper GND pin. We have a problem with the module in that the Arduino UNO 3V supply has inadequate current capabilities to power it. It still works but we get occasional garbage data from the packages sent sent from the module. Garbage transmissions seem to increase the longer the module is in use. It's possible to use an alternate 3.3V supply rather than the one from the Arduino but we didn't look into it, as it would take focus from some of the other work needed in the development.

Chapter 3

Website

Webserver

We chose to work with Microsoft's ASP.NET Core which is an open-source cross-platform framework for building cloud-based and internet-connected applications. We chose this framework because we familiar with it through the course 5022.16 Web Applications: ASP.NET with C# which we had right before this course. It is something both collaborators in this project like to use.

3.1 Hosting

We initially chose to host the webserver on Microsoft's Azure, which all university students get access to through Microsoft's student offer. We quickly found out that it didn't work for us, because it only allows port 8080 and port 443 to be open. We needed other ports to use for communicating between the Arduino and the webserver.

3.2 Login

We found out that .Net Core has something called Identity Core[1] which handles the login for an ASP.NET Core application. The user can create a membership for the application by using his or she's Facebook, Twitter, Microsoft, or Google login credentials. We could have added the other options, but we deemed it unnecessary as most faroese that would be willing to use a system such ours will most likely have a Facebook account. It's also possible to login with a regular email.

```
services.AddAuthentication().AddFacebook(facebookOptions =>
{
    facebookOptions.AppId =
            Configuration["Authentication:Facebook:AppId"];
    facebookOptions.AppSecret =
            Configuration["Authentication:Facebook:AppSecret"];
});
```

This code block adds Facebook authentication to our login service. facebookOptions.AppId and facebookOptions.AppSecret are plugins from chrome, which had to be created in Chrome to support facebook login for our project. We found the tutorial for it on Microsoft's support page for identity on. The idea with the login functionality is that plants can be attached to users in our system, and that he is able to view data about his plants on the website.

3.2.1   Project E.Garden Structure

Our .Net Core project is divided into three parts. It's good practice to have all business logic i.e. models that represents plants, users, and Arduino Data in a separate class library from the main program. It's done so the project is decoupled, and so that the classes can be included in a separate project. This class library is called Repository in our project, and structure of it is also divided into three parts. First part is the Model folder where the classes are and can be seen in B.1. ApplicationUser inherits from User, because that makes it so that the properties from ApplicationUser gets added to AspNetUsers in the database. That is a functionality of the previously mentioned Identity.

One part of the library is a folder that contains the skeleton i.e. interfaces for our business handling, then the implementation of those interfaces is located in the Concrete folder. This means that the EFPlantRepository model inherits from IPlantRepository, and then specifies what the methods User-Plants() and SavePlants() should do. These repositories are then initialized in the file Startup.cs, specifically the ConfigureServices(IServiceCollection services) method. It's initialized with something called Dependency Injection, and that can be seen in below.

Listing 3.1: Dependency Injection

```
services.AddScoped<IApplicationUserAccessor, ApplicationUserAccessor>();
services.AddScoped<IPlantRepository, EFPlantRepository>();
services.AddScoped<IArduinoDataRepository, EFArduinoDataRepository>();
```

This initializes at run time all instances of those three interfaces in the project as the EF implementation to the right of them. AddScoped or AddTransient establishes the lifetime of the services, and scoped means that it is created once per request. The code for the interfaces can be seen in **??**. The implementations and the explanations for the interfaces can be seen below.

Listing 3.2: Observer Class

```
public class EFArduinoDataRepository : IArduinoDataRepository
{
    private readonly EGardenerContext _context;

    public EFArduinoDataRepository(EGardenerContext context)
    {
        _context = context;
    }
    public void SaveData(ArduinoData data)
    {
        var context = _context;
        var plants = context.Plants.Include(x => x.Datas);
        var plant = plants.Single(x => x.PlantId == data.PlantId);
        var datas = plant.Datas;
        datas.Add(data);
        context.SaveChanges();
    }

}
```

It has one private datamember called _context of type EGardenerContext, which gets initialized in the EFArduinoDataRepository. This context is the context between the models and their corresponding database entities. All the EFRepositories have this EGardenerContext. The SaveData(Arduino Data) is the actual implementation of the IArduinoDataReposistory. It takes an ArduinoData object as a parameter. First a new context variable is initializied, then a plant gets initialized by dotting into the context, and choosing a plant in the database that has a PlantId that corresponds to plantId the data parameter has. A list of data gets initialized by assigning it to the data list in the newly created plant, and then the data parameter is added to that list. context.SaveChanges(); saves all the changes made.

Listing 3.3: Observer Class

```
public class EFPlantRepository : IPlantRepository
{
    private readonly EGardenerContext _context;
```

```csharp
    private readonly IApplicationUserAccessor _userAccessor;

    public EFPlantRepository(EGardenerContext context,
    IApplicationUserAccessoruserAccessor)
    {
        _context = context;
        _userAccessor = userAccessor;
    }


    public async Task<IEnumerable<Plant>> UserPlants()
    {

        var user = await _userAccessor.User;

        user = _context.Users.Include(x => x.Plants)
            .ThenInclude(x => x.Datas).Single(x => x.Id == user.Id);

            return user?.Plants ?? new List<Plant>();
    }

    public async Task SavePlant(Plant plant)
    {
        var user = await _userAccessor.User;
        var dbUser = await _context.Users.FindAsync(user.Id);

        plant.ApplicationUser = user;
        if (dbUser.Plants == null)
        {
            dbUser.Plants = new List<Plant>();
        }

        dbUser.Plants.Add(plant);
        var amount = await _context.SaveChangesAsync();
    }
}
```

This class has a private field of type IApplicationUserAccessor called userAccessor which is used to access the users in the database. UserPlants() returns an asynchronous task that takes an IEnumerable<Plant>. IEnerumable is a C# build in interface which makes it easy to work with any kind of containers. var user = await userAccessor.User initializes the variable with user that is currently logged in. Await means that the method has to wait that initialization to finish in order to continue. User is then reassigned to the entity in the database that shares the same Id. Include(x => x.Plants) includes the plant list from the database user. ThenInclude then includes datas to each plant in the previous list. Single(x => x.Id == user.Id) finds the

correct user. The method then either returns the user's plants or returns a new plant list. It returns the user plants when the user is not null and plants is also not null.

SavePlant(Plant pltant) is an asynchronous method that stores a plant in the database. It also initializes a user with the HttpContext, but it also creates one dbUser with one from the database. The dbUser is initialized similarly with how the user in UserPlants() got reassigned, and that is by locating it by user Id. The method suspends when it initialized those variables. The parameter, plant, then initializes its own user property to the user in this method. If db.User has an uninitalized plants property, then it gets initalized with a new empty one. The plant parameter then gets added to that plants property, and the _context asynchronously saves the changes.

3.2.2 Observer Pattern

We decided to use Gang Of Four's observer pattern. We think that it fit well in our implementation of the communication between the Arduino and our Webserver. Microsoft have their own implementation of the patternpattern

In short, the pattern is about having one subject that has a list of dependencies which it notifies when there's a change.

Observer

Our provider is called Observer and implements the IObserver with ArduinoData in the template brackets. That it implements the interface means that the class implements its own version of the methods

- OnCompletet()

- OnError(Exception Error)

- OnNext(T Value)

These can be called event handlers as they get called, when those events happen. The first method indicates that the provider is finished with transferring data. Second method informs the observer that an error occurred. The third method supplies the observer with current or new data. The class has one member variable called _unsubscriber, which is an IDisposable object, and it has two properties of type ArduinoData called Data and type String which is the Observer's name. IDisposable is a C# interface that provides a mechanism for releasing unmanaged resources by its method Dispose(). ArduinoData is one we made, and it is the container for the data in

which the arduino transfers to the webserver. It's properties can be seen in
listings **??**

Observable

Appendix A

Graphics of sensors and other components

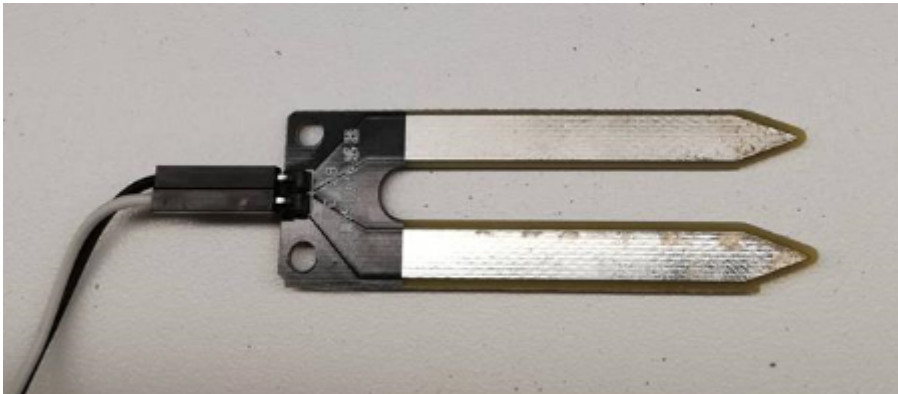The following are pictures of our sensors and other components.

Figure A.1: Corroded fc-28 after moderate use.

- 100% brand new and high quality

- DC Voltage:2.5-6V

- Maximum lift:40-110cm / 15.75"-43.4"

- Flow rate:80-120L/H

- Outside diameter of water outlet: 7.5mm / 0.3"

- Inside diameter of water outlet: 4.7mm / 0.18"

- Diameter:Approx. 24mm / 0.95"

- Length:Approx. 45mm / 1.8"

- Height:Approx. 33mm / 1.30"

- Material:engineering plastic

- Driving mode: brushless dc design, magnetic driving
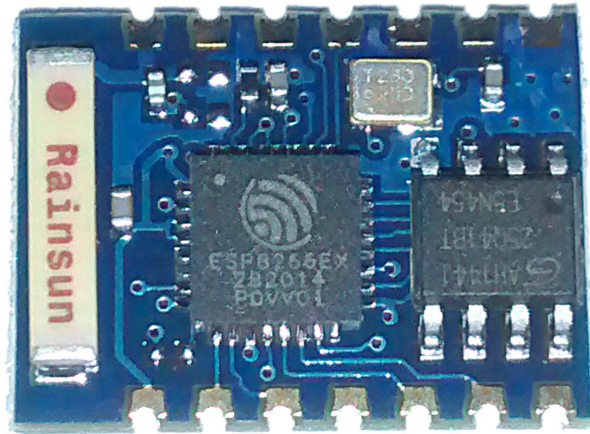
- Continuous working life of 500 hours

Figure A.2: ESP8266-01 module. You can see that distance between each pin is small which meant for us that it was hard to sold for someone inexperienced

Appendix B

Latex Listings

The code highlighting in this tex document was taken from trihedral's ArduinoLatexListing Github repository

Listing B.1: models

```csharp
public class ArduinoData
{
    public int Temperature { get; set; }
    public uint Moisture { get; set; }
    public Plant Plant { get; set; }
    public uint PlantId { get; set; }
    [Key]
    public long DataId { get; set; }
    public int Light { get;  set; }
    public int Water { get;  set; }
}

public class Plant
{
    public long PlantId { get; set; }
    public string Name { get; set; }
    public DateTime DateAdded { get; set; }
    public ApplicationUser ApplicationUser { get; set; }
    public IList<ArduinoData> Datas { get; set; }
}

public class ApplicationUser : IdentityUser
{
    public string Name { get; set; }
    public virtual List<Plant> Plants { get; set; }
}
```

Listing B.2: ArduinoDataRepository

```csharp
public interface IArduinoDataRepository
{
    void SaveData(ArduinoData data);
}

public interface IPlantRepository
{
    Task<IEnumerable<Plant>> UserPlants();
    Task SavePlant(Plant plant);
}

public interface IUserRepository
{
    ApplicationUser GetCurrentUser();
}
```

Listing B.3: Observer Class

```csharp
public class Observer : IObserver<ArduinoData>
{
    private IDisposable _unsubscriber;
    public EFArduinoDataRepository ArduinoDataRepository { get; set; }


    public void Subscribe(IObservable<ArduinoData> provider)
    {
        if (provider != null)
        _unsubscriber = provider.Subscribe(this);
    }

    public virtual void OnCompleted()
    {
    Console.WriteLine($"connection to observer closed.");
    this.Unsubscribe();
    }

    public virtual void OnError(Exception e)
    {
    Console.WriteLine($"Observer: Error in connection or transmission of data."
    }

    // This runs when data from arduino is received.
    public virtual void OnNext(ArduinoData value)
    {
        ArduinoDataRepository.SaveData(value); // save the read data
//into the data repository
    }
```

```csharp
        public virtual void Unsubscribe()
        {
            _unsubscriber.Dispose();
        }
}
```

Bibliography

[1] Rick Anderson. Introduction to identity on asp.net core.

[2] Arduino. Introduction to arduino.

[3] BC Robotics. Using a tmp36 temperature sensor with arduino.