Automatic Gardening System

Kristmund Ryggstein and Hergeir Winther Lognberg
University of the Faroe Islands

December 31, 2018

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 work 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.

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 in thanks to initiatives such as "Veltan" and "Eplafestivalurin". Trying to be self-sufficient with regards to food produce is, according us, a good idea, and that's were we think that our modest electronic gardening project fits well with that philosophy, and will be an interest 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. We used at the start a FC28 moisture sensor, but it generally uses DC current which means that it corrodes quickly because of electrolysis in the soil and alters the soil composition which could potentially damage the plant. That would also eventually lead to tainted readings. We changed it to use an AC pulse that can be seen in figure, but we decided to opt for Capacitive Soil Moisture Sensor V1.2 that is Corrosion Resistant. The FC28 sensor can be seen in B.1 and the Capacitive sensor can be seen in figure **??**.

2.1 Arduino

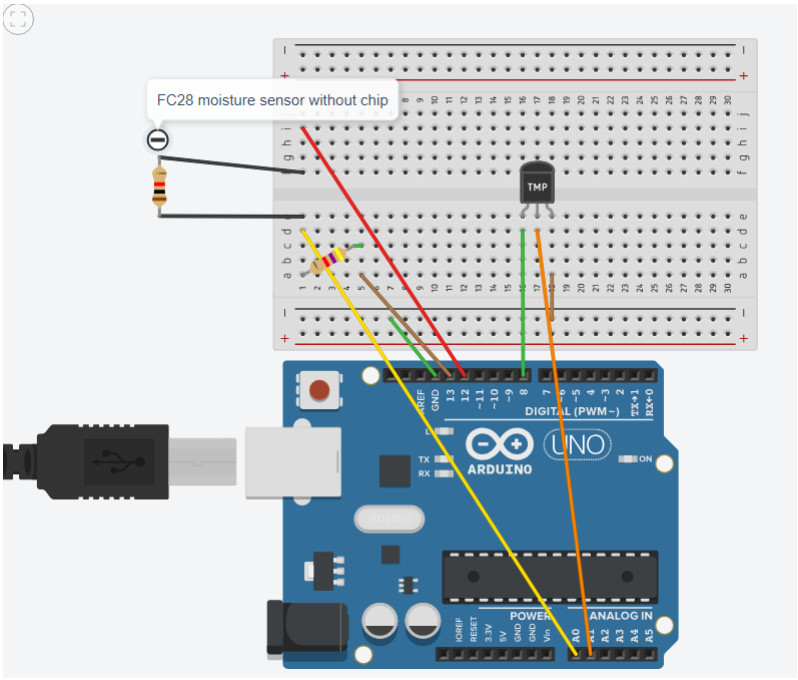
Arduino is an open-source electronics platforms which is based on user-friendly software and hardware components[2]. It can read outputs from these components e.g. readings from sensors, and it can give input to these sensors.

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 current, 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 between 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.



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. It is made by corrosion resistant material, 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.

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

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

  return valPercent;
}
```

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%.

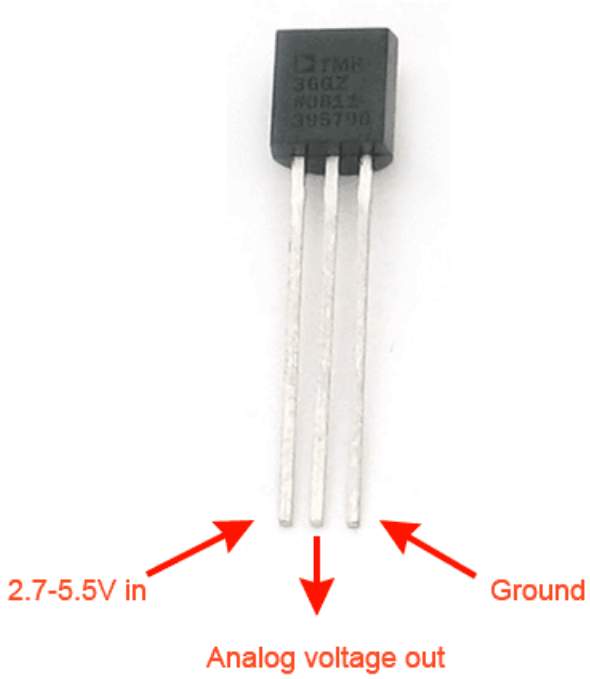
```
uint8_t CapacitiveMoistureSensor::read()
{
  digitalWrite(dPin,HIGH);
  delay(1000);
    uint16_t val = analogRead(aPin);

    return readPercent(val);
}
```

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.

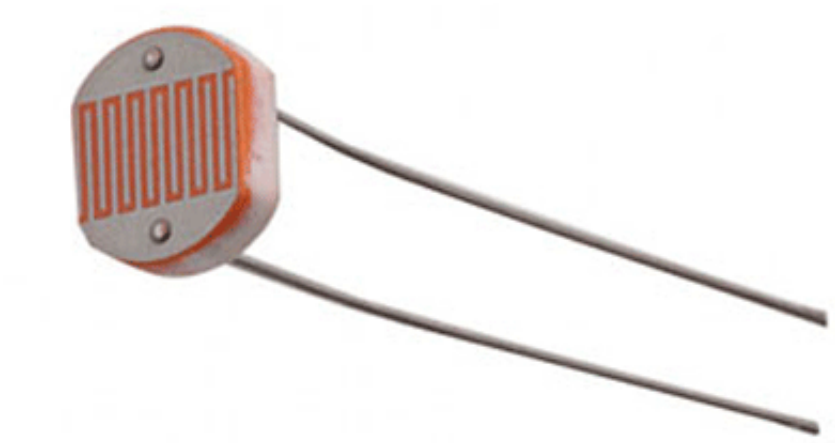
```
const int8_t TemperatureSensor::readCelsius()
{
    digitalWrite(triggerPin, HIGH);
    delayMicroseconds(200);
    uint16_t reading = analogRead(readPin);
    digitalWrite(triggerPin, LOW);
    // converting that reading to voltage, for 3.3v
       arduino use 3.3
```

```
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

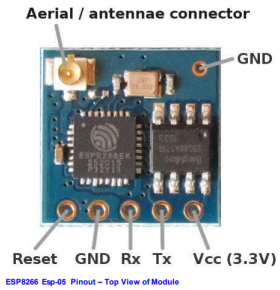


We use a LDR Resistor to detects the level of light in the room. The resistor changes its resistance based on how much light is in the room. If the room is unlit then the resistor will have a resistance of a few ohms, but will decrease if the room gets lighter. It's not ideal for our purpose because 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 if that is correct, but our tests to see if it worked deemed that it's pumping flow rate is more than enough for our use. It uses a DC current 2.5-6V. The rest of the product description is in B.1. 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 ESP8266-01B.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 B.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 [**?**] which is Microsoft's support page for Facebook external login. 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.

3.3 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 C.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 UserPlants() 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: EFArduinoDataRepository 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 initialized, 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: EFPlantRepository Class

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

3.3.1 Observer Pattern

We decided to use Gang Of Four's observer pattern[4]. 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 pattern[5] which we took inspiration from.

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. It can be seen in C.3. 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 _arduinoDataLoggerRepository for it to save. The class has two fields called _unsubscriber of IDisposable and one EFDataoggerPlantRepository called _arduinoDataRepository. IDisposable is a

C# interface that provides a mechanism for releasing unmanaged resources by its method Dispose(). ArduinoDataRepository 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 **??**. The Observer constructor initializes the field _arduinoDataRepository with a database context. The parameters in UseSQLServer are the ones for a server running on the localhost. The Subscribe(IObservable<ArduinoData> provider) method lets the provider subscribe to the current instance of Observer. Unsubscribe() and Dispose() both dispose of the resources they hold.

Observable

Our provides is the Observable class which can be seen in C.4. It implements the IDisposable, IObservable, and IHostedService interfaces. The implementation of Dispose() closes the socket, calls onCompleted on each observer in the observers list. The implementation of Subscribe() adds the observer parameter to the observer list, if it doesn't contain observers, and then returns a new instance of Unsubscriber, which was initialized with the observer parameter and the observers list. The Observable class has three private fields called Port, _socket, and the previously mentioned observers. The port is a constant int that is assigned with 8080. _socket is a TCPListener which is a standard C# class that belongs to the System.Net.Sockets namespace. It listens for connections from TCP clients. The public Observable(IServiceProvider serviceProvider) constructor initializes the list with a new list that contains IObserver<ArduinoData), and also initializes the _sokcet with any IPaddress that connects to field port. The constructor parameter, serviceProvider, then subscribes the singleton instance of Observer to the current instance of obserable, and subscribes the singleton instance of DataPusherObserver to the same instance.

Listen() listens for clients to connect to the observable class. _socket.AcceptTcpClient() accepts the pending connection request, and that request is assigned to a variable client. A new thread is then created which calls the ListenToClient(TcpClient client) method where the previously created client is the parameter. That thread is started the same time it is called.

ListenToClient() is run while a client is connected. The stream from the client is assigned to a variable reader, then the method enters a new while loop which is active as long as there's data to be received from the client. A byte array of size two and an ArduinoData object are then created. The reader then reads two bytes, then inserts those two bytes at index 0 in the array. data.PlantId = BitConverter.ToUInt16(buffer) takes the two bytes and converts it to an unsigned 16 bit intenger, and assigns data.PlantId to the

converted value. data.Temperature = reader.ReadByte() reads the current byte in the stream, advances the stream by one, and assigns the temperature with the read value. The same happens to data.Moisture. Then a new buffer is created with a byte size of 4, and the reader reads the subsequent 4 bytes in the stream to buffer from index zero. That buffer is then converted to a 32 bit integer, and data.Light is assigned to that value. data.Water is at last assigned to the last byte in the stream. The method then calls Notify(ref data), where the newly created ArduinoData is passed by reference. The method then exists the inner loop and pauses for one second and then closes the connection to client and the clause for the outer loop.

Unsubscriber is a private internal class for Observable. It also implements the IDisposable interface. It has two private fields, one List of observers and one observer. Those are initialized in the parameterized constructor which gets called in Observable's Subscribe() method as previously mentioned. This class' Dispose checks to see whether the observer field is null and that the list contains observers. The Subscriper's _observer is then removed from the list.

Notify(ref ArduinoData) takes a reference to ArduinoData as a parameter. First it checks to see if the data has a plantId equal to 21569; if that is true, then it gives the console a message a makes a return call. We had some problems with this Id, because it didn't trigger our UpdateData in DataPusher which meant that our view didn't update asynchronously. This data's other properties e.g. temperature were also peculiar, so we believe that its package was corrupted by the underpowered WIFI chip, but we don't understand how the same id seemed to reappear so often. The method then validates that data is not null. If it is then the observer at that time in the foreach loops calls OnError with an error message that says the data was corrupted. If data was valid, then the method tries to call the looped observer's OnNext(data) method with the data as a paremter, this time passed by value. If this triggers and InvalidOperationException, then the catch statement catches the exception and prints the exception message to the console.

StartAsync and StopAsync are the two methods that are inherited from the IHostedService. Those two return a task and each take a CancellationToken as a parameter. StartAsync starts the TcpListener, _socket, then starts a new thread that calls Listen and at last it returns a returns a Task that has completed successfully. StopAsync calls dispose and also returns a completed Task.

3.3.2 DataPusherObserver

DataPusherObserver is the class that connect the Arduino to the Webserver. It implements the IObserver<ArduinoData> interface. It has one field calledconnection which is of type HubConnection. This connection is initialized in the DataPusherObserver constructor, and it is the localhost/-datahub. This class implements the IObserver interface similarly as the Observer where it subscribes and notifies error the same.

OnNext takes an object of ArduinoData as a parameter. Then it asynchronously starts the connection to the server, and invokes the method "UpdateData" on the server with value as the method parameter. It's cancellationToken is set to none as well. At last it stops the connection to the server.

3.3.3 DataHub

DataHub is the class that gets triggered after the InvokeUpdateData call above. This is due to something called SignalR. This DataHub gets triggered because its name was mentioned in the url in the DataPusherClass's connection. When UpdateData gets called, it assigns a list of client to the clients assigned to the Hub. If there are no clients then the empty return statement breaks out of the method. Then it awaits for a SendAsync call for UpdateData to all clients where an ArduinoData object is the paramenter.

3.3.4 DataPusher

DataPusher.js is our javascript file that manages our plant updates on the server. It can be seen in listing **??**

3.3.5 DI class

The DI class extends the IServiceCollection with the method AddObserverLibrary(). It creates transient instances of Observer and DataPusherObserver, and adds a hosted Service with an Observable object. This method is called in Startup.cs.

3.4 Controllers

Controllers are the c part of ASP.Net core's MVC principal, which stands for Model View Controller. They are repsonsible for controlling the flow of the application execution. When you make a page reguest on the webpage to

MVC application, then a controller is responsible for handling that request. We have two controllers, HomeController and PlantController.

3.4.1 HomeController

The code for the HomeController can be seen in C.6. The controller has two private fields called _plantRepository and _userAccessor which are of type IPlantRepository and IApplicationUserAccessor. Those were dependency injected in3.1. Those fields are constructed in the parametered constructor.

Listing 3.4: Dependency Injection

```
[Authorize]
public async Task<IActionResult> Index()
{
    return View(await _plantRepository.UserPlants());
}
```

The Authorize tag is something called Data Annotation attribute. It ensures that the user must be logged in to be able to view the Index view. The Index() method returns an async Task<IActionResult>. This means that the method doesn't suspend all activities to wait for the view to fetch the user plants. When the task is completed, then then controller will present the Index view to user, and that view will have a list of plants. The code for the this view can be seen in C.7. You can see in the top that it takes IEnumerable<Repository.Models.Plant> as a model which is what the index() method supplies to the View, when it returns it. It is in layman's terms a list of plants. The html code is a table that gets filled by iterating through the plant list.

About() works exactly the same but presents a user instead of plants. The code for the view can be seen in C.8. It takes an AppilcationUser as a model, then presents that User's login name in a <p> tag. Afterwards it checks whether the user's has any plants and if he has attached a phonenumber to his accounts. The @{} tags are a way to write C# code in a cshtml file.

3.4.2 PlantController

The code for the plant controller can be seen in C.9. The whole controller is annotated by the authorize tag which means that the user must be logged in, if he wants so any one of its views. The controller has one private field of

type IPlantRepository called _repository which gets initialized in the PlantController() constructor. Its Index() method works exactly the same as the Index() one in HomeController() but the view is entirely different. The code for it is below.

Listing 3.5: models

```
@model System.Collections.Generic.IEnumerable<Repository.Models.Pla

@{
    ViewBag.Title = "Your Plants";
    Layout = "_Layout";
}

<h2>@ViewBag.Title</h2>

@await Html.PartialAsync("_plants_partial", Model)
```

The model this view uses is a Plant. The section surrounded by the curly brackets creates the title, "Your Plants" and assigns the layout type for this view to be the shared _Layout which is the skeleton that holds the navigation bar for example. The <h2> shows the title in the browser. The await statement loads asynchronously the partial view _plants_partial and supplies it the plant list. A partial is simply a view can be used in multiple views.

Listing 3.6: PlantsPartial

```
@using Microsoft.AspNetCore.Mvc.Rendering
@model System.Collections.Generic.IEnumerable<Repository.Models.Pla

<h1>Your plants: </h1>

@foreach (var plant in Model)
{
    @await Html.PartialAsync("_plant_partial",plant)
    <br/>
}

<script src="~/lib/signalr/dist/browser/signalr.js"></script>
<script src="~/js/DataPusher.js"></script>
```

This view loops through the plant list, and supplies the partial view, _plant_partial each element for each iteration. The bottom two tags includes the SignalR and Datapusher javascript libraries. The
 creates a linebreak.

Listing 3.7: plantPartial

```
@model Repository.Models.Plant

<div class="plant" id="@Model.PlantId">
    <div class="info">
        <div>PlantId: <span>@Model.PlantId</span></div>
        <div>DisplayName: <span>@Model.Name</span></div>
        <div>Date added: <span>@Model.DateAdded</span></div>
    </div>

    <div class="data">
        @await Html.PartialAsync("_arduino_data_partial.cshtml", @Model.Dat
        @await Html.PartialAsync("_data_chart_partial",@Model);
    </div>
</div>
```

This view uses a single plant model, one which was supplied to it in loop in the previous partial view. First a html div element is created as a container for the rest of the html element. Its class is "plant" which means it gets stylized after, what that html class says. It's id tag is whatever PlantId is. Id is used stylize single elements with by targeting the unique id. Then it creates a new div element as class info, and this div contains tree other divs with description about the each attribute of the printed model. The last div then loads the partial view and _chart _partial. The arduino data partial view gets either the last element in the datas list in Plant, or it gets a null value. Data_chart is supplied with the model

Appendix A

Circuit Diagrams

The following are diagrams of our components and Arduino system.

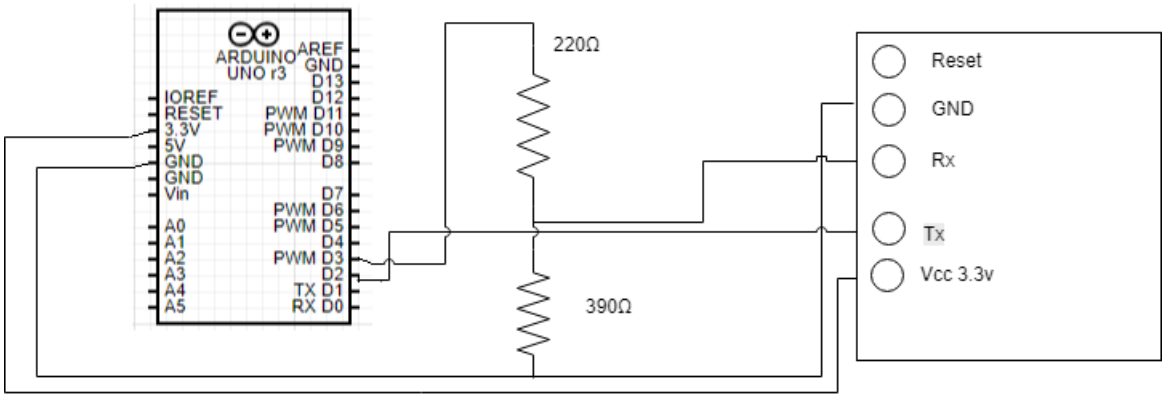


Figure A.1: Circuit diagram of the Wifi chip

Appendix B

Graphics of sensors and other components

The following are pictures of our sensors and other components.

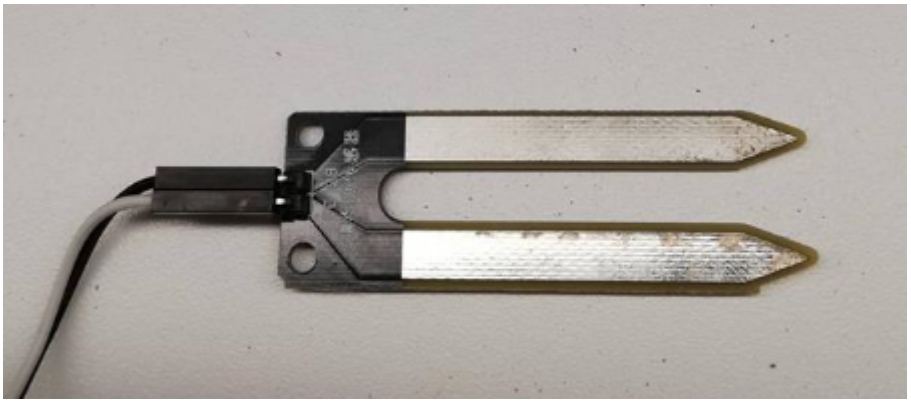


Figure B.1: Corroded fc-28 after moderate use.

B.1 Pump Description

- 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

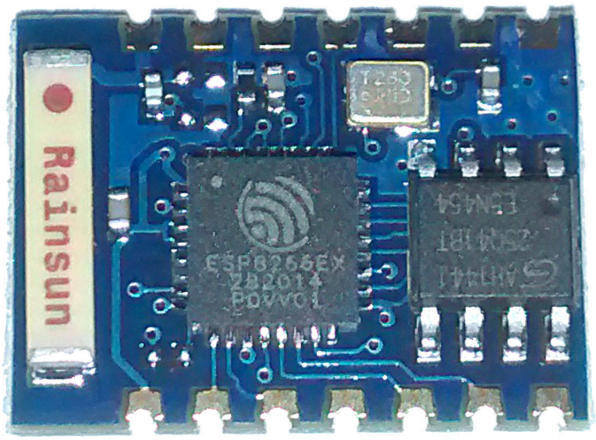


Figure B.2: ESP8266-01 module. You can see that distance between each pin is small which meant for us that it was hard to sold for someone inexperienced

Appendix C

Latex Listings

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

C.1 Models

Listing C.1: models

```
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; }
```

```
}
```

C.2 ArduinoDataLoggerRepository

Listing C.2: ArduinoDataLoggerRepository

```
public class EFDataLoggerPlantRepository
{
    private readonly EGardenerContext _context;

    public  EFDataLoggerPlantRepository(
    EGardenerContext context)
    {
       _context = context;
    }


    public void SavePlantData(ArduinoData data)
    {
        var plantQueryable = _context.Plants
            .Include(x => x.Datas);
             var plant = plantQueryable.Single(x=>
             x.PlantId == data.PlantId);

        plant.Datas.Add(data);
        _context.SaveChanges();
    }
}
```

Listing C.3: Observer Class

```
public class Observer : IObserver<ArduinoData>, IDisposable
{
    private IDisposable _unsubscriber;
    private readonly EFDataLoggerPlantRepository _arduinoDataRepository;

      public Observer()//, IObservable<ArduinoData> observable)
        {
            _arduinoDataRepository = new EFDataLoggerPlantRepository(new
            EGardenerContext(
                new DbContextOptionsBuilder()
                   .UseSqlServer("Server=127.0.0.1;Database=EGarden;
                   User Id=SA;Password=Password0").Options)
            );
        }
```

```
    public void Subscribe(IObservable<ArduinoData> provider)
    {
        if (provider != null)
        {
            _unsubscriber = provider.Subscribe(this);
        }
    }
    public void OnCompleted()
    {
        Console.WriteLine($"connection to  closed.");
        this.Unsubscribe();
    }

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

    // This runs when data from arduino is received.
    public void OnNext(ArduinoData value)
    {
       _arduinoDataRepository.SavePlantData(value);
    }

    public void Unsubscribe()
    {
        _unsubscriber.Dispose();
    }

    public void Dispose()
    {
        _unsubscriber?.Dispose();
    }
}
```

Listing C.4: Observable Class

```
public class Observable : IDisposable, IObservable<ArduinoData>, IHostedService
{
    private const int Port = 8080;
    private readonly TcpListener _socket;
    private List<IObserver<ArduinoData>> Observers { get; }

    public Observable(IServiceProvider serviceProvider)
    {
    //prep
        Observers = new List<IObserver<ArduinoData>>();
        _socket = new TcpListener(IPAddress.Any, Port);
```

```
        // subscribing necessary observers
        serviceProvider.GetService<Observer>().Subscribe(this);
        serviceProvider.GetService<DataPusherObserver>().Subscribe(this);
    }

    private void Listen()
    {
        while(true)
        {
            var client = _socket.AcceptTcpClient();
            new Thread(() => ListenToClient(client)).Start();
        }
    }

    private void ListenToClient(TcpClient client) {
        while(client.Connected)
        {
            var reader = client.GetStream();
            while(client.Available > 0)
            {
            // while(reader.DataAvailable)\\
             {
                byte[] buffer = new byte[2];
                var data = new ArduinoData();

                reader.Read(buffer,0,2);
                data.PlantId = BitConverter.ToUInt16(buffer);
                data.Temperature = reader.ReadByte();

                data.Moisture = (uint) reader.ReadByte();

                buffer = new byte[4];

                reader.Read(buffer,0,4);
                data.Light = BitConverter.ToInt32(buffer);
                data.Water = reader.ReadByte();

                Notify(ref data);
              }
                Thread.Sleep(1000);
            }
            client.Close();
       }


    public void Dispose()
    {
         // closing
```

```
        _socket.Stop();

        foreach (var observer in Observers.ToArray())
            if (Observers.Contains(observer))
                observer.OnCompleted();

            Observers.Clear();
        }


    public IDisposable Subscribe(IObserver<ArduinoData> observer)
    {
        if (!Observers.Contains(observer))
                Observers.Add(observer);

            return new Unsubscriber(Observers, observer);
    }

    private class Unsubscriber : IDisposable
    {
        private readonly List<IObserver<ArduinoData>> _observers;
        private readonly IObserver<ArduinoData> _observer;

        public Unsubscriber(List<IObserver<ArduinoData>> observers,
                                IObserver<ArduinoData> observer)
        {
            this._observers = observers;
            this._observer = observer;
        }

        public void Dispose()
        {
            if (_observer != null && _observers.Contains(_observer))
                    _observers.Remove(_observer);
        }
    }

    public void Notify(ref ArduinoData data)
    {
        if (data.PlantId == 21569)
        {
            Console.WriteLine("something something id invalid..-");
            return;
        }

        foreach (var observer in Observers)
        {
            if (data == null)
                observer.OnError(new Exception("Notifying corrupt data from arduino"))
```

```
            else
            {
                try
                {
                    observer.OnNext(data);
                }catch (InvalidOperationException exception)
                {
                        // TODO: ...
                    Console.WriteLine(exception.Message);
                }

            }
        }
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _socket.Start();
        new Thread(Listen).Start();
        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        Dispose();
        return Task.CompletedTask;
    }
}
```

Listing C.5: Datapusher.js Class

```
const connection = new signalR.HubConnectionBuilder().withUrl("/dataHub")
                                                     .build();

connection.on("UpdateData", function (data) {
    console.log(data);
    const plantDiv = document.getElementById(data.plantId);

    plantDiv.getElementsByClassName("id")[0]
        .getElementsByClassName("value")[0]
        .textContent = data.dataId;

    plantDiv.getElementsByClassName("temperature")[0]
        .getElementsByClassName("value")[0]
        .textContent = data.temperature;

    plantDiv.getElementsByClassName("moisture")[0]
        .getElementsByClassName("value")[0]
```

```
        .textContent = data.moisture;

    plantDiv.getElementsByClassName("light")[0]
        .getElementsByClassName("value")[0]
        .textContent = data.light;

    plantDiv.getElementsByClassName("water")[0]
        .getElementsByClassName("value")[0]
        .textContent = data.water;
});

connection.start().catch(function (err) {
    return console.error(err.toString());
});
```

Listing C.6: HomeController

```
public class HomeController : Controller
{
    private readonly IPlantRepository _plantRepository;
    private readonly IApplicationUserAccessor _userAccessor;


    public HomeController(IPlantRepository repository,
                          IApplicationUserAccessor userAccessor)
    {
        _plantRepository = repository;
        _userAccessor = userAccessor;
    }

    [Authorize]
    public async Task<IActionResult> Index()
    {

    return View(await _plantRepository.UserPlants());
    }

    public async Task<string> UpdateName()
    {
        await _plantRepository.SavePlant(new Plant()
        {
            Name = "hey"
        });
        return _plantRepository.UserPlants().ToString();
    }

    public async Task<IActionResult> About()
    {
        ViewData["Message"] = "Description of your profile";
```

```
        var user = await _userAccessor.User;

        return View(user);
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
                   NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
        HttpContext.TraceIdentifier });
    }
}
```

Listing C.7: IndexView

```
@model IEnumerable<Repository.Models.Plant>

<div align="center">
    <table class="table table-condensed">
        <tr>
            <th>Plant ID</th>
            <th>Name</th>
            <th>Added</th>
        </tr>
        <tr>
        </tr>
        @foreach (var plant in Model)
        {
            <tr>
                <td>@plant.PlantId</td>
                <td>@plant.Name</td>
                <td>@plant.DateAdded</td>
            </tr>
        }
    </table>
</div>
```

Listing C.8: AboutView

```
@model Repository.Models.ApplicationUser;
```

```
<h2>Your Profile Description</h2>
<p>Login user name: @Model.Name</p>
@{
    if (Model.Plants == null)
    {
        <p>Number of Plants: 0</p>
    }
    else
    {
        <p>Number of Plants: @Model.Plants.Count</p>
    }

    if (!Model.PhoneNumberConfirmed)
    {
        <p>No phone number attached to this user.
         You can add it by pressing
         your user name in the top right corner</p>
    }
    else
    {
        <p>Phone Number: @Model.PhoneNumber</p>
    }
}
```

Listing C.9: PlantController

```
[Authorize]
public class PlantController : Controller
{
    private readonly IPlantRepository _repository;

    public PlantController(IPlantRepository repository)
    {
       _repository = repository;
    }


    public async Task<IActionResult> Index()
    {
        return View(await _repository.UserPlants());
    }

    public async Task<IActionResult> Add(Plant plant)
    {
```

```
        if (plant != null && ModelState.IsValid)
        {
        await _repository.SavePlant(plant);
        return await Index();
        }

        return View(plant);
    }


    public PartialViewResult PlantPartial(Repository.Models.Plant plant)
    {
    return PartialView("_plant_partial", plant);
    }

    public async Task<PartialViewResult> PlantsPartial()
    {
        return PartialView("_plants_partial",await _repository.UserPlants());
    }
}
```

Bibliography

[1] Rick Anderson. Introduction to identity on asp.net core. URL https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-2.2&tabs=visual-studio.

[2] Arduino. Introduction to arduino. URL https://www.arduino.cc/en/Guide/Introduction.

[3] BC Robotics. Using a tmp36 temperature sensor with arduino. URL https://www.bc-robotics.com/tutorials/using-a-tmp36-temperature-sensor-with-arduino/.

[4] Wikipedia. Observer pattern. URL https://en.wikipedia.org/wiki/Observer_pattern.

[5] yishengjin1413, mairaw, guardrex, tompratt AQ, Mikejo5000, and xaviex. Observer design pattern. URL https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern.