Victor:

Building Monitored, Secured, And Self-sustaining IOT Hydroponic Gardens

Benjamin Carothers

Submitted to the Division of Natural Sciences

New College of Florida

in partial fulfillment of the requirements for the degree Bachelor of Arts

Under the sponsorship of David Gillman

Division of Natural Sciences

New College of Florida

May 2017

Abstract

I developed Victor to help me grow vegetables by monitoring my garden. Victor is a framework composed of three separate services: `Container-gardening`, `Gardeners-log`, and `Victor`. Together these three elements gather measurements about garden's environment, store the data in a cloud hosted database, and display the measurements graphically on a web application.

In this thesis I first detail the components, both hardware and software, used throughout my project and how they fit into the overall design and architecture of the framework. I then discuss how Victor could be abstracted from the use-case of hydroponic gardens entirely and used in a wide variety of distributed data acquisition applications.

Though I do not believe that Victor solves the issues of large scale Internet of Things deployments unanimously, I argue that, for systems that use a microcomputer as the central operator of IoT deployments, Victor mitigates the security and extensibility issues present in devices today.

Acknowledgements

Mauris eget blandit nisi, faucibus imperdiet odio. Suspendisse blandit dolor sed tellus venenatis, venenatis fringilla turpis pretium. Donec pharetra arcu vitae euismod tincidunt. Morbi ut turpis volutpat, ultrices felis non, finibus justo. Proin convallis accumsan sem ac vulputate. Sed rhoncus ipsum eu urna placerat, sed rhoncus erat facilisis. Praesent vitae vestibulum dui. Proin interdum tellus ac velit varius, sed finibus turpis placerat.

Table of Contents

List of figures

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
JSON	**J**ava**S**cript **O**bject **N**otation
HTTP	**H**yper**T**ext**T**ransfer**P**rotocol
RPC	**R**emote**P**rocedure**C**all
REST	**Re**presentational **S**tate **T**ransfer
IOT	**I**nternet **O**f **T**things

Introduction

The global population is projected to see an increase of 2.3 billion people by 2050. If current trends hold, a majority of the 9 million will live in cities and only around 3 percent will have any direct role in the production of their food.[1]

Most land suitable for farming is already being used, so 90 percent of the necessary increase is expected to come from increased yield; however, annual yield increases have stagnated to only half of the historical average. Agricultural scientists are facing new and unique problems stemming from the techniques and practices that have helped feed and sustain the twentieth century's population boom.[2]

The primary agricultural ideology was, and to some extent still is, that increasingly large, monocrop farms are the most cost effective way to increase yield to feed a global population. Monoculture farming facilitates maximizing both profits and crop yield by minimizing the competition for nutrients between plants, reducing the amount of specialized machinery required to operate, and

[1] http://www.gartner.com/newsroom/id/3236718
[2] https://you.stonybrook.edu/environment/sustainable-vs-conventional-agriculture/

simplifying the processes of fertilizing and managing the health of crops.[3][4] However, the practice has some serious environmental and operational implications.

The reliance on monocrops led to farms filled with nutrient depleted soil. Increased use of pesticides, necessary to keep up with increasing demand, have damaged soil ecology even further due to the indiscriminate destruction of fungi and bacteria, and crop yield has largely plateaued.[5][6]

Though advances in farming technology and practice have managed to sustain a relatively massive population, the input costs are still drastic and unsustainable. Today 40% of land and 85% of water withdrawal is used solely to grow food.

Furthermore, the increased production of food tends not to reach those who need it most. In fact, it is likely that those who need the food most are those that are affected least by recent technological advances. The International Assessment of Agricultural Science and Technology for Development (IAASTD) has commented that "despite significant scientific and technological achievements in our ability to increase agricultural productivity, we have been less attentive to some of the unintended social and environmental consequences of our achievements." Beyond the wastefulness and unsustainable nature of industrial agriculture, the fact that the consolidation and monopolization of the food industry is contributing to the inequity of food dispersal is particularly

[3]http://www.newworldencyclopedia.org/entry/Industrial_agriculture

[4]http://www.ecifm.rdg.ac.uk/monoculture.htm

[5]http://www.networkworld.com/article/3135270/security/fridays-ddos-attack-came-from-100000-infected-devices.html

[6]https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2984095/

worrying.[7]

Agriculture has become much less accessible and little effort has been made to make small-scale, personal farming easier to engage in. The number of farms globally has shrunk dramatically – in the US alone the number of farms has halved over the last 50 years, decreasing by over 2 million.[8] Farmland has also become much less dispersed, and the average size of farms has increased greatly despite 85 percent of farms being less than 2 hectares.

With the looming global food crisis, environmental impact of rising global temperatures, and ever growing global population, how can we improve on the process of growing food? The necessity for higher yield farming accessible to many and the quickly approaching projected natural maximum capacity of industrial agriculture has led to a rapid shift toward sustainable agriculture. One Japanese company, Mirai, working with GE has spearheaded an experiment proposing indoor hydroponic gardens. After purchasing a retired 25,000 square foot Sony semi-conductor factory plant physiologist Shigeharu Shimamura began converting it into the world's largest indoor farm illuminated by LED. The constantly lit, environmentally controlled grow room immediately showed some of the benefits of the practice. Produce waste decreased from nearly 50 percent to just 10, productivity per square foot was nearly 100 times the average, and water usage was just 1 percent of what a conventional farm would consume. The success of their flagship factory has led to increased interest in creating factories all over the world, suggesting that hydroponics could be the future of agriculture.

[7] http://encyclopedia.uia.org/en/problem/140183
[8] http://www.sciencedirect.com/science/article/pii/S0305750X15002703

Hydroponics is a method of growing crops using only a nutrient solution administered directly to the roots of the plants being grown and an inert medium used to suspend the roots of the plants into the given solution. Crops grown hydroponically use significantly less water, yield many times more than their soil-grown counterparts, and can grow healthy, nutrient-rich vegetables despite inhospitable soil conditions or temperatures.

The promise of more sustainable produce has led to an increased adoption of hydroponics both commercially and by amateur gardeners all over the world; however, there are two important complications that have kept soilless gardens from being widely accessible.

First, there is a large learning curve associated with growing plants hydroponically. In comparison to traditional soil gardens, hydroponic gardens require strict monitoring of growing conditions, nutrient solutions, and root health. Furthermore, hydroponic gardens are prone to a collection of diseases and pests that are not commonly seen in soil gardens.

Second, hydroponic gardens are costly. The initial start up costs, cost of labor, and cost of upkeep are all significantly higher for hydroponic gardens. In commercial grow operations this cost is often offset by the greater yield, but the necessity of more higher-paid operators is a serious business consideration. For smaller, hobbyist gardeners looking to build a hydroponic garden in a community space or at home the increased cost can be incredibly prohibitive. Even if the garden's yield outweighs the cost of materials, the opportunity cost associated with constantly monitoring and servicing a hydroponic garden is often impossible to commit to.

These two glaring problems are not entirely avoidable. For one, the initial costs of materials and continuing cost of maintenance and upkeep will most likely always be more than a traditional soil garden. However, I believe that the benefits of hydroponic gardens are enticing enough that by lessening the burden of monitoring and upkeep the prospect of hydroponic gardens could become immediately more viable and likely less expensive as well.

Nutrient Film Technique (NFT) is a sub-genre of hydroponic gardens that utilizes a shallow film of nutrient enriched water running separate channels on a slight decline to deliver sustenance to plant's roots, which are suspended in a medium absent of any nutrients on the top of the aforementioned channels. NFT gardens provide an ideal environment in which plants are consistently exposed to water, oxygen, and nutrients.

NFT is considered an active hydroponic method because it relies on mechanical pumps to continuously deliver recirculated water and nutrients to the plant's roots. Other methods, like Deep Water Culture and Kratky, rely on capillary or wick systems. Though maintenance and monitoring of passive systems is generally much simpler, these systems generally subject their plant's roots to constant submersion. As a result, passive systems generally prohibit the flow of oxygen. Furthermore, unlike NFT, many hydroponic systems are not able to recover the unused nutrients.[9]

With a good setup, Nutrient Film Technique gardens will consistently produce high-quality vegetables at high yields. Certain crops, especially those that are not as reliant on their roots for structural support, can produce even higher

[9]http://www.doityourself.com/stry/hydroponic-systems-active-vs-passive

yields than other hydroponic methods. However, this comes at a significant cost. Because NFT gardens are so reliant on the constant flow of nutrients they also require constant monitoring. These gardens are susceptible to a handful of disastrous issues regarding flow that are capable of killing the plants affected within a matter of hours. In the case that the the pump supplying water from the reservoir to the channels were to lose power the plants would dry up and die within a few hours. Similarly because of the tight plumbing and inevitability of sediment being introduced into the reservoir a clogged line could kill all of the plants in a single channel or even an entire garden if it was the main line. A less dire, but still pressing complication occurs because as the plants consume the nutrients from the solution some of the water evaporates through the openings at the top of each channel. If the amount of water that sweats out of the system is greater than the consumption rate of the nutrients, then the plants are at risk of dying from being exposed to a pH imbalanced solution.

In an effort to explore methods of making hydroponic gardens, a high-yield, sustainable, space saving means of growing fresh produce, more available I built Victor – a collection of services used in conjunction to mitigate the amount of time spent taking measurements to monitor the growing conditions of your garden, provide you with real time alerts in the case that something seems to be going wrong, and give you an interface allowing you to manage some of the upkeep remotely. I believe this can easily both cut back on the total number of hours required to maintain a hydroponic garden and lower the barrier of entry by providing gardeners with baseline numbers to compare to and programmable warnings.

Requirements

As the primary user of this service, I approached development with a strict set of requirements.

First, there must be an attractive web-facing dashboard that is accessible by any and every user that knows where to find it. The dashboard should be extensible so that the numbers of gardens being displayed or connected is entirely up to the user.

Secondly, the connected gardens should, in real time, update the user on the status of the health indicators being monitored. The data should be transparently stored and accessed through a third party API, and while read access is completely open, any posted data to the API should be monitored and allowed only for credentialed users. In a secure manner, any user with the authorization to do so should have the ability to control the configuration of the garden and any number of configured manual actions.

Lastly, and most importantly, the product should exist as a fully extensible framework. Though I'm designing the project with a finite set of sensors

and parameters in mind, adding another sensor, choosing to use a different number of components, and monitoring a greater number of gardens should be expected and accounted for. The system for modification should be simple and documented.

These requirements ensured that I created a useful, secure product that was abstracted from the configuration of the gardens being monitored and the hardware configurations of each.

Summary of Chapters

In chapters 1 and 2 I will outline how I designed the application. In chapter 1, **Processes**, I will outline tools, technologies, and components of particular interest. A great deal of planning went into the construction of each particular service, and understanding the deliberate decisions of each should provide context of the overall scope of the project.

Chapter 2, **Implementation**, will then describe in detail how each item's functionality was built and show how they interact in order to display the role and extensibility of each.

In Chapter 3, **Security**, I'll briefly outline the threat model I used while attempting to control the flow of data.

Chapter 4, **Future Work**, I'll identify planned additions and how the extensi-

bility of the framework would allow Victor to adapt to a wide variety of fields and applications.

Chapter 1

Processes

1.1 Introduction

Two separate test gardens were monitored during the course of this project. The first is a mobile Nutrient Film Technique (NFT) hydroponic garden consisting of four channels each of which are four feet in length. The channels are mounted on a tabletop in a staggered configuration in which the two middle channels are a few inches above the outer ones to regulate temperature and sun exposure. (See Figure X.)

A 30 gallon reservoir is affixed on a shelf underneath the channels in which a small pond pump feeds water to the higher end of the channels.

The second garden was devised after a plumbing incident which stopped the

flow of nutrients and killed every plant in the four channels of the first garden. The second garden is also a NFT hydroponic garden, but it consists of a single channel. Its reservoir is a five gallon bucket and its stand is made of a PVC frame. I chose to scale the system down to a bare minimum in the hopes that it would be easier to manage during testing.

In both instances, on top of the reservoir sits a small water-tight enclosure that houses the electronics used to monitor the garden. The heart of the system is the Raspberry Pi – a credit card sized microcomputer. My garden incorporates three separate networked Raspberry Pis. Though only one is truly necessary, I chose to use multiple in order to emulate a large scale garden consisting of many separate table – each table would in this case have a separate Pi. These computers each control a handful of sensors constantly measuring health indicators, which they push to a cloud hosted database. The operator of the garden can manage each computer by accessing it directly or through a remote provisioning tool. To display the data collected by the garden's computer, or any collection of garden's computers, another cloud hosted site provides a dashboard with real time graphics and analytics that are generated from the most recent data stored in the database.

After the initial setup, by accessing the dashboard site we can gather a quick and comprehensive view of the current state of the garden.

1.2 Hardware

1.2.1 Sensors

1.2.1.1 Temperature

The build includes three separate sensors that gather measures of temperature – a sensor each for ambient, reservoir, and channel temperature. The ambient and reservoir temperatures are measured via a waterproofed DS18B20 sensor. The DS18B0 is reasonably accurate and IO efficient.

Using the 1-Wire protocol it's possible to gather separate measurements from multiple, identifiable devices using only a single pin of input.

This is incredibly beneficial because each of the garden computers can incorporate more devices without the need for a hardware extension that includes additional general-purpose input/output pins. Furthermore, each sensor can be identified programmatically, because each sensor has a unique serial number, which means that a single process can interface with all of the sensors.

One additional sensor is used to measure the temperature inside one of the top-side growing channels. Instead of the DS18B20 the internal channel temperature is measured using an AM2315. This sensor includes a waterproof housing that includes an identical DS18B20 temperature sensor, a capacitive humidity sensor, and a small microcontroller to provide a simple I2C commu-

nication interface. Though interfacing via I2C adds some complexity, having a measure of humidity inside the channels provides insight into the roots' growing conditions as well as propensity for harmful algae blooms.

It is assumed that monitoring of a single channel is sufficient because the channels exist in close proximity; however, barring monetary constraints additional sensors could be added very easily. Adding additional DS18B20's to each of the remaining channels for instance would only require mapping each of their serial ids to their respective locations.

1.2.1.2 Ph

Coupled with temperature, pH is a very descriptive health indicator for a hydroponic garden. I've chosen to use Atlas Scientific's EZO pH circuit and silver electrode pH probe. After the initial calibration this sensor is expected to provide accurate measurements of pH between 0 and 14 for two years. This circuit is capable of communicating asynchronously via serial with the universal asynchronous receiver/transmitter (UART) on the Raspberry Pi. Because communication is handled via serial, the sensor occupies the RX and TX pins on the RaspberryPi 2 rev b.

1.2.1.3 Flow

A flow sensor is attached to the tubing that runs from the pond pump located inside the main reservoir to the entry point of each of the four channels. Al-

though it's unlikely that each of the four entry tubes will be blocked at any given point, any blockage should impact the flow rate because of increased resistance. A sudden, or even gradual, decrease in flow should be a warning sign that urges garden maintainers to check for plumbing issues. The sensor includes a pinwheel with a magnet attached that turns as the water flows through it. A hall effect magnetic sensor on the other side of the plastic tube measures how many spins the pinwheel has made over a certain amount of time. This particular sensor requires only a single pin of digital input to read the pulse output.

1.2.1.4 Wind

An anemometer is included on the table top to gather the wind speed. This measurement is ancillary, but because NFT is a medium-less method of growing plants, windspeed can affect structural health. High wind speed may provide early warning for hazardous conditions allowing the garden's maintainer to provide some sort of cover. The anemometer included in this build provides an analog voltage between 0.4 and 2.0 volts corresponding to a 0 and 32.4m/s wind speed respectively.

The component is designed to interface with microcontrollers primarily. The required voltage is much higher than that which can be comfortably provided by a Raspberry Pi, so an external power source is required. Furthermore, Raspberry Pi's have no onboard analog to digital converter, meaning an external chip had to be included in the circuit.

In this build I decided to use an MCP3008. The Raspberry Pi, like most other modern microcomputers, is capable of reading 17 digital GPIO pins. As a 5V device, it's able to determine whether input is on (5V) or off (0V). This isn't especially useful when the device outputs voltages between those two values. The MCP3008 provides 8 channels of 10 bit conversion at serviceable precision. This allows for an explicitly digital device to interface with analog inputs like the anemometer.

1.2.1.5 Depth

Routine maintenance of the reservoir is largely unavoidable. Monitoring pH and temperature helps mitigate harmful algae growth and promotes plant growth, but bimonthly water cycling is one of the most efficient ways to maintain a healthy garden. A depth sensor is included in nutrient reservoir to help plan appropriate cycles. The rates of absorption and evaporation can differ depending on the temperature, state of the plumbing, and health of the plants. Once a predetermined amount of water has been expended it's likely a reasonable time to cycle the water. This build uses an eTape liquid level sensor that uses a resistive output that varies depending on immersion to determine depth. The resistive output of the sensor is inversely proportional to the height of the liquid because as more tape is submerged the sensor's resistance decreases. This sensor is also analog, so much like the anemometer it requires a single channel of the external analog to digital converter.

1.2.1.6 Light

A small, cheap visible light sensor was added to determine the amount light reaching the plants each day. Because the table is mobile, it's possible to determine the optimal position for the current crops by monitoring sun exposure. The SI1145 is a small breakout board that measures visible light and IR to approximate UV index. The sensor is digital and communicates via I2C.

1.3 Software

1.3.1 Angular

Angular is a Javascript framework developed and maintained by Google that is built around the idea that by extending HTML, web application developers can construct modular, declarative components that fit together to build dynamic web pages. Database operations are largely abstracted and the results of any AJAX calls can be bound to, and thus automatically update, the previously constructed, reusable DOM elements. Angular adds the ability to extend HTML and create Directives, which are encapsulations of HTML and client-side Javascript that allow developers to create and reuse custom elements. This leads to declarative markup, and means that reading the HTML alone is usually enough to get a quick idea of an application's purpose. Furthermore, by defining options that can be passed into directives, it's possible to create multiple similar components without an egregious reuse of code.

The web application, aptly called Victor, has two primary functions – gathering data and displaying data. All of the information is time series data, so displaying the garden's data via graphs is a very repeatable process. This is one way in which I leveraged Angular to create an extensible dashboard to consume garden data.

Loading the dashboard initiates a request to the API that grabs a list of the most recent database entries.

```
$http.get('https://aadrsu3hne.execute-api.us-east-1.amazonaws.com/dev/datum').
```

This function retrieves an array of entries asynchronously, so the the following `.then()` calls a function to manipulate the data once it is received.

The resulting array is a bag of unsorted measurements from each of the sensors, so I do a bit of manipulation before assigning it to a variable that is visible to the HTML Directives.


```
[{"parameter":"light","createdAt":1491252532082,"value":"352.000",
"id":"f1f4d520-18ae-11e7-b313-5d5b225fc2b6","updatedAt":1491252532082},
{"parameter":"pressure","createdAt":1491252725837,"value":"1013.061",
"id":"65717fd0-18af-11e7-b313-5d5b225fc2b6","updatedAt":1491252725837},
{"parameter":"temperature","createdAt":1491254237495,"value":"29.723",
"id":"ea763470-18b2-11e7-b313-5d5b225fc2b6","updatedAt":1491254237495},
{"parameter":"pressure","createdAt":1491253037765,"value":"1013.145",
"id":"1f5def50-18b0-11e7-b313-5d5b225fc2b6","updatedAt":1491253037765},
{"parameter":"pressure","createdAt":1491253204313,"value":"101477.794",
```

”id”:”82a32490-18b0-11e7-b313-5d5b225fc2b6”,”updatedAt”:1491253204313},
{”parameter”:”temperature”,”createdAt”:1491253022715,”value”:”30.408”,
”id”:”16657cb0-18b0-11e7-b313-5d5b225fc2b6”,”updatedAt”:1491253022715},
{”parameter”:”depth”,”createdAt”:1491253267173,”value”:”2.8”,
”id”:”a81ad150-18b0-11e7-b313-5d5b225fc2b6”,”updatedAt”:1491253267173},
{”parameter”:”Luminosity”,”createdAt”:1491253948475,”value”:”2”,
”id”:”3e3140b0-18b2-11e7-b313-5d5b225fc2b6”,”updatedAt”:1491253948475},
{”parameter”:”depth”,”createdAt”:1491252841556,”value”:”2.6”,
”id”:”aa6ad140-18af-11e7-b313-5d5b225fc2b6”,”updatedAt”:1491252841556},

Each of these entries is transformed into key value pairs of the form `{”value”: XXX, ”date”: XXX}` and collected into separate arrays for each measurement. Now this collection of parameters and their massaged values are assigned to a $Scope variable, another Angular construct that makes the data visible and modifiable via the HTML, so that I’m able to cycle through and create a data dashboard programmatically.

```
<div class=”dash-page”>
 <div class=”dash col-md-12” ng-controller=”DashboardController as ctrl”>
  <div class=”row” ng-repeat=’row in ctrl.charts | groupBy:2’>
    <div ng-repeat=”item in row” class=”col-md-6”>
      <chart data=”item.data”
        options=”item”
        ng-if=”item.data”
        convert-date-field=”{{date}}”>
      </chart>
    </div>
```

```
  </div>
 </div>
</div>
```

This small snippet of code is able to create a full dashboard of approximately eight time series graphs featuring real-time, updatable data.

`ng-repeat='row in ctrl.charts | groupBy:2` cycles through every element in the previously created object of measurement arrays by groups of two. Then, while looping through each group of two, I'm able to create a row of two charts by using the `<chart>` directive to render `metrics-graphics` time series charts. Under the hood, the `<chart>` directive is constructing an object consisting of the necessary Javascript and HTML to create and render a chart with the given parameters, but this is abstracted to make the HTML page's components encapsulated and expressive. Modification of the data exposed by `item.data` through the web page is reflected in the Javascript and updates to the data in the Javascript are made visible immediately by the HTML, which is called two-way data binding.

Though this is a small example of the benefit provided by using Angular as the underlying structure of the application, it demonstrates two of the key mechanisms that made Angular an appealing option.

The web framework space is heavily flooded and also incredibly opinionated. My main criteria were that I wanted a structure that was both pluggable and maintainable, meaning I wanted every file to encapsulate only one module of logic, I needed the act of updating previously stored data to be transparent

and fast, and I wanted a framework that was well maintained.

During my research I was immediately confident that Angular would provide a stable base considering the fact that it's backed by Google and has over 50,000 stars on GitHub; however, Angular's directives and two-way data binding are what drove me to ultimately believe Angular would work well within the structure I had devised for Victor.

1.3.2 REST

Victor is a built using the basic principles of service oriented architecture, which is a style of software design where self-contained modules of functionality are composed and interconnected. The framework consists of a composition of three services, namely Victor, Gardners-log, and Container-gardening, each of which have a single function.

In the previous section I showed how through a top-level, front-end service, Angular helps load and display all of the data collected from the garden. However, in order to view and manipulate data, the front-end service needs to gather information in a parseable format. Historically, the dashboard and garden would have been heavily coupled. As hardware components collected measurements they would write them to a relational or flat file database, which the dashboard would then use to gather and display information. Also, any commands to be executed by user input would likely entail direct access of the data collection process.

This monolithic architectural style presented a few problems. In order to access the data you need to be able to access the machine running the program, know the format of data storage, and have a direct connection to the data. Furthermore, when any portion breaks or is scheduled to be updated or modified then then entire application goes down. Lastly, gaining access through the use of any vulnerability present in the application opens the door to any and all other services involved, which means that if you find a client-side vulnerability in the user-facing dashboard then you likely have full access to the data collection process.

Each of these issues is serious, but by implementing a service based application I was able to mitigate some of these operational issues.

The central, facilitating service of Victor's composition is the service called `Gardners-log`. This service consists of a cloud hosted database, a list of functions that read and update that database, and listener service that sits waiting for users requests to initiate one of those functions. `Gardeners-log` provides a RESTful API that the other services can interact with to create and consume data. An API, or Application Programming Interface, is a list of functions and a designation of how they can be executed and with what parameters. What makes an API RESTful is that a RESTful API provides interoperability between web based services by allowing requesting services to access and manipulate textual representations of data using a uniform and predefined set of stateless HTTP based operations.

In terms of technology, this means a database hosted in the cloud sits behind a small bit of marshalling code that reads and writes JSON represented garden

measurements based on standard HTTP requests it receives. For instance, if the API receives a POST request a function is initiated that grabs the data in the body of the HTTP POST and writes that data to the database. If it receives a GET request a function queries the database records and responds with a textual representation of what it found.

This service is incredibly valuable because it makes Victor much more robust, secure, and accessible. For instance, the data coming from my garden is valuable and should be made available to any number of clients to read. In the future I may want to create different an application that reads data from gardens that exist all over the country and compare their yield based on weather and other health indicators. It could also be the case that I want to create a native application that can send updates straight to my desktop. By creating an API using standard RESTful practices the clients that intend to read in my data don't need any previous knowledge aside from the URL of where it resides to access it. All modern, interconnected devices speak HTTP, meaning clients can be entirely heterogenous.

Furthermore, I want it to simple to be able to digest the data in a standardized way so that future applications are able to easily manipulate it. Data is sent using JSON. The entire service is textual, so transmission is efficient and cheap. Parsing though the raw data is not computationally expensive, and because JSON is a standard form of representation, constructing data structures in any native language should be straight forward.

1.3.2.1 Security Implications

Because the API is a stand-alone service it is important to consider access controls and the flow of control. Anyone with the resources and interest should have access to view the data, but it should be secure and protect against unauthorized writes. For this reason, GET requests are largely left unattended. Aside, from overwhelming the server, these types of requests are not much of a threat because the marshalling code does not rely on any input from the user. Any POSTs or RPC style calls need to be credentialed access.

The design consideration was that the database would need some means of keeping track of users that were authorized. This could be handled simply enough by constructing a table of users and hashed passwords, but after logging in a user should not immediately need to authenticate to issue another request. Instead I opted to implement a token style authentication. Completely transparent to the user, on login the dashboard checks in with the API in an attempt to authorize. In the case that authentication is successful, the API issues a token that is valid for a customizable amount of time. This way the system is still secure, but not at the cost of the user experience.

Taken one step further, the dashboard and API allow for the control of a few features that could be very destructive to the connected gardens. For these functions two factor-authentication is required.

I also felt it was necessary that the data should stand alone and act as the single point of contact that connects the garden network to the greater internet. As the most valuable resource, the data should be siloed and kept segmented from

any other means of control. Hosting the API on a virtual machine provider allows for firewalls, load balancing, and monitoring that I wouldn't have access to on my own. Furthermore, by only accepting controls from this single machine, I can easily dismiss a large portion of invalid commands issued to garden machines.

1.3.3 Python

Victor as a framework is meant to be explicitly language agnostic. Data is transferred using very standard RESTful web services meaning the lone requirement for a program to successfully submit data to the API is that it must be able to construct a valid HTTP request. Though this leaves a vast number of capable options I gravitated strongly toward Python for code that is directly interfacing with the sensors.

Python facilitates really rapid development, many of the sensors in this build have far reaching community support and open-source libraries written in Python, and lastly RPi.GPIO is shipped with Raspian, the RaspberryPi foundations Linux distribution. This module provides a direct interface to the boards General Purpose Input/Output pins. Each and every external component interfaces through these pins, so opting to use Python almost exclusively helped keep provisioning and deployment as simple as possible.

1.4 Docker

Every sensor is controlled by a single, separate Python script. Most of the scripts share at least some code including a collection of utility methods for timing and reporting findings I wrote, however, they each have very separate dependencies. Some require specific hardware-level systems packages, whereas others might only need a tagged version of a Python library hosted on Github. It's expected that every component has a unique set of dependencies and it's assumed that many of these will conflict. Furthermore, the scripts are also separate in their access roles. A single process is expected to be able to communicate outside of my home network, and all of the sensor's programs are expected to be able to communicate amongst each other. Likewise, only a single process, the same program that is able to send messages outside of the home network, should be reachable via the internet. That same program should be able to send messages to the other sensor running scripts.

I chose Docker as the primary tool to handle sensor's code deployment, separation, and management. Docker is a technology and framework built around developing, building, and deploying applications inside of software containers. Containers are a virtualization technique in which and application and its dependencies are packaged in isolated processed. Like standard virtual machines, Docker containers ensure that I can customize and specify all of the dependencies at the user level. This solves the dependency collision issues because each container has its own unique and unshared user space. Containers are initialized and provisioned via a scripted Dockerfile, which means that environments are consistent and shareable. Similarly, Docker containers, like the code that they host, work well with version control.

However, unlike virtual machines, containers are very lightweight. Once built, a process that generally takes a few minutes depending on internet speed, a container generally starts within seconds. Whereas, each virtual machine is a full operating system with both memory management and virtual devices, Docker containers attempt to save space and resources by sharing a kernel and systems level libraries. This cuts overhead significantly, but decreases the true separation. Containers don't have a full systems level separation like virtual machines, which leaves them somewhat susceptible to breakout attacks. However, Docker containers by default are comfortably secure, and with proper configuration – namely, not running available processes as root – the risk of containers is much, much lower than the risk of stand alone processes.

Beyond the security and isolation benefits, containers make the victor framework incredibly scalable. Adding a sensor to the deployment takes three steps – two of which can be largely automated. The first step is to write the code to interface with the sensor. The file can reside anywhere, but the organization is standardized by storing the file in a directory named after the parameter it Measures. To run in it's own separate container the sensor's code needs a defined Dockerfile. The Dockerfiles used in this project are very similar, so much of the declaration can be reused. Dependencies unique to the sensor's code need to be defined. Lastly, an entry is added to the a DockerCompose YAML file. A Docker Compose file is declared for each deployment. Each machine can host a different configuration of sensors. To keep builds customizable the Docker Compose file defines build instructions, runtime parameters, and names for each sensor container to be run on any given machine. Once defined, start up on any give deployment is as simple as docker-compose up. The configuration in the compose file handles all networking, storage, and environment

management.

Figure 1.1: First Garden

Figure 1.2: Second Garden

Figure 1.3: Inside

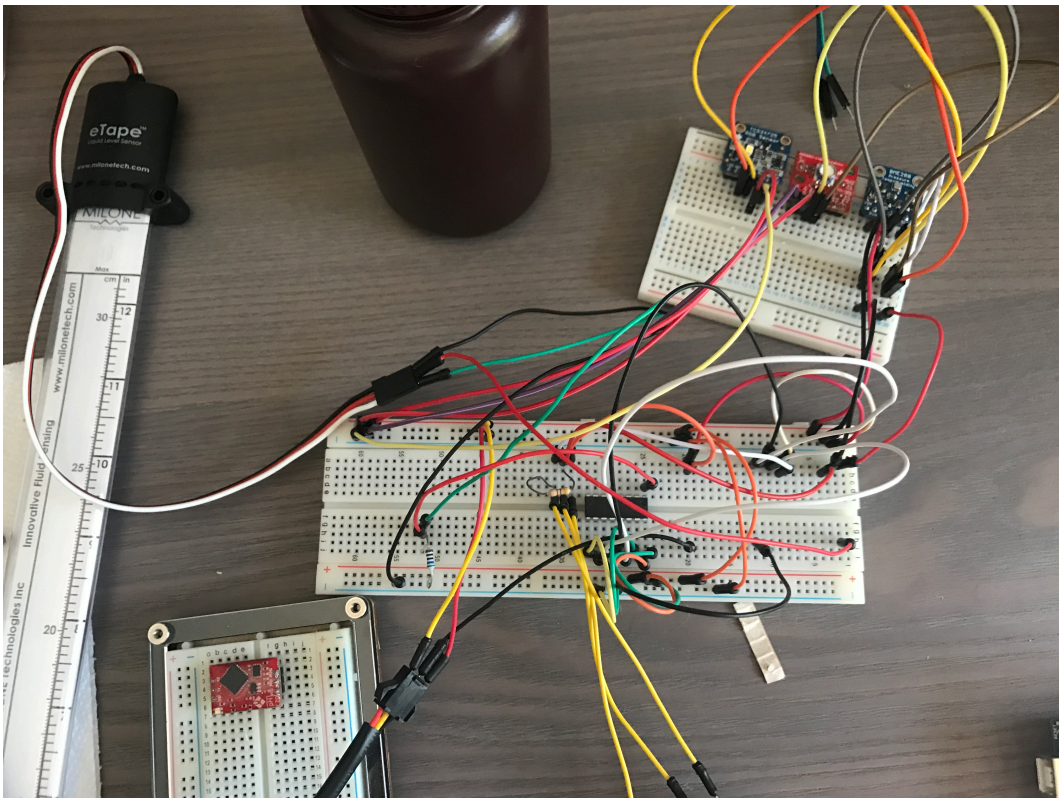


Figure 1.4: Circuit

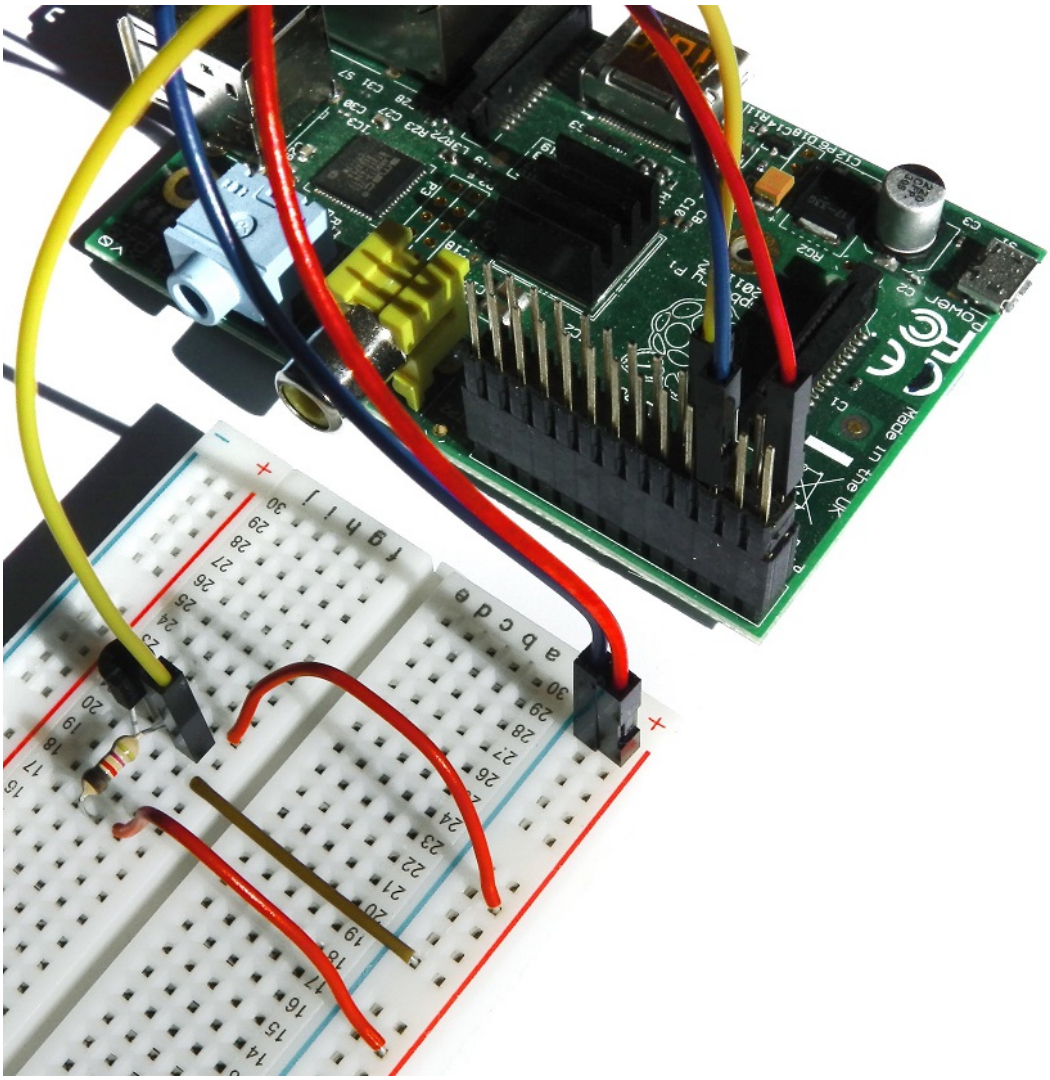


Figure 1.5: The yellow jumper cable in this image is the only wire attached to any input pin. Each other wire deals with supplying power.

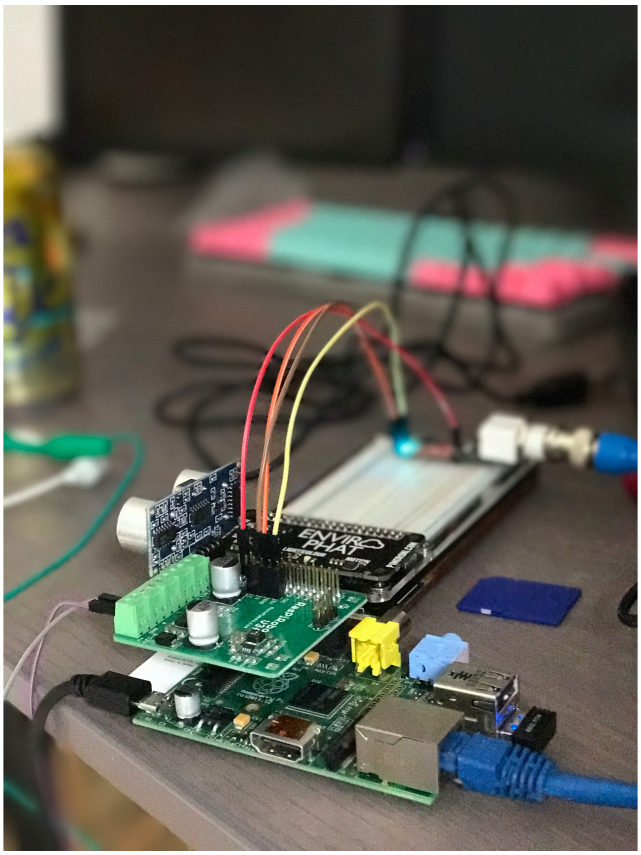


Figure 1.6: pH Circuit

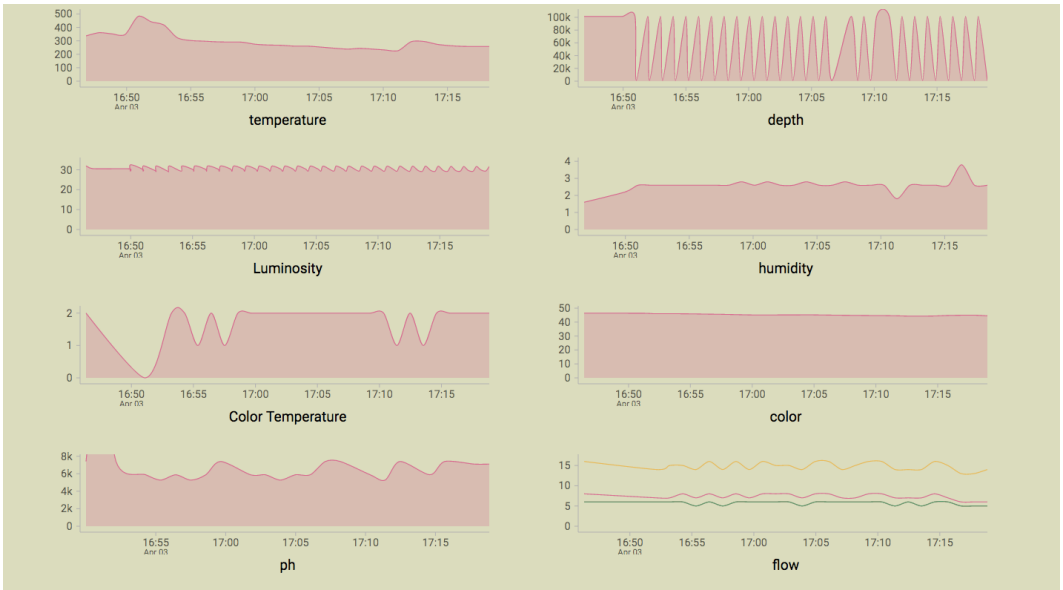


Figure 1.7

Chapter 2

Implementation

2.1 Introduction

In this chapter I will first outline the flow of information through Victor's services in it's most general use case. Then, working from the bottom up, I'll detail the implementation process including how working parts of the services are tied together and how each of the three services, the garden, the api, and the dashboard, interact.

2.2 Flow Control

Victor uses a role based access control system in which only two types of users are considered to exist. The vast majority of users that might access the ap-

plication are uncredentialed, outside users. This group makes up person who wishes to view the data, but has not been given access to make any changes. These users will almost always be accessing the site remotely, meaning requests to the dashboard will not come from the same network that the garden is connected to. The other group is much smaller and is made up of garden maintainers who want to and have the access to make changes to the garden remotely. These credentialed admin users have access to all of the same functions as the outside users; however, once their access has been properly vetted, they gain the additional power of manipulating data via PUT and DELETE requests to the API and issuing commands to be evaluated by the garden machines.

As a outside user, the benefits made available by Victor are in consuming data about the garden's environment. Outside user's can use this a point of reference or comparison for their own gardens, so they want an easy way to consume recent relevant data. To do this the user must access the URL of the dashboard. Though no means of collaboration is included in the framework, access to the dashboard is made available through the projects Github repository. Upon accessing the main page, the dashboard service immediately realizes that the user's request is unauthorized, so a call to the API is necessary to identify roles. The The API seeing a request coming from the user, knows that read requests from an uncredentialed user are allowed, so it gathers the most recent set of data from the garden and sends it back as a JSON object to the dashboard. Once the dashboard gets the data it cleans it and constructs the appropriate graphs and widgets. Knowing that more relevant data is saved and stored with the API service every few minutes, the dashboard makes sure the check back in periodically to maintain relevance.

An outside user has no direct communication with the garden network itself, but rather the data that is sent from the garden's sensors is summoned and displayed by the dashboard on request. Communication between the garden and the API occurs only through one head device. The garden can be constructed using any number of machines, but communication is limited to being sent and received by a single elected proxy. Each machine that is a member of the garden's network hosts sensors contained by Docker containers. These contained processes continuously, generally based on a time increment though there is no enforced standard, take measurements about the garden environment and relay them to garden's head communicator device. The communicator maintains a service for relaying data points to the API to be stored. Data points are sent in the same manner regardless of which sensor they came from. This is a implementation decision meant to allow for the addition of any number of arbitrary sensors and controls. Doing so made the load a bit heavier for the dashboard service because data needs to be sorted and accounted for, but because there are large extensibility gains and the data is purely textual and relatively light weight it seems to be worth it.

As a credentialed user this is still the most commonly used flow of data. Sensor data is sent to the API which is sent to dashboard after being asked for by the user. The control is augmented, however, by being able to send messages through the api directly to the garden. As an admin user after accessing the garden's dashboard page I have the option of logging into the service. Logging in requires only a username and password. Submitting credentials sends a message to the api, which handles identification and authorization. If the user exists and the password is correct a token is sent back to the dashboard and access is granted. credentialed access allows the user to view another

page that facilitates additional control interfaces. For instance, if the user wants to turn on the water pump manually for some amount of time they can elect to send a command to the API that is then relayed to garden's communicator and executed. Sending this message requires only two things aside from using the dashboards UI to send the proper message. First the token value that was received previously after signing in must match one of those that is currently valid and dispatched. Secondly, to prevent sniffing token values, the API must also be sent a One Time Password from an authorized two factor authentication key. Ultimately this means, to control the garden you need both a username and password as well as a physical device like a Yubikey to command the garden components. Once the API verifies that your request is valid it then sends the appropriate request to the communicator, which is then parsed and dispatched to the proper process.

These two roles and the parties that are allowed to communicate maintain a very robust and secure service despite the number of moving parts. In the following sections I'll identify how the tools outlined in the previous chapter were used to build the components of Victor's ecosystem and how they can be customized to remain relevant and deliberate for any garden configuration.

2.2.1 Container-Gardening

Container-Gardening is the name I've given to the repository for all of the resources and processes to be run on the microcomputer that lives physically connected to the garden.

In the case of my specific test build I've chosen to use three separate boards, each of which are produced and sold by RaspberryPi. The three microcomputers are the RaspberryPi 3, RaspberryPi Zero, and RaspberryPi 2 rev b.

Each board has drastically different cost, IO limitations, and power requirements. For instance, the RaspberryPi Zero, a $5 microcomputer, has a power rating of 160ma and a 1 Ghz single core processor, whereas the RaspberryPi model 3 costs $37.95 for a 1.2 Ghz quad-core processor and an 800ma power rating.

Though the total power usage of the build and the computational power required to host the required sensors will ultimately impact the choice in platform – there are a small set of requirements. First the garden's computer needs to run some variant of linux in order to leverage the Docker ecosystem. I've chosen Ubuntu ARM, but any modern Linux OS would reliably and comfortably facilitate using Docker. The chosen board also needs a GPIO interface. A strong majority of sensors sold today interface over GPIO. I2C and SPI communication protocols are relatively efficient, so the total number of external components per board can be much greater than alternatives using input like USB. Lastly, the board needs some means of networking. At least one board needs access to the internet, but all need some way to communicate whether over the wire, HTTP, or some other short range mesh networking.

Every sensor thats attached to a garden's computer is liable to interface in a different way; however, in order to exist in the context of Victor they're all handled in a very similar way. First, every component needs to establish a physical hardware connection with their respective board. In most cases

this entail three or four wired connections – two of which provide power and ground.

Assuming the connection is successful, the sensor needs a respective program to intermittently collect a measurement and forward it on. Each sensor has it's own directory in the container-gardening repository. Sensors are categorized by parameter, and subdivided if two similar measurements are being taken. For instance the directory for the DS18B20 temperature sensor can be found at `container-gardening/temperature/1wire`. There is more than one temperature sensor being used in my build so I chose to designate this particular one by the protocol it uses. This directory houses the code that schedules measurement and any additionally required libraries. Whereas many sensors have provided open-source libraries, some sensors require low-level calibration and communication code. The flow for measurement scheduling is loosely standardized.

A job function is designated to be run on a predetermined interval. The job in most cases has three responsibilities. For the components it's responsible for, the job gathers measurements, formats the data in API digestible messages, and sends the readings to the `Gardener`, the container tasked with handling communication to and from the garden.

```
import w1thermsensor
import schedule
import requests
import json
import time
```

```
import os

def job():
    for sensor in w1thermsensor.W1ThermSensor.get_available_sensors():
        send_data(sensor.get_temperature())

def send_data(temperature):
    entry = json.dumps({'parameter': 'temperature', 'value': str(temperature)}
    url = (os.environ['API'] + '/dev/datum')
    requests.post(url, data=entry)

schedule.every(15).minutes.do(job)

while True:
    schedule.run_pending()
    time.sleep(1)
```

In this example, we're gathering temperature data from the DS18B20. This particular sensor has an API, which we include using `import w1thermsensor`. This sensors job() function is simply iterate through every DS18B20 attached – there are multiple because this sensor interfaces using the 1wire protocol – package each measurement into a simple JSOn message, and send it along. The jobs initialization is handled by a library called `schedule`, which is a single-process program level implementation of chron. Using schedule we can determine how frequently to gather measurement.

Once we can interface with our sensor programmatically on a timed interval,

it's time to define a Dockerfile to containerize the sensor in question. In doing so, we isolate the sensor's dependencies, define a local network by which the containers can communicate, and prevent unwanted access between processes. Each Dockerfile, which defines the process of building the sensor's environment, is defined a directory in the `container-gardening` repository called `dockerfiles`. The directory mirrors the sensor's organization exactly for the sake of simplicity.

This is the Dockerfile that runs and manages the code from the previous example (`container-gardening/dockerfiles/temperature/1wire/Dockerfile`):

```
FROM resin/rpi-raspbian:jessie

RUN apt-get update && apt-get install -y \
    git-core \
    build-essential \
    gcc \
    python \
    python-dev \
    python-pip \
    python-virtualenv \
    --no-install-recommends && \
    rm -rf /var/lib/apt/lists/*

RUN pip install w1thermsensor
RUN pip install schedule
```

```
RUN git clone https://github.com/btcrs/container-gardening.git /data/container

# Define working directory
WORKDIR /data
VOLUME /data

CMD ["python", "/data/container-gardening/temperature/1wire/simple_temperature
```

Every Dockerfile begins with a `FROM` directive. This defines the base image that our image will ultimately be built up from. I've chosen `resin/rpi-raspbian:jessie` for the majority of my RaspberryPi containers. This is a bare-bones version of Raspian, which is a Debian based Linux OS optimized for the Raspberry Pi hardware, it has most of the tools I've needed and for the most part has just worked.

The next three RUN directives define all of the OS and program dependencies necessary to run the program. Here I define which `apt-get install` and `pip install` commands need to be run to provision the container.

The next command clones the entire container-gardening repository to the container. This step is somewhat wasteful because it's very redundant, but I opted to grab the entire repository for simplicity. Some of the sensors have multi-file directories and includes, and I've abstracted some utility functions to their own module. If the number of sensor declarations and resources dramatically hindered storage and build time it might make sense to define the container-gardening repository as a collection of sensor submodules.

The `VOLUME` directive defines a directory to be made accessible from the host machine and `WORKDIR` sets the context from which the following `CMD` command should be runs

Lastly, the build process executes the defined `CMD` command which is programmed to run indefinitely.

At this point we could build the container manually and run it, which would begin to show sensor output and send newly acquired entries. One further step is required to include this newly defined container into the Victor framework.

Each machine that exists as a part of the garden's configuration has what's called a `docker-compose.yml` file. Docker-compose, which is one of the tools provided in the Docker ecosystem that's used to define and run multi-container applications. It's assumed that different machines in the garden's configuration will have unique sets of sensors, sol docker-compose is used to define which collection programs we want to run on any given machine.

The DS18B20 is hosted on the RaspberryPi 3 in my build along with a handful of other sensors. The docker-compose.yml file used for the RaspberryPi 3 provides a canonical name, build instructions, and run parameters for each of theses sensors, so that they can be run in conjunction using a single command.

```
version: '2'
services:
  temperature:
    build: ./dockerfiles/temperature/1wire/
```

```
    privileged: true
    devices:
     - /dev/ttyAMA0:/dev/ttyAMA0
     - /dev/mem:/dev/mem
    volumes:
     - /data
  uv:
    build: ./dockerfiles/uv/lux/
    privileged: true
    devices:
     - /dev/ttyAMA0:/dev/ttyAMA0
     - /dev/mem:/dev/mem
    volumes:
     - /data
  flow:
    build: ./dockerfiles/flow/
    privileged: true
    devices:
     - /dev/ttyAMA0:/dev/ttyAMA0
     - /dev/mem:/dev/mem
    volumes:
     - /data
  multi:
    build: ./dockerfiles/multi/
    privileged: true
    devices:
     - /dev/ttyAMA0:/dev/ttyAMA0
```

```
    - /dev/mem:/dev/mem
  volumes:
    - /data
depth:
  build: ./dockerfiles/depth/
  privileged: true
  devices:
    - /dev/ttyAMA0:/dev/ttyAMA0
    - /dev/mem:/dev/mem
  volumes:
    - /data
depth:
  build: ./dockerfiles/pressure/5803
  privileged: true
  devices:
    - /dev/ttyAMA0:/dev/ttyAMA0
    - /dev/mem:/dev/mem
  volumes:
    - /data
```

Each of service defined under the services tag defines a separate docker container. In this case the requirements are all very similar. The `temperature` tag defines the name to be given to the container. `build` delineates where to find the dockerfile associated with this container. `privileged` defines that the contianer may need sudo access to run appropriately. `volumes` explicitly states any of the volumes, which may be defined in the dockerfile. One of the most important declarations is `devices` which specifies a runtime parameter

allowing the container to directly interface with the hardware.

The incredibly powerful implication of this tool is that a single command `docker-compose up` from within the proper directory builds every defined container and starts collecting data rapidly in a secure, manageable way. Furthermore, the containers are ephemeral, so changes can be deployed with ease and downtime can be strongly mitigated because of the speed and ease of start up. Lastly, though each sensor requires a few important pieces, this architecture keeps the deployment of a system nearly identical regardless of the complexity and size of the configuration, which is a massive gain for extensibility.

2.2.2 Gardeners-Log

At this point each of the sensors is capable of measuring and reporting data, but we haven't defined a location where the data should be sent. `Gardeners-Log` is the service that handles the manipulation of data, authentication, and communication between the dashboard and `container-gardening` level processes. At its core `Gardeners-Log` is a simple RESTful API; however, the architecture on which it's built and the security implementations associated allow for a scalable, protected channel of information transfer.

Victor uses a serverless architecture for it's API because it hands off the responsibility of managing, scaling, and provisioning servers to the serverless provider. We're already maintaining a semi-complex deployment of IOT devices, so minimizing the complexity and cost of the other services helped keep

everything as manageable as possible. Victor's API is currently hosted on AWS Lambda, Amazon's serverless platform. Architecturally, the API is made up of a composition of a few of Amazon's cloud services interacting with Lambda's compute platform. The entry point for any application wishing to interact with garden data is an Amazon API Gateway. This gateway is an HTTP listener configured to initiate a certain function correlated to the path used to access it. The gateway also provides an interface for using custom authorizer functions and API key authorization. Behind the API gateway sits some number of Lambda functions. Functions are stand alone module exports exposing a single method. Though lambda supports a few languages, I chose to write my functions in Node. The functions used for Victor's API define CRUD operations for the Garden's database. These functions interact with the last Amazon cloud service involved in this architecture, DynamoDb. The data is light and relatively consistent in shape, so most storage implementations would be suitable, but DynamoDb was an easy choice because it is managed, pretty simple, and exists within the Amazon ecosystem. Though the API gateway is constantly listening for new requests, the Lambda functions spin up on demand and are metered by ticks of 100ms. Unlike a standard always on servers the runtime costs should be very minor for a REST API.

As mentioned in the section detailing Flow Control there are two primary users, credentialed and uncredentialed. However, another communicating party is the `Gardener` container hosted on one of the Garden's computers. Unlike, the aforementioned user types, the `Gardener` isn't able to navigate through a multi-step authentication process. The data being sent to the API is the most valuable resource provided by Victor, so naturally I wanted to put some measures in place to preserve data integrity. The API Gateway service has a

concept of protected paths with which I implemented relatively simple, secret based authentication. Through AWS management I created an API key for each of the garden's microcomputers. In the configuration of the POST function I designated the function as protected and allowed any of the three keys to authenticate. In the docker-compose configuration I define the respective keys as environmental variables which are accessed by the sensors code. In doing so, I have a log of which machine accessed the POST method of the API gateway and the data that it submitted.

Credentialed users, however, may need to update or delete data. I wasn't comfortable with secret based authentication for admin level operations, so I exposed another set of Lambda functions on the separate API gateway to be used as authorizer function. When a user requests to authorize to the API they're redirected based on a chosen provider to either Facebook's or Google's authentication page. The provider asks the user if they authorize Victor read access of their account. If the user allows authorization of Victor then the provider sends an authorization code to the API. With this code the API sends an authorization request back to the provider and in turn receives an access token.

This flow represents standard Oauth authentication. The authorizer function of both the PUT and DELETE entry points of the API's gateway is set to the authorization function of the of my second Lambda based authentication service. This means that for any PUT or DELETE request sent to the API must also pass the authorization function.

After a user authenticates they receive an access token that is in turn sent with

every subsequent request. This token is passed to the authorizer. The authorized constructs and sends a request to the correct provider with the access key. If the access token is valid, the API will process the request according to its API specifications. If the access token is expired or otherwise invalid, the API will return an "invalid_request" error. If the Authorizer passes successfully then the originally requested operation is performed otherwise an error message is returned.

The resulting API is simple, but exposes all necessary operations in a very deliberate manner. Creation of data entries is handled by POST requests to `/dev/datum` and authorized via the API keys. Manipulation of data is handled by PUT and DELETE requests to `/dev/datum/{id}` and authorized via the separate serverless Oauth service. Obtaining a list of every entry is handled by a get request to `/dev/datum/` and getting a single entry by a GET request to `/dev/datum/{id}`. Neither of the read requests are credentialed so that unauthorized users are still able to view the garden's data. In the following subsection I'll outline the construction of one of the serverless function.

2.2.2.1 Serverless

All of AWS Lambda development was done using a framework aptly named `Serverless`. Each collection of functions is called a service. Every service is contained within its own directory and defined by a file name `serverless.yml`.

The first portion of this file details AWS configuration metadata.

```
service: victors-api

frameworkVersion: ">=1.1.0 <2.0.0"

provider:
  name: aws
  runtime: nodejs4.3
  environment:
    DYNAMODB_TABLE: ${self:service}-${opt:stage, self:provider.stage}
  iamRoleStatements:
    - Effect: Allow
      Action:
        - dynamodb:Query
        - dynamodb:Scan
        - dynamodb:GetItem
        - dynamodb:PutItem
        - dynamodb:UpdateItem
        - dynamodb:DeleteItem
      Resource: "arn:aws:dynamodb:${opt:region, self:provider.region}:*:table/
  apiKeys:
   - DataLogger
```

This snippet defines the language the functions are written in, environmental variables and api keys, and identity management configuration for the defined resources.

The next section provides a declaration for each function contained in the

service.

```
functions:
  create:
    handler: datum/create.create
    events:
      - http:
          path: datum
          method: post
          private: true
          cors: true
          integration: lambda
```

Here I define the create function, whcih is located at the path `datum/create`. The `events` tag defines the API gateway path that will trigger this function. `integration`, `path`, and `method` declare that it will be triggered by a POST to an API gateway at the path datum. `private` indicates the request will require a key defined in the previous snippet called `DataLogger`. `cors` allows the request to be called by another resource rather than directly by a user.

Finally, a DynamoDB instance is defined in the resources section declaring the database and table that the functions will interact with:

```
resources:
  Resources:
    DataDynamoDbTable:
      Type: 'AWS::DynamoDB::Table'
```

```
      DeletionPolicy: Retain
      Properties:
        AttributeDefinitions:
          -
            AttributeName: id
            AttributeType: S
        KeySchema:
          -
            AttributeName: id
            KeyType: HASH
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1
        TableName: ${self:provider.environment.DYNAMODB_TABLE}
```

The function itself is a small, but not by necessity, Node file.

```
const uuid = require('uuid');
const AWS = require('aws-sdk');

const dynamoDb = new AWS.DynamoDB.DocumentClient();

module.exports.create = (event, context, callback) => {
  const timestamp = new Date().getTime();
  const data = JSON.parse(event.body);
  if (typeof data.value !== 'string' || typeof data.parameter !== 'string') {
    console.error('Validation Failed');
```

```
  callback(new Error('Couldn\'t create the data value.'));
  return;
}

const params = {
  TableName: process.env.DYNAMODB_TABLE,
  Item: {
    id: uuid.v1(),
    parameter: data.parameter,
    value: data.value,
    createdAt: timestamp,
    updatedAt: timestamp,
  },
};

dynamoDb.put(params, (error, result) => {
  if (error) {
    console.error(error);
    callback(new Error('Couldn\'t create the data entry.'));
    return;
  }

  const response = {
    statusCode: 200,
    headers: {
      "Access-Control-Allow-Origin" : "*" // Required for CORS support to wo
    },
```

```
      body: JSON.stringify(result.Item),
    };
    callback(null, response);
  });
};
```

The file defines a single function that takes the parameters event, context, and callback. These define event data, runtime information of the Lambda function, and a function used to return information to the caller.

This particular function takes the data posted in the event's request body and the current time and then constructs an object to be stored in the dynamoDB database defined in the resources section. The function passes a 200 response back to the caller as long as the database call did not error out.

Once every other function is defined `serverless deploy` translates `serverless.yml` to a single AWS CloudFormation template, zips the functions, and publishes a new version for each function in the service.

In line with the emphasis on extensibility this architecture allows for a modular, secure, and highly modifiable API that implements and exposes every necessary action safely.

2.2.3 Victor

With `container-gardening` consistently gathering sensor data and `gardeners-log` providing a means of both storage and access, a single service stands in the way Victor being a transparent and pleasant user experience. Simply named `victor`, the framework's front end is the view of the application in totality to most users. Though `victor` the web application, is the simplest component of victor, the framework, the front end is responsible for masking the fact that there exists any separation of duties among services while also offering a beautiful and useful interface.

I chose to build this application using Angular, and I've hosted it using GitHub pages. All of the data garden's data is accessible via a single HTTP call, so the application is composed of entirely static resources. This proved incredibly useful because Github pages, though limited to only static pages, is hosted completely for free. Furthermore, any changes made to the page, on the `gh-pages` branch of the repository, are immediately deployed to the site.

`victor` is a web application with two primary views. As a user makes a request for any of the site's page, the base authentication service examines the request's cookies looking for an access token. If no token is found, the page request is immediately redirected to the login page. The login view is relatively bare. The user can choose to login via either Facebook or Google or if they aren't interested in authenticating they have the option of accessing a read only view of the garden.

If the request does contain an access token, the user successfully authenticates,

or they choose to view the read-only data then they're routed to the garden dashboard view. This view's controller immediately makes a request to the API to gather the most recent data. The data is received as a large JSON array, which makes manipulation and sorting quick and easy.

From the received array, the controller constructs an objects matching each parameter's name to an array of objects representing that parameter's measurements. Each element of the array takes the form `{'value': XX, 'dateCreated': mm/dd/yyyy}`. The view uses an Angular directive `ng-repeat` to iterate through the object of parameters and constructs a time series graph for each.

Though collecting and sorting all of the data adds some complexity to the client-side service, doing so allows for any configuration of sensors. Designation of data is handled via the parameter, and the datas representation is parameter agnostic. Parameter specific view configuration can be handled and defined within the controller's view if necessary.

For credentialed users a small icon button is added to the parameters with configuration or manual controls. Clicking the button opens a window with the appropriate inputs and a form mapped to the API endpoint. Submitting the form sends a simple HTTP request in the same exact format as submitting or manipulating data. User's controls are never multi-step, so submission fully hands off responsibility to the API. Furthermore, any command that invalidates the controller's current data initializes a subsequent request for a data update. Neither call requires the page to reload, but because of Angular's two way data-binding any change in data is immediately reflected in the view

resulting in a near real-time display. If no change occurs on the page, the data is refreshed based on the time since the last API call.

As previously mentioned, this service is very simple, which is largely due to the architecture of the other components. I chose to develop a web application for the widest availability, but there's very little preventing the front end being replaced or duplicated. The data is fully abstracted from the application, so a mobile application, native client, or something more novel like a skill for Amazon's Alexa.

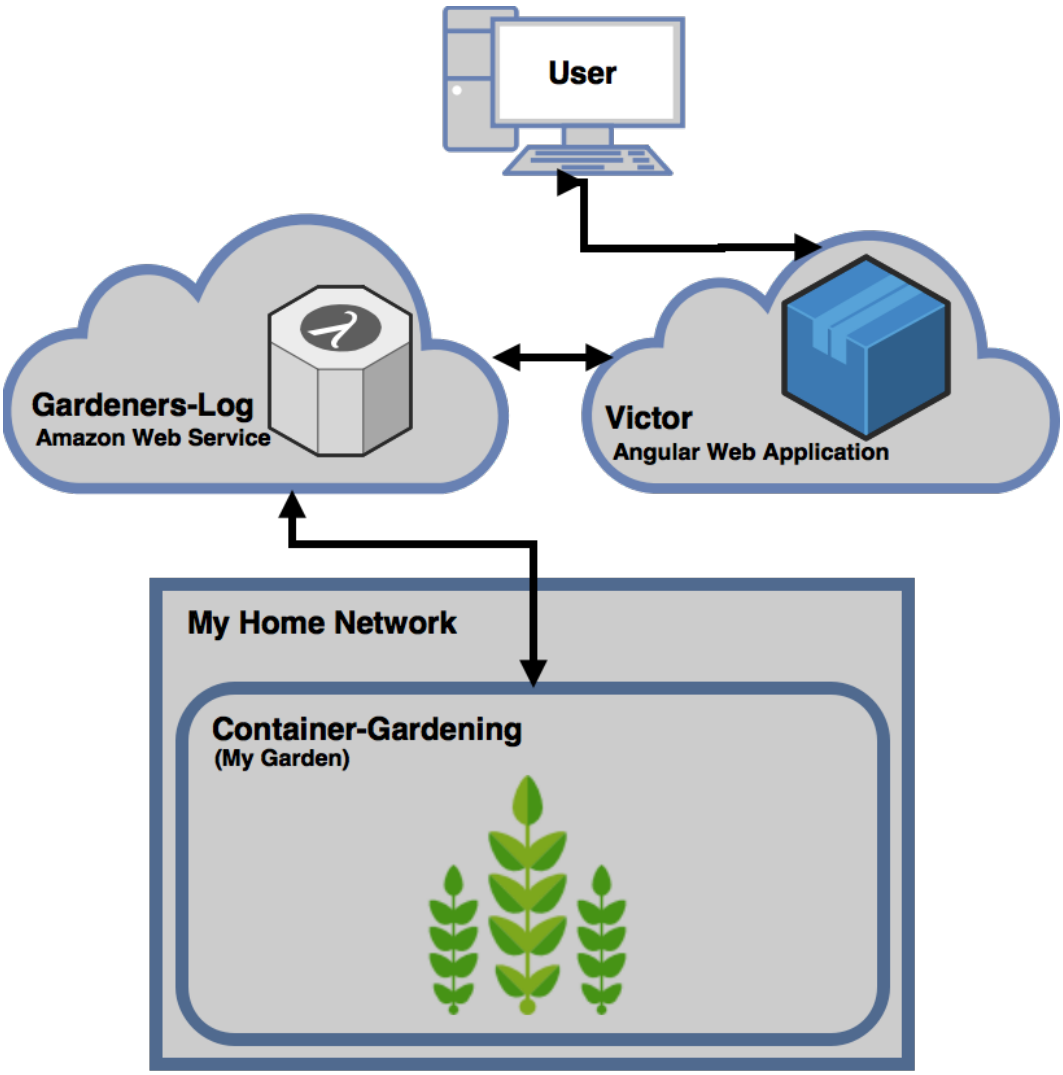


Figure 2.1: Flow

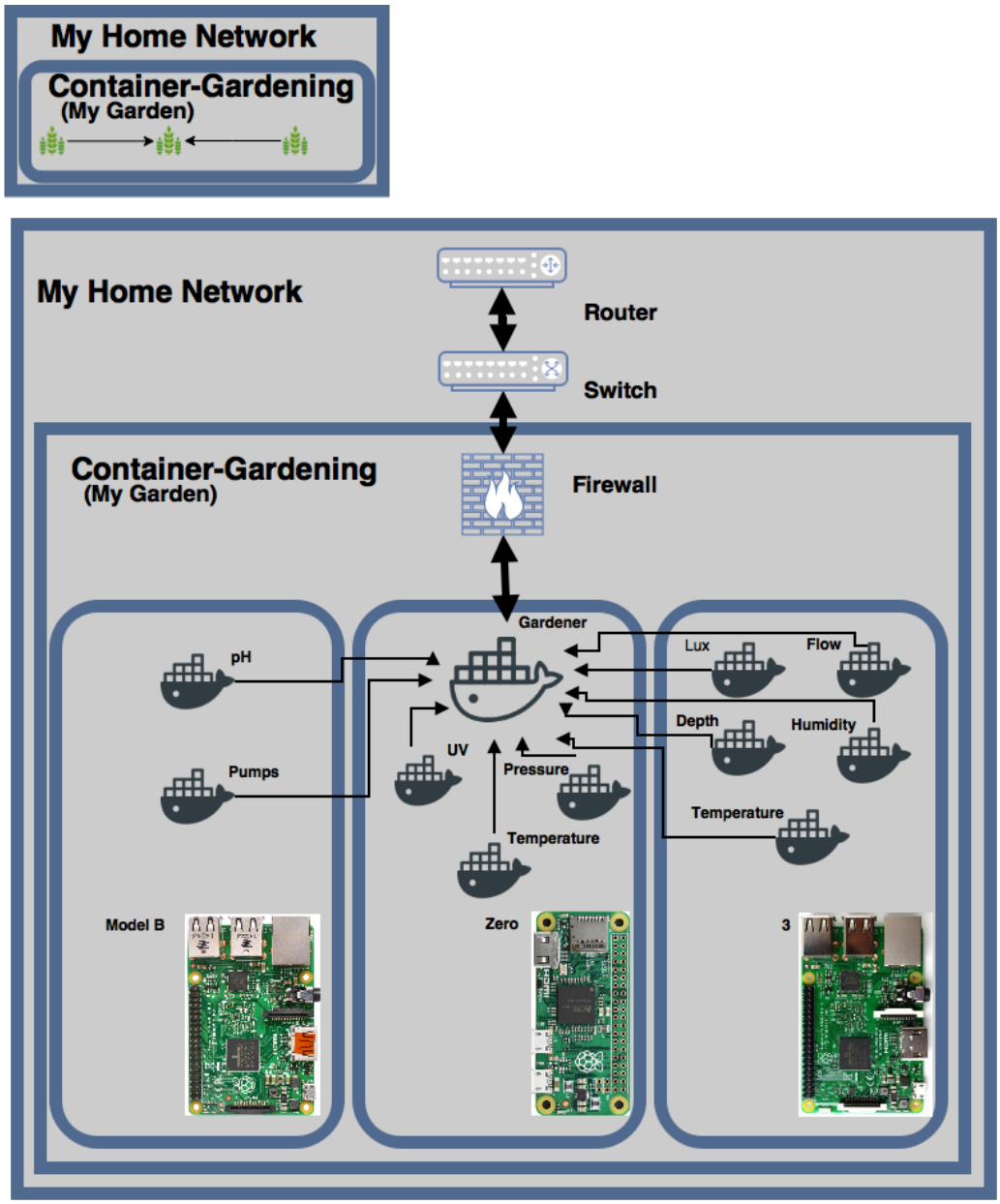


Figure 2.2: Garden

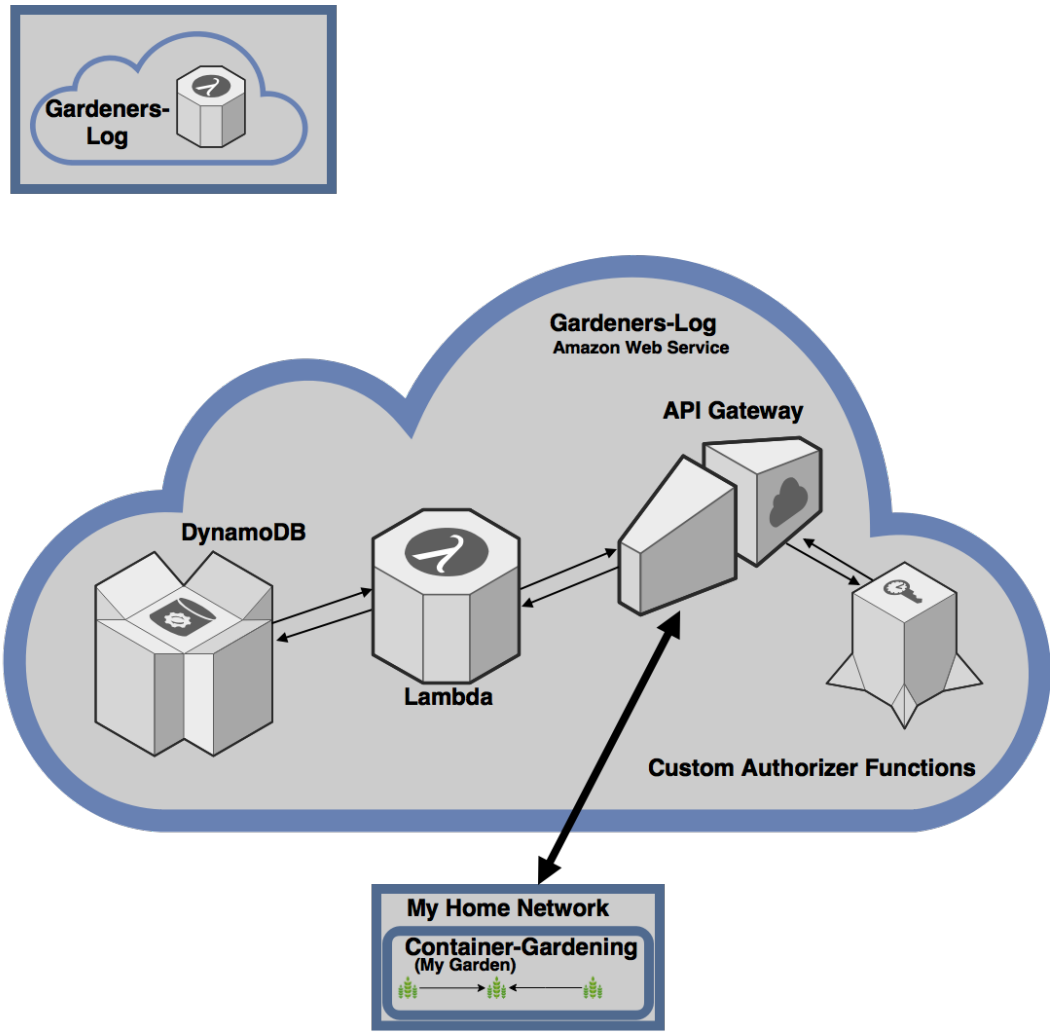


Figure 2.3: API

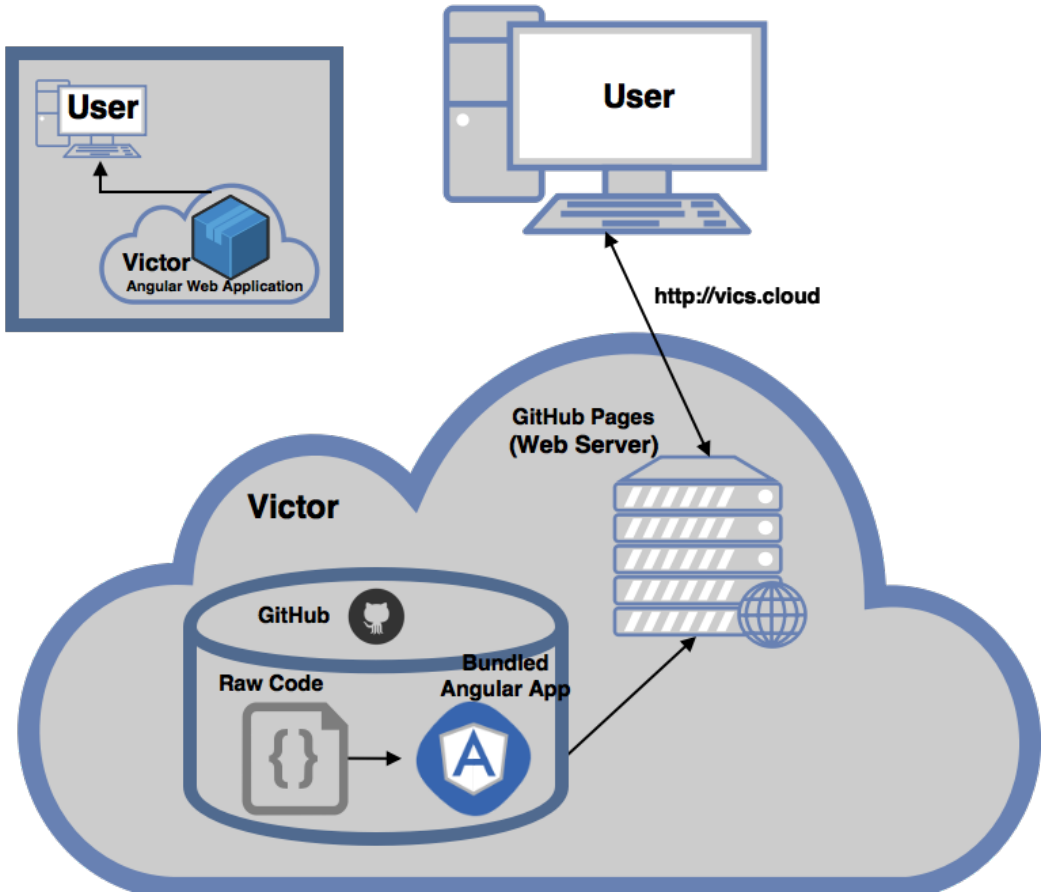


Figure 2.4: Victor

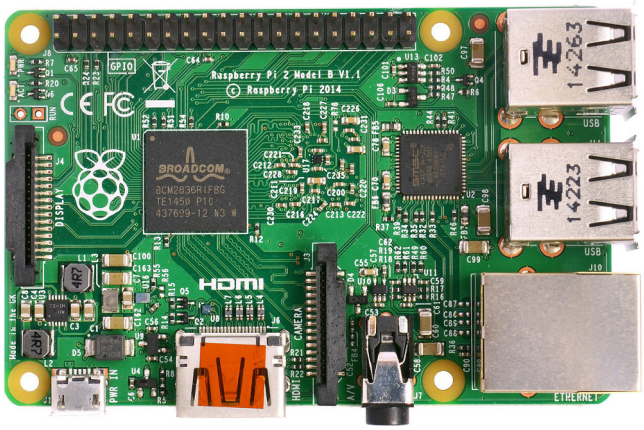


Figure 2.5: Model B

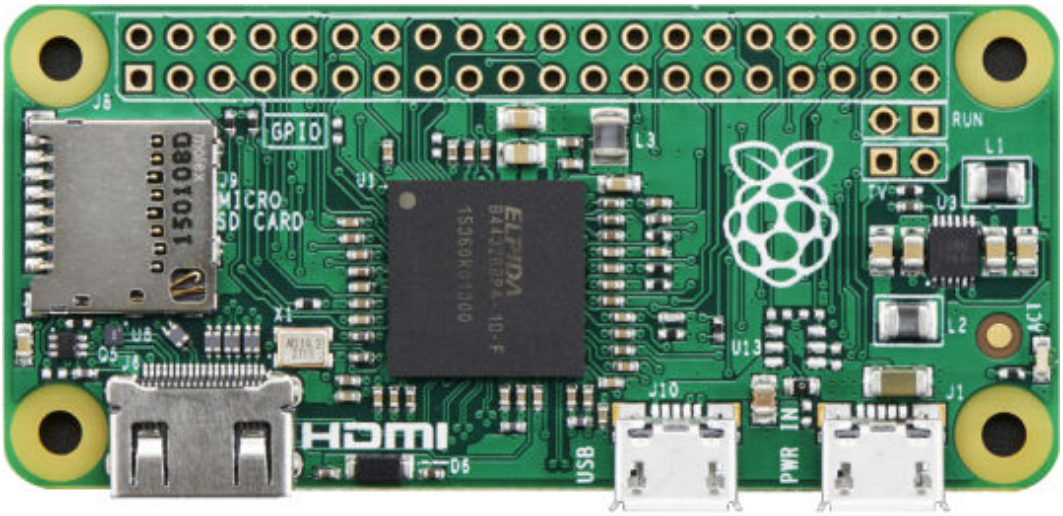


Figure 2.6: Zero

Figure 2.7: 3

Chapter 3

Security

3.1 Introduction

The most important part of Victor is its data. Victor makes it easy to collect more data and new types of data. It also lets users view data as the composite of a few visually pleasing graphs rather than a wall of text. Getting an overview of the garden's health is quick even with a huge amount of data. Yet none of this functionality is important without the data itself.

For this reason, it is very important to me that the data is accurate and current. When I open Victor's web application I expect to see graphs that represent the state of the garden. Wrong data or a lack of data means Victor is not providing value and could be causing harm.

For instance, what if the water depth graph shows that the reservoir is close to full while it is empty or emptying? In this case, Victor is not providing any value and is very explicitly causing harm by misleading me. Opening the app to see no data at all is still bad, but less dangerous. A lack of data could arise from a simpler hardware failure or software bug, but it could also point to some attacker gaining access to the database service or blocking the web application and the database's communication. In any case, this lack of data availability is much quicker to spot and diagnose.

When I see a healthy reservoir depth I expect to be reading accurate measurements from a sensor that is still online. This expectation is important to keep the framework useful and accountable. Though, I do not believe it is something that I can take for granted.

There are two things that could keep Victor's data from being available and accurate. Either the system could fail or someone, or something, could attack it. There is always some risk of the system failing. Hardware is bound to fail because electronics do not fend well outdoors, and software always needs maintenance. I built the framework to be resilient, which I detailed in the previous chapter. One of the ways I accomplished this, however, was by making routine maintenance easier. These sorts of failures are unpreventable but detectable. Hardware failure, for instance, is much more likely to report unrealistic or no data than it is to show believable, fake data. The latter issue is a bit more unnerving, but should be largely preventable. In this chapter, I'll discuss the ways an attacker could abuse Victor's services and use the system to compromise the data's accuracy and availability. I'll then outline how I weighed the difficulty and potential damage of these attacks while deciding where to focus

the frameworks defense and the mechanisms currently put in place to protect the system.

3.1.1 Who is the Attacker

The simplest definition of "the attacker" is anyone who attempts to use Victor in an unintended, malicious way. Though short and simple, this definition implies a great deal of difficulty in identifying any single attacker. First, this concept of "the attacker" emphasizes intentionality over concrete actions. A wide variety of things, notably fame, boredom, and financial gain, can motivate cyber attacks. Determining possible outcomes of an attack can help identify likely motivations. Second, there are many more unintended, malicious actions than intended ones. Since I developed the framework I am immediately disadvantaged. I have a very clear, but limited, perspective of Victor's services and how they communicate. An attacker's view is much more fluid. They should not have a very clear understanding of the framework, but they are only limited by their imaginations.

Thinking about who are we protecting the system against is always valuable. It helps identify what exactly we are protecting and what is necessary to protect it. Finding a good answer is difficult because there is almost never one attacker. Victor is likely to elicit some interest from a small subsection of all possible attackers. Though I cannot identify a single attacker, I can try to predict the most likely types of attackers. By doing so I can also start to determine the most common ways attackers will try to abuse Victor's services.

In most cases, categories of attackers can be defined using two variables, motivation and resources. Motivation in this context defines the amount of effort an attacker would commit to my garden's network specifically. Resources include the tooling, experience, personnel, and time available to that attacker.

The most dangerous and most difficult to protect against are the attackers that are highly motivated with a large pool of resources. This group includes nation states and organized, professional rings of hackers. Any garden network, including mine, would be invisible to these sorts of attackers unless it feeds a huge number of people or is integral to generating large amounts of money.

Attackers with little or no motivation and only moderate resources are more plentiful. Novice hackers find or write scripts to find possible targets. These programs wander the open internet looking for machines exposing services, such as a Raspberry Pi running SSH. Then they spend a small amount of time trying to execute a novel attack or guess the password before moving on to the next target. Their motivation is low, but their resources are more plentiful than one would expect. Tools to automate mindless attacks like these are widely available and free. Many automated bots scan the internet in the same way novice hackers do. They look for vulnerable devices that they are confident they can compromise. These vulnerable devices help grow a network of infected machines used to serve ads or perform DDoS attacks. Their motivation is even lower than novice hackers because they are looking for a very specific signature. However, they have the benefit of running perpetually and autonomously. I expect that a very strong majority of attackers interested in Victor would be in this group of attackers. Because of this, most of my defensive measures are to protect against simplistic, automated attacks.

The rest are in place to guard against the smaller group of possible attackers that have slightly greater motivation. By committing more time and effort to learning about Victor itself these attackers could conceivably construct a more elaborate attack using the framework's data flow and the communication between services. These scenarios are the most likely to compromise the data accuracy.

I expect the types of attackers, their motivations, and the tools they have available to differ. As I work through the possible attacks and how I worked to prevent them throughout the rest of this chapter I will include a quick synopsis of expected attacker type at the beginning of each section.

3.1.2 Trust

The three services that comprise Victor are all very different. They each use very different technologies to provide separate, complementary functionality. For the services to work together transparently, they need to be able to communicate.

Transferring data in Victor requires HTTP communication between each component. Luckily, the messages, senders, and receivers are all very predictable. For instance, the web application is never supposed send messages directly to the garden network. We can expect that any message received by the garden and sent from the web application is malicious.

Using the expected flow of communication, I have defined trust relationships

between the services. These relationships define what messages the framework expects, trusts, and needs to function. Using that list, I am able to deny any messages that do not fit the expected characteristics.

The garden's computers live in a self-contained subnetwork of my home network. A firewall sits at the entry point of the garden network as a first line of defense. The rules in this firewall deny internet connections to the Raspberry Pis that are not running the Gardener container.

Communication to and from the Gardener is only expected to be with the database API service. The database service is built on a serverless architecture, so it is not expected to use the same IP address more than once. Amazon Lambda, the serverless hosting service, does not have a set list of addresses that it uses. Amazon owns and publishes a list of their IP addresses, but this list is liable to change at any point.

These reasons make implementing a IP address whitelist error prone. A whitelist would deny any messages sent by an IP address that is not on the list. This would cut down a lot of malicious automated traffic, but in this case it introduces maintenance complexity.

Instead, to be trusted, messages from Amazon need to be signed by the Amazon API Gateway that manages database service's requests. To sign a request, the Gateway first forms the request. Then it generates a string to use as input for a cryptographic hash function. The signature is then created using this string and a secret derived key created using a series of hash-based message authentication codes.

This signature is then verified on the Garden's end. Messages that contain a valid signature are trusted, but the contents are treated as being potentially dangerous.

The Gardener container first reads the message to make sure it is safe and then forwards it. Messages sent within the garden network are all expected to be trustworthy because the Gardener container is looking out for the rest of the machines and containers in garden. The construction and format of the messages are still standardized, so spotting issues in the logs is straightforward.

The database service trusts messages sent by both the gardener container and web application's authenticated users. Messages sent from the gardener are standardized and must be in an expected format. A request from the garden must also contain a secret API key that matches the one defined by the API's endpoint. Messages from the web application are trusted if they can pass the OAuth authentication check. After a user goes through the process of authenticating with either Facebook or Google, they are given an access key to include in every request.

The web application's expectations are the simplest. It asks for data from the database service, so it trusts that the data that it receives is in the right format and vetted data. The database service is not expected to do a lot of data processing, so the web application is expected to check it over quickly before reflecting it directly to the web page.

3.1.3 How Is the Data Sent

In the previous section I defined which services are expected to communicate. I defined some high-level ways to confirm the sender of the message and when the contents of message should be treated cautiously.

Now I will detail exactly why and how the messages are sent from and received by the services.

The Gardener container sends measurements to and receives commands from the database service. Victor uses HTTP as a standard protocol for communication. The garden allows connections over port 80 and port 443 for outgoing HTTP and incoming HTTPS, a secure version of HTTP, requests.

Each Raspberry Pi, including the one hosting the Gardener, also expects communication over port 22. The Pis use port 22 for SSH which is used for both remote deployment via docker-compose and standard management over SSH. Port 22, 80, and 443 are the only externally facing allowed ports. Internally, machines send HTTP messages between each other containing new measurements and commands on port 8000.

The database service sends commands to and receives data from the garden. It also receives requests for data from the web application service. All data transferred is textual and is represented through a standardized JSON-based format.

The API expects requests over both port 80 and 443 for HTTP and HTTPS.

Standard HTTP is used for most requests to the Amazon API gateway that are asking for data. Consumption of data is both non-confidential and is meant to be available to any user. HTTP in this case is allowed so that the data is available to as many users as possible.

HTTPS is used for any request that will create, delete, or manipulate data on the database service. The use of HTTPS when authenticating and performing authenticated requests, like deletion or configuration, is crucial. In this case, using encrypted messages prevents an attacker from gaining unauthorized access to credentialed actions because it protects the tokens that authorize these actions.

The front-end sends commands to and receives data from the database service. Any message that the user wants to send to the garden is mediated by the database service. I limit the number of possible senders to simplify the process of defending the communication.

The front-end expects requests over both port 80 and 443 for HTTP and HTTPS. For this build I chose to use GitHub Pages, GitHub's free hosting service, because it is free and very easy to manage. Ideally, the service would default to HTTPS, but GitHub Pages does not allow this configuration. Instead messages and commands are sent in a protected way programmatically.

3.1.4 Threats

I have now defined clear expectations for how Victor's services are meant to interact. Using these expectations I will begin to consider the ways in which an attacker might try to access the data or prevent others from accessing it.

There are two main ways an attacker could modify the data to make it inaccurate. Both of these have the primary goal of tricking the database service into trusting the faulty message. The attacker in these scenarios would be motivated to attack my garden specifically because these attacks require a good amount of knowledge and time.

First an attacker could pretend to be part of the garden network and send fake measurements to the database service.

To send messages to the API the attacker would either need to steal the API key used to validate the Gardner container's messages and then send fake updates to the database, or they would need to gain access to the garden's network and send messages to the Gardener container, which would be forwarded and trusted by the database service.

Getting into the garden's network could include either physically accessing one of the Raspberry Pis and modifying the code or trying to access the garden's subnetwork with their own device and pretending to be a new garden machine.

Alternatively, an attacker could try to send messages from the web application to modify or delete the data already stored in the database service. If

the attacker were able to make authenticated requests they could also send dangerous commands to the garden.

If they were unable to authenticate through the web application, they could also try to send requests directly to the API and hope that their credentials were not checked there.

There are also two places in Victor's data flow that an attacker could keep a user from being able to access the data. The attacker in these scenarios would be much less motivated. In fact, the garden would not even need to be the primary target. The attacker would most likely be a bot or a large network of bots called a botnet.

The two places where data could be prevented from making it to the user are between the garden's network and the database service and between the database service and the web application.

Blocking communication between the database service and the garden prevents a user from viewing data and issuing commands.

If the database service or garden network are overloaded with junk traffic, they will not be able to communicate. The user no longer has access to the most recent data because the garden cannot send messages to the API. The messages will be sent once the connection is resumed, but the latency can be an issue. Any configuration or control messages that need to be sent to the garden will not make it. Both of these issues can be severe in time-sensitive applications, like a rapidly dropping reservoir.

Blocking communication between the database service and web application causes slightly different issues. The user will not be able to view any data through the web application even though the data would likely be complete and recent. The user will also be unable to send messages to the garden. In a dire situation, the maintainer of the garden would have to access the Raspberry Pis directly over SSH.

Lastly, each service includes some form of authentication, so they are all susceptible to a brute force attack. If any attacker found the credentials to one of the services they would be able to upload their own code, which could contain backdoors allowing them to perform any action they wanted, or invoke unauthorized actions.

Discovering the credentials of the database service administrator would allow the attacker to create their own API endpoints to perform any action. Bypassing a garden machine's SSH authentication would give the attacker direct access to the API key allowing them to create fake data. Circumventing OAuth authentication on the web application service would allow the attacker to modify and delete data and send commands through the API to the garden.

These three scenarios show that bypassing or breaking authentication on any of the services allows an attacker to easily manipulate the data, so protecting against these sorts of attacks is an important part of keeping the data accurate.

3.2 What needs secured

Implementing secure authentication and proper session management is absolutely key. Gaining unauthorized access to any of the services, but especially the database, is highly damaging. It is critical that passwords are strong and two-factor authentication is used where possible. When a user authenticates with the web application, their session should not last forever. Instead, they should be required to confirm their session or reauthenticate periodically.

The security of the garden's network is a weakness of the framework. No unnecessary services should be running on any of the garden's machines. Only the single machine hosting the gardener container should be able to access the internet. The firewall should only allow messages through if they are expected and should fail closed. Failing closed, as opposed to open, means that if some received packet makes it through the machine's list firewall rules without being accepted or denied, then it should be denied by default rather than allowed through. The goal should ultimately be to harden the machines to the greatest extent possible using all modern best practices.

Data received by the database service and sent and received within the network should be checked and sanitized. To protect against the situation in which a message is able to bypass the authentication of the web application service and the identification of the database service, the database service and garden machines should sanitize it to limit its potential damage.

Likewise, the web application's controls should be checked and sanitized to prevent any unnecessary web vulnerabilities, like cross-site scripting or request

forgery, to eliminate the tools an attacker has to elevate privileges. These are two instances in which we want to protect against dangerous input, and they are both handled in a very similar way.

Any data that is going to be reflected to the web application's view, used to set configuration variables, or invoke manual action needs to be treated as potentially dangerous. Reflected text in the web application can be dangerous if it is rendered on the page as valid HTML. For this reason, user input is either sanitized or escaped. This is another example of how Angular provided a lot of value right out of the box.

Sanitization of user input includes removing potentially dangerous characters entirely. This is commonly handled using regular expressions by looking for strings that match a particular pattern – in this scenario we would try to identify and remove strings that begin with `<` and end with `>` because this is a pattern followed by all HTML tags. Angular sanitizes items by default if they are assigned in the HTML using the `ng-bind-html` and can be sanitized manually by using the `$sanitize` service.

```
function cleanContent(string) {
  return ”<span>” + $sanitize(string) + ”</span>”;
}
```

There is no current use case where we need to store and display HTML or text with potentially dangerous characters, so sanitizing the input is the safest option. Escaping, where the dangerous tokens are escaped so that they are not recognized as HTML, is handled by default through standard two-way data

binding.

All the graphs are constructed using parameter values stored in the database service's entries, so every title string is escaped to prevent the possibility of an attacker inserting malicious code into the database as a parameter title causing it to be rendered on the dashboard to every subsequent user.

It is also possible for dangerous input to be sent to the Gardener container as an RPC request. The gardener is capable of executing a handful of commands to modify the environment remotely. These commands can be changes to software, such as the propagation of new configuration, or hardware, like turning on the pumps for a set amount of time.

It is certain, however, that we can list each possible valid command. For this reason, I created a dispatch table including all the possible commands mapped to their function. Functions are first class objects in Python, so they can be stored in Dictionaries by default and invoked by key. This prevents execution of unwanted code because any command sent to the Gardener container's RPC server is cross-referenced with the dispatch table. This creates a whitelist for incoming requests. Furthermore, the parameters expected to be passed to these functions each have defining characteristics that can be checked before the functions execution. the dispatch table can also store an upper or lower bound on the parameter.

Under this system, the Gardener container might receive a request from the database service. This request contains a signature proving that it is indeed from the API and has a body that contains the following:

```
turnPumpsOn 5
```

This message is split and expected to be in the form `method parameter`. The dispatch table includes a line, which matches the string `turnPumpsOn` to the appropriate method.

```
{
  'turnPumpsOn': pumps.turnOn
}
```

In turn, this method is called with the passed parameter. This flow means that, barring a serious breach, a validated message is sent from the API containing a method known to exist and be available for remote execution.

3.3 How Did I Secure It

I implemented a handful of security mechanisms to keep Victor safe while remaining extensible, powerful, and user friendly. Each of these preventative measures is intended to help defend both the weakest and most important parts of the framework.

3.3.1 Physical

Physical security only protects against a very motivated attacker that is interested in doing long term damage. If the attacker already has direct physical access to the garden and its devices, they can already do a lot of physical damage. The following attacks are harder to detect and capable harming the system over longer periods of time.

Physical security is ultimately a deterrent rather than a means to an end. It's possible to interface with the Raspberry Pi machines over both ethernet and usb, so protecting physical access to these ports is important. Furthermore, the filesystem is contained on a single SD card, so direct access to this card could allow an attacker access regardless of whether or not the ports are protected.

I am using three primary methods of protecting the devices from an attacker with physical access. The hardware is kept in a secured enclosure, unused and unnecessary ports are removed, and SD cards are fixed and secured.

An enclosure is a necessary component of the build because the garden is outside at most times. It's possible to grow plants indoors with the proper lighting, but, even then, an enclosure protects the electronics from possible leaks. Though the enclosure is primarily in place as protection from the elements, adding a simple lock is an easily implemented first line of defense. The enclosure is fully acrylic, however, so a motivated attacker would have little problem gaining access to the devices.

With the device in hand, an attacker's first step would be attempting to gain

local access over USB or ethernet. Though we may require these ports for wireless adapters, there are unused ports on each device. Unused ports can easily be desoldered and removed. Used ports can be coated in epoxy making their connections semi-permanent making it much more difficult to physically interface with the device. Lastly, the SD card contains all the data necessary to send messages to the proper API endpoint, so some means of obfuscation and protection is necessary. Encryption of the entire filesystem is the most secure way of solving this issue, but it can be problematic in the case of power failure. An authorized user may not be around to enter the decryption key for every device on reboot. In addition, in a large-deployment this could be a huge waste of time. For Victor's purposes, the creation of fake data is not a large enough concern to risk the increase in device management, so the SD card is handled much like a port that is in use and covered in epoxy. Coating the SD card reader in epoxy is enough to create a semi-permanent connection. Card failure happens much less often than reboot, so maintenance issues are less of a concern. There is always the chance of an attacker desoldering the entire reader and physically connecting it to another device, but there is a fine line when it comes to balancing the interplay between usability and overall system security. In this instance, mitigating the risk would compromise the management of the garden, and likely result in the implementation of a costly secured enclosure. The risk of an attacker removing a device's storage in the hopes of gaining write access to the API is too low to justify the costs associated, especially when considering that an attacker would still need to brute force the password of the device.

3.3.2 Network

The following security mechanisms used to protect the garden's network are primarily used to prevent attackers with low motivation and low resources. These best practice methods of keeping Linux machines safe and secure should prevent the sorts of attacks bots and novice attackers are capable of performing.

Only one Raspberry Pi is allowed access out of the subnetwork to the open internet. This is the machine that the gardener container, which is used to send and receive messages to the database service. Though the machine with the Gardener container has some additional measures, each Raspberry Pi was hardened by the following actions.

-Every machine's username and password were changed from their defaults.

Though this step seems trivial, the number of computers with unchanged credentials that are accessible over the internet is unnerving. For instance, using Shodan, which is a search engine that regularly and randomly fingerprints internet facing machines, it is trivially easy to find vulnerable devices. Searching for the phrase "default password" results in over 80,000 devices that contain those two words in their response, most commonly in the banner.

```
Cisco Router and Security Device Manager (SDM) is installed on this device.
This feature requires the one-time use of the username "cisco"
with the password "cisco". The default username and password have a privilege
```

This is a real response indexed on April 8th, 2017. Furthermore, determining a list of internet facing Raspberry Pis running SSH is just as simple. Though Raspberry Pis are capable of running multiple operating systems, they are distributed with the Raspian OS. Searching for internet facing hosts that are running Raspian with port 22 open results in an index of over 50,000 devices. Since Raspian ships with default credentials and was created primarily for users with little Linux experience, it is likely that many of these services are entirely open.

-All unused services were shut off

Raspberry Pi's OS comes default with a handful of unnecessary running services. None of my devices require FTP, Apache, or SQL to run properly. Furthermore, any service they do need will be running inside of a Docker container.

-Machines are kept regularly updated.

One of the easiest ways to keep the OS and necessary services secure is by updating them regularly. To cut down on maintenance requirements this is handled with `cron-apt` a tool used to automatically update packages at regular time intervals.

The main gardener machine also included the following two measures because of its increased responsibilities.

-The machine used to communicate with the API needs comprehensive logging.

The gardener machine has the largest SD card at a whopping 64 Gigabytes, so it handles extensive logging. Standard Debian logging keeps track of log ins and failed attempts.

-Protect against unwanted packets with a firewall.

Iptables is a standard firewall included in most Linux distributions that allows for user-defined configuration of the tables provided by the Linux kernel firewall. The configuration I've used is simple, but it allows me to define the ports and networks from which I allow communication. A huge benefit of the firewall is that it allows me to drop any packet that's not sent to the expected ports – 80, 443, and 22. Furthermore, a third-party tool, Fail2ban, runs on the Raspberry Pi and monitors the logs. It autonomously modifies the iptables rules to ban any IPs that appear malicious. Too many password failures or inappropriate snooping are circumstances that can be configured as a means for ban.

3.3.3 Database Service

One of the benefits of using a serverless architecture hosted within the cloud is that, I'm not responsible for hardening any of the servers that my API will ultimately run on. I do, however, need to properly manage authentication to the endpoints I want protected. The two mechanisms that I used to protect the endpoints were Amazon authorizer functions and API keys.

API keys are not inherently a great form of security because they rely on

a single secret shared between the main garden device that is running the Gardener container and the API. Because the garden's network and devices are secured and the Gardner is only sending encrypted messages, API keys provide a cost effective and simple means of decent protection. The key is created and stored within the Amazon Management Portal and set as an environmental variable on the main garden machine. Every request sent to the API from the garden contains the key in its headers.

Though I'm confident about the API key's role in this build, future applications may wish to utilize IOT frameworks, like AWS Greengrass or AT&T's M2X, for management, deployment and security. These frameworks facilitate local development for remote deployment, handling intermittent connectivity via data synchronization, and permission policies through X.509 certificates. The strongest benefit provided by these alternative services is that, unlike Victor's database service, they are able to identify the devices that are sending them data, which creates fine-grained control over the set of API actions each identity is authorized to invoke. These benefits come at a cost both monetarily and in the sense that the sender and receiver of the data become much more coupled.

When a request is sent to an Amazon API Gateway it first checks to determine whether or not any custom authorizers are associated with the Lambda function it is trying to access. If one exists the gateway grabs the authorizer token in the request and forwards it to the authorizer function. The API implments the OAuth authentication protocol so functions protected via the authorizer functions are perfectly secure, assuming a valid OAuth configuration is in place and barring any breach of Google or Facebook.

In terms of availability, AWS Lambda is infinitely scalable. The service is ultimately reliant on Amazon, but because the code is spun up directly in response to the requests it receives, its infrastructure is as large as the number of people or entities trying to access it. Furthermore, the servers owned by Amazon that are capable of running Victor's functions are spread geographically. The risk of company-wide outage or denial of service is slim compared to any alternatives.

3.3.4 Authentication

As I mentioned earlier, there are many places in the framework where a failure to protect authentication can cause a lot of problems. Authentication to the management of any of the three services is critically damaging, but I am only personally protecting the garden network's management. I have to trust in the fact that I have vetted the security practices of the services I have chosen to use, but I also need to ensure that I use best practices where I have control.

As I mentioned in the network section, each of the Raspberry Pis have a defined set of credentials used to authenticate over SSH. The passwords are all sufficiently long and randomized. Theoretically, they could be broken using a brute force attack, but it would be prohibitively expensive both computationally and financially.

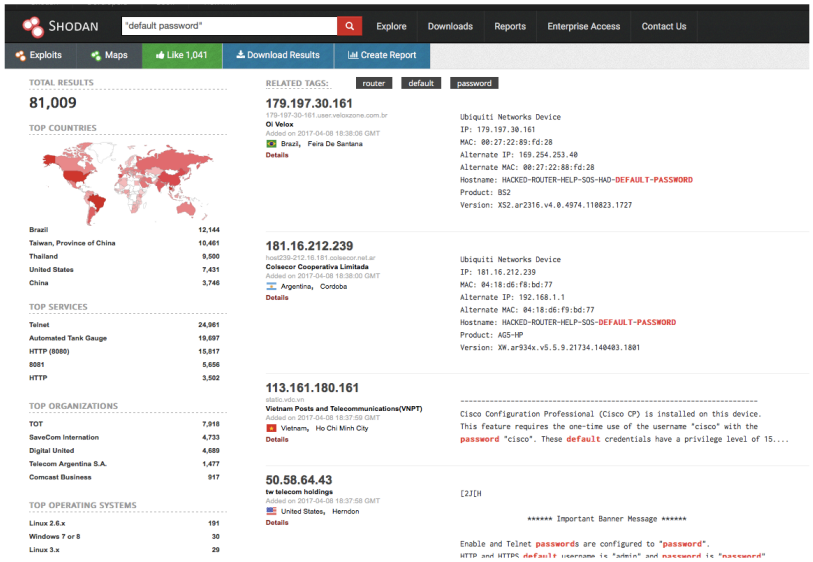
The front-end web application utilizes the best practices for using the OAuth authentication protocol. In this case, the provider, Facebook and Google, set the username and password requirements, so I can be confident in the

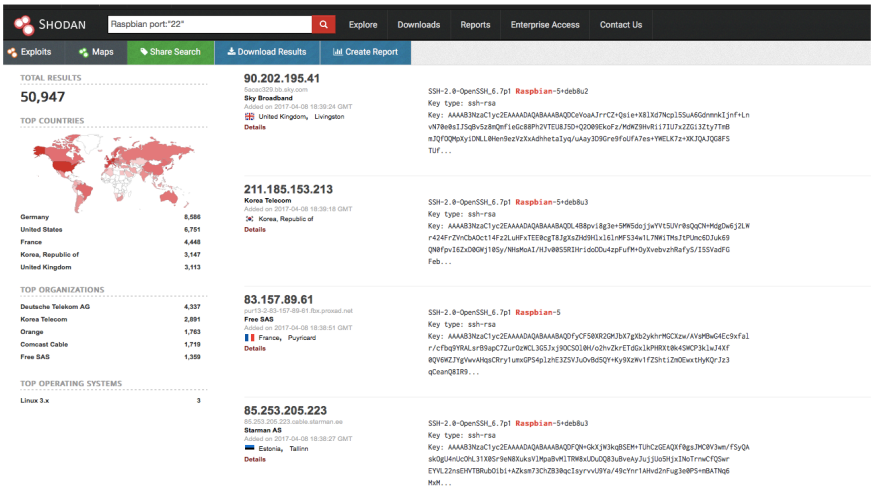
strength of credentials. Furthermore, both of these services are continually adopting two-factor authentication, which is ideal for authenticating to make credentialed API calls. The custom Lambda authorizer, upon receiving a success response from the chosen provider, sends the access token protected in the request's body. The token is stored in a cookie on the user's machine and used in subsequent requests that require the key. Secure coding of the web application helps protect against a remote attacker gaining access to or using another user's token. After a predetermined amount of time the token is invalidated and the user is required to authenticate once again.

Authentication can be a scary thing to secure because there are many things that can go wrong, I'm confident that by following best practices and using cryptographically secure passwords that I have done my due diligence.

3.3.5 Closing Thoughts

The security of the system is a balance of strength and convenience. The resources and actions made available to the user vary in both value and vulnerability. Extra measures of security are put in place both where I find the most value and where it will not impact the user's perception of the tool. Furthermore, security is handed off to third party providers when I felt that they can reliably provide a deliberately secure and well-maintained service, which keeps the operator's duties relatively low. At a significantly greater cost to the owner of the system it would be possible to implement a much more secure framework, but considering the overall risk of breach, the mechanisms put in place allow for a secure environment.


SHODAN
"default password"
Explore
Downloads
Reports
Enterprise Access
Contact Us
Exploits
Maps
Like 1,041
Download Results
Create Report
TOTAL RESULTS
81,009
TOP COUNTRIES
Brazil 12,144
Taiwan, Province of China 10,461
Thailand 9,500
United States 7,431
China 3,746
TOP SERVICES
Telnet 24,961
Automated Tank Gauge 19,697
HTTP (8080) 15,817
8081 5,656
HTTP 3,502
TOP ORGANIZATIONS
TOT 7,918
SaveCom Internation 4,733
Digital United 4,689
Telecom Argentina S.A. 1,477
Comcast Business 917
TOP OPERATING SYSTEMS
Linux 2.6.x 191
Windows 7 or 8 30
Linux 3.x 29
RELATED TAGS:
router
default
password
179.197.30.161
Oi Velox
Added on 2017-04-08 18:38:06 GMT
Brazil, Feira De Santana
Details
Ubiquiti Networks Device
IP: 179.197.30.161
MAC: 00:27:22:89:fd:28
Alternate IP: 169.254.253.40
Alternate MAC: 00:27:22:88:fd:28
Hostname: HACKED-ROUTER-HELP-SOS-HAD-DEFAULT-PASSWORD
Product: BS2
Version: XS2.ar2316.v4.0.4974.110823.1727
181.16.212.239
Colsecor Cooperativa Limitada
Added on 2017-04-08 18:38:03 GMT
Argentina, Cordoba
Details
Ubiquiti Networks Device
IP: 181.16.212.239
MAC: 04:18:d6:f8:bd:77
Alternate IP: 192.168.1.1
Alternate MAC: 04:18:d6:f9:bd:77
Hostname: HACKED-ROUTER-HELP-SOS-DEFAULT-PASSWORD
Product: AGS-HP
Version: XM.ar934x.v5.5.9.21734.140403.1801
113.161.180.161
Vietnam Posts and Telecommunications(VNPT)
Added on 2017-04-08 18:37:59 GMT
Vietnam, Ho Chi Minh City
Details
Cisco Configuration Professional (Cisco CP) is installed on this device.
This feature requires the one-time use of the username "cisco" with the
password "cisco". These default credentials have a privilege level of 15....
50.58.64.43
tw telecom holdings
Added on 2017-04-08 18:37:58 GMT
United States, Herndon
Details
[2J[H
****** Important Banner Message ******
Enable and Telnet passwords are configured to "password".


SHODAN
Raspbian port:"22"
Explore
Downloads
Reports
Enterprise Access
Contact Us
Exploits
Maps
Share Search
Download Results
Create Report
TOTAL RESULTS
50,947
TOP COUNTRIES
Germany 8,586
United States 6,751
France 4,448
Korea, Republic of 3,147
United Kingdom 3,113
TOP ORGANIZATIONS
Deutsche Telekom AG 4,337
Korea Telecom 2,891
Orange 1,763
Comcast Cable 1,719
Free SAS 1,359
TOP OPERATING SYSTEMS
Linux 3.x 3
90.202.195.41
Sky Broadband
Added on 2017-04-08 18:39:24 GMT
United Kingdom, Livingston
Details
SSH-2.0-OpenSSH_6.7p1 Raspbian-5+deb8u2
Key type: ssh-rsa
211.185.153.213
Korea Telecom
Added on 2017-04-08 18:39:18 GMT
Korea, Republic of
Details
SSH-2.0-OpenSSH_6.7p1 Raspbian-5+deb8u3
Key type: ssh-rsa
83.157.89.61
Free SAS
Added on 2017-04-08 18:38:51 GMT
France, Puylaurent
Details
SSH-2.0-OpenSSH_6.7p1 Raspbian-5
Key type: ssh-rsa
85.253.205.223
Starman AS
Added on 2017-04-08 18:38:27 GMT
Estonia, Tallinn
Details
SSH-2.0-OpenSSH_6.7p1 Raspbian-5+deb8u3
Key type: ssh-rsa

Chapter 4

Alternate applications

4.1 Introduction

Though I designed the framework around the use case of monitoring and controlling hydroponic gardens, I put a lot of time and care into making the services as extensible as possible.

Internet of things devices already make up a huge portion of all internet connected devices globally. By 2021 IoT is expected to be a 2.9 trillion dollar industry made 20 billion separate interconnected devices.[1] A majority of these devices are, and will be, sensors because IoT is an incredibly efficient, cost-effective means of using analytics to increase productivity and ultimately decrease operating costs.

[1] http://www.gartner.com/newsroom/id/3236718

Though still very young, IoT is facilitating a number of very interesting projects in production environments. Disney, during its renovation of Disney Springs, recently built two five-level parking garages that fully leverage Internet of Things technology. Each parking spot has a small device attached to the ceiling directly above it. The device using a combination of high-power LEDs and a proximity sensor determines whether or not the parking spot is open and illuminates the LEDs if so. The devices also passes on the status of their parking spot so the total open spots per row, floor, and garage can be displayed on a number of signs to mitigate congestion and make parking much more efficient and simple.[2]

The city of Barcelona is funding a 90 million dollar project to construct a network of sensors installed throughout the city. As many as 3000 motion detecting street lights with the capability of sensing pollution and humidity have already been installed. Furthermore, they're attempting to install a network of sensors used to gather sound levels to create and urban map of noise pollution. Notably, the project necessitates 1500 WiFi access points that will provide city-wide internet access for free.[3]

The breadth applications is endless, and the number of people with access to start developing is vast because microcomputers are cheap, available, and come with active and helpful communities. However, because of the state of the industry, there are a number of key issues that hinder deployment and development. Victor was designed to solve at least some of these problems, while allowing for a deployment of multiple configurations.

[2]http://www.pcworld.com/article/3167268/internet-of-things/as-iot-sales-surge-consumers-still-lead-the-way.html

[3]http://www.networkworld.com/article/3135270/security/fridays-ddos-attack-came-from-100000-infected-devices.html

4.2 Standards

In many ways IoT is still in its infancy. As hardware gets smaller and cheaper, the market is being flooded with vendors each of which hope to secure a foothold in the space and lobby to create their ideal standard.

To most the devices themselves are irrelevant, but the real value comes from the way, or ways currently, in which they communicate.

I devised Victor using an opinionated set of standards, but understanding that the industry trends shift constantly, I favored extensibility by completely abstracting communication.

One assumed constant is that communication to and from the API is explicitly HTTP traffic. The API is RESTful meaning its operation depends on HTTP requests to specifically constructed endpoints, so there's little leeway in the communication channels between the services. However, these REST is already a web standard in wide use. Any device capable of speaking HTTP has access to the data.

We give the gardener container on the root garden machine the ability to speak HTTP over the internet, but communication between machines is less solidified.

Communication between containers is abstracted into shared messaging modules. A module handles two separate tasks shared by any data logging application. First, the module is tasked with formatting a measurement. Though the

API expects measurements are passed according to a specific JSON schema, the API could mandate an alternate form of transfer and to conform a developer would only need to change the communication module.

Second, the communication module is tasked with sending messages between containers. For the sake of conformity and simplicity, I built the module to send messages in a JSON format over HTTP much like those that will be sent on to the API. An internal network is defined by the docker-compose file, but the channels of communication are pretty straight forward because the messages are coming to and from the gardener container. Though alternate networking technologies are gaining popularity for use in power conscious applications.

The Open Connectivity Foundation, a consortium dedicated to establishing the importance of interoperability it IoT, has consistently pushed for ZigBee's Dotdot as the unifying language of interconnected devices. ZigBee offers a suite of high-level communication protocols, each of which are used to create short range mesh networks with small, low-power digital radios. In the case of relative contained installment, or at least one in which devices can establish line of site with at least one neighbor, ZigBee protocols are beneficial because they use significantly less power and are intended to be cheaper than WiFi alternatives.

The communication module means redesigning the sensor's network is as simple as reimplementing or overriding the send method. ZigBee requires specialized hardware, but the framework is designed to easily adapt to the software changes necessary to leverage the hardware changes.

Furthermore the network architecture currently in place is totally agnostic to the protocol used to send messages. The garden's machines are configured in a star network in which one centralized point in the network delegates a majority of the communication, a topology that works well with both of the aforementioned communication protocols.

4.3 Maintenance

Maintenance is a huge concern with large installations of complex IoT configurations. As billions of devices flood the internet, it's imperative to have some means of automating maintenance. A centralized method of provisioning, deployment, and management is crucial for system administrators

Maintenance of a machine can be further separated into two separate subcategories, maintenance of the OS and maintenance of the processes it's running. I've leveraged three tools to keep Victor as maintainable as possible in order to promote extensibility and scalability.

The first two, `cron-apt` and `iptables configured with fail2ban` were mentioned in the previous chapter. None of the machines require a very extensive list of running services, so a majority of the OS maintenance is required for security. Regular updates and a firewall that can adapt to new threats handles 90% of regular OS maintenance. Most importantly, however, the machines need a sturdy stable connection to both their network and their external components.

Each container is capable of determining the state of it's sensor. Though runtime errors can occur for a number of reasons, the calls to the the sensors fail in an explicitly verbose way. If the measurement method fails the script does not stop. Instead a message is sent to the gardener with a declaration of the failure. The gardener is tasked with parsing the message and determining what to do. I've chosen to send a simple email with the sensor's status, but this can be tailored to use whatever method the maintainer finds most convenient like a text, slack message, or push notification.

The connection status of the device is a bit harder to determine. For instance, a device measuring temperature and humidity could lose connection to garden's network. If this machine configuration was unique it may be possible to determine simple because of a lack of temperature and humidity measurements, but its highly possible this is a commonly used configuration in the network. Though I've not been able to implement this in a production setting, one way I've been working to solve this issue is with docker-swarm, another service provided by the third tool Docker. Docker-swarm allows me to cluster each of the separate garden machines around a central manager. Each of the nodes, every connected machine, is then capable of running a predefined service, much like docker-compose services. It's possible then to define a service for every duplicated configuration and run them simultaneously on each of the nodes responsible for that action. In doing so we create a network of N managers, where N is the number of garden configurations, and Y nodes, where Y is the number of machines utilizing that configuration for each node respectively. Swarm implements a consensus algorithm, Raft, by which the group is capable of surviving (N-1)/2 manager losses, where N is the current number of managers. This modified architecture would allow Victor to main-

tain a consistent internal state of the entire swarm and all the services running on it

Regardless of future direction, Docker is already facilitating huge gains in scalability and continuos integration. Firstly, the amount of downtime in the garden system is incredibly low. In the case of failure, loss of power, or standard reboot the containers are set to restart automatically. In the case of unexpected stoppage the Docker daemon attempts to restart the container until it succeeds. If restarting continues to fail the daemon keeps trying to start the container using an incremental back-off algorithm. The garden architecture is fault tolerant, so regardless of the number of sensors a maintainer chooses to interface with, a single-failure will not compromise a deployment.

Second, the use of docker-compose promotes constant development and modification. Changes made to the `container-gardening` service can be propagated quickly and unobtrusively. With a version of a configuration already running, a maintainer can remotely build the entire new set of Docker container that make up the compose file on the target machine, run each one of them, and shut down the previous version. This set up provides a three command deployment process that takes minutes to build and causes no downtime.

4.3.1 Organization

Machines on the garden's network are heterogenous by design but not by necessity. Despite a varying number and collection of services, each ma-

chine's processes are defined in exactly the same way. Each machine's docker-compose.yml file is a textual representation of its running services. Docker-compose.yml configurations can be consolidated a set collection of builds or tailored specifically to every garden machine, but the format will be identical regardless.

The organization of the fleet is defined by the organization at the system level. In the pursuit of quick development, easy deployment, and clean devices, none of the Raspberry Pis are required to directly host the code that will ultimately run on them.

Garden code is self-contained within a single directory and is comprised of scripts to interface with sensors, Dockerfiles to containerize the scripts, and Docker-compose files to automate the process of maintaining configurations.

The top level directory contains three directories used to separate these three. The Dockerfiles and sensor program directories are fully mirrored. Sensor code is separated by the parameter being measured. If multiple sensors are being used to measure the same parameter then they're each given their own directory within parameter's directory named after the sensor in contention. The Docker-compose files are separated into their own separate directories named by machine configuration and contain services named that match their code's repository name. Each of the services reference the relative path to the Dockerfile that matches their name.

The well-defined organization has a few strategic extensibility gains. First, adding a component is incredibly straight-forward because the three neces-

sary steps are highlighted by the directories contained within the top level directory. The configuration of existing machines can change very rapidly. Adding a service to a configuration is a small amount of change – especially if the sensor's code and Dockerfile has already been written. Lastly, this configuration allows for totally remote deployment. All of the path references are relative, so the docker-compose file can be run remotely. As I mentioned in the `implementation` chapter, the Dockerfiles each pull the entire `container-gardening` repository. Though this is wasteful, it keeps the organization and deployment incredibly simple.

4.4 Service Based Architecture

The sensor network is easily the most extensible part of the framework because in many ways it's the most integral. However, implementing a service based architecture allows for redundancy or change at the service level.

The API is an opinionated decision that can be substituted with a number of alternate choices. I decided REST was a suitable framework because JSON is relatively human readable compared to XML and nearly every internet connected device has the capability of sending JSON data over HTTP, but ultimately the framework needs only a way of storing and retrieving data. We could conceivably include a stand alone database on either the garden network or front-end service.

Furthermore, using a web application as the front-end is largely arbitrary.

There are a huge number available mediums to display and manipulate data. Mobile or native apps, voice controlled skills via Alexa or Siri, or textual bots implemented within a tool like Slack are all totally viably and arguably more efficient.

These two services can be substituted entirely assuming they provide the same functionality, whereas the garden's configuration is largely tied to it's physical implementation, which is one of the main reasons that the API and front-end were delegated to third-party providers.

Victor as a framework provides an ideal platform for facilitating a secure and configurable IoT while limiting the amount of code necessary to make changes. Though a large amount of variability is expected and allowed within the garden's network, the other services limit active management by passing it off to third party providers. The framework is capable of adapting to a great deal of change, and should thrive by acting reliably regardless of the developer's modifications.

4.5 Closing Thoughts

The IoT ecosystem is likely to mature drastically in the next few years. The desire for a fully standardized market is unlikely, but as the market share consolidates it appears it may converge on a common means of communication and discovery. Victor was developed to adapt and conform to standardization as a model of how an ecosystem can be situated to provide necessary features

without stagnating the growth of deployment.

The organization and tooling that Victor is able to offer through secure, quick, and emerging technologies like Docker allows for constant development and deployment. Provisioning new devices is seamless, and the architecture of the application is cost-efficient and capable of scaling to as many or few devices as necessary.

The design provides a lot of value for networked, data acquisition focused deployments of IoT hardware.

Chapter 5

Conclusion

The Internet of Things (IoT) is expected to grow tremendously over the next decade, and I believe one of the driving causes of this is the direct and immediate benefit of scalable data aggregation and the analytics thereof. Gartner estimates that 43% of all businesses have already implemented some form of IoT.[1] IoT is already making an impact by increasing workflow efficiency and cost savings. The consumer market for IoT sees many of the same benefits and is also growing at an unprecedented rate with an expected $725 billion being spent on IoT in 2017.[2]

Though the cost of production for connected devices has been driven down over the past few years, two challenges still stand as obstacles for future growth – security and extensibility. Incidents like October 2016's huge Mirai distributed denial of service attack from a botnet of insecure IoT cameras and DVRs has

[1] http://www.gartner.com/newsroom/id/3236718

[2] http://www.pcworld.com/article/3167268/internet-of-things/as-iot-sales-surge-consumers-still-lead-the-way.html

consumers much more interested in the security of the devices they are purchasing.[3] The companies and developers of such IoT devices have yet to match the consumer's concern with a strong emphasis on device security. Furthermore, the market is shifting rapidly and the application of these devices are incredibly varied. A strong platform for extensibility, both in hardware and software, would mean that a collection of IoT devices would be able to grow with their application rather than serve a singular isolated purpose. Extensibility prevents a future of disposable IoT.

First and foremost, Victor was built to help me maintain a hydroponic garden. I cannot say that it accomplished that goal in the first two seasons that it has been running because I do not have any vegetables grown to prove it. As it turns out, growing food is still hard regardless of how much data you have about your garden. However, Victor is one way in which I can iterate on my failures in a quantifiable way. The conditions of the garden and their results can be catalogued and referenced during future seasons meaning ultimately, given enough time, I will know the optimal environment necessary to grow productive, healthy vegetables.

Furthermore, Victor acts as one example framework built with the current issues in IoT in mind. Using current, light-weight virtualization tools, Victor makes extensibility simple to develop and deploy. However, this is only feasible if the IoT device or platform is running on a microcomputer. Likewise, virtualized encapsulation, sane device hardening, and protected communication among the entities of a service based architecture facilitate a strongly secured IoT environment without compromising usability or functionality.

[3] http://www.networkworld.com/article/3135270/security/fridays-ddos-attack-came-from-100000-infected-devices.html

The future applications of IoT devices are vast and constantly growing. I think it is important for the market to adopt a framework and architecture for large scale networks of interconnected devices for the sake of the security of the network and the feasibility of long-term, large-scale adoption. Though I do not argue that Victor is the ultimate solution to the problems faced by IoT, it has solved many of those issues for my garden installation, and I hope I have the lettuce to show for it as soon as I can.

Chapter 6

References