IoT Platform Orchestration and Management Tool

MyT Platform

Semester 2 Report

Daemon Macklin

20075689

Supervisor: Joe Daly

BSc (Hons) in The Internet of Things

MyT Platform

Abstract

The Internet of Things(IoT) is a relatively new idea which involves a network of smart devices which share data. IoT is being used in a wide range of applications, such as the Smart home, Smart Agriculture and other fields. All of these applications have the same core ideas. Collect data, Analyse data and Store data, typically this is all done in the cloud, with services such as Wia, Wyliodrin or AWS IoT. MyT aims to find a new way to deploy and manage IoT services in the cloud. By giving the user control of their own IoT platforms at design, creation and deployment so they get the IoT application that best suits them.

Introduction

MyT - What is it?

MyT is a cloud agnostic orchestration and management tool which allows users to design an IoT platform and then deploy it on a cloud service or server of their choice. Allowing the user to customize some aspects of their platform. For example different database solutions offer different features and are typically designed for different data. If a user has very structured data they will be more suitable for a relational database. On the other hand, if the user does not have structured data, or does not know what data they will be collecting they would be more suited to a NoSQL solution. This will provide an IoT platform that will suit the needs of the user. The MyT application will be a front end to manage the users platforms. Automating parts of maintaining your own IoT stack, allowing the user to focus on the rest of their IoT system.

Why would you use it?

The issue of data ownership is becoming a major concern in modern society. Especially in technology, many companies allow users to utilize their services for “free”. However data they collect is worth more than most users realize. The practice of selling user data is very common, so it is hard to tell who to trust. A solution to that is to trust no one. Other IoT platforms MyT aims to compete with like Wia technically own data you upload to their service and could potentially sell or view that data with little consequence. From the point of view of a person or company who wants to monitor systems and produce sensitive data, owning and being able to confidently secure or even delete that data is vital. By design MyT will put the user in control of their data, so that they have complete ownership of it and can be sure if the data has been deleted it is gone forever.

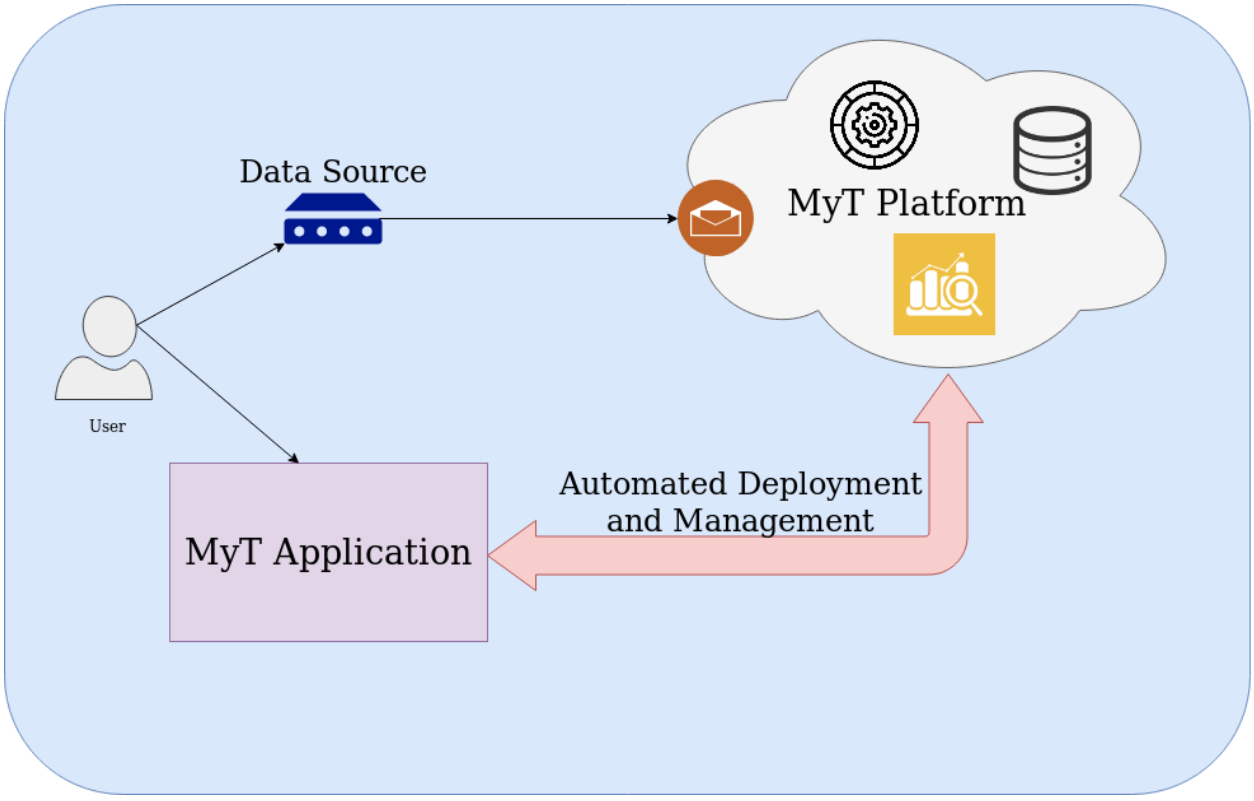
The second motivator is that when you use an IoT platform like Wia for your IoT application you are relying on their service to be working and fully functional at all times. Not only that if the IoT platform you are using goes out of business your IoT application goes with it. MyT aims to make it easy to cut out this third party application. This will give the user control over their IoT platform. Being able to directly manage their platform or fix issues if they occur or blow it up and upload your data to a completely new MyT platform.

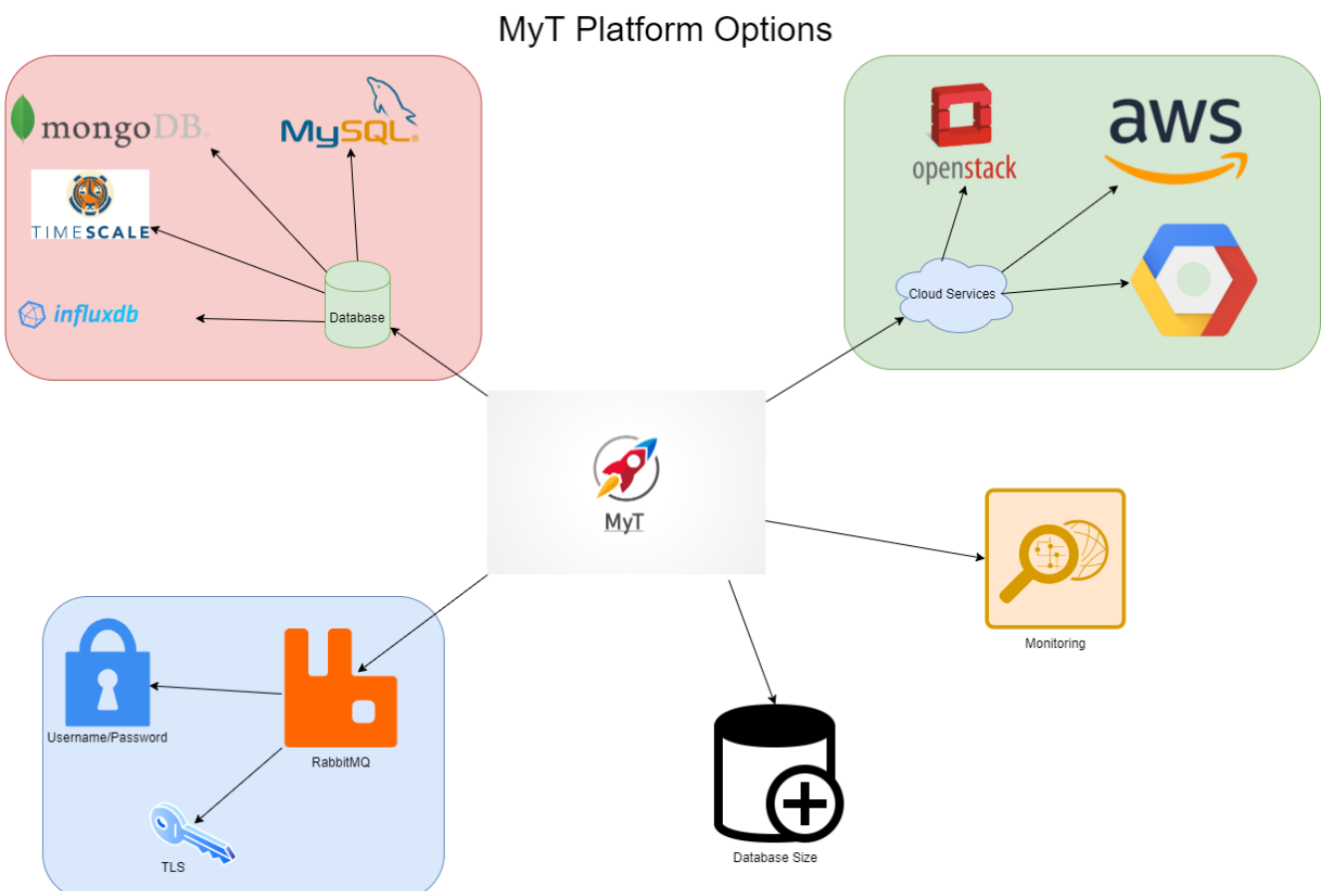
How will it work?

MyT has a number of moving parts. The first thing the user will see is the MyT application. This will be the portal where the user can create, delete and manage their IoT platforms. This front end must be always available to the user, so they always have control of their platforms. This application will allow the users to run scripts which will go off and manage their infrastructure. There are a number of tools that could be used for this process such as Cloud-Formation, Terraform, Ansible, etc.

When the user wants to create an IoT platform they will first pick out the technologies they want to use, pick their cloud service provider and hit launch. MyT will then use a combination of orchestration and management tools to create the users desired infrastructure.

Once the user has created their platform. MyT will continue to monitor it, and give the user details about it. When the user wants to take the stack down, MyT will be able to use the same combination of tools it used to create it to terminate the services.





Getting it to work

The end goal of the project is to have a service that;

1. Can easily and reliably deploy a customised IoT platform.
2. Keep track of the users platforms.
3. Interact with the platforms; update data processing scripts, downloading a database dump and tearing down a platform.

As well as having an Iot Platform which;

1. Take in user data.
2. Host users data processing scripts.
3. Store user data.
4. Provide a graphing platform for users to analyze data.

This report will go through the development and results of the project.

Semester 1 Report

https://docs.google.com/document/d/1o_p90dCqxdXoYkrqOSAYQs2H05MpincbRqKlk4ZSh78/edit?usp=sharing

Git Repos

Vue Frontend:

https://github.com/Daemon-Macklin/MyTFrontEndService

Flask Backend:

https://github.com/Daemon-Macklin/MyTBackendService

MyT Platform:

https://github.com/Daemon-Macklin/MyTPlatformDataProcessingContainer

Example Usage

Use Case

Randy Johnson is a web designer. He is very interested in monitoring the temperature in the different rooms in his house. Randy investigates a number of Iot platforms such as Wia and Wyliodrin. Randy likes the simplicity of the platforms but does not like the basic graphing functionality that each service provides. Randy wants to be able to create complex graphs which overlay different sensor readings.

After further research Randy comes across MyT on GitHub. He reads the details of the application and likes the idea of being in complete control of his data. Being able to write complex data processing scripts natively. Most importantly for him he is drawn to the graphs he can create with Grafana. He signs up for an AWS account and installs the application on his laptop.

Randy sets up his AWS credentials on his account and goes through the installation process. He chooses to use a MongoDB as he is not too sure about the data he is going to be collecting. He presses install and his platform is created on his AWS account. The application gives him the MQTT endpoint to send his data to.

Randy then starts to set up his devices to take ambient temperature data and send it to the MQTT endpoint given to him from the MyT application. He then discovers a slight problem. The temperature data is just the raw resistance value instead of the actual temperature value. Instead of rewriting the code on each of his devices to fix this. He decides to write some Python code to convert the data on the MyT platform. The MyT application gives Randy the template code to use. He adds his data conversion code to the template and uploads it via his MyT application to his platform. Now raw data sent to the platform is being converted to actual temperature data and that is stored. Randy then spends the rest of his evening creating the graphs he wanted on his grafana instance.

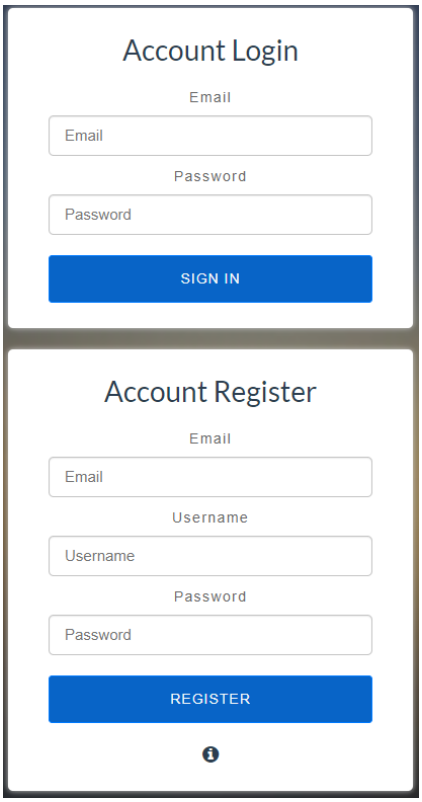
Using MyT

A full walkthrough of using MyT will be available on youtube:

https://youtu.be/sW3mF0uK5Hc

This section will go through it in a little more detail.

Creating an account

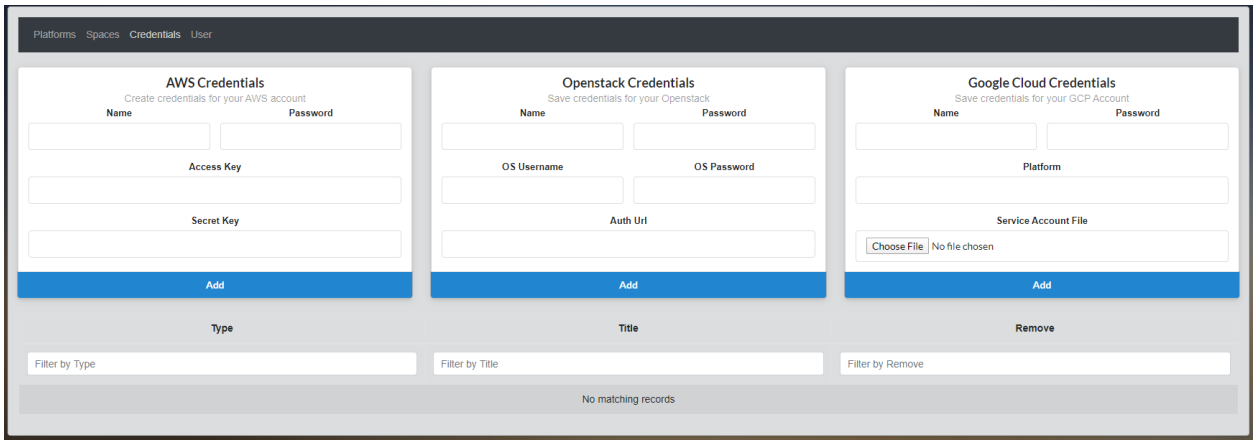


When first going to the MyT website the user will be prompted with a login screen. The user will need to register an account before they use the service.

All the user will need for an account is a username, an email and a password. Once the user is logged in they can manage their IoT platforms.

Credentials - Creating

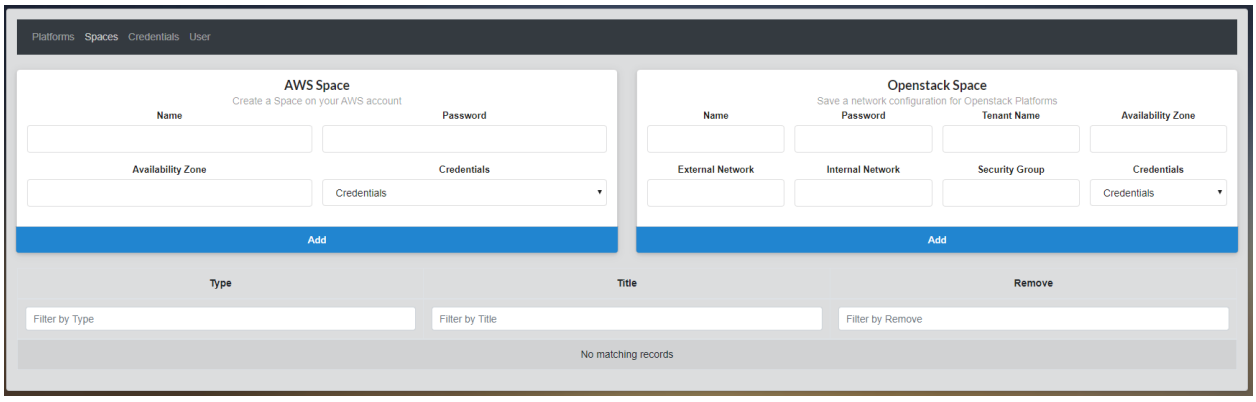
Before deploying anything, Randy will need to set up their cloud service credentials.



The credentials page is where a user can set up their credentials. Each of the different cloud services have their own credentials requirements. For this example Randy will be using AWS so all he needs is his Access Key and Secret Key he can get from AWS. Once added he can move on to a Space.

Spaces - Launching

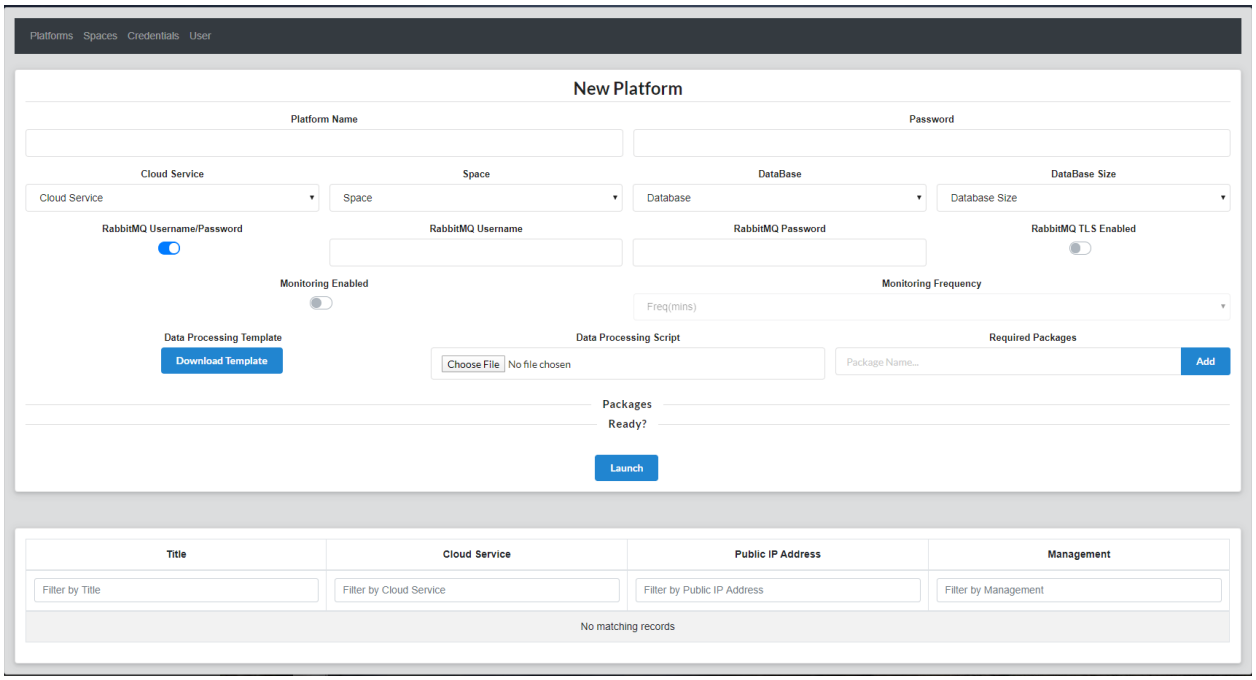
A Space is a network configuration for MyT platforms. Only required for AWS or Openstack platforms, they are used to create a VPC on AWS or just save networking information for Openstack.



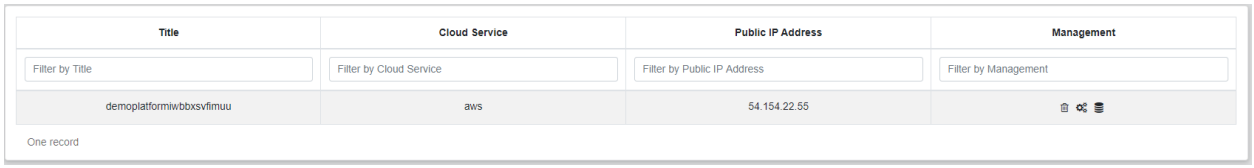
Once the user has created a space they can place multiple platforms in that space. Randy will create a AWS Space and this will create a VPC on AWS for him to put his platform.

Platforms - Launching

With all the prerequisites met the user can now create a platform.

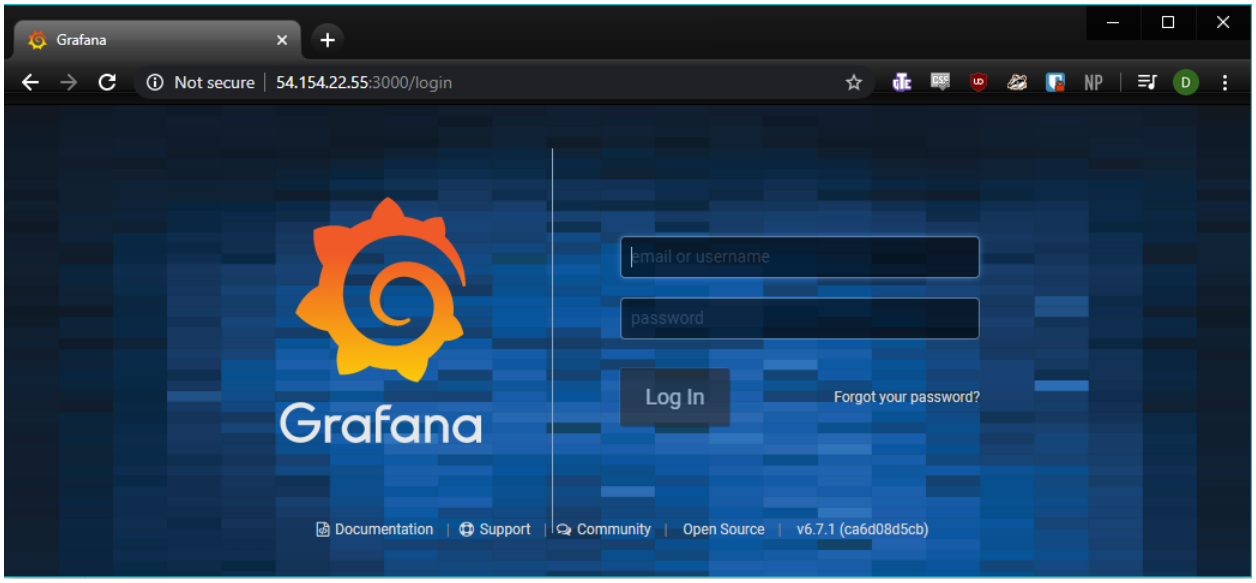


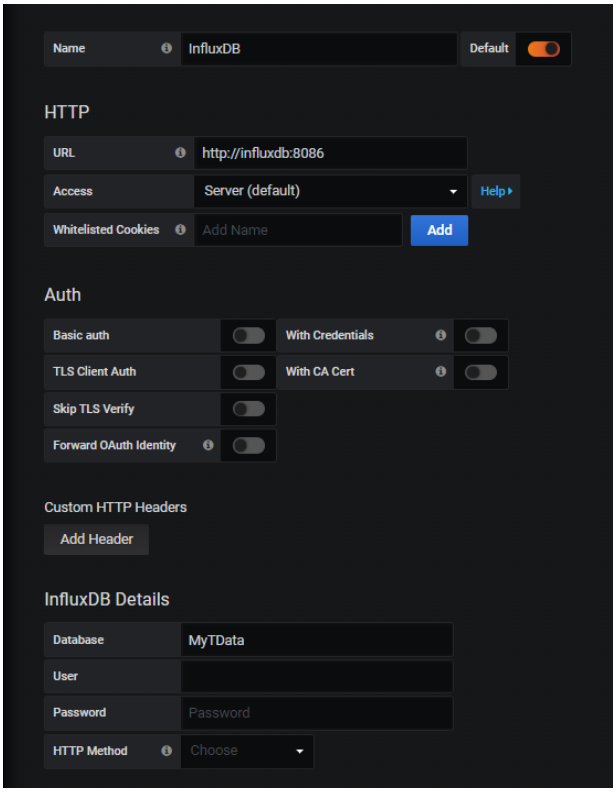
This page the user will fill out the form with all of their options. Ranging from cloud service to their data processing script and vary between cloud services. When the user is satisfied with their choices they press “Launch” and after a couple of minutes they will have a working platform. Once the platform has been created it will appear in the platform list.



Grafana

Going to the IP address on port 3000 e.g (http://54.154.22.55:3000/) will take the user to the Grafana login page.





Using the grafana default username and password you can login. Once logged in all the user must do before creating graphs is adding the data source.

The database URL is obtained from the database name, “db” and it’s default port. In this example Influx is used so the URL is simply; http://influxdb:8086.

The database name is always MyTData. Once the database setup is complete the user will be able to start creating their graphs.

RabbitMQ

The user will need to use some sort of RabbitMQ client to send data to their platform. A large number of languages have RabbitMQ packages. This example uses the pika Python package (pika, 2020).

```python
import pika
import sys
import json
import ssl
# Putting in Username or Password
#creds = pika.PlainCredentials(username='user', password='pass')

# Setting up SSL credentials
#context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
#context.load_verify_locations("cacert.pem")
#context.load_cert_chain(certfile="cert.pem", keyfile="key.pem")
#ssl_options = pika.SSLOptions(context)

connection = pika.BlockingConnection(pika.ConnectionParameters(host="54.154.22.55"))
channel = connection.channel()
channel.exchange_declare(exchange='data', exchange_type='fanout')

message = '{"measurement": "sensorData","sensor": "pi1", "data" : { "temp" : 22, "pressure" : 15, "device" : "pi1"}}'
message = json.loads(message)

channel.basic_publish(exchange='data', routing_key='', body=json.dumps(message))
connection.close()
```

With a simple bit of code like this a use can send data to their platform. The commented out code is more advanced features that are not required. Randy will add this bit of code to his project and replace the message with data from his sensors.

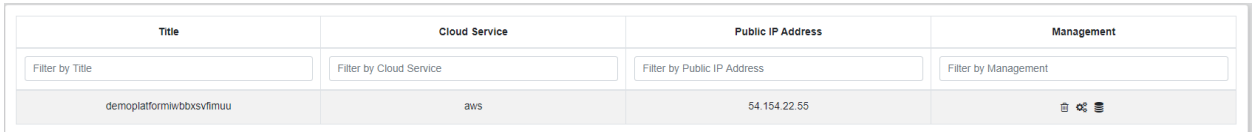
Data Processing

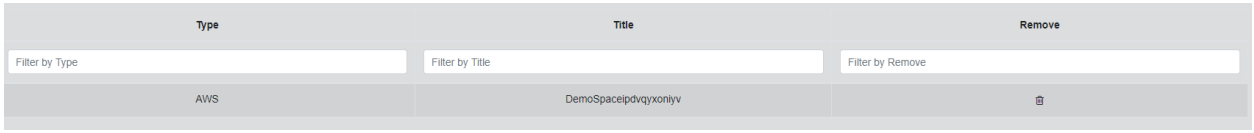
A data processing script can be added to a platform. This process includes adding required Python3 packages. The user supplies the file and the list of packages and the script based on a template. Then the script will run whenever data is uploaded to the platform.

The script can also be updated on existing platforms; this is done in the platform list by pressing the cog icon. Supplying the new file and required packages then the data processing script on the platform is updated. A database dump can also be gotten from the platform list by pressing the database icon. The user will then get a zipped folder containing their data.

Removing items

When the user is finished with their infrastructure they can delete it with the press of one button. Each of the Platform, Space and Credentials pages have a list of all of the users items. By pressing the trash can icon the user can delete items.





Project Environment

Development Environment

PyCham

Most of the development of the MyT Project was done with the JetBrains tool PyCharm. PyCharm is primarily a Python IDE, but with a few plugins it is easy to develop the Terraform scripts and Ansible playbooks all in one IDE. This made the development of the backend easy as I did not have to constantly be switching between programs while developing.

PyCharm also has a number of very useful features to make it easy to implement a continuous Integration system. Since the machines used for development were windows they are not able to run the Ansible and Terraform scripts locally, which in turn meant that they couldn't run the backend service. It was then decided that the application would be run on a ubuntu1804 docker container. Using a virtual machine on the TSSG openstack, I created a docker runner. A Docker file was written for the backend service, something that was already planned. Then using a PyCharm run template and a small bit of server configuration setup a pipeline so whenever the application was run in PyCharm the program would automatically be built and launched in a Docker container on the server, fully deployed and accessible on the internet.

WebStorm

Webstorm was used for the Frontend development. Writing the code for the VueJS component of the project. Webstorm is also a JetBrains product that comes with all of the features that PyCharm has. As a result I was able to use the same "Docker runner" system that I did with the backend service, for a continuous integration of the frontend.

Cloud Services

The aim of this project was to create a cloud agnostic IoT platform. As a result I needed to use a number of different cloud services. I started development with the TSSG openstack, as it was free for me to use, then moved on to Amazon Web services and Google Cloud, using student accounts with free credits for both.

The TSSG openstack was graciously made available to me by the people in the Infrastructure group, whom I did work placement with last year. It was not only used by the application to deploy platforms, but It was where I deployed my Docker runner service and other application demos.

Cloud Storage/Version Control

GitHub

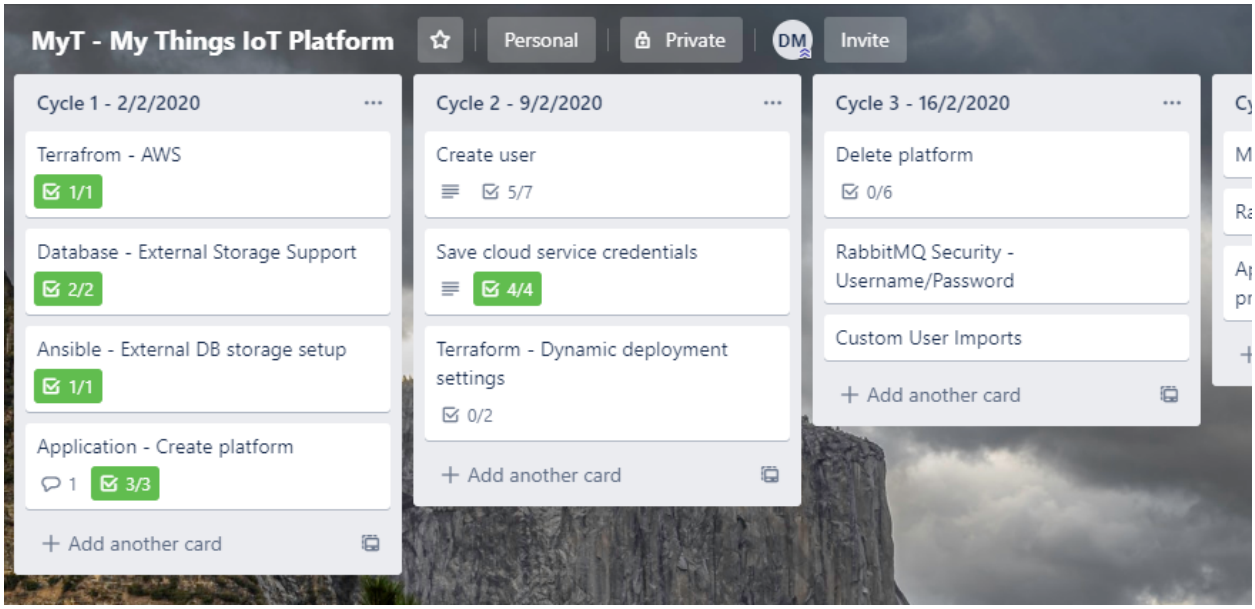
GitHub was used extensively in this project as the main version control. Using the development model (nvie.com, 2020), made it easy to keep track of progress and maintain the code base. Especially when I made a mistake and had to roll back to an earlier version of the project having a number of feature branches makes it easy to roll back and keep moving forward.

The development model was used on all of the projects repositories git repos with the exception of the MyT Platform: repo which was used to create the MyT application. This did not follow as all of the features had to be on the master branch so that the MyT deployment script always uses the most up to date version of the application.

Project Management and Development

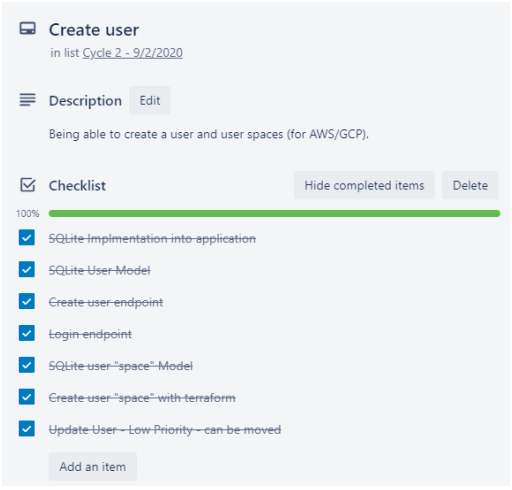
Trello

Trello (Trello, 2020) is a web-based Kanban-style task management application. Trello made it really easy to plan out the development of the project and monitor progress.



Iterative Cycles

Before being able to start development of the project, the project was broken down into smaller achievable goals for each week, following the iterative development methodology (Beck, K., et al, 2001).



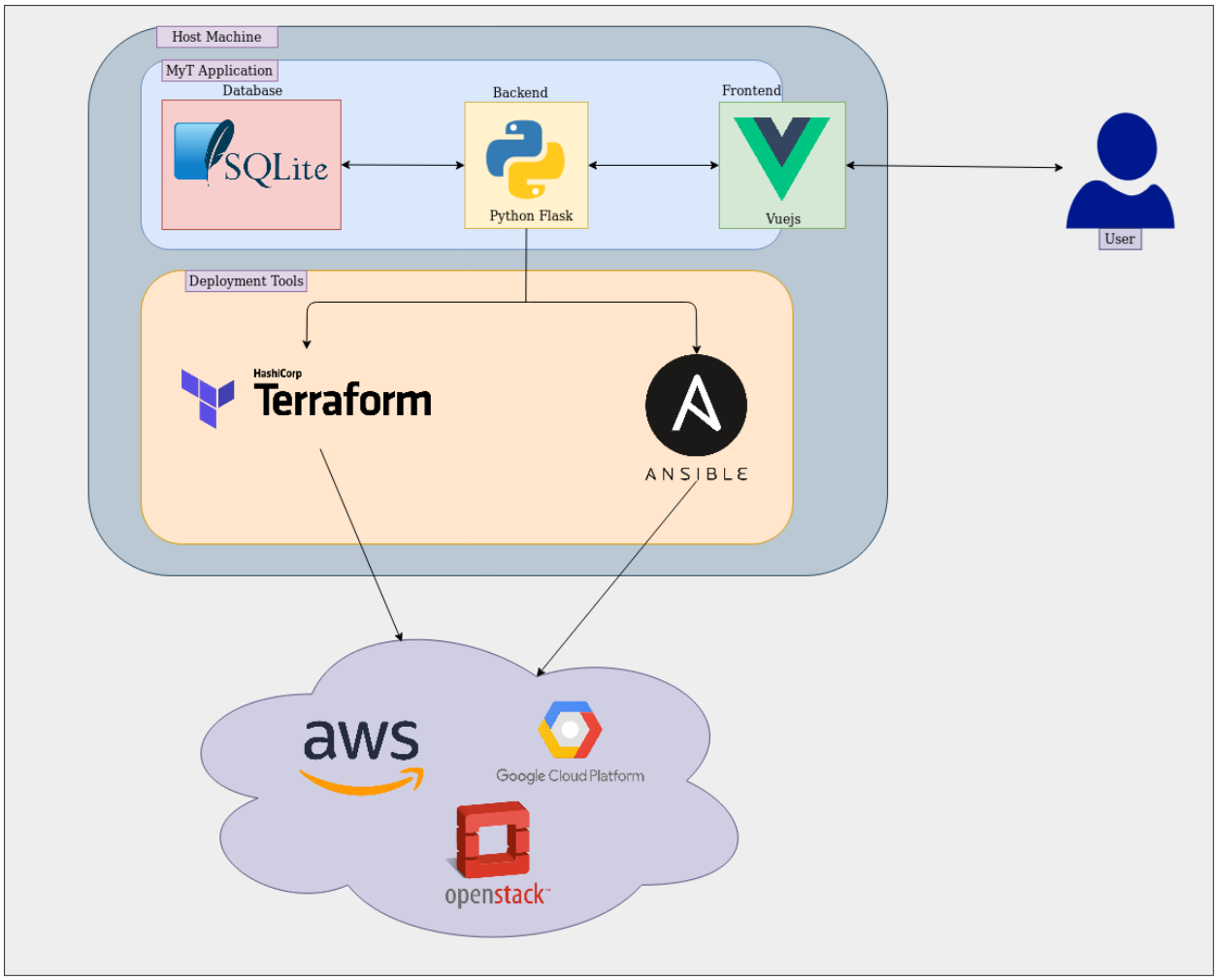
Dividing the development into 8 weekly cycles with 2-4 features each week. Each feature was further broken down into a number of tasks. This gave the project a clear pathway to completion.

Trello was also used to track issues. Any issue that occured was added to an Issues card. Making it easy to go back and fix issues without forgetting about them. When an issue was fixed it was moved to the Fixed Issues Card.

Implementation

Application

Architecture



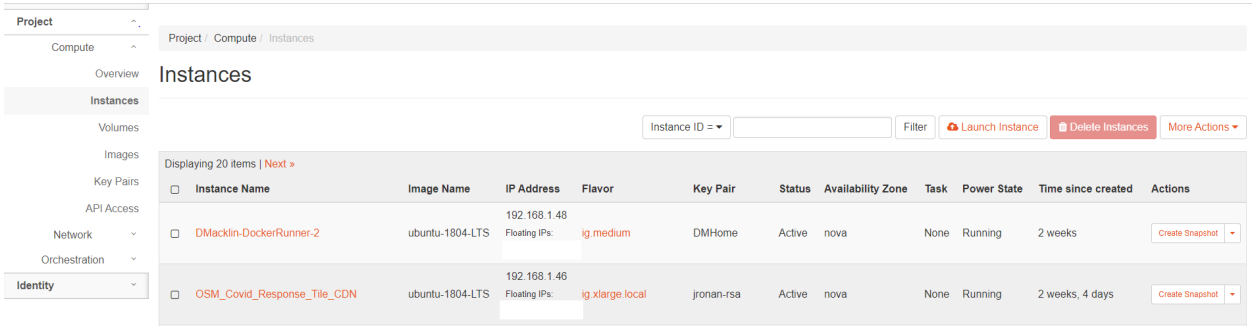
The MyT application is a web application using a Vue Frontend and a Flask backend which serves as a tool which the user will input data and be given a IoT platform.

Vue Frontend

As early as the design stage of the MyT project, the frontend was not a major feature of the project. However this does not mean that it was not important to have an aesthetically pleasing frontend which allows users to easily and securely interact with the backend and their infrastructure.

Designing the Frontend

The main idea of the project and the application is to give a user a space where they can easily create and manage infrastructure. Before designing a layout it was important to look at how similar applications look in terms of a user experience. The MyT frontend took inspiration from the Openstack Dashboard.

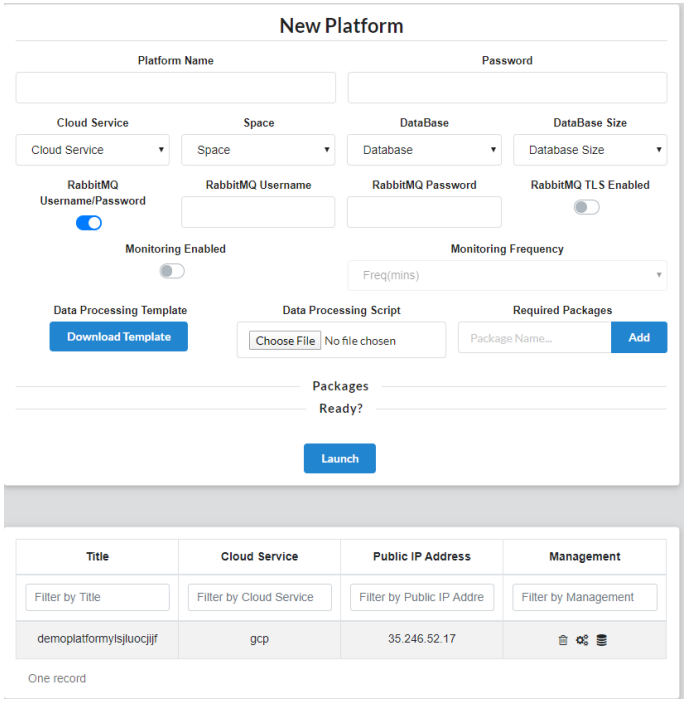


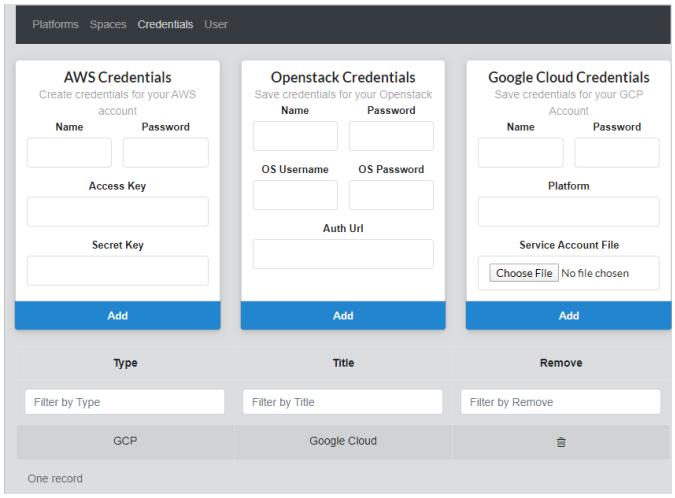
Openstack makes it easy for users to easily get information on their virtual machines at a glance. The most useful information such as IP address and Key Pair are shown up front. This is the user experience that MyT attempts to replicate. Making it easy for users to get the information they need.

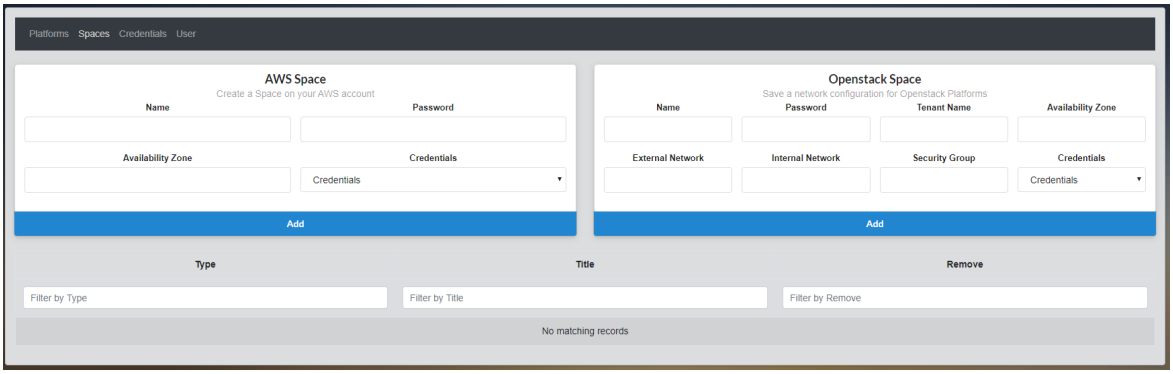
Openstack's simple and easy to use UI succeeds in minimizing how much time users spend using their application and giving them more time to use the virtual machines they deployed. This is also a concept the MyT frontend design wants to emulate. Requiring as little effort and time as possible from the users to deploy their IoT infrastructure.

Looking at each of the Platform, Space and Credentials pages on MyT shows that it is easy for users to get information on their infrastructure without having to go through a confusing maze of menus. Having the most important information the IP address and Cloud Service easy for users to find.

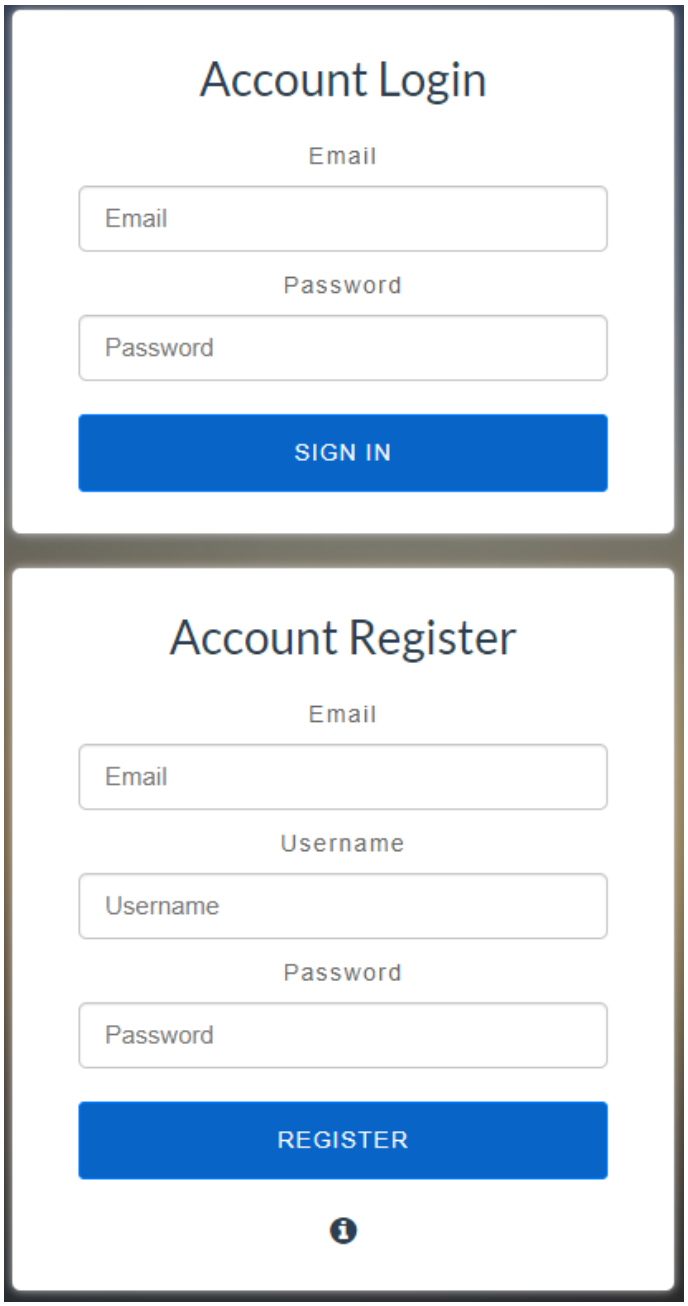
The design also makes it quick to get an IoT platform deployed. The users only have to fill out a minimum of 14 fields across three pages, if using AWS with none of the additional options on their platform, before they can get their deployment started. But once the fields are filled out and they press launch they don't have to do anything but wait for their platform to be deployed. Making it quick and easy for them to get what they need from the application and progress with the work they want to do.







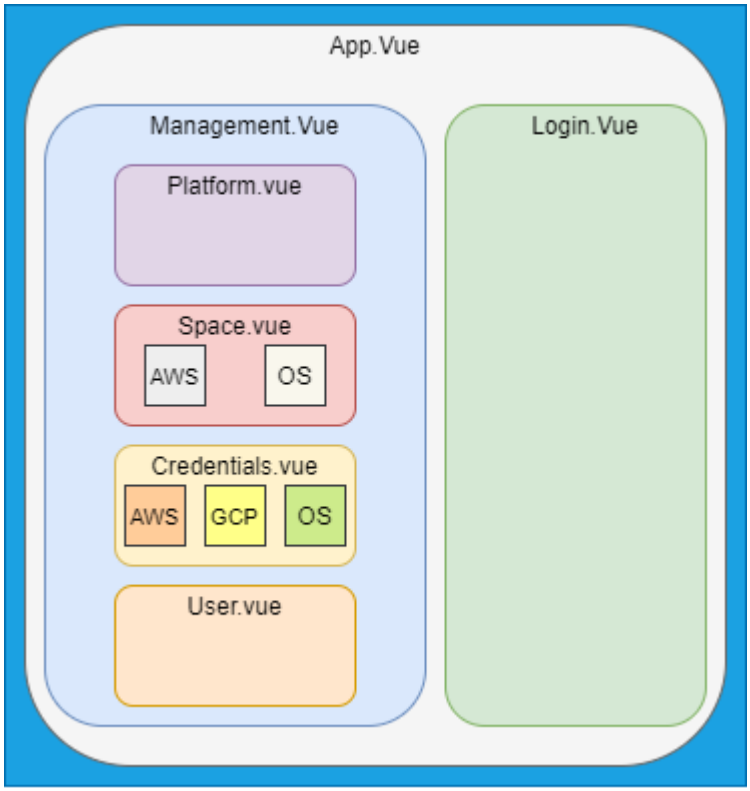
The login and register page also follows suit in the simple and easy to use schema.



Application Architecture and Components

Vue is designed for single page applications. Having one main page that components are injected into. This was advantageous for MyT for two reasons, firstly it will keep the MyT frontend simple. And secondly it will make it much easier to lock down the application from a security point of view. Only having one route means that there is no need for access control on certain pages because you can't skip past pages by going to a specific URI.

MyT has a number of components which are injected into each other. This is done to keep the amount of code in each file to a minimum to make the application easier to manage and build upon in future.



This diagram gives a good view into the architecture of the application. The code snippet shows how this is achieved in the main App.vue component. The v-if flag is used extensively in the MyT application.

```
</div>
  <Login v-if="signedIn === false"/>
  <Management v-if="signedIn === true"/>
  <FlashMessage></FlashMessage>
</div>
```

App.vue

The App component is the main component and only contains the code for the nav bar as it is the only part of the application that is persistent across the app. Past this is either the management or login components. This depends if there is a user signed in, the process for determining this is detailed in the security section.

Login.vue

The Login component is a small component which is used for logging in and registering an account. All the user needs for an account is a username, email and password. Due to the nature of the application it is crucial that MyT ensures the user has a secure password. So this component contains a check to see if the user has a valid password. A valid password contains one number, one symbol and has at least 7 characters, this is checked using a regular expression.

```
let passwordRegex = new RegExp( pattern: "^(?=.*[0-9])(?=.*[!@#$%^&*])(?=.{7,})")
```

Management.vue

The Management component is the main dashboard for the users, and is responsible for displaying the different sections such as the Platforms section. This is done using a navbar for the user to select which section they would like to view.

```
<b-navbar type="dark" variant="dark">
    <b-navbar-nav>
    <b-nav-item v-on:click="showSelection( page: 'platform')"> Platforms </b-nav-item>
    <b-nav-item v-on:click="showSelection( page: 'space')"> Spaces </b-nav-item>
    <b-nav-item v-on:click="showSelection( page: 'credentials')"> Credentials </b-nav-item>
    <b-nav-item v-on:click="showSelection( page: 'user')"> User </b-nav-item>
    <b-navbar-toggle target="nav-collapse"></b-navbar-toggle>
    </b-navbar-nav>
</b-navbar>
```

When the user presses on one of the buttons it runs the showSelection method which will change which of the components is shown below. Which is handled below.

```
showSelection(page){
    this.show = page
},
```

```
<br>
    <Platform v-bind:user="user" v-if="show==='platform'"/>
    <Space v-bind:user="user" v-if="show==='space'"/>
    <Credentials v-bind:user="user" v-if="show==='credentials'"/>
    <User v-bind:user="user" v-if="show==='user'"/>
</div>
```

The result of these is having a navbar for the users to decide which component they want to view with that component displayed below, making it easy for the user to navigate the application as well as easy for a future developer to add more components.

User.vue

The User component is a page that allows users to view their username and password as well as download the ssh-keys that MyT uses for their infrastructure. MyT allows users to download their keys whenever they want. But as this is sensitive data and MyT is targeted to users who mightn't be familiar with cyber security principles it is important to inform the user to keep their keys safe and not to share it.

Credentials.vue

```
<div class="ui three wide cards">
    <AwsForm v-bind:user="user" />
    <OSCredForm v-bind:user="user" />
    <GCPCredForm v-bind:user="user" />
</div>
```

The Credentials component is the page where users manage their credentials. They can create and remove credentials easily. However the Credentials page is mostly made up from other smaller components that make up the forms for the different types of credentials. One for each of the cloud services. This method of further breaking down components into smaller modules that can be added into larger components make the application easy to manage and build upon.

Space.vue

The Space component is where users manage their spaces. Similar to the credentials page it is made out of smaller components that make the forms for creating spaces.

Platform.vue

The Platform component is the largest of the components. As it needs to pull in a lot of data to allow the user to create their platforms and have them displayed. However this one is not broken down into smaller components. This is due to an oversight as to how many fields would be required as the number of cloud services and options increased. Despite this it is an attractive and functional web page.

CSS Framework and Tools

In order to create a modern and aesthetically pleasing web application in the limited development time, it was evident that I needed to use a framework and a number of smaller tools ontop of VueJs (Vue.js, 2020).

Semantic UI

Semantic UI (Semantic UI, 2020) is a lightweight framework that is used in MyT for formatting of the web pages, creating the forms and most of the components that did not require a lot of logic behind them. Semantic UI made it simple to create a modern and aesthetically pleasing UI without having to spend time tinkering with CSS.

Vue-login-form

The Vue login form (vue-login-form, 2020) module is a module that provides an easy way to make a login and register form without having to manually create the HTML. Just install the package and it can be used like so:

```
<div id="app">
        <login-form :data="loginFormData" @sign-in="signIn"/>
        <br>
        <login-form :data="registerFormData" @sign-in="register">
            <br>
            <i class="fa fa-info-circle fa-lg" aria-hidden="true"  v-b-popover.hover.top=
                    "'Passwords must be at least 7 characters long, contain at least 1 number and 1 symbol'"
               title="Password Specifications"></i>
        </login-form>
</div>
```

With this HTML plus a small bit of Javascript to define the inputs the MyT login and register page was complete.

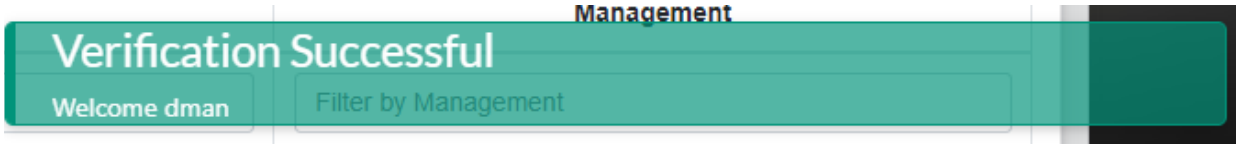
Vue-spinner

The Vue Spinner (vue-spinner, 2020) module is used to generate the loading spinners when waiting for infrastructure. Just import the package and use this html to get a nice loading spinner which is a small detail which adds a lot to make MyT look polished and responsive.


```
<div align="center" justify="center" v-if="loading">
    <RingLoader size="60px"></RingLoader>
</div>
```

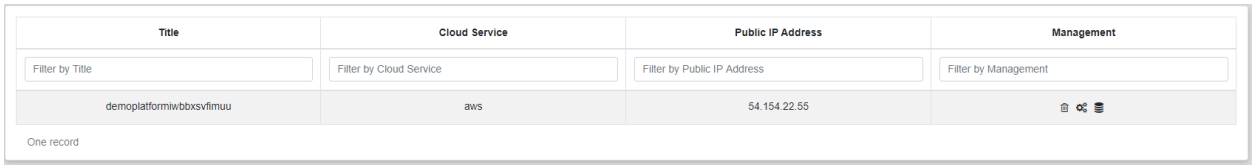
@smartweb/vue-flash-message

The Vue Flash Message (@smartweb/vue-flash-message, 2020) module is responsible for making pop-ups occur on the frontend application. These range from error messages to informing the user that their infrastructure is ready. Similar to the Vue-spinner module it is a small detail that makes MyT a much more responsive and user friendly application.



Vue-tables-2

The Vue Tables (vue-tables-2, 2020) module was used to easily display a list of users Platforms, Spaces and Credentials. It is a very useful plugin as it handles a lot for you. It provides filtering functionality right out of the box, which is always useful. Also allowing for buttons to be inserted into the rows making it easy to create a component with which users can easily view and manage their infrastructure.




```
<div class="ui raised segment">
    <v-client-table :columns="columns" :data="platforms" :options="options">
        <template slot="id" slot-scope="props">
            <i class="center aligned fa fa-trash-o" style="padding-right: 5px" v-on:click="removePlatform(props.row.id)"></i>
            <i class="center aligned fa fa-cogs" style="padding: 5px" v-on:click="updateProcessing(props.row.id)"></i>
            <i class="center aligned fa fa-database" style="padding: 5px" v-on:click="getDump(props.row.id)"></i>
        </template>
    </v-client-table>
</div>
```

Connecting to the Backend

The Vue application serves only to be a place where the user can interact with the actual MyT flask application. This is done using HTTP (RFC 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2), 2020). There are many NPM modules that will do this. I chose to use Axios (axios, 2020), simply because I was familiar with it and it ticks all of the necessary boxes in terms of functionality.

Using Axios makes it easy to implement HTTP requests into the design pattern. For example on the User page MyT allows the user to download their public and private ssh-key that MyT generated. To get this the frontend must get the key from an endpoint on the backend. To implement this using Axios we first define the URL to the backend service in *"api.js"*.

```
export default() => {
    return axios.create({
        baseURL: 'http://87.44.16.158:5000/v1'
    })
}
```

All other routes will now use this URL as the base of the URI. So the next step is to define the specific URIs in *"mytservice.js"*. We wrap these in functions that have arguments so we can pass necessary data into them.

This example shows how the, get user ssh-keys endpoint it accessed. Passing in the users id and token as well as the user supplied password. Then the response is returned and handled after the function call, as shown on the next page.

```
getKeys(uid, password, token){
    let json = {"password": password}
    return Api().post( url: "/users/sshKey/"+uid, json,
        config: {headers: {'Authorization': 'Bearer '+token}})
},
```

```
getKey() {
    if(this.key_password === null){
        this.flashMessage.error({title: 'Authentication Error', message: "Password is required to get Key"});
    } else {
        let access_token = this.$cookies.get('access_token')
        mytservice.getKeys(this.user.uid, this.key_password, access_token).then(
            response => {
                this.flashMessage.warning({title: 'Key received', message: 'BE SURE NOT TO SHOW THIS TO ANYONE'})
                let blob = new Blob( blobParts: [response.data.privateKey.toString() + "\n" + response.data.publicKey.toString()], options: {type: "text/plain;charset=utf-8"})
                console.log(blob)
                try {
                    saveAs(blob, this.user.username + "-key.txt")
                } catch (e) {
                    console.log(e)
                }
                this.key_password = null
            }
        ).catch(
            error => {
                console.log(error)
                if (error.response.status === 401) {
                    this.flashMessage.error({title: 'Error', message: error.response.data.msg});
                    this.$parent.$parent.isSignedIn()
                }
                else if(error.response.status === 400){
                    this.flashMessage.error({title: 'Error', message: error.response.data.errors.message});
                }
            }
        )
    }
}
```

This is how the functions like getKeys are called. The data required from the user is checked and validated in this case we check if the user has input a password. Then any data stored in cookies is accessed in this case it is the Json web token, used to authenticate the user. Once all of the data is prepared we can call the function. The *".then"* and *".catch"* flags are used to differentiate what should happen if a successful response is returned in the case of *".then"* or an unsuccessful response *".catch"*. This ensures that regardless of what happens there is a response for the user.

Data and Security

Cookies

Cookies (RFC 6265 - HTTP State Management Mechanism, 2020) are an integral part of modern website design. Providing an easy way to manage session data for users. The MyT frontend uses cookies to store the users data and their web token. This is done so that the user does not have to constantly log in and out or provide their user id or other data, except their password. Vue-cookies (vue-cookies, 2020) is used to handle the use of cookies.

Name	Value	Do...	P...	Expires / ...	Size	Http...	S...	Sam...
access_token	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1...	loca...	/	2020-04-0...	287			
email	daemon1%40gmail.com	loca...	/	2020-04-0...	24			
refresh_token	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1...	loca...	/	2020-04-0...	270			
uid	891ed894-37b9-4b48-9175-53d18...	loca...	/	2020-04-0...	39			
username	dman	loca...	/	2020-04-0...	12			

Security

Security is a major part of this project, and as a result MyT must take precautions with the frontend. The first thing that becomes clear is that the user is required to input their password for just about everything. This is done as it is required in the backend to encrypt and decrypt the users data. The password could easily have been stored in the cookies along with the rest of the users data. But since the user information stored in MyT is very sensitive, it makes sense to sacrifice a small bit of expediency for an extra level of security.

The next level of security is the use of Json web tokens to further secure endpoints. When the user first logs in they are given an access token and a refresh token. The access token is used whenever a user requests data from the backend service, and will be made invalid after 15 minutes. Then the next time the user requests anything from the backend. The response returns an authentication error and this will prompt MyT to automatically use the users refresh token to generate a new access token. The refresh token itself will become invalid after 30 days.

Flask Backend

The backend service is the central piece of the MyT application. It is essentially a synthetic systems administrator, managing all of the users data and infrastructure. As per the spec of the project the backend needed to be lightweight and be able to easily interact with Terraform and Ansible. It was decided to use Python Flask as it met all of the requirements and meant being able to use Python for development.

Flask Basics

Python Flask (Welcome to Flask — Flask Documentation (1.1.x), 2020) is a web framework written in Python. One of the main advantages to using Flask is that there is little boilerplate code that would be found in a Node application. Starting the application is very simple.

```python
# Create app
def createApp():
    app = Flask(__name__)
    app.config.from_pyfile('app/config.py')
    CORS(app, resources={r"/*": {"origins": "*"}})

    jwt = JWTManager(app)

    for bluePrint in EXPORT_BLUEPRINTS:
        app.register_blueprint(bluePrint)

    db.init()

    return app


app = createApp()

if __name__ == '__main__':
    app.run(host="0.0.0.0", debug=True)
```

Running this code will first run the *"createApp"* function. This function creates a new Flask instance, loads the config file, sets up cross-origin resource sharing, sets up json web tokens, loads all of the routes, known as blueprints and lastly initializes the sqlite database. Once all of this is done the application is run, using the host of *"0.0.0.0"*. This is done so that when deployed the application can be accessed by all IP addresses.

Routes

Creating new routes is a simple process. The first step is to define a blueprint. A blueprint is an application component which contains a number of routes which should revolve around the same function. For example this is how the users application component is setup;

```
users = Blueprint('users', __name__, url_prefix=URL_PREFIX)
```

This line of code defines the blueprint in the *users.py* file.

```
EXPORT_BLUEPRINTS = [platform_crud, users, spaces, credentials]
```

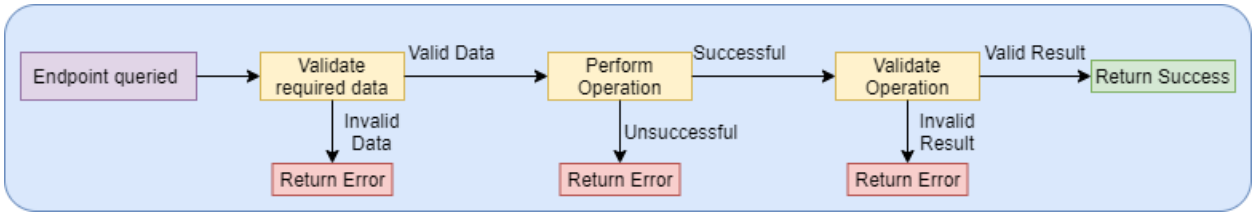
Then we create a list of them in the *__init__.py* that are to be used in the app when it is created. Once these steps are done we can define new routes like so;

```
@users.route('users/create', methods=["Post"])
def createUser():
```

When the endpoint *"users/create"* is queried then the function "*createUser*" will be run.

Handling Requests

When writing a function to be run when an endpoint is queried MyT has a design pattern that is adhered to.



This design pattern ensures that whatever happens that the backend returns a meaningful response. Which makes it easy to figure out where issues are. As well as preventing unnecessary crashes.

We can see this design pattern implemented into the login function.

```python
@users.route('users/login', methods=['Post'])
def login():
    data = request.json

    # Verify data
    if 'email' in data:
        email = data["email"]
    else:
        return Response.make_error_resp(msg="Email is required", code=400)

    if 'password' in data:
        password = data["password"]
    else:
        return Response.make_error_resp(msg="Password is required", code=400)
```

The login function requires an email and password. So we can check if the data is present in the request. In the event that the data is missing an error response is made informing the user what the issue is.

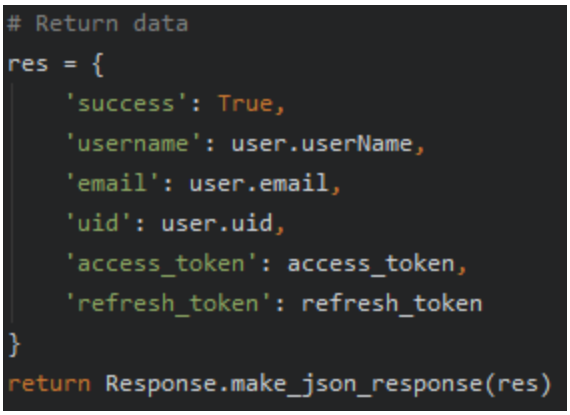
```python
    try:
        user = Users.get(Users.email == email)
    except Users.DoesNotExist:
        return Response.make_error_resp(msg="No User Found")
    except:
        return Response.make_error_resp(msg="Error reading database", code=500)

    # Verify password
    if pbkdf2_sha256.verify(password, user.password):

        # Generate User tokens
        access_token = create_access_token(identity=user.userName)
        refresh_token = create_refresh_token(identity=user.userName)
```

```python
    else:
        return Response.make_error_resp(msg="Password Incorrect", code=400)
```

Next we perform the operation. In this instance we search the database for users with the supplied email. And in the event no email is found we return with a meaningful error message. In the event of a success we move on to the next part of the operation, this being password verification. Again we prepare for an event where the password is correct and one where it is incorrect. Once we are finished with the operation we move on to the final stage in the design pattern.

The last thing we do is return the relevant data. In this case we return the user data and the generated tokens.

There are a number of more complex routes, but they still conform to this design pattern.



```
# Return data
res = {
    'success': True,
    'username': user.userName,
    'email': user.email,
    'uid': user.uid,
    'access_token': access_token,
    'refresh_token': refresh_token
}
return Response.make_json_response(res)
```

Response

There are a number of ways to return data from a Flask application. The simplest way is just to use *"return 'string'".* However this method is error prone and not suitable for more complex Json objects. As a result it was important to write a number of functions which will handle returning data to the user. Functions in the *"response.py"* file are used throughout the application to handle the different situations when returning data.

Managing Deployments

The main feature of this project and the main purpose of the MyT application is to act as a wrapper for Terraform and Ansible in order to make it much easier for inexperienced users to deploy IoT infrastructure. So most of the functionality performed by the application is configuring and running Terraform scripts and Ansible playbooks.

Terraform

Terraform (Terraform by HashiCorp, 2020) is an orchestration tool that MyT uses to create the infrastructure that MyT is deployed on. MyT uses the python-terraform (python-terraform, 2020) python package to run the scripts. However there are a number of steps that need to be taken before we can launch the Terraform scripts. Using the create platform as an example this section will detail the steps the Flask application takes before launching a Terraform script.

After all of the required data has been validated as per the handling requests design pattern the first step is to find which cloud service the users is deploying their platform on. This is important as there are a number of differences between the cloud services, and as a result the Flask application requires different data depending on which cloud service is in use. For example;

```
tfPath = ""
platformPath = ""
if cloudService == "aws":
    tfPath = "terraformScripts/createPlatform/aws"
    externalVolume = "/dev/nvme1n1"
    try:
        space = SpaceAWS.get((SpaceAWS.id == sid) & (SpaceAWS.uid == uid))
    except SpaceAWS.DoesNotExist:
        return Response.make_error_resp(msg="Error Finding Space", code=400)
    platformPath = os.path.join(space.dir, "platforms", safePlatformName)
```

An AWS platform requires a space, so when using it as a cloud service the MyT application must find the space that the user has chosen to deploy their application in.

```
    elif cloudService == "gcp":
        tfPath = "terraformScripts/createPlatform/gcp"
        externalVolume = "/dev/sdb"
        if 'zone' in data:
            zone = data['zone']
        else:
            return Response.make_error_resp(msg="Zone Required for GCP")

        if 'cid' in data:
            cid = data['cid']
        else:
            return Response.make_error_resp(msg="Credentials Required for GCP")

        try:
            creds = GCPCreds.get((GCPCreds.id == cid) & (GCPCreds.uid == uid))
        except SpaceOS.DoesNotExist:
            return Response.make_error_resp(msg="Error Finding Credentials", code=400)
        platformPath = os.path.join("gcp", "platforms", safePlatformName)
```

Contrast this to GCP, which does not require a space so it instead requires only the users GCP credentials as well as an availability zone, data which would otherwise be stored in a space for AWS or Openstack.

The next step in the process is for the application to make a new folder to put the terraform and ansible scripts. Each space has their own folder, in which contains their terraform scripts, ansible playbooks. And platforms that have been deployed to them, which themselves have their own terraform scripts and ansible playbooks. This is necessary because of how Terraform operates, which will be discussed in the Terraform section.

Creating this folder is done simply by copying the required terraform script into a new folder with platforms name.

```
try:
    shutil.copytree(tfPath, platformPath)
except FileExistsError as e:
    return Response.make_error_resp(msg="Platform Name already used", code=400)
```

The next step is to take the information supplied by the user and generate the Terraform Variables file. This will tell Terraform how, where and what to deploy for the user.

```
if cloudService == "aws":
    print(dbsize)
    varPath = awsGenVars(user, password, space, dbsize, safePlatformName, platformPath)
    if varPath == "Error Finding Creds":
        return Response.make_error_resp(msg="Error Finding Creds", code=400)
```

As there are different Terraform files for the different cloud services, their variables file have different requirements too.

```
# Function to generate variables file for aws platforms
def generateAWSPlatformVars(keyPairId, securityGroupId, subnetId, secretKey, accessKey, dbsize, platformName, platformPath):
    string = 'variable "key_pair_id"{\n  default = "' + keyPairId + '"\n }\n\n\
    variable "aws_secret_key"{\n  default = "' + secretKey + '"\n }\n\n\
    variable "aws_access_key" { \n  default = "' + accessKey + '"\n}\n\n\
    variable "security_group_id" { \n  default = "' +securityGroupId +'"\n}\n\n\
    variable "subnet_id" { \n  default = "' + subnetId + '"\n}\n\n\
    variable "db_size" { \n  default = "' + str(dbsize) + '"\n}\n\n\
    variable "platform_name" { \n default = "' + platformName + '" \n } \n'

    path = platformPath + "/variables.tf"

    f = open(path, "w")
    f.write(string)
    f.close()

    return path
```

This is how the Terraform variables files are generated. It is a simple process as Terraform does not use indentation so the number of new lines are mostly for human readability.

The next and final steps come after the Ansible setup detailed in the next section. These are the initialization and creation of the infrastructure.

```
# ------------Terraform Create------------#
initResultCode = tf.init(platformPath)

output, createResultCode = tf.create(platformPath)
```

This will run first the initialization function which will get terraform in install the necessary plugins and setup everything it needs to run the scripts. After which the create function is run.

```
# Function to run create infrastructure scripts
def create(pathToInf):

    # Initalize a terraform object
    # Make the working directory the openstack directory
    terra = Terraform(pathToInf)

    # Apply the IAC in the openstack directory and skip the planning Stage
    return_code, stdout, stderr = terra.apply(skip_plan=True)

    # Get the outputs from the apply
    outputs = terra.output()

    return outputs, return_code
```

This function takes in the path to the Terraform scripts be it for a platform or space then runs the apply command. Which will run the scripts and create the infrastructure. There is also a destroy function which operates in the same way, goes to the directory with the scripts and runs the command on the scripts.

An important idea which inspired Infrastructure as code solutions such as terraform is the ability to deploy infrastructure repeatedly and reliably. Most of the time the MyT terraform script executes without error, however there are some occasions when it fails, typically due to user error. As a result MyT needs to be able to handle these situations.

When Terraform returns a result before moving on MyT checks the result code which will indicate if it ran successfully. So when it runs unsuccessfully MyT will take action.

```
if createResultCode != 0:
    # Add destroy function here
    print("Removing Inf")
    tf.destroy(platformPath)
    shutil.rmtree(platformPath)
    return Response.make_error_resp(msg="Error Creating Infrastructure", code=400)
```

In any situation MyT will attempt to destroy the infrastructure even if none was created, Terraform will accept the command and destroy what was created. Then the folder created for that platform will be removed.

In the case of platforms, after Terraform is finished deploying the infrastructure we use the *"outputs"* variable to get the output data in the terraform script, in this example we are interested in the IP address of the new virtual machine.

Before we hand off the deployment to Ansible MyT will ensure that the Infrastructure is active. By pinging the IP address output from Terraform we can check if the Infrastructure is ready.

This function is called after the Terraform is run successfully. And when a ping returns successfully MyT moves on to Ansible.

```
def serverCheck(floating_ip):
    counter = 0
    isUp = False
    while counter < 12:
        response = os.system("ping -c 1 " + floating_ip)

        if response == 0:
            time.sleep(10)
            isUp = True
            break
        else:
            time.sleep(10)
            counter += 1

    return isUp
```

Ansible (Ansible, 2020) is a management tool that MyT uses to install and configure the MyT platform and all of it's requirements on the created infrastructure. There is no suitable python package to run the ansible scripts for MyT so it uses subprocess to run the commands on the shell. Just like Terraform, there are a number of steps that need to be taken to prepare for a deployment. Again using the create platform as an example.

Since Ansible is simply installing and configuring on a ubuntu machine it does not matter what cloud service is being used. However there is one small difference between the three cloud services and that is the default location for external volumes. MyT will automatically use the right location based on the chosen cloud service.

Since there are no major differences in the Ansible playbook between the different platforms we can just copy the playbook into the new platforms directory. This is done so the same playbook isn't constantly being edited and run.

```
shutil.copytree(createAnsibleFiles, ansiblePath)
```

Now that MyT has the files ready, next is to configure the user's data before launching the script.

```
if script:
    script.save(os.path.join(ansiblePath, "roles", "dmacklin.mytInstall", "templates", "dataProcessing.py"))

if dbFields:
    ab.createSQLInit(ansiblePath, dbFields, database)

ab.updateAnsiblePlaybookVars(cloudService, externalVolume, database, rabbitTLS, monitoring, monitoringFreq, ansiblePath)

ab.generateMyTConfig(rabbitUser, rabbitPass, rabbitTLS, database, ansiblePath)

ab.generateRequirementsFile(packages, ansiblePath, "dmacklin.mytInstall")
```

This bit of code is responsible for getting the ansible playbook ready for deployment. The first bit in the *"if script"* section is used to check if there is a custom data processing script that needs to be deployed. If so then the script is saved into the templates section of the *"dmacklin.mytInstall"* role.

The second if statement checks if there are any database fields, a feature when creating an SQL database. If there is an SQL init script will be created with the given fields.

The next section calls the *"updateAnsiblePlaybookVars"* function. This will edit the variables in the ansible playbook.

```
def updateAnsiblePlaybookVars(cloudService, externalVolume, database, rabbitTLS, monitoring, monitoringFreq, ansiblePath):

    monitoringFreq = "*/" + monitoringFreq

    with open(ansiblePath + "/installService.yml", 'r') as file:
        # read a list of lines into data
        data = file.readlines()

    # now change the 2nd line, note that you have to add a newline
    data[3] = "    cloudService: '" + cloudService + "'\n"
    data[4] = "    externalVol: '" + externalVolume + "'\n"
    data[5] = "    database: '" + database + "' \n"
    data[6] = "    rabbitmqTLS: '" + rabbitTLS + "' \n"
    data[7] = "    monitoring: '" + monitoring + "' \n"
    data[8] = "    monitoringFreq: '" + monitoringFreq + "' \n"

    # and write everything back
    with open(ansiblePath + "/installService.yml", 'w') as file:
        file.writelines(data)
```

After that, *"generateMyTConfig"* will set up the configuration of RabbitMQ, editing some of the variables in the Docker-compose script that will enable password or TLS. And generating a config file for the RabbitMQ python wrapper in the data processing container.

```
def generateMyTConfig(rabbitUser, rabbitPass, rabbitTLS, database, ansiblePath):

    configPath = os.path.join(ansiblePath, "roles", "dmacklin.mytInstall", "templates", "config.ini")
    # Create config parser object
    conf = configparser.ConfigParser()

    if rabbitUser != "" and rabbitPass != "":
        setRabbitmqComposeData(rabbitUser, rabbitPass, database, ansiblePath)

    if rabbitTLS == "true":
        enableRabbitMQTLS(database, ansiblePath)

    # Generate config file data
    conf['rabbitmq'] = {
        'user': rabbitUser,
        'password': rabbitPass,
        'tlsEnabled': rabbitTLS
    }

    # Write data to file
    with open(configPath, 'w') as configfile:
        conf.write(configfile)
```

Lastly the *"generateRequirementsFile"* will take the list of requirements supplied by the user and generate a requirements.txt file to be used when deploying the data processing container.

```
def generateRequirementsFile(packages, ansilbePath, roleName):

    string = ""
    for package in packages:
        string = string + package + "\n"

    path = os.path.join(ansilbePath, "roles", roleName, "templates", "requirements.txt")

    f = open(path, "w+")
    f.write(string)
    f.close()
```

After all of the configuration MyT is ready to run the Ansible playbook. But first we must get the user's privateKey, so Ansible can ssh into the machine. Once decrypted we can call the *"runPlaybook"* function.

```
# ------------Ansible Create------------#
privateKey = encryption.decryptString(password=password, salt=user.keySalt, resKey=user.resKey,
                                      string=user.privateKey)

aboutput, aberror = ab.runPlaybook(output["instance_ip_address"]["value"], privateKey, ansiblePath, "installService")
```

```python
def runPlaybook(floatingIp, privateKey, ansiblePath, playbooknName):
    # Get the path of the inventory file
    inventoryPath = ansiblePath + "/inventory"
    keyPath = ansiblePath + "/id_rsa"

    # Open the inventory file and add the floating ip address
    f = open(inventoryPath, "w+")
    f.write("[vms]\n" + floatingIp + "\n")
    f.close()

    # Open the private key file and add the private key
    f = open(keyPath, "w+")
    f.write(privateKey)
    f.close()

    os.chmod(keyPath, 0o600)

    # Run the ansible-playbook command
    executeCommand = "ansible-playbook -i inventory " + playbooknName + ".yml  -e 'ansible_python_interpreter=/usr/bin/python3'"
    print(executeCommand)
    process = subprocess.Popen(executeCommand.split(), stdout=subprocess.PIPE, cwd=ansiblePath)
    output, error = process.communicate()

    # Delete the key
    os.remove(keyPath)

    # Return the output and error
    return output, error
```

This function is used to run all MyT ansible playbooks. The first thing it does is generate the inventory file which contains the IP address of the virtual machine to deploy the platform on and then generates the private key file.

Once the required files have been created it will execute the ansible command. After Ansible has run its course it will remove the private key.

After this the MyT platform will be deployed and ready for use.

Post deployment

The last step in the deployment of the MyT platform or space is to save the infrastructure data to the database as it will be required for other features. Keeping with the platform example;

```
newPlatform = Platforms.create(dir=platformPath, name=safePlatformName, uid=user.uid, sid=sid, cid=cid,
                               cloudService=cloudService, ipAddress=output["instance_ip_address"]["value"],
                               packageList=data['packages'], database=database, dbsize=dbsize, id=str(uuid.uuid4()))

try:
    platform = Platforms.get(Platforms.id == newPlatform.id)
except Platforms.DoesNotExist:
    return Response.make_error_resp(msg="Platform Not Found", code=400)
```

This code will save all of the data on the platform that is required into the database. Then it is important to validate that the data has been saved, this is done for both platforms and spaces. The final step is to return the required data. For both platforms and spaces we just return the name of the platform and the id. However in the case of a RabbitMQ TLS enabled platform we also return the certs.

Security

Json Web Tokens

As discussed in the Frontend section Jwt's were used to secure routes on the backend. Using the Flask-JWT-Extended (Flask-JWT-Extended, 2020) module it is easy to implement the tokens. From generation;

```
# Generate User tokens
access_token = create_access_token(identity=user.userName)
refresh_token = create_refresh_token(identity=user.userName)
```

Renewing a token;

```
@users.route('/refresh', methods=['Get'])
@jwt_refresh_token_required
def refresh():
    current_user = get_jwt_identity()
    res = {
        'access_token': create_access_token(identity=current_user)
    }
    return Response.make_json_response(res)
```

And making the use of a token required on an endpoint is done by just adding a flag when defining a new route;

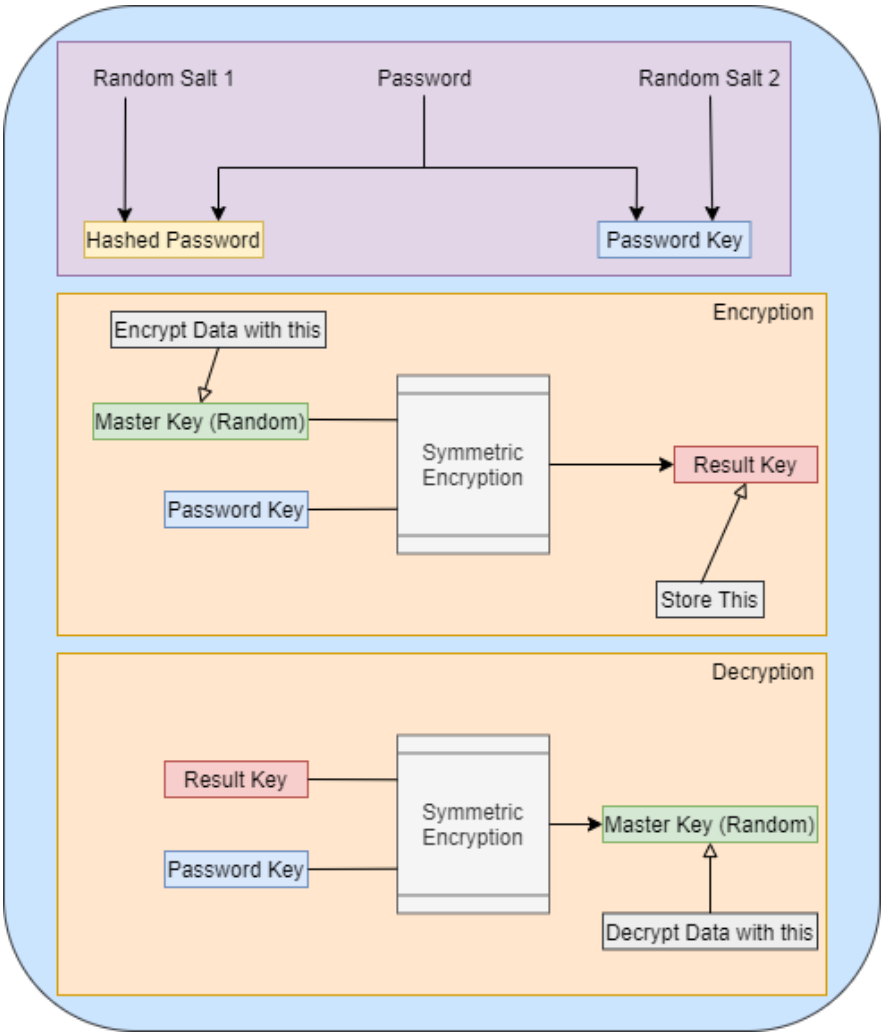
```
@users.route('/users/sshKey/<uid>', methods=["Post"])
@jwt_required
```

These seemingly small steps contribute to the very necessary security measures for particularly this application but all other modern web applications.

Encryption

MyT requires users to trust it with very sensitive credentials. It is all too easy for a malicious party to rack up thousands of dollars in cloud service bills if they had access to a user's credentials. As a result it is vital that MyT handles users sensitive information correctly. As discussed in the spec MyT uses a series of keys to encrypt and decrypt user data. This method ensures that even if someone was able to access the MyT database they would be unable to view the user's encrypted data.

This method ensures that all of the users data is secure and it is only accessible by them through their password. For this reason it is vital to ensure that a user has a strong password that is suitable to generate a strong password key.



To implement this into the MyT application a number of functions in *"encryption.py"* are used to handle all of the security operations. All of the cryptographic functionality uses the Python Cryptography (cryptography, 2020) package.

```
# Function to generate the result key
def generateResKey(password, salt):

    # Set up the key derivation function
    kdf = passwordEncryptionSettings(salt)

    # Generate the password key based on the password
    passwordKey = base64.urlsafe_b64encode(kdf.derive(str.encode(password)))

    # Randomly generate a key to use to encrypt user data
    masterKey = Fernet.generate_key()

    # Setup Fernet with the password key
    f = Fernet(passwordKey)

    # Encrypt the master key with the password key
    resKey = f.encrypt(masterKey)

    # Return the result key for storage
    return resKey
```

This code is used to generate all of the keys used to store the users data. First the Password Key is generated using the users password to derive the key. Then the Master Key is randomly generated then using Fernet which is an implementation of symmetric encryption, we encrypt the Master Key with the Password Key to create the Result Key which is then stored. A similar function is used to decrypt the Result key when we need the Master Key to encrypt or decrypt the user's data.

The *"encryption.py"* file also contains functions that handle the encryption and decryption of data. Making the implementation of encryption simple and easy to replicate.

```
secretKey = encryption.decryptString(password=password, salt=user.keySalt, resKey=user.resKey, string=creds.secretKey)
```

```
privateKey = encryption.encryptString(password=data['password'], salt=keySalt, resKey=resKey, string=privateKey)
```

Database

The MyT backend service uses Peewee (peewee, 2020) to handle the connection to the Sqlite database. Peewee is an Object Relational Mapper (Object-relational Mappers (ORMs), 2020). This means that there are no SQL commands written in the Python code. This prevents an SQL injection attack, as well as making it easier to query the database.

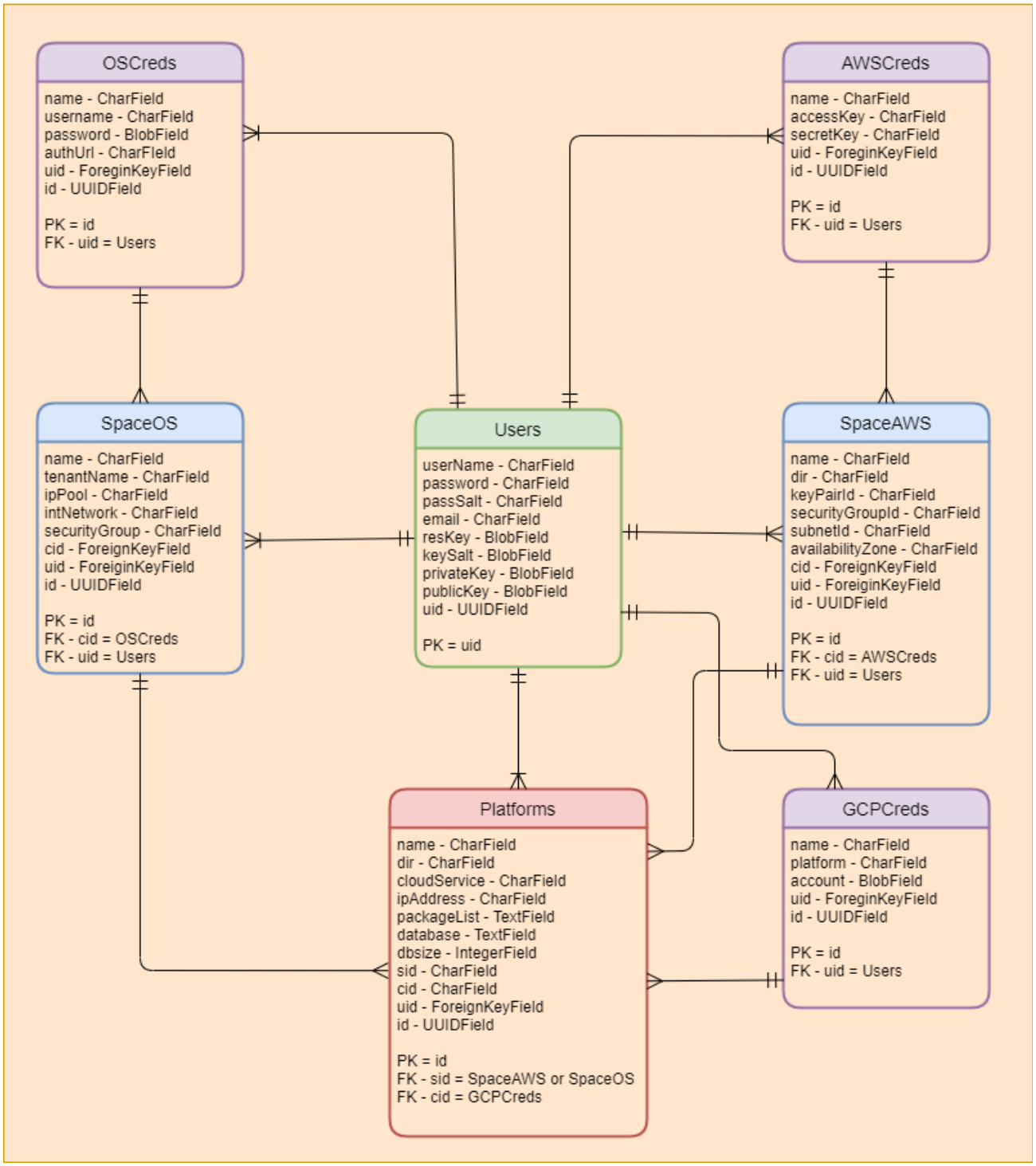
```
Users.create(userName=userName, email=email, password=password, passSalt=passSalt,
             resKey=resKey, keySalt=keySalt, privateKey=privateKey, publicKey=publicKey, uid=str(uuid.uuid4()))
```

This code shows how easy it is to save data to the database, instead of SQL commands it's more like Object Oriented Programming.

```
user = Users.get(Users.email == email)
```

And the same idea is also used when querying the database.

SQLite Database



Users

- userName - User's username.
- password - User's hashed password
- passSalt - User's password salt
- Email - User's email.
- resKey - User's result key (encrypted master key).
- keySalt - User's key generation salt.
- privateKey - User's encrypted private key.
- publicKey - User's encrypted public key.
- uid - User's unique identifier.

Platforms

- name - The platform's name.
- dir - Path to the directory the platform is saved in.
- cloudService - The cloud service which the platform is deployed on.
- ipAddress - The IP Address of the virtual machine.
- packageList - The Python packages used for data processing.
- database - Database used on the platform.
- dbsize - Size of the database.
- sid - ID of the space used - Can either be SpaceAWS or SpaceOS
- cid - ID of the credentials used - Only used in GCP platforms.
- uid - User's id
- Id - Platforms unique identifier

Spaces

SpaceOS

- name - The space's name.
- tennantName - Openstack tenant name.
- ipPool - Openstack IP address Pool.
- intNetwork - Openstack internal network.
- securityGroup - Openstack security group.
- cid - Openstack credentials id (OSCreds).
- uid - User's id
- Id - Space's unique identifier.

SpaceAWS

- name - The space's name.
- dir - Path to the directory the space is stored in.
- keyPairID - ID of the AWS key pair created.
- securityGroupID - ID of the AWS security group created.
- subnetID - ID of the AWS subnet created.
- availabilityZone - AWS availability zone Space is deployed to.
- cid - Openstack credentials id (AWSCreds).
- uid - User's id
- Id - Space's unique identifier.

Credentials

OSCreds

- name - Credential's name.
- username - User's encrypted Openstack username.
- password - User's encrypted Openstack password.
- authUrl - User's encrypted Openstack authentication Url.
- uid - User's id
- Id - Credential's unique identifier.

AWSCreds

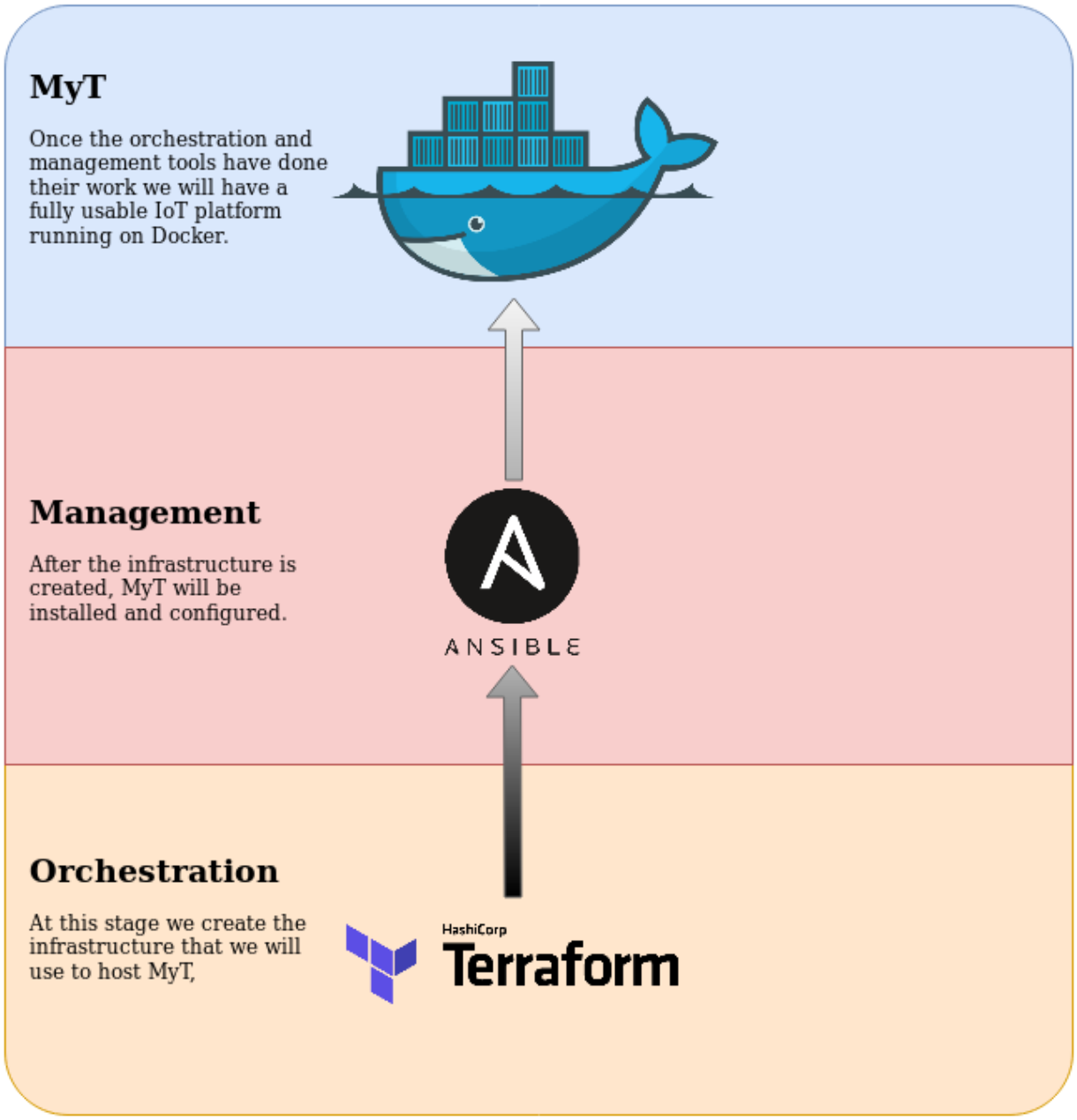
- name - Credential's name.
- accessKey - User's encrypted AWS access key.
- secretKey - User's encrypted AWS secret key.
- uid - User's id
- Id - Credential's unique identifier.

GCPCreds

- name - Credential's name.
- account - User's encrypted GCP account.json file.
- platform - User's GCP platform name.
- uid - User's id
- Id - Credential's unique identifier.

Deployment

Deployment Summary



The automated deployment of the MyT platform is done by using Terraform to create the cloud infrastructure. Ansible to install and configure MyT and docker to run and manage it.

Terraform

Terraform (Terraform by HashiCorp, 2020) is a cloud orchestration tool. It is used in MyT to create infrastructure on the various cloud services.

Terraform Usage

Terraform uses Hashicorp Configuration Language to define the desired infrastructure. When a user wants to deploy their infrastructure they use the *"apply"* command, then Terraform will read the terraform files and deploy the infrastructure. While this is happening Terraform will remember what the infrastructure "looks like", meaning it will store the state of the deployed infrasture. It does so in *".tfstate"* files, which is why the MyT application creates new directories for each of the spaces and platforms. The state storage has the added benefit of making it easy to remove infrasture, by simply running the *"destroy"* command in the infrastrures directory, and Terraform takes care of the rest.

Before letting Terraform handle the deployment of infrastructure we must first define the Infrastructure as Code "IaC". The IaC used to deploy infrastructure across all of the cloud services is broken down into three different files.

Provider

```
provider "aws" {
  region     = "eu-west-1"
  access_key = var.aws_access_key
  secret_key = var.aws_secret_key
}
```

The *"provider.tf"* file defines which of the cloud services is to be used. In this example it is Amazon Web Services. Each of the cloud services have different required and optional variables.

Variables

```
variable "aws_access_key" {
  description = "AWS Access Key"
  default   = "NotMyAWSAccessKey"
}

variable "aws_secret_key" {
  description = "AWS Secret Key"
  default   = "MostCertainlyNotMyAWSSecretKey"
}
```

The *"variables.tf"* file is where all of the variables for the script are deployed. These differ between the cloud services as they require different authentication data and data for the infrastructure. This file is generated by MyT using data input from the user.

The above image shows how a variable is defined. With the data defined in the *"default"* being the value. The data in the variable is accessed with the format; *"var.variable_name"*.

Deploy

```
resource "aws_instance" "web" {
  ami = "ami-02df9ea15c1778c9c"
  instance_type = "t3a.small"
  key_name = var.key_pair_id
  security_groups = [var.security_group_id]
  subnet_id = var.subnet_id

  tags = {
    Name = var.platform_name
  }
}
```

Lastly *"deploy.tf"* is where all of the actual infrastructure components are defined. These files vary significantly across the different cloud services as they have their own requirements. The example above shows how an AWS ec2 instance is created. Note how it gets most of its data from the variables file. This is how the users data is converted into their infrastructure.

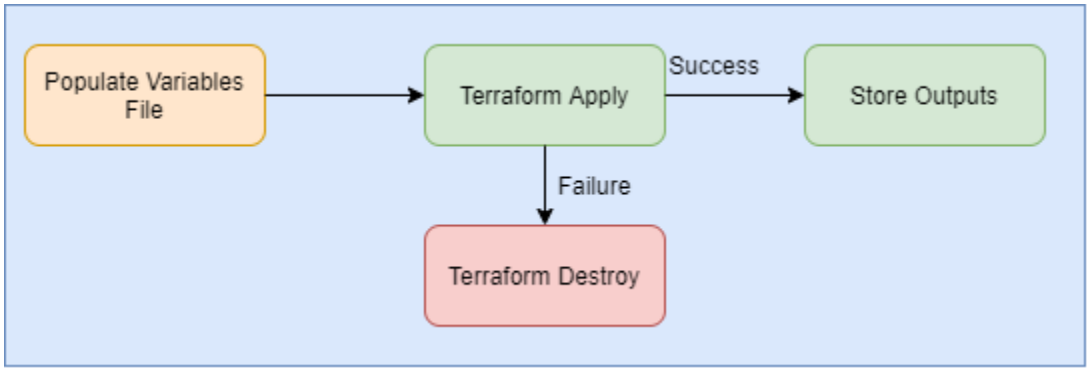
Outputting Data

In order to effectively manage cloud infrastructure MyT must store data on the platform such as a virtual machines Ip address. Terraform provides output functionality, which allows terraform to output data from the infrastructure. This is implemented into MyT when creating platforms or spaces.

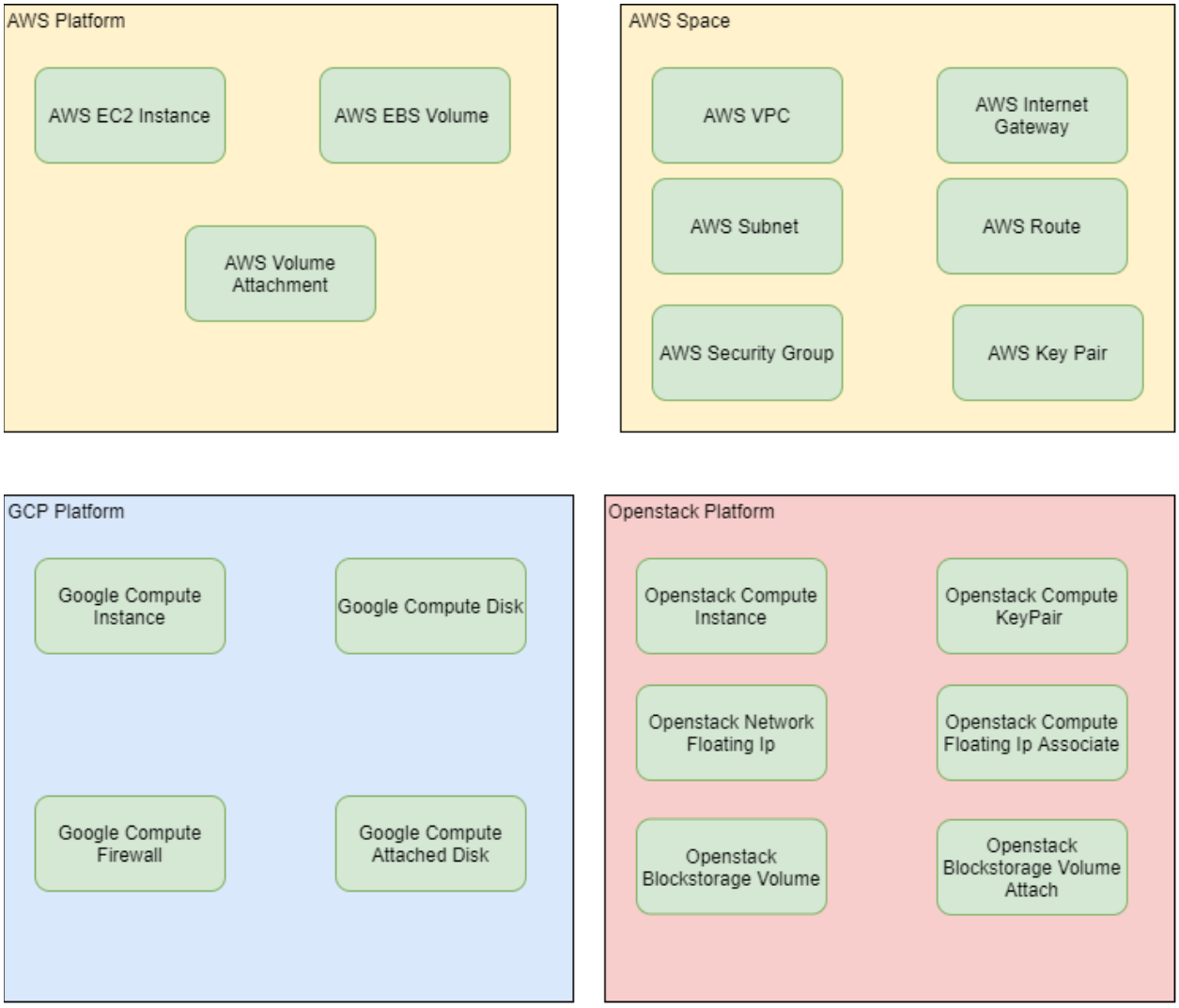
```
output "key_pair" {
  value = aws_key_pair.MyT-Key.id
}

output "security_group" {
  value = aws_security_group.MyT-SG.id
}

output "subnet" {
  value = aws_subnet.my_subnet.id
}
```

These are the outputs used when a AWS space is deployed. MyT needs to store the key pair id, security group id and the subnet id in the SQLite database because new platforms to be deployed in that space will need that information.

Terraform Design Pattern



Terraform Resources



Above is a list of each of the resources that MyT deploys across the different cloud services, required to create the same platform.

Differences Across Cloud Services

The aim of MyT was to allow users to use any cloud service of their choice. This means that MyT must provide the same service across all of the cloud services. This is easier said than done as the three cloud services that MyT supports have different ways they want you to do things.

Spaces

AWS was the service that MyT used when going through most of development, and the way that AWS wants you to manage your infrastructure revolves around VPCs. They want you to make a VPC and then deploy your infrastructure into it's subnets. This is where the idea of the Platforms and Spaces came from. A Space was initially intended to be a network that a user would deploy Platforms into. This idea works perfectly for AWS. When implementing Openstack this idea still fits. However there was no need to create a VPC because Openstack doesn't force you to, so then the Space idea was adapted from a VPC to a network configuration for Openstack. So a user can deploy multiple Platforms with the same network settings. Finally when moving to GCP the Space concept became unnecessary. Google Cloud makes users Create a platform to deploy their infrastructure to, which contains most of the network configuration in it. As a result MyT only collects the platform name and the platforms account.json file and it will have all of the data required to deploy MyT platforms on the Google Cloud Platform.

All of this results with only AWS Spaces deploying any infrastructure, GCP platforms not needing a space at all and Openstack being somewhere in the middle.

Platforms

The changes also occur when creating a platform. For various reasons the data required to create a platform on the various cloud services is different. For example when creating a GCP platform the user will be required to input an availability zone. This is not required for AWS or Openstack platforms. Openstack also has a number of its own requirements. Specifically flavor name and image name. These are needed because unlike AWS and Google cloud this data is dependent on the users Openstack installation.

Ansible

Ansible (Ansible, 2020) is a cloud management tool used to setup and configure the MyT platform.

Ansible Usage

Ansible uses yml to allow the users to define a number of operations that they want executed on the target machine. Unlike Terraform there is no state being saved. Ansible simply logs onto the machine and executes the commands. However MyT still puts the ansible playbook into the platforms directory then configures it and executes it there. This is done so the same playbook isn't constantly being edited and executed as this could lead to a number of issues.

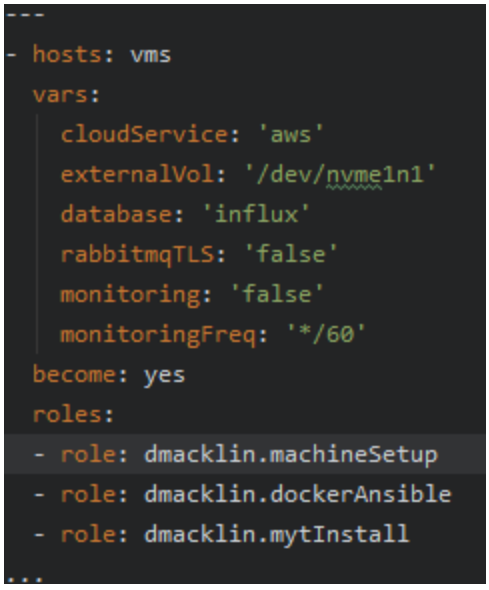
With the exception of the location of the external database volume, the result from Terraform is identical across all of the cloud services. This means there is no need for a new role for each of the cloud services like Terraform. Instead we can use variables to handle the differences.

An Ansible Playbook is broken into a number of different components.

The Main yml file

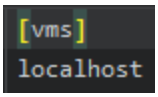
This image shows the main yml file for the createPlatform playbook.

- The hosts variable defines the target machine or machines. In this case it references a section of the inventory file.
- The vars variable defines a list of variables used in the different roles.
- The become variable means the playbook should become the root user when executing commands.
- The roles variable defines a list of roles that are to be executed and in what order.



Inventory

The *"inventory"* file contains a list of target machines. For MyT there will only be one IP address used, which comes from the Terraform outputs.



Ansible.cfg

The *"ansible.cfg"* file contains the configuration for ansible.

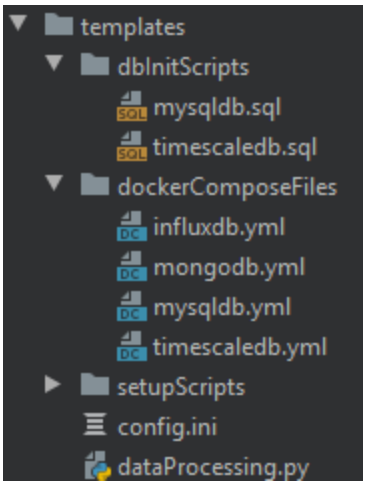
Roles

Roles further break down an ansible playbook. In MyT all of the roles contain a tasks folder, this contains a yml file with all of the commands to be executed. For example the *"dmacklin.mytinstall"* roles task file;

```
---
- name: Remove MyT MyTApplication
  file:
    path: /opt/MyTApplication
    state: absent

- name: Pull MyT git repo
  git:
    repo: https://github.com/Daemon-Macklin/MyTPlatformDataProcessingContainer
    dest: /opt/MyTApplication
```

This is the first two commands executed in the role. There are a number of others, but this shows how the ansible modules are used. There are modules that are used to perform different operations. In the image we see the file module (file – Manage files and file properties — Ansible Documentation, 2020) and the git module (git – Deploy software (or files) from git checkouts — Ansible Documentation, 2020) being used to remove the MyT application folder if it is there and then pulling down the application from git.

Some of the other roles also have a templates folder. This contains all of the files and configs that the role might need. Looking at *"dmacklin.mytinstall"* role again we see what the templates folder contains a number of different files and folders.



- The *"dbInitScripts"* folder contains both of the base init scripts for the SQL databases.
- The *"dockerComposeFiles"* folder contains the different docker-compose files which are used depending on the database in use.
- The *"setupScripts"* folder contains old db init scripts that are no longer in use.
- The *"config.ini"* file is the data processing containers configuration file.
- The *"dataProcessing.py"* file is the data processing script that will get put into the data processing container. If the user has uploaded one when creating the platform this will be it.

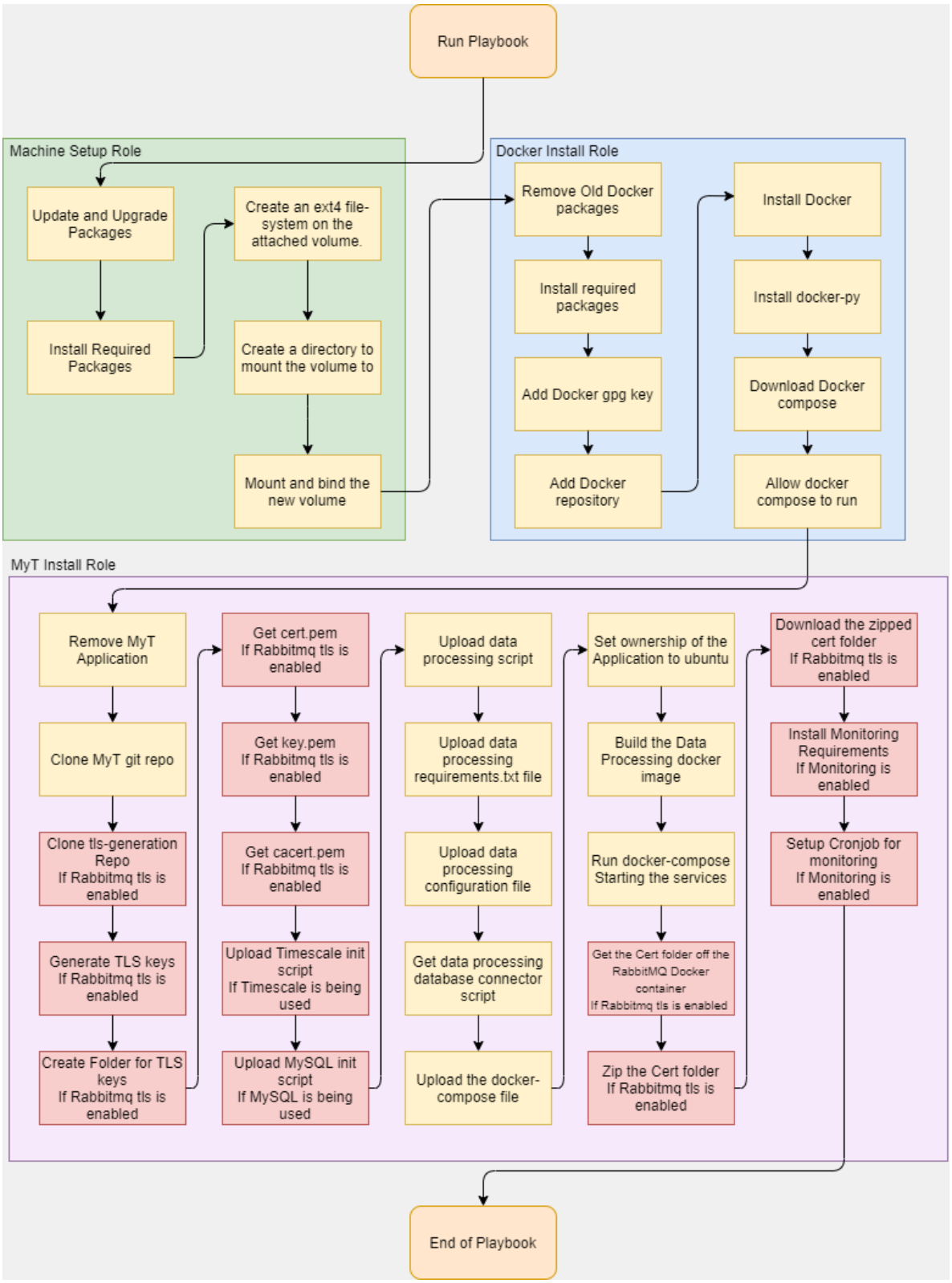
Each of these files will be edited by the MyT application before running the playbook, to add the users data. We can see how these files are used and how variables are used by looking at how the docker-compse files are used.

```
- name: Configure docker-compose.yml file
  copy:
    src: "../templates/dockerComposeFiles/{{ database }}.yml"
    dest: /opt/MyTApplication/docker-compose.yml
```

MyT uses the copy module to copy the file from the host machine to the target machine. The templates folder is where all of the files that will be copied up are stored. As seen in *"src"* the file being used is dependent on the *"database"* variable. Which is defined in the main file. That variable is set by the MyT application based on which database the user has chosen. This is how the choices made in the frontend are used to create their custom platform.

```
database: 'influxdb'
```

Ansible Platform Deployment



Docker-Compose

Docker-Compose (Overview of Docker Compose, 2020) is a tool used to define and run multi-container Docker Applications.

Docker-Compose Usage

Docker-Compose is used in MyT to run all of the docker containers that make up the MyT IoT platform. Docker-Compose allows users to define a number of containers they want to run along various configuration settings.

There are four Docker-Compose scripts used in MyT, this is to account for the different databases and their requirements. There is a script for each database, most of the scripts are the same but defining the database container is different along with some other differences.

Containers

Docker containers are defined in Docker-Compose scripts like so;

```
influxdb:
  image: influxdb
  ports:
    - "8086:8086"
  networks:
    - myt-network
  container_name: influxdb
  volumes:
    - /var/lib/mytDatabase:/var/lib/influxdb
  environment:
    - INFLUXDB_DB=MyTData
```

- *"Image"* is the docker image to be used to build the container. If the image is not already on the local machine Docker will pull the image down from DockerHub.
- *"ports"* specifies which ports should be open on the machine.
- *"networks"* specifies which network the container should be on.
- *"container_name"* defines the name of the container.
- *"volumes"* are used to bind folders on the container to folders on the host machine. In this example we bind where influxdb stores data from the database to the folder where on the host machine that we mounted the external volume to, ensuring data is safe.
- *"environment"* is used to pass in environments to the container, they vary across containers but this example shows how the MyT database is created.

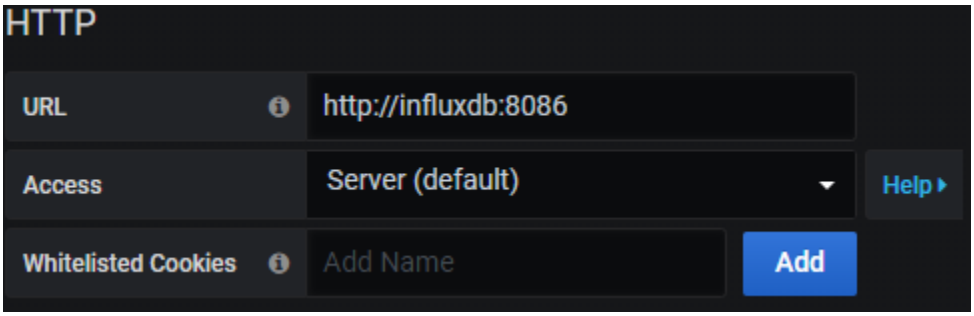
Networks

Docker-Compose allows for the ability to create networks which can easily connect containers together. MyT uses this to make it easier to manage the connections between containers for the platform itself and how users interact with it.

```
networks:
  myt-network:
    name: myt-network
    driver: bridge
```

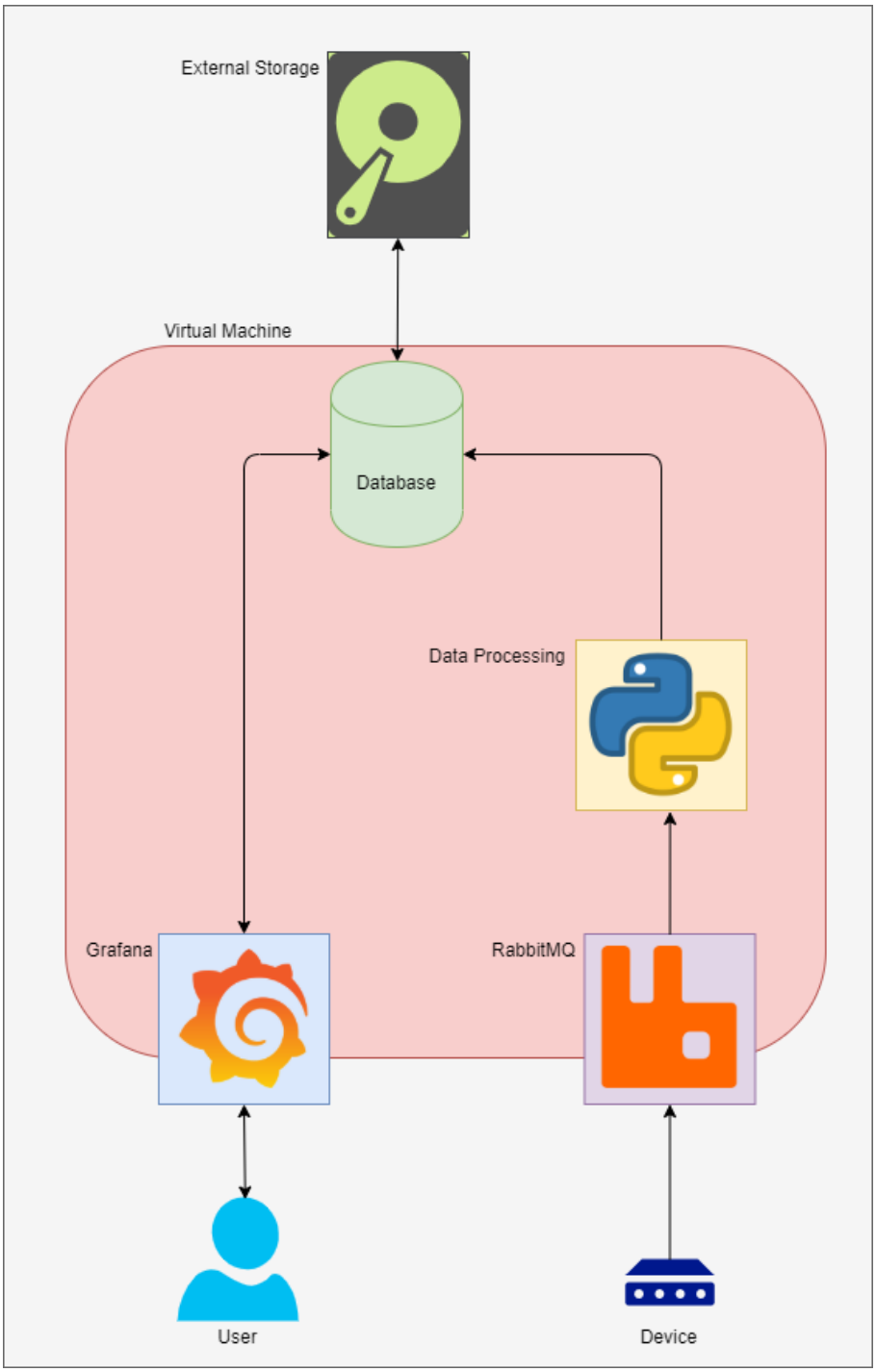
- *"name"* is the name of the network to be used.
- *"driver"* is the set of rules used to define the network. *"bridge"* is the default setting.

The biggest reason for using networks is that they automatically set up dns names for all of the containers in an internal dns service. This means that when the user has to set up the data source in Grafana they just need to type in the name of the database they chose, instead of using an Ip address.



Platform

Architecture

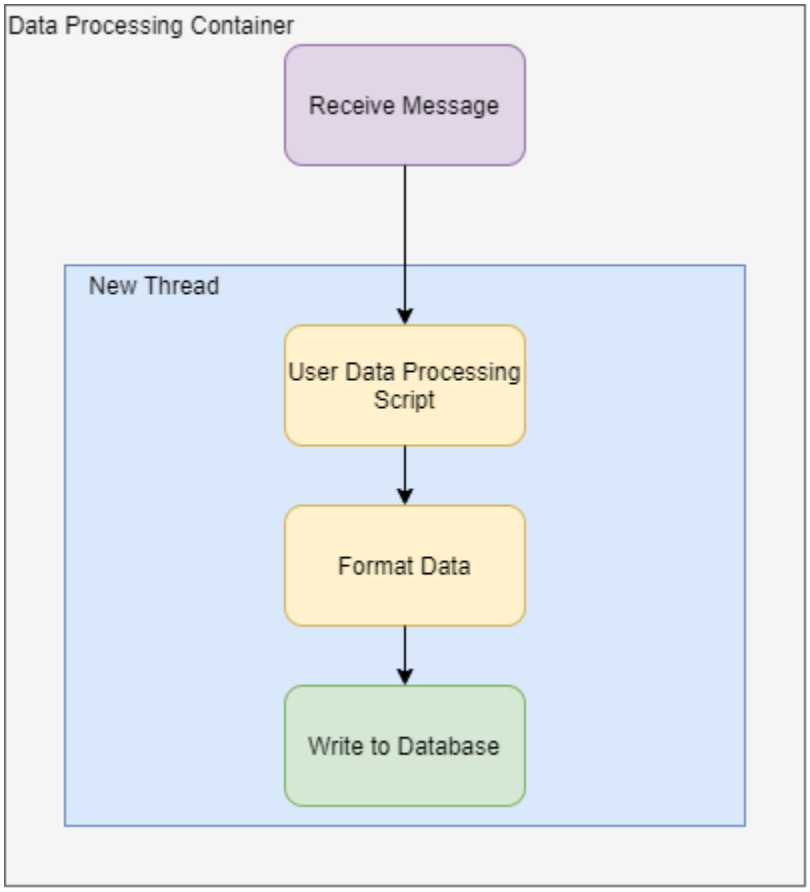


Components

RabbitMQ

RabbitMQ is a AMQP messaging service that MyT uses to collect data from the user's sensors. The user will write code to connect and send messages to the broker. The RabbitMQ broker has a number of security options. A username and password can be configured on the broker so that only people with the username and password can send messages. The other feature is enabling TLS on RabbitMQ. This will result in MyT generating a number of keys which can be given to the user to use when writing their RabbitMQ client code.

Data Processing



Listening to RabbitMQ

The first job of the Data Processing container is to act as a RabbitMQ wrapper. The first thing it does on startup is to connect to the RabbitMQ broker and listen to all channels.

When a message is received the function *"handle_message"* is called. This message handles the creation of a new thread which will perform all of the necessary steps with the data. A new thread is started on the *"do_work"* function, this function is used to read and process the data. The first thing it does is pass it into the data processing script, if the user has correctly written their script the data will be processed. Then the data is pushed onto the database component to save the data.

```
def do_work(connection, channel, delivery_tag, body):
    thread_id = threading.get_ident()
    fmt1 = 'Thread id: {} Delivery tag: {} Message body: {}'
    loggerHelper.getLogger().info(fmt1.format(thread_id, delivery_tag, body))

    body = json.loads(body)
    body["data"] = dp.main(body["data"])
    db.writeData(body)

    cb = functools.partial(ack_message, channel, delivery_tag)
    connection.add_callback_threadsafe(cb)
```

Data Processing

The data processing script the user provides is based on a template that must be used for it to run with the platform.

```
def main(data):
    # Perform data processing here
    loggerHelper.getLogger().info("Processing")

    # All data in the dict data will be saved to the database
    return data
```

The user will write code into the *"main"* function. This code would get executed and the new data will be returned and saved into the database.

Saving to the Database

Since there are a number of databases that could be used for the MyT application there needs to be a database connector to handle each one. As a result there are four database connector files which all use the same function names. When the platform is being deployed Ansible will select the correct connector to use when creating the data processing container.

SQL vs NoSQL

MongoDB and InfluxDB are both NoSQL databases, as a result there is no need to define tables or schemas. All that needs to be done is converting the data into a JSON object and writing it to the database. Influx and Mongo have different JSON structures but the procedure remains the same and simple.

When using MySQL or Timescale, both SQL databases MyT needs to account for the schemas in the database. The user will define the fields in the database when creating a platform. MyT also relies on the users sending valid data to the platform when using SQL databases. The user sends a JSON object to RabbitMQ, then when writing the data to the SQL database, the code uses the keys as the variable names for their corresponding values.

```
def writeData(body):
    global client
    data=body["data"]
    date=datetime.datetime.now()
    varnames = ""
    formatting = ""
    vals = ()
    for key, value in data.items():
        varnames = varnames + key + ","
        formatting = formatting + "%s,"
        vals = vals + (value,)
    varnames = varnames + "ts"
    formatting = formatting + "%s"
    vals = vals + (date,)
    loggerHelper.getLogger().info(varnames)
    loggerHelper.getLogger().info(vals)

    command = "INSERT INTO SensorData (" + varnames + ") VALUES (" + formatting + ")
    loggerHelper.getLogger().info(command)

    client.cursor().execute(command, vals)

    client.commit()
```

Database

One of the main features in MyT is that the user chooses which database that the platform uses. This adds a lot of flexibility to the MyT platform. The database naturally serves to store all of the user's sensor data. There are two types of databases being used.

NoSQL

MongoDB (MongoDB, 2020) and InfluxDB (InfluxDB, 2020) are NoSQL databases. This means there is no need for schemas or predefined tables. Any data can be inserted into collections (Mongo) or measurements (InfluxDB). This makes them very flexible and reduces the work required to set them up for the user.

When setting up both databases all MyT needs to do is create a database, then users can upload data to collections and measurements that are created when data is inserted into them. The databases themselves are created using an environment variable passed through the docker file.

```
environment:
  - INFLUXDB_DB=MyTData
```

```
environment:
  - MONGO_INITDB_DATABASE=MyTData
```

SQL

MySQL (MySQL, 2020) and Timescale (Timescale, 2020) are both SQL databases. This means that they require databases and tables to be set up before the user can start storing data into them.

MyT gets around this issue by asking the user to define all of the variables they are going to need when they create the platform. The define the variable name and which data type it is. This data will be inserted into an SQL initialization script. This initialization script is placed in the database container in the location where the database reads data when starting up. The commands in the script are run and the database, and table are created.

```
volumes:
  - /var/lib/mytDatabase:/var/lib/postgresql
  - ./init.sql:/docker-entrypoint-initdb.d/init.sql
```

```
volumes:
  - /var/lib/mytDatabase:/var/lib/mysql
  - ./init.sql:/data/application/init.sql
```

The init scripts themselves are just a list of commands to be run.

```
CREATE DATABASE IF NOT EXISTS MyTData;

USE MyTData;

CREATE TABLE IF NOT EXISTS SensorData (
        id int(11) NOT NULL AUTO_INCREMENT,
        ts timestamp NOT NULL,
```

The script contains the creation of the database and the beginning of the command to create the table, with the two default variables, for MySQL databases. The syntax for Timescale is slightly different. When the user chooses to use an SQL database the variables they provide are added to this script by a function on the MyT backend.

```
def createSQLInit(ansiblePath, fields, database):

    # Check which database is being used
    if database == "timescaledb":
        # Get the Timescale base file
        path = os.path.join(ansiblePath, "roles", "dmacklin.mytInstall", "templates", "dbInitScripts", "timescaledb.sql")
        # Update the keys to use Timescale syntax
        for key, value in fields.items():
            if value == "int":
                fields[key] = "integer"
    else:
        # Get the MySQL base file
        path = os.path.join(ansiblePath, "roles", "dmacklin.mytInstall", "templates", "dbInitScripts", "mysqldb.sql")

    # Generate a string which will add turn the dict into a list of variables for an SQL command.
    fieldsStr = ""
    for key, value in fields.items():
        fieldsStr = fieldsStr + "         " + key + " " + value + ",\n"
    # Add the last line
    fieldsStr = fieldsStr + "        PRIMARY KEY (id));\n"

    # Append the string to the end of the file
    with open(path, 'a') as file:
        file.write(fieldsStr)
```

This allows MyT to provide limited SQL support. Only allowing the user to create on table, and not being able to update tables once created.

Grafana

Grafana (Grafana, 2020) is a dashboarding tool that MyT provides as the platforms graphing and data analysis service. It is what the user will have the most interaction with as they create graphs and analyze their data.

When the platform is created and the user goes to port 3000 on the platform's IP address, the user is given a login screen to their dashboard. The user will put into the default Grafana username and password, then be prompted to change it. Once logged in all the user will have to do is add the data source. Since Docker provides a dns service across the containers the user just types in "*http://dbnamedb:port*" in the add datasource page and their database will be connected, and they can start making graphs.

Testing

Http Endpoint Testing

Testing the api endpoints of the MyT backend service was done using the Python request library. A script was written which would go through the process of creating a user, creating a space and platform and then removing all of the infrastructure and the user.

This script would test one endpoint then using data in the response move on to test the next endpoint.

```
# sending post request and saving response as response object
response = requests.post(url = API_ENDPOINT + "users/create", json = createData)
print(response.json())
uid = response.json()["uid"]


credsData = {
    "name" : "AWS",
    "accessKey" : [redacted],
    "secretKey" : [redacted],
    "uid" : uid,
    "password" : password
}

response = requests.post(url = API_ENDPOINT + "credentials/create/aws", json = credsData)
print(response.json())
cid = response.json()["id"]
```

This example shows how a user was created then using the uid from the response of that request in the next request to create a set of user credentials.

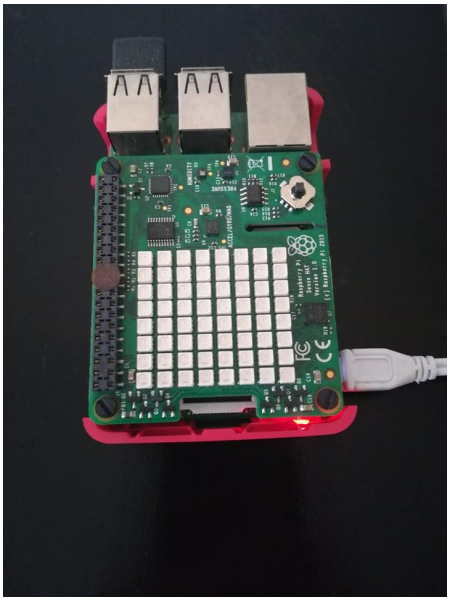
This method simultaneously tests the endpoint itself, the response it generates and shows that data is being stored in the database.

Platform Testing

The creation of the MyT platform is an interesting and cool process. However once it is created it needs to stand on its own and operate reliably. To test this I have set up a platform on GCP and have been sending it data every 5 minutes.

Using the MyT application a platform was created which uses InfluxDB and has both RabbitMQ username/password and TLS enabled. After a few minutes my platform was ready.

The next step was to find a way of simulating an IoT device. The best way to do this was to create an IoT device, so a Raspberry Pi (Raspberrypi, 2020) and Sense HAT (SenseHat, 2020) was used. The Sense HAT provides temperature, humidity and pressure sensors that are being used to gather data from a room.



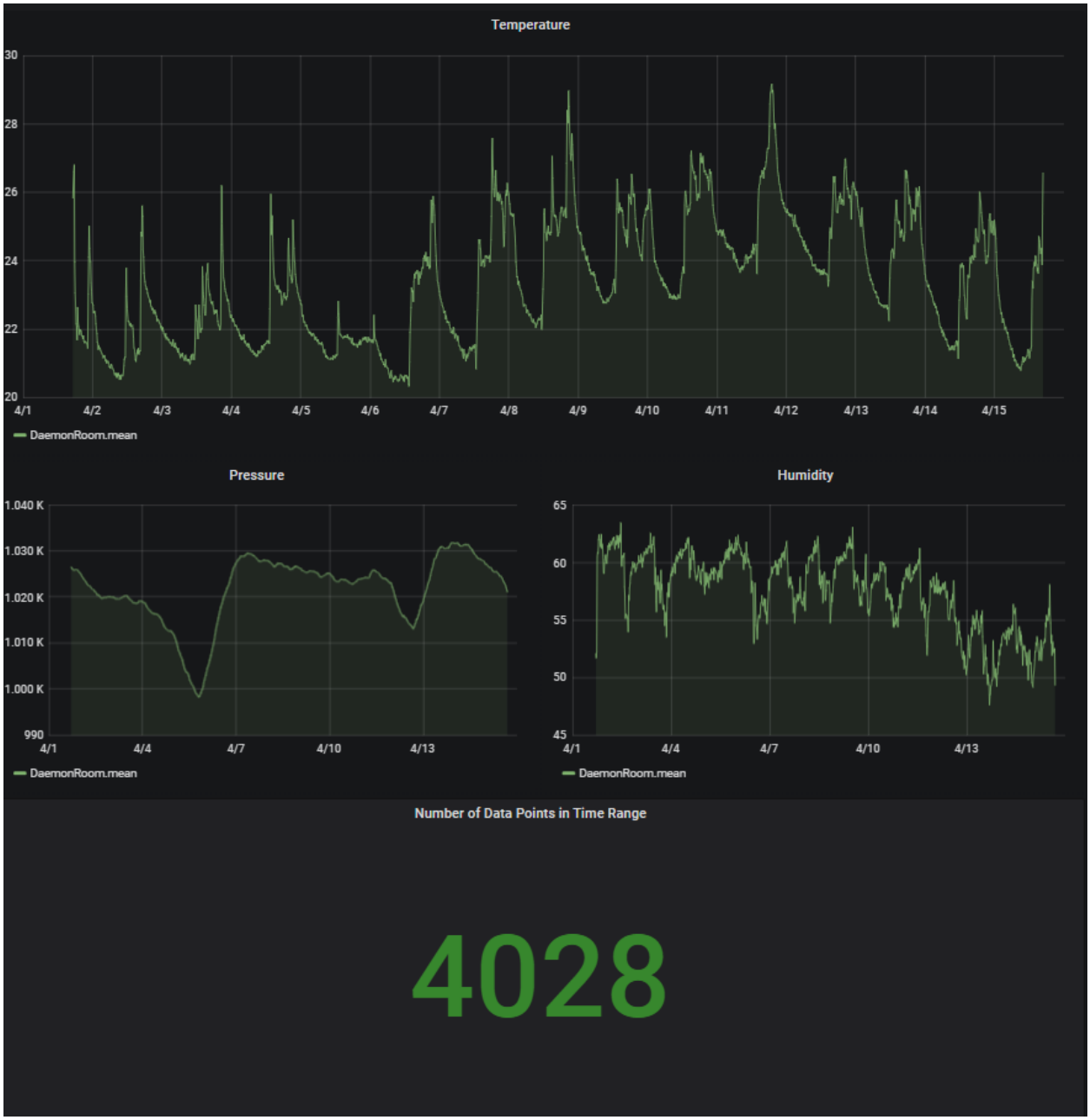
```python
#!/usr/bin/python3

import pika
from sense_hat import SenseHat
import sys
import json
import ssl

sense = SenseHat()

creds = pika.PlainCredentials(username='dman', password=[redacted])


context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
context.load_verify_locations("rabbitmq_cert/cacert.pem")
context.load_cert_chain("rabbitmq_cert/cert.pem", "rabbitmq_cert/key.pem")

ssl_options = pika.SSLOptions(context)

connection = pika.BlockingConnection(pika.ConnectionParameters(
    host="35.246.52.17", credentials=creds, ssl_options=ssl_options))

channel = connection.channel()

channel.exchange_declare(exchange='data', exchange_type='fanout')

message = {
    "measurement": "DaemonRoom",
    "sensor": "pi1",
    "data" : {
        "temperature": sense.get_temperature(),
        "pressure": sense.get_pressure(),
        "humidity": sense.get_humidity()
        }
    }

print(message)
channel.basic_publish(exchange='data', routing_key='', body=json.dumps(message))
print(" [x] Sent %r" % json.dumps(message))
connection.close()
```

To access the data in the senors a python script was written that would get the data from the sensors and then publish the data to the MyT platforms RabbitMQ broker. To run this script a cron job was used.

```
# m h  dom mon dow   command
*/5 * * * * cd /home/pi/Desktop && /home/pi/Desktop/daemonsRoom.py > /home/pi/Desktop/log.log 2>&1
```

This cron job runs the python script every 5 minutes. Meaning the platform is getting sensor data every 5 minutes.

This test started on 1/4/2020 at 17:10, and has been running continuously. As of 15/4/2020 at 16:50 there are 4028 data points in the database.



The results of this on-going test can be seen on the platforms Grafana. At;

http://35.246.52.17:3000/

Using the username: Jon and password: Jon123 will let you see the graphs.

Challenges and Problems

Setting up Databases

Initially the database for the MyT platform was set up by Ansible running a python script which would create the database after the docker-compose script was run. When initially testing it seemed to work. After further development and more additions to the docker-compse file it was noticed that the database was not being created. The problem was first thought to be an issue with the script itself, however running manually would create the database flawlessly.

The next thought was that it was an issue with the Ansible command, so additions were made and it again seemed to work. However not long after it was found that it was again not working. When thinking about the issue it occurred to me that error messages were saying that there was no database server to connect to, this led to the realization that docker had not finished creating the database container before the Ansible command had run. So the database server wasn't ready when the command was being run.

This problem led to the realization that container related actions could not be performed by Ansible post deployment due to unreliability. This forced MyT to use the environment variables in Docker-Compose to configure things in the docker containers.

Downloading Zips

The database dumps created on user request are zipped before being returned to the user. There were a number of challenges in handling the data and returning it to the user. The one which gave the most issue is how the frontend gave the data to the user. Javascript file writer allows you to pass in a blob object and it will generate a file. This was implemented for the dumps however it would always end up corrupted. Initially it was thought to be an issue with the writer how it dealt with the bytes from the zip file. So many different ways of writing a file were tried. After hours of trial and error it was found that the issue was that the request was simply using the wrong request header and would return the data as a string instead of bytes. Reverting the code to its original state and updating the request header fixed the issue.

Testing Ansible Playbooks and Terraform Scripts

Ansible and to a lesser extent Terraform were challenging to test when used on the application. Terraform was okay to use as it produced an easy to read error message that made it easy to identify errors. Ansible on the other had given error messages which were nearly impossible to read when displayed through the application. This made it challenging to debug Ansible playbooks, particularly when testing out new features which involves adding variables to the playbooks with the application. The solution was to ssh into the backend container and manually run the scripts. However you need the users key so Ansible can ssh into the machine and MyT removes the keys after they are used as a security measure. So to test them, the key would need to be retrieved using the Application and then scp the file upto the docker runner and then copy it into the container and then the playbook could be run manually to be debugged.

This issue was also part of a larger issue when developing with Ansible and Terraform is the time it takes for them to run. When testing a new feature, the code gets updated, it will take a minute for the updates to be pushed to the testing container. Then run the code being tested which usually involves creating a new platform and that could take upwards of 5 minutes, just waiting to see the results. This problem made development a slow and sometimes painful experience.

Checking for Valid Python Packages

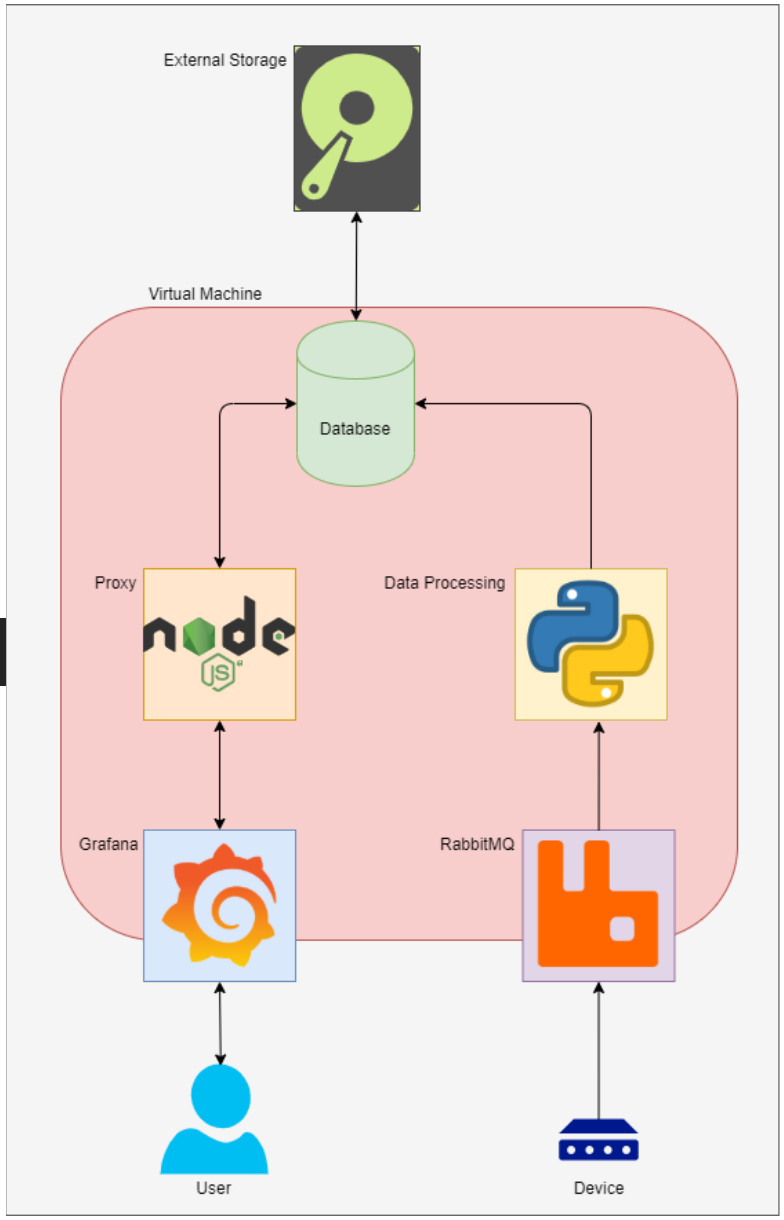
One of MyT's features is the user's ability to pull in python packages for their data processing scripts. One of the issues with implementing this was ensuring the package was valid before the creation of the platform. As a result a function was written that would check the users list of packages to ensure they are valid, and inform the user which package was an issue.

```
def checkPackages(packages):

    for package in packages:
        response = requests.get("https://pypi.python.org/pypi/{}/json".format(package))
        if response.status_code != 200:
            return package

    return ""
```

MongoDB Grafana

When implementing MongoDB it was found that Grafana did not support Mongodb data sources natively. This would cause an issue as it would be pointless to store data in Mongo if there was no way to access it. Thankfully many people have had this issue and have developed solutions to allow this. One of the most popular solutions uses a proxy to access the data. Using [JamesOsgood/mongodb-grafana: MongoDB plugin for Grafana] application and plugin it is possible to create dashboards.

As a result MongoDB platforms have an extra container which contains the proxy application. Then the user sets up a custom MongoDB data source and then can access data from the MongoDB.



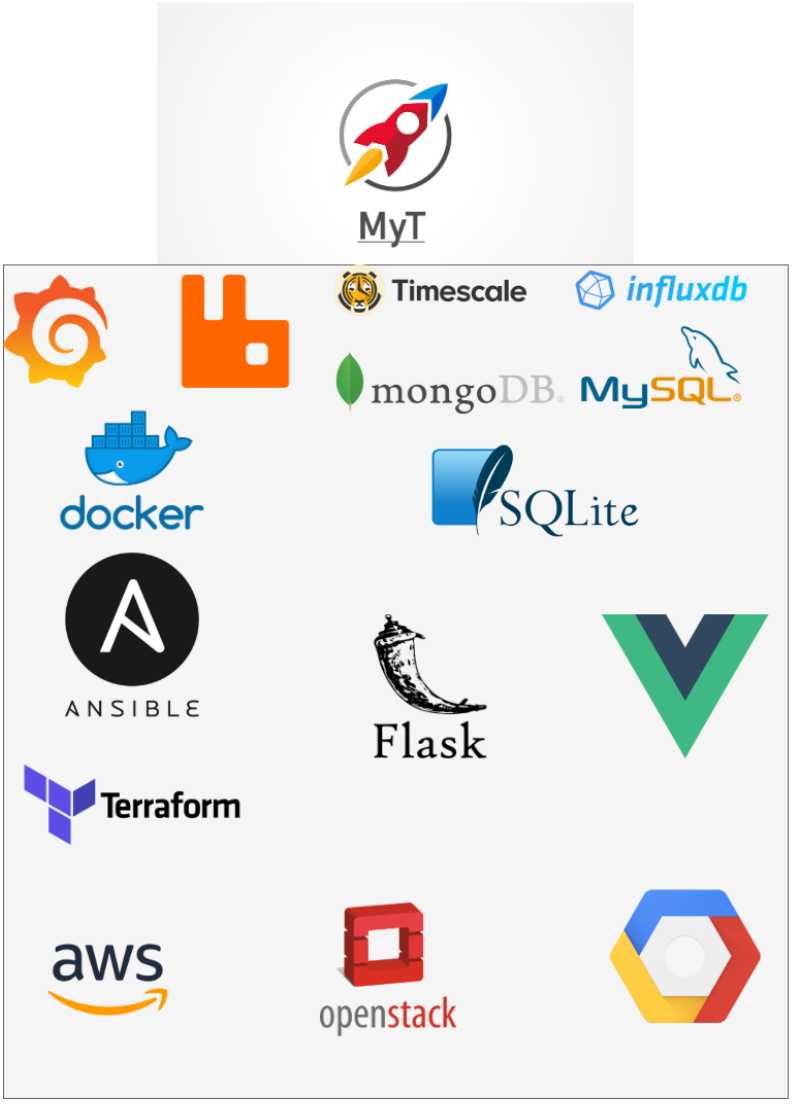
Grafana instances on MongoDB platforms also have a custom plugin (Grafana Labs. 2020) installed to allow this.

```
environment:
  - "GF_INSTALL_PLUGINS=grafana-simple-json-datasource"
```

Conclusion

The aim of this project was to develop an IoT platform and a way to easily customize, manage and deploy it. It was designed to provide users with their own IoT platform that they can use without having to put in the work to deploy it themselves, and give them the reassurance that their data is completely owned by them. Using a web frontend for users to interact with and industry standard tools being used to deploy and manage the platforms.

A wide array of technologies were used to make this project possible, and giving the user more freedom with their IoT platform.



All of these technologies were implemented into one functional application in order to overcome the challenges and complexity of deploying and maintaining an IoT platform. However this project was only scopped to deploy IoT platforms. The idea and a lot of the design patterns in the MyT project would allow for the deployment of other applications. This along with a few other things would be what could be done in future.

- Improve SQL database support
- Improve displaying the progress of platform deployment on the frontend.
- Add more cloud services and databases.

Overall I am very happy with how the project went. I achieved all of the goals I have set for myself and I have learned a lot. Not only did I gain a lot of knowledge on the technical aspects of the project, but more importantly I gained insight into developing a large project. Throughout this project and others during my 4 years at WIT I have learned to look at large complex issues and break them down into smaller achievable goals which will build into the solution. I think it is the most important skill to have as a developer as it not only applies to my future professional career but also my personal life.

Appendix

Supervisor Meeting Minutes

Week 1:

Date: 8/10/19
Time: 12:15
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:
N/A

Plan for upcoming weeks work:

- Do "unstructured" research on Deployment tools, Decision Trees and Desktop applications.
- Look at functional and nonfunctional requirements.

Additional Notes:

Week 3:

Date: 15/10/19
Time: 12:15
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:

- Showed Electron-Vue proof of concept application.

Plan for upcoming weeks work:

- Project report - Justification for Tech used, Cloud architecture, Deployment strategy, front end.

Additional Notes:

Week 4:

Date: 22/10/19
Time: 12:15
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:

- Project report headings.

Plan for upcoming weeks work:

- Literature Review.
- Project report - Abstract and introduction.
- Project report - Cloud deployment technologies.
- Project report - Cloud architecture technologies.
- Project report - Front end Desktop vs Web App.

Additional Notes

Week 5:

Mid Term Break

Week 6:

Date: 5/11/19
Time: 12:15
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:

- Discussed progress on report.

Plan for upcoming weeks work:

- Make Writing style of report consistent - 3rd person.
- Compare Terraform and Pulumi.
- Add references.

Week 7:

Date: 12/11/19

Time: 12:15

Attendance: Daemon Macklin, Joe Daly

Report on previous week's work:

- Discussed progress on report.

Plan for upcoming weeks work:

- Plan for what the proof of concept demo will be
- Who would use MyT and why?
- Speak to potential users
- Development methodology
- Gantt chart

Additional Notes

Week 8:

Date: 19/11/19

Time: 12:15

Attendance: Daemon Macklin, Joe Daly

Report on previous week's work:

- Chosen technologies for MyT

Plan for upcoming weeks work:

- User data security.
- Potential Applications of MyT

Additional Notes

Week 9:

Date: 26/11/19

Time: 12:15

Attendance: Daemon Macklin, Joe Daly

Report on previous week's work:

- Met with Jason Berry to discuss project

Plan for upcoming weeks work:

- Use cases.
- Being reviewing paper.
- State diagram of entire project.
- Start working on proof of concept.

Additional Notes

Week 10:

Date: 5/12/19

Time: 12:15

Attendance: Daemon Macklin, Joe Daly

Report on previous week's work:

- Report first draft finished.

Plan for upcoming weeks work:

- Revise report.
- Format references correctly.

Additional Notes

Week 17:

Date: 20/1/20
Time: 2:45
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:

Plan for upcoming weeks work:

- Investigate SQL feasibility.
- Scope Project development.

Additional Notes

Week 18:

Date: 27/1/20
Time: 2:45
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:

- Creation of Weekly cycles for developments
- Aim to have the project finished by 23rd March.

Plan for upcoming weeks work:

- Terraform For AWS
- Extenal Storage for DBs
- API - Create platform

Additional Notes

Week 19:

Date: 2/2/20
Time: 2:45
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:

- Terraform For AWS
- Extenal Storage for DBs
- API - Create platform

Plan for upcoming weeks work:

- Create User
- Save Cloud Service Credentials
- Terraform - Dynamic deployment settings

Additional Notes

Week 20:

Date: 10/2/20
Time: 2:45
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:

- Create User
- Login
- SQLite Models
- Save Cloud Service Credentials
- Encrypt Cloud Service Credentials
- Terraform - Dynamic deployment settings

Plan for upcoming weeks work:

- Delete Platform
- RabbitMQ Username/Password

- MongoDB

Additional Notes

Week 21:

Date: 24/2/20
Time: 2:45
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:

- Delete Platform/Space/User
- RabbitMQ Username/Password
- MongoDB
- Vue Frontend - Caught up with backend

Plan for upcoming weeks work:

- Catch up on cycle 4
- DB backups
- RabbitMQ TLS

Additional Notes

Week 22:

Date: 2/3/20
Time: 2:45
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:

- Update user data processing
- Custom user imports done
- Database dump nearly complete

Plan for upcoming weeks work:

- Catch up on cycle 5 and 6

Additional Note- One week behind

Week 22:

Date: 9/3/20
Time: 2:45
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:

- Catch up on cycle 5 and 6
- Database DUmps
- TLS Support for RabbitMQ
- Platform Montoring

Plan for upcoming weeks work:

- GCP + Openstack support
- SQL support
- Seek feedback on project

Additional Notes

Week 23:

Date: 23/3/20
Time: 2:45
Attendance: Daemon Macklin, Joe Daly
Report on previous week's work:

- SQL Done
- GCP + Openstack Done
- Tidy frontend

Plan for upcoming weeks work:

- Poster and report

Additional Notes

References

Ansible, R., 2020. Ansible Is Simple IT Automation. [online] Ansible.com. Available at: <https://www.ansible.com/> [Accessed 8 April 2020].

axios. 2020. Axios. [online] Available at: <https://www.npmjs.com/package/axios> [Accessed 7 April 2020].

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R. and Kern, J., 2001. Manifesto for agile software development.

Docker Documentation. 2020. Overview Of Docker Compose. [online] Available at: <https://docs.docker.com/compose/> [Accessed 14 April 2020].

Docs.ansible.com. 2020. File – Manage Files And File Properties — Ansible Documentation. [online] Available at: <https://docs.ansible.com/ansible/latest/modules/file_module.html#file-module> [Accessed 13 April 2020].

Docs.ansible.com. 2020. Git – Deploy Software (Or Files) From Git Checkouts — Ansible Documentation. [online] Available at: <https://docs.ansible.com/ansible/latest/modules/git_module.html> [Accessed 13 April 2020].

Flask.palletsprojects.com. 2020. Welcome To Flask — Flask Documentation (1.1.X). [online] Available at: <https://flask.palletsprojects.com/en/1.1.x/> [Accessed 7 April 2020].

Fullstackpython.com. 2020. Object-Relational Mappers (Orms). [online] Available at: <https://www.fullstackpython.com/object-relational-mappers-orms.html> [Accessed 10 April 2020].

Grafana. 2020. [online] Available at: <https://grafana.com/> [Accessed 15 April 2020].

Grafana Labs. 2020. Simplejson Plugin For Grafana. [online] Available at: <https://grafana.com/grafana/plugins/grafana-simple-json-datasource/installation> [Accessed 15 April 2020].

InfluxDB. 2020. Influxdb: Purpose-Built Open Source Time Series Database | Influxdata. [online] Available at: <https://www.influxdata.com/> [Accessed 15 April 2020].

MongoDB. 2020. The Most Popular Database For Modern Apps. [online] Available at: <https://www.mongodb.com/> [Accessed 15 April 2020].

Mysql. 2020. Mysql. [online] Available at: <https://www.mysql.com/> [Accessed 15 April 2020].

npm. 2020. Vue-Tables-2. [online] Available at: <https://www.npmjs.com/package/vue-tables-2> [Accessed 6 April 2020].

nvie.com. (2020). A successful Git branching model. [online] Available at: https://nvie.com/posts/a-successful-git-branching-model/ [Accessed 6 Feb. 2020].

PyPI. 2020. Cryptography. [online] Available at: <https://pypi.org/project/cryptography/> [Accessed 10 April 2020].

PyPI. 2020. Flask-JWT-Extended. [online] Available at: <https://pypi.org/project/Flask-JWT-Extended/> [Accessed 8 April 2020].

PyPI. 2020. Peewee. [online] Available at: <https://pypi.org/project/peewee/> [Accessed 10 April 2020].

PyPI. 2020. Pika. [online] Available at: <https://pypi.org/project/pika/> [Accessed 25 March 2020].

PyPI. 2020. Python-Terraform. [online] Available at: <https://pypi.org/project/python-terraform/> [Accessed 8 April 2020].

Raspberrypi. 2020. [online] Available at: <https://www.raspberrypi.org/> [Accessed 15 April 2020].

Semantic-ui.com. 2020. Semantic UI. [online] Available at: <https://semantic-ui.com/> [Accessed 6 April 2020].

SenseHat. 2020. [online] Available at: <https://www.raspberrypi.org/products/sense-hat/> [Accessed 15 April 2020].

Terraform by HashiCorp. 2020. Terraform By Hashicorp. [online] Available at: <https://www.terraform.io/> [Accessed 8 April 2020].

Timescale. 2020. [online] Available at: <https://www.timescale.com/> [Accessed 15 April 2020].

Tools.ietf.org. 2020. RFC 6265 - HTTP State Management Mechanism. [online] Available at: <https://tools.ietf.org/html/rfc6265> [Accessed 7 April 2020].

Tools.ietf.org. 2020. RFC 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2). [online] Available at: <https://tools.ietf.org/html/rfc7540> [Accessed 7 April 2020].

Trello.com. 2020. Trello. [online] Available at: <https://trello.com/> [Accessed 1 April 2020].

vue-cookies. 2020. Vue-Cookies. [online] Available at: <https://www.npmjs.com/package/vue-cookies> [Accessed 7 April 2020].

vue-login-form. 2020. Vue-Login-Form. [online] Available at: <https://www.npmjs.com/package/vue-login-form> [Accessed 6 April 2020].

vue-spinner. 2020. Vue-Spinner. [online] Available at: <https://www.npmjs.com/package/vue-spinner> [Accessed 6 April 2020].

Vuejs.org. 2020. Vue.Js. [online] Available at: <https://vuejs.org/> [Accessed 6 April 2020].

@smartweb/vue-flash-message. 2020. @Smartweb/Vue-Flash-Message. [online] Available at: <https://www.npmjs.com/package/@smartweb/vue-flash-message> [Accessed 6 April 2020].