Roan Wassink

P1721217x

BSc Computer Science

De Montfort University  
Supervisor: Sarmadullah Khan

MicroMonitor

Final year project report

Contents

[1. Introduction 2](#_Toc40553557)

[Project Specification 3](#_Toc40553558)

[Scope 3](#_Toc40553559)

[Deliverables 3](#_Toc40553560)

[Justification 3](#_Toc40553561)

[Constraints 3](#_Toc40553562)

[Assumptions 3](#_Toc40553563)

[2.0 Background 4](#_Toc40553564)

[Aim 4](#_Toc40553565)

[Alternative solutions 4](#_Toc40553566)

[Nagios 4](#_Toc40553567)

[Zabix 4](#_Toc40553568)

[2.1 Glossary 4](#_Toc40553569)

[2.1 functional requirements 4](#_Toc40553570)

[Interface requirements 4](#_Toc40553571)

[3. 0 Specification and design 5](#_Toc40553572)

[3.1 Interface overview, functions and use cases 5](#_Toc40553573)

[3.1.1 Index page log 5](#_Toc40553574)

[3.2 Data flow 9](#_Toc40553575)

[3.2.1 Front-end 9](#_Toc40553576)

[3.2.2 Backend 11](#_Toc40553577)

[3.3 Structure 11](#_Toc40553578)

[4.0 Implementation 13](#_Toc40553579)

[4.1 Python files 13](#_Toc40553580)

[4.1.1 \_\_init\_\_.py 13](#_Toc40553581)

[4.1.2 routes.py 13](#_Toc40553582)

[4.1.3 models.py 17](#_Toc40553583)

[4.2 templates 18](#_Toc40553584)

[4.2.1 base.html 18](#_Toc40553585)

[4.2.2 Index page 19](#_Toc40553586)

[4.2.3 CPU and memory page cpu.html, memory.html 20](#_Toc40553587)

[4.2.4 registration page 21](#_Toc40553588)

[4.3 Client and server script 22](#_Toc40553589)

[4.3.1 Client.py 22](#_Toc40553590)

[4.3.2 Server data.py 23](#_Toc40553591)

[5. Results and evaluation 24](#_Toc40553592)

[5.1 Testing 24](#_Toc40553593)

[Test environment 25](#_Toc40553594)

[5.2 Evaluation 27](#_Toc40553595)

[6.0 Reflection 28](#_Toc40553596)

[6.1 Future work 28](#_Toc40553597)

[Database 28](#_Toc40553598)

[Missing functions 29](#_Toc40553599)

# Introduction

This is a full development report for the 3rd year final year project for Computer Science degree at De Montfort University Leicester. The report describes the monitoring system known as MicroMonitor. MicroMonitor is a system monitoring application that is developed using Python 3. The application can monitor the:

* CPU usage of a system
* Memory usage
* Disk usage

This report is intended to be read by the project supervisor and any reader that is interested in the development of a web application using a non-standard method. The report will contain the planning of the application, overview of the application, the technical details on how that application is coded and the evaluation.

## Project Specification

### Scope

This project involves writing an application that can gather the following metric: CPU usage, CPU temperature, RAM usage and disk usage. And then display and visualize the metrics on a web interface accessible from any device in the local area network. The web interface must support graphs and tables. The gathered metrics need to be logged so that they can be accessed at any time. The application needs to be accessible only to those with the correct credentials.

### Deliverables

The following deliverables are included in the scope of the project:

* Project contract
* Literature review to decide if Python is a viable option
* Project documentation (this document)
* Monitoring application front end
* Data gathering client script
* Data gathering server

### Justification

The project is started to complete the BSc requirements mandated. The topic of monitoring application has been chosen out of interest for data visualization and the need for a simple application that allows for the monitoring of various servers in the local network. The experience gained is very useful for later stages of personal professional development.

### Constraints

The main constraint is the lack of experience managing the complete development cycle of a fully functional application. Various skills will have to be self-taught along the way. Furthermore, no extra hardware will be available, any testing and all development is limited to virtual machines.

### Assumptions

A project runs under the assumption that certain issues will not arise as they are unable to be prevented if they do arise.

No material defects, loss of data or financial constraints are encountered during the development of the application. Material defects, such as a failing power supply, encountered on the development workstation hinder the progression of the development. Loss of data can cause setbacks if there are no mitigations in place. Financial or health issues arising from the Covid-19 pandemic can cause setbacks as well. The project will commence under the assumption that no such issue will arise.

The skill and knowledge necessary to complete the project are mastered or can be mastered within the time limit.

Necessary libraries have no unforeseen limitations that prevent the project from implementing the entire scope.

# Background

## Aim

The project is launched with the goal to develop an application to make it easier to monitor servers and clients. Existing solutions often provide this functionality with many features, but the configuration is too complicated. MicroMonitor is an attempt to provide a simple solution that requires little configuration. The goal is to make a monitoring application that does what it needs to do without any extra bells and whistles. The installation should be as simple as possible and the interface should be functional.

## Alternative solutions

### Nagios

Nagios provides an incredible number of features such as process monitoring. The scope of Nagios and Zabix is clearly much larger. The setup however is advanced and requires a great amount of dependencies and a great deal of steps before it becomes functional. [Nagios setup on CentOS](https://support.nagios.com/kb/article/nagios-core-installing-nagios-core-from-source-96.html#CentOS).

### Zabix

Zabix is aimed at large organizations and they even sell a course for the setup. It is not ideal if you are looking for a quick and easy solution that does not require many features. [Zabix installation course](https://www.udemy.com/course/zabbix-network-monitoring-essentials/?LSNPUBID=jf7w44yEft4&ranEAID=jf7w44yEft4&ranMID=39197&ranSiteID=jf7w44yEft4-KG6cdue6aHz5DU63PjBcxw)

## Glossary

|  |  |  |
| --- | --- | --- |
| Item | Explanation | Further reading |
| Matplotlib | Matplotlib is a Python library that draws a multitude of graphs. The most common use draws line graphs. See Monitor CPU usage. | https://matplotlib.org/ |
| Mpld3 | Python library that converts Matplotlib graphs into a JavaScript plot so that the client can run it client side. | https://mpld3.github.io/ |
| SQLalchemy | SQLalchemy is essentially an abstraction layer on top of a database that provides Python syntax and context. | https://www.sqlalchemy.org/ |
| Flask | Flask is a microframework that allows for websites to be written with Python. | https://flask.palletsprojects.com/en/1.1.x/foreword/ |
| Jinja | Jinja is a templating engine that works in conjunction with Flask. They are developed by the same person. | https://Jinja.palletsprojects.com/en/2.11.x/ |

## functional requirements

The functional requirements were drawn during the first deliverable. They will be compared to the finished product in section 5.1.

### Interface requirements

* Users must be able to authenticate
* Users must be able to access monitoring page only when authenticated
* The application must have a navigation bar at the top of the screen for quick navigation
* Interface must be consistent throughout the application
* Application must be accessible from a web browser
* Dashboard overview presented when logged in
* Selection of devices available for monitoring
* Monitoring of processes
* Monitoring of temperatures
* Viewing of warnings
* Users can select which device to view
* Users can add device on the network
* User can change password
* Logging of events

# Specification and design

The application is called MicroMonitor named because it is a small application that monitors devices. MicroMonitor is a browser app made to make monitoring of devices on the local network possible. MicroMonitor currently supports the monitoring of Windows and Linux based devices.

It can read several different metrics from connected. These are:

* CPU usage
* Disk space
* RAM Usage
* Amount of CPU cores
* CPU frequency
* Hostname
* Operating system
* Total memory
* Total disk space

CPU temperature was originally planned as a feature but is not implemented in the final version. This is due to a library incompatibility. More about that in section 4.3.1 Client.py.

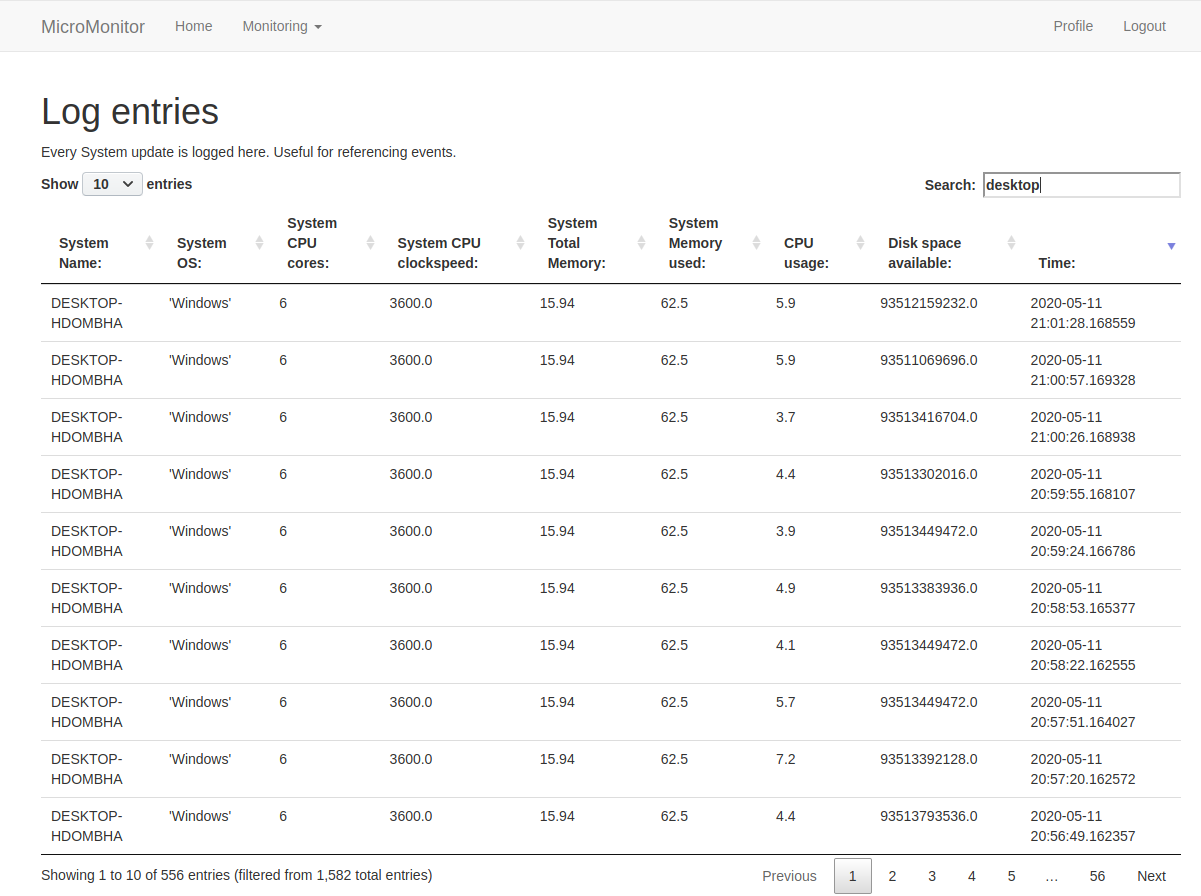


Image Application homepage

## Interface overview, functions and use cases

When first opening the interface, the administrator is greeted with a login screen. The login screen prevents unauthorized users from accessing the main website. While logging metrics might not be considered sensitive data, it can still be used to gain knowledge about which devices are on the local network.

To use the application a new account must be made first. Currently this has no email verification attached, as it is meant as a place holder. The application in production would have a feature to predefine user accounts. After logging in the user will be redirected to the index page after which they have access to all the site functions.

### Index page event log

The index page is an overview page displaying all the log entries of each metric that has been inserted into the database. It can be used to retroactively view the history of each device. System crashes can now be traced and troubleshooted by matching the time of the crash with the timestamp in the system.

#### Sorting headers

The log entries can be sorted by clicking the headers on the page. [See image 1] This is the default sorting. All the columns with text data make use of alphabetical sorting, either ascending or descending.

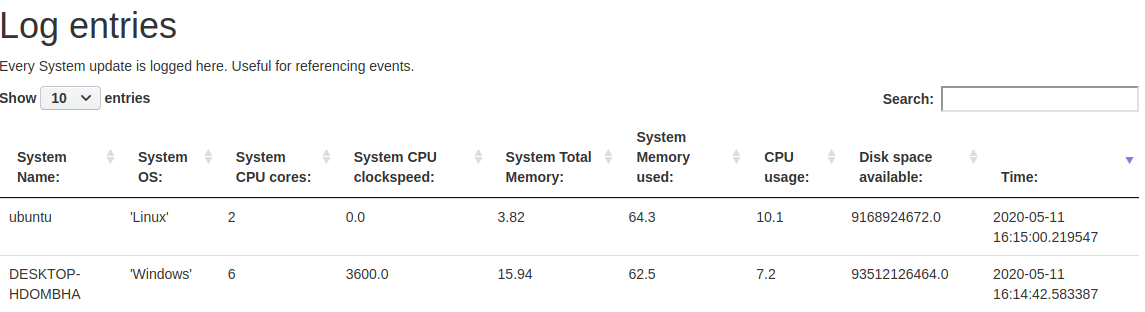


Image sorted by time (not every result is shown)

#### Filtering displayed data

The table can be filtered by using the search function. For example: find the highest CPU usage for system “ubuntu”. (The system name is the hostname of the registered system.) “ubuntu” in the search box and sorted by CPU usage descending. [image 2]

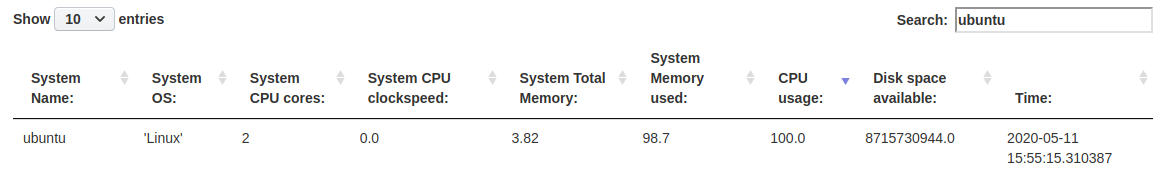


Image highest CPU usage for ubuntu

#### Filtering by date

Similar operations can be performed for all the other fields. Data can be entered to see all the entries from that date. [see image 3]

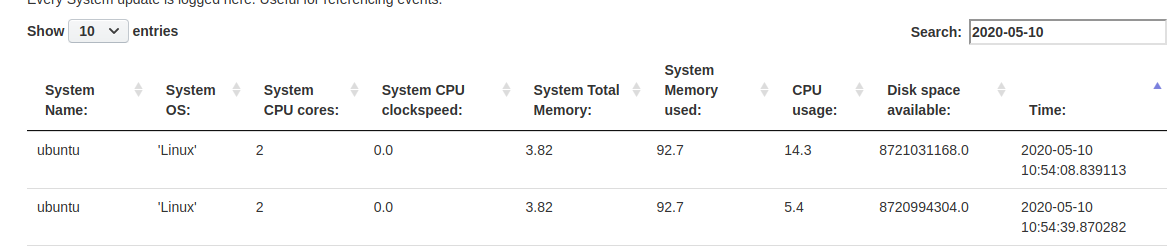


Image all the entries from 2020-05-10

### Navigation bar items

The navigation bar at the top has a dropdown menu titled “monitoring”. This dropdown contains all the currently implemented monitoring items.

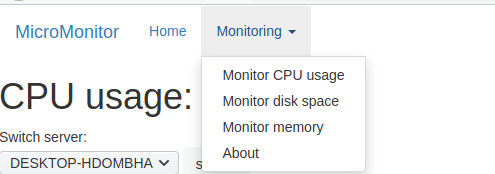


image Opened dropdown menu

#### Monitor CPU usage

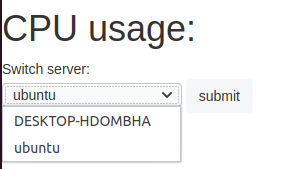
The CPU monitoring page contain an interactive graph and a dropdown menu. The dropdown menu is automatically populated with each available system.   


image system selection

The graph shows the CPU usage on the y-axis and the time on the x-axis. Hoovering over any point on the graph will show a tooltip with the exact CPU usage and time.

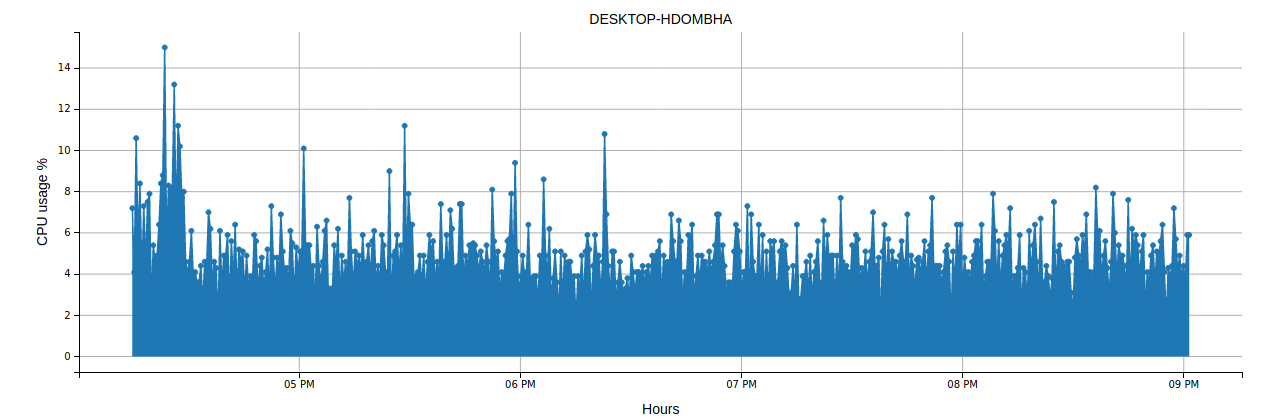


image CPU usage graph

The graphs allow for resizing and moving the graph in every direction. This can be helpful if a bigger picture is needed or certain time needs closer examination. When viewing the site on a mobile device the zoom function is necessary because of the smaller screen.

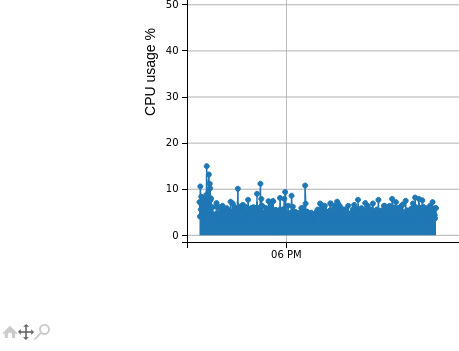


image Resizing with toolbar notice toolbar in the bottom left

#### Visualize disk space

The disk space page shows a pie chart of the given system. The free and used space unit is changed according to the size of the respective section. I.e. 0.1 GB will be displayed as 10MB.

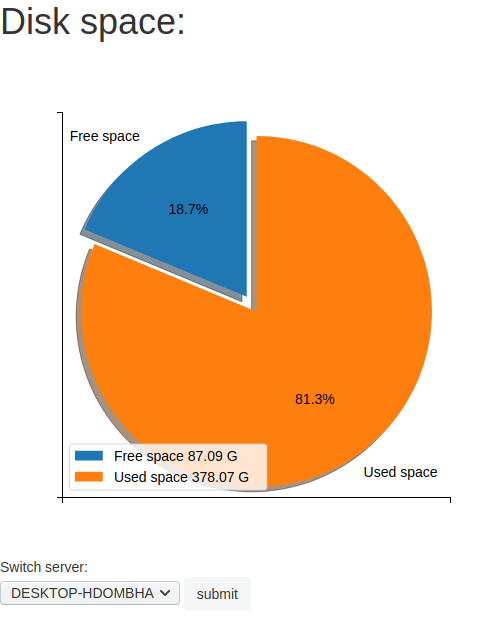


image Disk space pie chart

#### Memory usage

The memory graph works the same as the CPU graph but with different values. Screenshots omitted for the sake of brevity.

## Functional design

### Basic application data flow

The workstations and servers that are to be monitored, called clients, have a Python script running. The client script sends data to the monitoring server using a socket. The socket supports up to 50 clients at once.

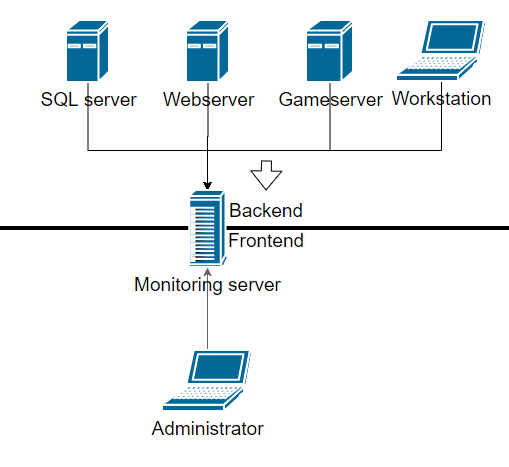


image Scenario example

The monitoring server similarly runs a script that waits for the clients to connect. As soon as it receives data through the socket it writes it to the local database for the front-end to display. The client script sends out its data every 30 second by default. The time can be shortened or lengthened if needed. Keep in mind that a shorter time means a larger database and incurs some performance penalty.

### Front-end architecture

The front end is ultimately the part of the code that is going to be displayed on the browser. The front end renders the page and makes sure the web client can execute scripts.

The front end makes use of JavaScript, Jinja2 and Twitter Bootstrap.

* JavaScript is used to draw the graphs and for the display of the table.
* Jinja is a popular template engine that allows for syntax within the HTML documents. It works in conjunction with Python to retrieve and pass variables to and from the backend.
* Twitter Bootstrap is style sheet engine made to stylize webpages according to a template. It provides for navigation bars, tables, padding and more.

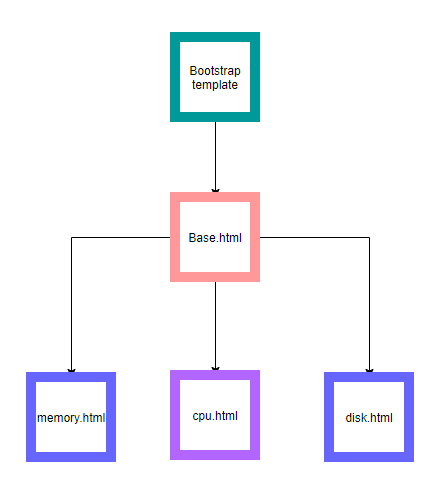


image Page hierarchy

The frontend makes use of a 3-tier hierarchical structure. Bootstrap sits on top as it defines the main layout of the webpages. After that comes the base which includes the navigation bar exports an app content block, using Jinja, where other webpages are inserted in a container. This way all the other pages follow the layout rules mandated by the base file and in turn the Bootstrap template.

The URL’s are dynamically defined with the “url\_for” syntax. The URL for a specific function is defined at the backend and the url\_for points at that code. This means that the URL can be easily changed without having to reshuffle and rename pointers per HTML file. It also allows for a modular development of new features. New pages only need to be specified in the backend and possible links need to be made in the navigation bar.

The frontend renders the graph using a Python plugin, “mpld3”, that essentially converts Matplotlib objects to a JavaScript object. This then allows for the graphs to be interactive. Mpld3 itself has the functionality to specify plugins for it. The graphs drawn make use of a plugin that provides for hoover-over tooltips that provide more detail about a specific point in the graph.

The index page makes use of another JavaScript called Datatables. Datatables makes the tables interactive in a similar fashion as mpld3.

### Backend description

#### Used libraries and their function

MicroMonitor makes use of a variety of different libraries and frameworks to render the webpages. The entire application is built on top of Flask framework. Flask interacts with Jinja to pass variables into an HTML compliant format. Flask is chosen for this project as it very modular by nature. It has many functionalities, but they do not have to be used. Flask also has a community making extensions for it.

Flask works together with Werkzeug. Werkzeug is a flask extension that handles the actual creation of the WebSocket using WSGI, and this allows for connections to be made to it. It also provides security functions such as encryption and validation.

Another extension is SQLalchemy. SQLalchemy is used to provide an abstraction layer on top of a database service. In this case SQLite. It is possible to use Pythonic language with the database instead of SQL. The database structure is specified in the models.py file inside the project structure. This allows for easy creation and changing of tables. The database ERD is shown in image 12.

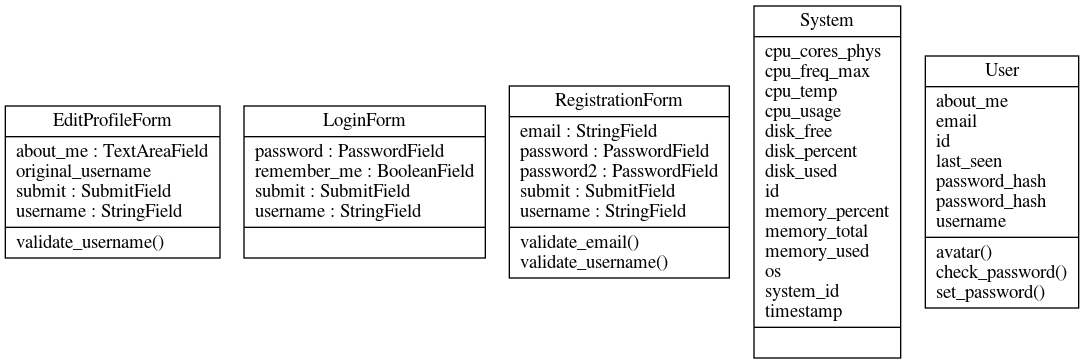
MicroMonitor currently makes use of a single table that stores all the System metrics. This is not optimal and is due for change. That unfortunately requires quite extensive rework of the queries. The current solution is however functional and is thus good enough for now. 

image Database tables and fields

Matplotlib as mentioned earlier, is a pure Python library that turns lists into a variety of different graph types. MicroMonitor uses it to draw line graphs and pie charts. Matplotlib on its own, can turn a plotted object into an image. This image can be displayed on the webpage; however, it will not support interactions such as zooming. Mpld3 is then called to turn it into a JavaScript object which means it can be loaded client side. This allows for the graphs to be interactive.

## Structure

Each webpage is defined within routes.py which in return calls for the Werkzeug and Jinja to render the webpage. The application makes use of the separation of concern and inheritance concepts. Each file has its own job and only knows things about other files on a need to know basis.

The \_\_init\_\_.py file is where flask, flask’s addons, SQLalchemy and Bootstrap are defined. The other files then import from that. This is to keep the code readable, modular and reusable.

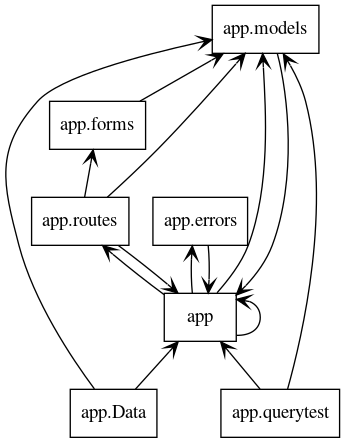


image Class diagram app is essentially \_\_init\_\_.py

#### Index page (log page)

The routes function for the index page queries the database for each field that needs to be included and passes it on to the Jinja template for index.html. Jinja then runs a for loop on that to populate each row according to the header. Datatables makes the table pretty.

#### CPU and memory page

The routes.cpu function reads the selected system in the dropdown [image 6]. The chosen value is passed on by html using GET and uses that to query the database for that systems timestamp and CPU usage data.

Ie: System.timestamp, System.cpu\_usage, System.system\_id [See image 11]

The page on startup has no system selected so a default one is used instead. The database returns a tuple that is then used to plot the data with Matplotlib. Mpld3 takes the Matplotlib and this object is then returned to Jinja to render it in the HTML file.

Unfortunately, mpld3 has no way of knowing about the zoom properties of the page so viewing it on a mobile device with a low resolution might break it. A solution for this needs to be implemented in a future version.

The memory page works the same but with different database queries.

System.timestamp, System.memory\_percent, System.system\_id

#### Disk space

The disk space page is fundamentally the same as the page in the previous section. With the difference that it is a pie chart that is rendered instead of a graph.

#### Application logging

The application keeps track of events using logs. These show detailed error reports but also startup and closing events.

# Implementation

This section will follow the same general structure as the Specification and design section. It will go into more detail and will contain code snippets that are deemed important or interesting. Pages that work in similar fashion as one already described will also be omitted simply because it is repetitive. Cruxes encountered during the development will also be mentioned per section.

## Python files

### \_\_init\_\_.py

This file as mentioned in section 3.3 is where all the environmental variables are called and made available to the rest of the package.



Note the import statement for routes, models and errors.

### routes.py

Most of the application logic reside in this file. It defines the routes that url\_for refer to and it makes variables available to Jinja. It can also get variables from Jinja something which was quite confusing at first but turned out to be simple. The code snippet below is only a small part of the full code which can be found in the GitHub submission.

#### Route\_for Index page

Below is the query used for the logging/index page. It simply requests specific fields from the database and the .all() function retrieves it in the form of a dictionary (key value pairs).

systems = session.query(System.system\_id, System.os, System.cpu\_cores\_phys, System.memory\_total,System.memory\_percent, System.cpu\_freq\_max, System.cpu\_usage, System.timestamp, System.disk\_free).all()

The query is then simply made available to the index.html Jinja template which then uses it to generate the interactive table.

return render\_template('index.html', title='Home', systems=systems)

#### Route for CPU

A little more interesting page is the CPU usage monitoring page. It has some logic that makes the “select system” selection box functional. systemSelection requests the value entered in the box using GET. If there is no system selected a default selection will be used. This prevents 404 errors.

The queries retrieve System.cpu\_usage and System.timestamp separately filtering the results based on system\_id = systemSelection.

systemSelection = request.form.get("system")

# Prevents error when first opening the page. Effectively makes 'ubuntu' the default selection.

if systemSelection is None:

# Query cpu\_usage

query = session.query(System.cpu\_usage).filter\_by(system\_id='ubuntu').all()

# query Timestamps

query2 = session.query(System.timestamp).filter\_by(system\_id='ubuntu').all()

systemSelection = 'ubuntu'

else:

query2 = session.query(System.timestamp).filter\_by(system\_id=systemSelection).all()

query = session.query(System.cpu\_usage).filter\_by(system\_id=systemSelection).all()

The result is a dictionary. To support hoover-over tooltips the dictionary needs to be made into a flat list of values. This is done using a for loop to loop over each entry and add it to a new list removing the nesting.

cpu\_list = []

timestamp\_list = []

# add cpu usage to x

for row in query:

cpu\_list.append(row)

# add timestamp to y

for row2 in query2:

timestamp\_list.append(row2)

The following code is what generates the graph for this page.

First the figure object is made on which the axes are defined “ax”.   
Then the lines for the axes are made and some options are set such as the labels and grid.

figure = pyplot.figure(figsize=(15, 5))

figure.autofmt\_xdate()

ax = pyplot.axes()

lines = ax.plot(time\_list, cpu\_list, marker='o', ls='-', ms=5)

ax.fill\_between(time\_list, cpu\_list)

ax.set\_title(str(systemSelection))

ax.set\_xlabel("Hours")

ax.set\_ylabel("CPU usage %")

ax.grid()

To get the proper format for the hoover-over tooltips some string formatting had to be done inside a for loop:

for time, cpu in zip(time\_list, cpu\_list):

timehour = [

"cpu " + str(cpu) + " time " + " " + str(time.hour) + ":" + str(time.minute) + ":" + str(time.second)]

hour.append(timehour)

Lastly mpld3 is called to generate an html compatible graph that supports user interaction. Tooltip is the variable which uses the previously changed string format together with the values of the lines in the graph. Plugins.connect connects the plugins to the mpld3 object which is then finalized at the html\_text variable. Everything is then passed on to Jinja like every other page defined in routes.py

tooltip = plugins.PointHTMLTooltip(lines[0], labels,

voffset=10, hoffset=10, css=css)

plugins.connect(figure, tooltip)

html\_text = mpld3.fig\_to\_html(figure)

#return the graph, page and the list of available systems.

return render\_template('cpu.html', plot=html\_text, systems=uniqueList)

#### Route for registration page

The registration page has not come up before. It is simply not that interesting. Make an account, login and you are good to go. The technical implementation, however, is interesting enough for it to be featured.

When a user registers and chooses a new password, the password is hashed, and the hashed password is stored in the database. In the User table to be precise [see image 11]. The function for register inside routes.py is just specifying which fields the form contains and the actual data handling is left to the model.

@app.route('/register', methods=['GET', 'POST'])

def register():

# If the current user is already authenticated send them back to the main page.

if current\_user.is\_authenticated:

return redirect(url\_for('index'))

#make a new form called form

form = RegistrationForm()

#if the form is validated enter data from the form into the database User table

if form.validate\_on\_submit():

user = User(username=form.username.data, email=form.email.data)

#special function specified in the model’s class. See section 4.1.3.

user.set\_password(form.password.data)

db.session.add(user)

db.session.commit()

#confirmation message

flash('Congratulations, you are now a registered user!')

#redirect to login page

return redirect(url\_for('login'))

#return the form to the Jinja template “register.html”

return render\_template('register.html', title='Register', form=form)

This function is not very exciting, but it is required to understand how the data gets handles on both the front and back end. The validation is also not handled in the routes class.

### models.py

The models’ file is where the structure of the data is decided. It also handles data send throughout the entire application. SQLalchemy reads this file when generating a database.

class User(UserMixin, db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(64), index=True, unique=True)

email = db.Column(db.String(120), index=True, unique=True)

password\_hash = db.Column(db.String(128))

about\_me = db.Column(db.String(140))

last\_seen = db.Column(db.DateTime, default=datetime.utcnow)

def avatar(self, size):

digest = md5(self.email.lower().encode('utf-8')).hexdigest()

return 'https://www.gravatar.com/avatar/{}?d=identicon&s={}'.format(

digest, size)

def \_\_repr\_\_(self):

return '<User {}>'.format(self.username)

def set\_password(self, password):

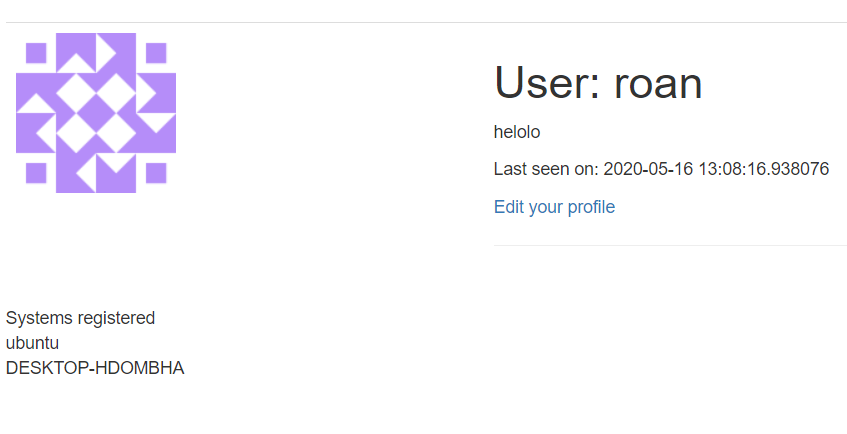
self.password\_hash = generate\_password\_hash(password)

def check\_password(self, password):

return check\_password\_hash(self.password\_hash, password)

The columns of the User table are defined by SQLalchemy in this file. The datatype and which properties each column has is also set here. SQLalchemy reads this, tracks the changes and updates the database file accordingly. The function set\_password and check\_password take care of storing the non-hashed password in a hashed format and loading the hashed password in a human readable format. The actual hashing is done by Werkzeug. UserMixin is another function that comes with Flask. It allows for the users’ login state to be tracked. This works with the @login\_required handler.

The gravatar() function fetches a profile picture from Gravatars service, if it does not exist it will generate a geometric image.



## templates

### base.html

The base.html page decides how the other pages are displayed and defines the navigation bar. This is how it work:

Jinja works with blocks. Blocks start with {% something %} and close with {%endblock%}.

The extends block makes the base.html comply to the layout rules defined by Bootstrap. [see image 11]. After that is the title block which sets the title to use throughout the entire site. The navbar block defines a navbar which is a function defined within a Bootstrap which is in turn inherited by base.html.

{% extends 'bootstrap/base.html' %}

{% block title %}

{% if title %}{{ title }} - MicroMonitor{% else %}Welcome to MicroMonitor{% endif %}

{% endblock %}

{% block navbar %}

Navbar code goes here

{% end block %}

The content block specifies where the actual application content fits. The base.html itself does not provide any content. That is why the block app\_content is specified. It functionally imports other html pages and inserts it in the app\_content block. The block itself is also in a container so that it fits on a specific place on the screen.

% block content %}

<div class="container">

{# application content needs to be provided in the app\_content block #}

{% block app\_content %} >Content goes here< {% endblock %}

</div>

{% endblock %}

### Index page

Every page starts with an {% extends base.html %} block. The application content fits within the app\_content blocks which insert into the base.html blocks. All the normal html tags are used from then on. The table is specified here with the <table id> being important for DataTables. The rows are inside a Jinja for loop making use of the “systems” variable provided by Routes.index [see 4.1.2: Route for index]. This is how the table is populated. This shows how Jinja and Python-flask work together.

#### Script block

After the content block comes the scripts block. In order for DataTables to work JavaScript is needed. However, Bootstrap already includes JavaScript so importing JS in the normal way would reinitialize the table removing all the content. This was difficult to figure out and more time was spent on this than expected. The solution is to put the script imports in a script block and call the super() function. This overrides the Bootstrap JS. And prevents it from altering the table. The DataTables then imports normally and the table is displayed correctly.

{% block app\_content %}

<body>

<table id="systemTable" class=table table-hover">

<table headers go here, one for each column>

{% for system in systems[1:]%}

<tr>

<td>{{system.system\_id}}</td>

...

</tr>

{% endfor %}

</tbody>

</table>

</body>

{% endblock %}

% block scripts %}

{{super()}}

<script src="https://code.jquery.com/ui/1.12.1/jquery-ui.js"></script>

<script type="text/javascript" language="javascript" src="https://cdn.datatables.net/1.10.20/js/jquery.dataTables.min.js"></script>

<script type="text/javascript" class="init">

$(document).ready(function() {

$('#systemTable').DataTable( {

"order":[[8, "desc"]]

})

} );

</script>

### CPU and memory page cpu.html, memory.html

Once again because the CPU and memory page work identically only code from the CPU will be shown here.

The page content is inserted into the app\_content block. The dropdown menu called system is seen here. This where Routes.cpu gets the systemSelection from [see 4.1.2: Route for CPU]. The mpld3 graph is displayed with {{plot|safe}}

{% block app\_content %}

<h1>CPU usage:</h1>

<span class="dropdown-text">Switch server:</span>

<form name = "system" action = ' 'method ="POST" >

<select name="system" method="GET" action="/">

< !--<option value="{{systems[0]}}"{ selected>{{systems[0]}}</option>-->

{% for system in systems[1:] %}

<option value="{{system}}">{{system}}</option>

{% endfor %}

</select>

<input class="btn" type="submit" value="submit">

</form>

{{plot|safe}}

{% endblock %}

### Registration page

The registration form is rendered by bootstrap wtf. It automatically renders a form based on the variables defined in 4.1.2: routes for registration page. The advantage is of course that there is no need to painstakingly match each form with a new variable.

% extends "base.html" %}

{% import 'bootstrap/wtf.html' as wtf %}

{% block app\_content %}

<h1>Edit Profile</h1>

<div class="row">

<div class="col-md-4">

{{ wtf.quick\_form(form) }}

</div>

</div>

{% endblock %}

## Client and server script

### Client.py

The client script has three main functions that need some explaining.

1. Establishing a socket connection to a specific IP address/port
2. Collecting data from the system using Pythons’ psutil package.
3. Sending the data over the socket

Establishing a connection is done as follows. This is one of the only things that need to be configured in order to use the application.

s=socket.socket()

s.connect(('192.168.10.11', 5002))

Collecting data works as follows within a while loop so the variables refresh. Note the get\_size() function which dynamically appends the correct post fix on a memory size.

svmem = psutil.virtual\_memory() memory\_total = float(get\_size(svmem.total)) memory\_used = float(get\_size(svmem.used))

Sending the data works within the same while loop. The metrics are converted to a string format to make sure that it is of a uniform type which makes decoding easier. Then the list of strings is send over the socket. It makes use of TCP so it will receive a confirmation.

metrics = [system\_id, cpu\_usage, disk\_percent, disk\_free, disk\_used, os, cpu\_cores\_phys, cpu\_freq\_max, memory\_total, memory\_used, memory\_percent]

data = str(metrics)

s.send(data.encode())

One limitation has been encountered and that is the inability to read CPU temperature data from Windows machines and from any Virtual machine. There are some workarounds, but they require more dependencies and some closed source dll’s. In the end the feature has been put on hold.

### Server data.py

The server script is made to support multiple connections at once and close connections that are unused. It does that using a reader. A reader simply checks if a connection is active and if it receives data through that connection. If there is no data, it disconnects and removes the client from the list of clients.

If there is data, it first splits it up using the comma symbol. Then it removes all the square brackets and quote symbols. This is needed so that each value is of the correct type.

Then the script assigns each index in the resulting list to a variable. This allows the script to put each variable in the database in the right format.

with socket.socket() as server:

server.bind(('192.168.10.11',5002))

server.listen(50)

to\_read = [server] # add server to list of readable sockets.

clients = {}

…

data = reader.recv(4096)

data = data.decode('utf-8')

if not data: # No data indicates disconnect

#print('disconnected',clients[reader])

to\_read.remove(reader) # remove from monitoring

del clients[reader] # remove from dict as well

..

data = data.strip('][').split(', ')

system\_id, cpu\_usage, disk\_percent, disk\_free, disk\_used, os, cpu\_cores\_phys, cpu\_freq\_max, memory\_total, memory\_used, memory\_percent = data

db.session.add(p)

db.session.commit()

# Results and evaluation

## Testing

The full test plan can be found in the first deliverable. Only the case, expected result, the actual result and the method will be shown here. The environment on which the tests are conducted is also essential for testing integrity.

### Test environment

* Application runs inside an Ubuntu 19.10 virtual machine on VMware
* Python 3.7
* VM is assigned 2 cores
* VM is assigned 4GB of memory
* VM runs of a virtual disk which is stored on a NVME SSD
* Host machine
  + I5 8600k
  + 16GB ram
  + Windows 10 pro

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Expected result | Actual result | Method |
| Register account | Account is made and is inserted into the database | User created. Login successful | Username: testaccount1  Password: password |
| Log in with correct credentials | Log in successful, gets directed to the index page. | User is redirected to the index.html page. | Log in using values created at the previous case and make note of which page is shown. |
| Log in using incorrect password/ incorrect username | Log in failed. Message: “incorrect username or password” | Invalid username or password message. | Username: wrongaccount  Password: password  Username: testaccount1  Password: wrongpassword |
| Non-Authenticated user views monitoring page | No link shown in navigation bar. Or 401 error if accessing URL directly. | Non-authenticated user cannot leave the login page. | Log out of account and try opening each link. |
| CPU monitoring | Temperature and usage are the same in the web-interface as with MSI afterburner | Temperature cannot be tested. CPU usage on Afterburner roughly matches to the one recorded in the application | Start the monitoring script and compare the CPU usage on both applications using the timestamp 14:29 is 11% on both |
| Disk monitoring | Compare with windows explorer or Linux command line | Disk usage gives the same result on both methods | MicroMonitor 93369315328.0 bytes  Windows explorer:  93369315328.0 bytes |
| Service monitoring | Stopping a service should reflect on the services on MicroMonitor. | Function not implemented. Unable to test. No result. | N.A. |
| Test Chrome | Website should look the same on Chrome as on the other browsers. | Confirmed | Open the website using Chrome |
| Test Edge | Website should look the same on Edge as on the other browsers. | Confirmed | Open the website using Edge |
| Test Firefox | Website should look the same on Firefox as on the other browsers. | Confirmed | Open the website using Firefox |
| Resizing | The interface scales correctly. And resized back to original when maximized again. | Navigation bar automatically collapses when reaching a size threshold. | Using ctrl+scrollwheel to zoom in and out. |
| Responsiveness | The application should be responsive with the maximum time until response being no more than 1500 ms. | 8 msLoading  109 msScripting  38 msRendering  29 msPainting  65 msSystem | Use the Chrome developer toolkit to record how fast the page loads for /cpu |
| Clarity towards the user | Username, password field should be mandatory. Message that reflects this should display when one or the other isn’t filled in. |  |  |
| Data consistency | The data put in the database should be retrieved the same. No truncation is supposed to happen. | Data entered is the same as the data retrieved |  |
| Data integrity | CPU and memory usage cannot exceed the system RAM present. | CPU percent and memory percent has a ceiling. |  |

## Evaluation

This section will compare the finished product to the planned product comparing it to the functional requirements. Comments are there when deemed necessary.

|  |  |  |
| --- | --- | --- |
| Requirement | Implementation | Comment |
| Users must be able to authenticate | Authentication is present |  |
| Users must be able to access monitoring page only when authenticated | Limited access is present | Users cannot access any functions without logging in |
| The application must have a navigation bar at the top of the screen for quick navigation | Navigation bar is implemented |  |
| Interface must be consistent throughout the application | Application makes use of inheritance for a consistent interface |  |
| Application must be accessible from a web browser | Application is accessible with the servers IP address with a browser |  |
| Dashboard overview presented when logged in | Dashboard is present | Dashboard is what is now called the logging page |
| Selection of devices available for monitoring | Selection of devices is present | Selection box is available on every monitoring page |
| Monitoring of processes | Process monitoring is not currently implemented | Process monitoring was too complex to implement when dealing with multiple operating systems |
| Monitoring of temperatures | Temperature monitoring is not currently implemented | Psutil does not support temperature monitoring on Windows and virtual machines have no access to the host’s CPU temperature.  No access to a non VM ware Linux system to test |
| Viewing of warnings | Automated warnings are not present | Users must watch the logs manually. |
| Users can add device on the network | Theoretical maximum of 50 devices can be added at once | Devices are added automatically once they start transmitting their metrics. |
|  |  |  |

# Reflection

The prototype made during the first deliverable was a platform to further build features upon. The flask architecture really allows for a modular approach that works especially well with an agile planning. Each sprint results in a new feature, sometimes small, and when it goes well, it results in an entirely new functionality. The architecture makes it obvious where each feature is defined due to the separation of concerns design principle.

The first hurdle that had to be overcome was a functional selection box for choosing a server to monitor. Solving this problem required an understanding of the architecture. It turned out to be a simple solution. This hurdle was eventually overcome and after that the entire development went much smoother due to better understanding how variables are passed from Flask to Jinja.

The way scripts are loaded in a classical web development project is inside of the HTML. Styling is loaded from the static folder. Flask has built in functions for this and so these are loaded somewhere else. Not fully understanding this at the start led to some difficulties getting JavaScript dependent functions to load. Due to the very specific nature of Flask + DataTables and mpld3 this was very hard to find answers to. Learning to use the web browser console made it eventually possible to determine how and importantly when dependencies are loaded [see 4.2.2.1].

Getting DataTables to show an interactive graph after troubleshooting and many hours of trial and error was rewarding. After this the product was essentially done. Graphs are displaying correctly, table works, and the layout looks good.

The graphs while displaying all the data correctly, were still lacking customizability. A function to somehow choose how much data was showing was needed. It could have been done with another dropdown somehow filtering the data by date, say the last 24 hours. Mpld3 allows for the graphs to be moved around and zoomed in on. Solving the problem that way. This is another example of a functionality that was not originally planned.

## Future work

There are some functions that were not implemented because they were either to complex making it impossible to finish in time or were not possible using the current design. There were also some parts that are functional but could be improved upon with the knowledge that I have now.

### Database

The database structure needed to be done differently. With hindsight this is obvious and easy to see. At the time however, using SQLalchemy to design a 3F database seemed unnecessary and daunting. A SQL database should never have redundant data in it. To prevent that the database should be split up into multiple tables that are joined together when writing queries. Also known as normalization.

Currently all the System data is inside a single table. With a new primary key being made every time a new log entry is made. The log entry contains the system name, and some other repeating groups that are entered into the database adding a lot of redundant data. This made implementing some functions, those that rely on the database queries, a lot harder than necessary.

The database needs a full rework with the data being split up in most likely 5 or more tables using one-to-many relationship. One system has one hostname, one CPU model and a set amount of RAM. But as System has many different CPU usage, Ram Usage and disk space readings. So, a new table would have to be made for each. The queries would in turn have to be changed and the backend that draws the graphs would work differently also.

### Missing functions

Some functional requirements are missing from the final design.

* CPU temperature monitoring
* Services monitoring
* Automated warnings

These would ideally be added in a later version of MicroMonitor. The most pressing one being the CPU temperature monitoring. This is easy to add for Linux based systems not running on a virtual machine. This could not be tested due to not having a Linux machine to test on. Windows uses a different method of exposing certain information about a system, Python does not have access to those methods without the use of closed source DLL’s. Eventually it could be possible to write a function that makes use of said DLL’s, but it does go against the open-source philosophy of Python.

The ability for the user to receives warning if user inputter thresholds are reached needs to be implemented in the future also as it would remove the need to constantly observe the application. Monitoring running services such as an Apache webserver would be ideal, but for now there too many variables to consider.

## Conclusion