**Analysis**

I will be developing a system for Bert Whitehead who is the Domain Administrator at King & Shaxson. King & Shaxson is a specialist investment firm located in the City of London, they focus on asset management; securities dealings and treasury consultancy. Bert wants a system where he can check details on his servers such as runtime and disk space without having to VNC into them and looking at system information. The system will be used to monitor servers via a web page, this is because it is the easiest to access from a wide range of devices. The current system is completely manual and takes multiple steps. The system I develop should make this process more efficient and wastes less time.

One of the jobs my client, Bert, does is checking the servers are kept up to date and operational. With the increase in security breaches in the commercial world it is important servers are kept up to date since many of the vulnerabilities are patched before or very quickly after an exploit is published/used.

•Objectives: what is the proposed system to do? – The system will need to quickly display information about the servers such as, Disk Space, Runtime, notifications (For example, when a server has been running for over 3 months) and the ability to check if updates are available. A secure login is also required so users on the network cannot access the web app without being authenticated to do so.

•What are the problems with the current way of doing things? – At the moment the current system involves using VNC to access the servers remotely and checking system information manually. There are more than 18 servers on the network, with upgrades and expansion scheduled for the future, and to manually check them all for updates and uptime takes a long time and sometimes doesn’t get done.

**Interview.**

•What data or information is recorded in the current system? – With the current system there is no data recorded apart from logs on the server itself.

•What data or information is to be recorded in the proposed system? How much data will the proposed system record? – With the new system necessary system information will be stored in the database and accessible via the webapp. One idea is to have information such as runtime stored on a graph in the webapp.

•How frequently will the data need to be updated? – Data will be updated every 5 minutes to keep the statistics up to date.

•Will new records need to be added or old ones deleted? How often? – New computers/servers will be added to the system over time. When a new server is commissioned it will be added to the database.

•How important is the data or information that is recorded? – The data that is recorded is important and can be useful to figure out problems with servers.

•What processes or functions are to be performed by the new system and what algorithms will be used? In the new system there will be a sorting algorithm to organise the servers by runtime.

•What inputs are required for the proposed system? The main input is a form on the website where the new server is added. The IP for the server is passed to WMI where it gathers the necessary information.

•What outputs will be required from the proposed system? Output includes a text file, graphs and charts in the webapp and some logs on the python console.

•How often will outputs be required? Textfile is only output when the app is first run. The outputs on the webapp are updated every 5 minutes.

•What computing resources does the client possess? The client has many resources and the app will be run on its own VM on one of the servers.

•Is the client prepared to purchase hardware/software resources? The client already has enough resources.

•Is security an issue? Yes the data needs to be protected and passwords need to be encrypted.

•Should there be restricted access to particular areas? Restricted access to anything that could affect the servers. Some users have more servers accessible to others. Users will need to login to do access most pages in the app.

•How are exceptions and errors handled in the current system? N/A

•What errors and exceptions should be reported in the proposed system? Log file and email notifications. Some exceptions will also be output on the console.

•How should they be reported? Email or console.

•Are there any constraints on hardware, software, data, methods of working, cost, time, etc.? There is no real limits apart from time.

•Does the user have a particular solution in mind? The client has recommended using WMI

The current system at King & Shaxson is completely manual. It requires using VNC to access the server and checking for updates or other system information. This can lead to late updates on critical servers since it can be easily forgotten. By having to manually VNC into each system it uses a lot of time especially since there is 18+ servers on the network. The proposed system should use the WMI group on the domain and feature a clear web interface which includes a login page with different user types. This would increase the efficiency of managing the servers which allows the IT staff to spend more time on other projects.

**Investigation**

**Current System**

**Data sources and destinations**

|  |  |  |  |
| --- | --- | --- | --- |
| **Source** | **Data** | **Example Data** | **Destination** |
| Domain Admin (IT Staff) | IP address & Password for VNC | 192.168.1.6 – Pa55word | VNC programs input forms. |
| Windows update | Update name and Date of update | Microsoft security patch – 26/02/2018 | Windows OS log files. |
| IT Staff | Updates have been completed | The <server> has been updated the update was “Microsoft security patch” and the server is ready to be used again. | Other IT staff or developers |

**Current system algorithm.**

The current system is manual since the IT staff must remote into each server if they need to find information on it.

**Pseudocode**

In the code below VNC represents how the user would interact with the Windows GUI.

If updates\_available:

VNC(IP = 192.168.1.6, Password = Pa55word)

VNC.connect

Startupdate = input(“Do you want to update Y/N?)

if Startupdate = Y:

VNC.update

Else:

VNC.exit

Def check\_runtime:

VNC(IP = 192.168.1.6, Password = Pa55word)

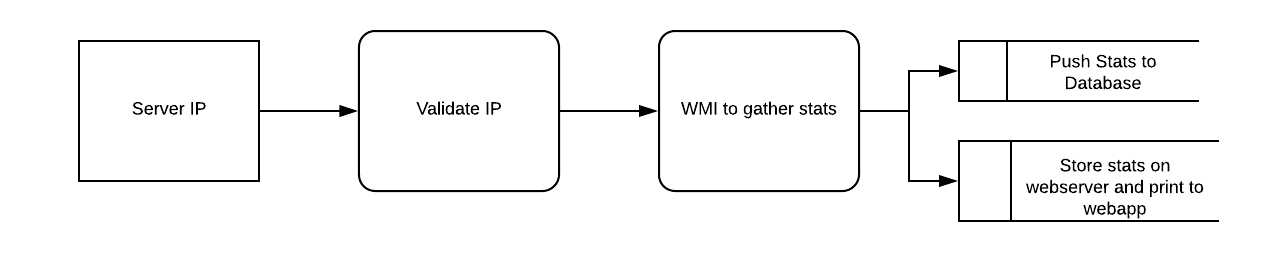
VNC.connect

VNC.opentaskmanager

Print(runtime)

VNC.exit

**DFD**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | **Data Type** | **Length** | **Validation** | **Example Data** | **Comment** |
| ip address | String | 15 Chars | 11 Chars must exist | 192.168.6.1 | Unique to each server |
| UserAccount | String | 15 Chars | 5 Chars must exist | MahoneyS OR Admin | Each user has a unique name they use to login with |
| AccountPassword | String | 25 Chars | Atleast 6 Chars | Pa55word | Each user account has a password. This is used in conjunction with the IP and username to use VNC |

**Data Dictionary**

**Volumetrics**

The current system doesn’t process or store any data. It currently only processes one request at a time since each server needs to be updated manually.

**Proposed System**

**Data sources and destinations**

|  |  |  |  |
| --- | --- | --- | --- |
| **Source** | **Data** | **Example Data** | **Destination** |
| Admin User on WebApp | Commission date and name of server | KASLAD64 – 01/01/2016 | Database.  Webapp |
| Database system | Updates – (Boolean) | Updates = Y/N | Database.  Webapp updates available. |
| Webapp login page | Username and Password | Username = Example  Password = Example | Database/hashtable? |

**Proposed system algorithm**

The proposed system uses many different algorithms to preform different tasks and solve problems. Some algorithms that will be used are a user login, password encryption and sorting algorithm.

**Pseudocode**

BubbleSort algorithm.

For theElementsinList

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

UserLogin

Username = input()

Password = input()

Loginuser = DB.Query.filter\_by(Username)

If user:

If checkpassword(password == db.Query.filter\_by(password)

LoginUser()

Encryption

**DFD**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | **Data Type** | **Length** | **Validation** | **Example Data** | **Comment** |
| ip address | String | 15 Chars | 11 Chars must exist | 192.168.6.1 | Unique to each server and used when adding a server to the database |
| UserAccount | String | 15 Chars | 5 Chars must exist | MahoneyS OR Admin | Each user has a unique name they use to login with |
| AccountPassword | String | 25 Chars | Atleast 6 Chars | Pa55word | Each user account has a password. This is used in conjunction with the IP and username to use access WMI and fetch the server stats. |

**Data Dictionary**

**Volumetrics**

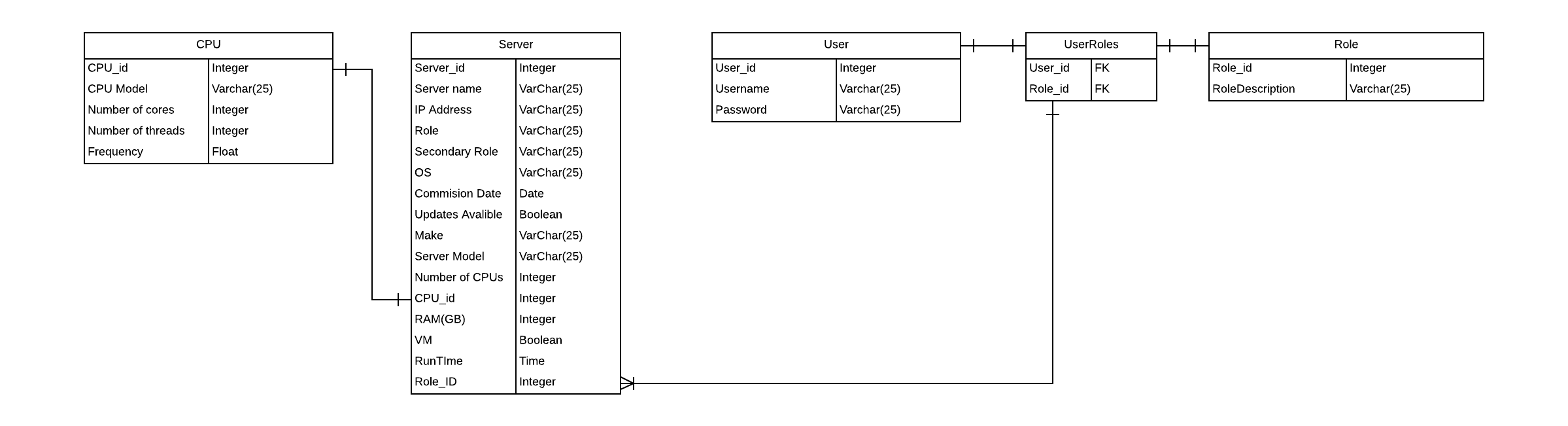
The proposed system could be able to store up to 30 servers initially, but this can easily be expanded up to 200 servers. The system will only process one WMI request at a time, so each record will be updated before moving on to the next one.

**Objectives**

1. Create a login system for users and redirecting them to the home page.
2. Create a page where users can add servers to the database
3. Create a page where users can see the latest server information
4. Automate gathering information from servers
5. Allow users to add and remove emails from a file
6. Encrypt domain login details and decrypt them and use them to access WMI
7. Send email notifications if a server has a problem
8. Create a page where users can input an IP and get the latest stats on that machine
9. Automatically generate and admin account and password when the app is first run

**Entity Relationship Diagram**

**Move this to design.**

 This shows how I am going to implement different account types in the system. The user table will contain the basic information much as User\_id, which is the primary key, and the username and password. The UserRoles contains the User\_id (foreign key) and the Role\_id (Also a foreign key from the UserRoles table). The Role table contain the Role\_id which is the primary key and the RoleDescription which states what role it is for example, administrator. In the Servers table the CPU\_id is a foreign key that links to the CPU table which contains information on the CPU.

It shows all the attributes for a server. The Server\_id is the Primary Key and the Role\_ID will need to match with the role for the user trying to access it in the user roles table. This will give certain users access to more servers than others depending on how critical the servers are to the business. The diagram can also show the attributes for the object “Server” in the program. Other objects might include “User” and “Admin”.

This is the list of current servers on the network and the attributes for each server listed.

**Constraints**

Hardware – The system will be run on a local server, so it has access to the network. However, this server may be a virtual machine.

Software – There are no preferences apart from a clear interface.

Time – The project deadline will be set by the teacher and exam board.

Knowledge – Since I am creating the project for a domain administer there is no constraints of knowledge in terms of IT.

Access right – It would be preferred if the WMI user group was used to access the servers on the domain

**Alternate Solutions**

These are all the possible ways of creating the proposed system. It consists frameworks, languages and modules that may be used in the final program.

**Flask vs Django**

I had the choice to either use the Flask or Django web framework. I have chosen to use a web framework over a windows GUI or command line application since it can be accessed by anyone on the network without needing to install any additional software. Flask is a micro web framework which only implements a basic webserver. This leaves the developer to add additional modules and build the application from the ground up. This makes flask more flexible and simple but the trade of is that it sacrifices the features already present in Django. The first popular release of Django was in 2005 in comparison Flask was first released in 2010. Although Django more common in the industry it does has a steep learning curve which is why I chose to use Flask since it adds more control and is easier to learn. One of my favourite things about Flask is the ability to create simple web apps quickly and easily which is good for testing ideas and prototyping. Since both are based on python and Jinga2 it is also possible that once I have learnt to use flask I can also apply the skills to Django.

**Python Modules**

One module that I might use is PyWin32 which is a set of extension modules that provide access to many of the Windows API functions this is important because I will need to access statics from the servers. Another module I might use is the OS module which provides a way of using operating system functions which can be useful when accessing the servers or even finding files on the host machine. I may also use the platform, sys and ctypes alongside these. Another option could be using the WMI (Windows Management Instrumentation) module which would allow python to talk to the WMI API with psutil which provides information on CPU, Memory and Disks. Currently WMI is the one I will most likely use since it provides better security as domain users can be assigned WMI privileges.

Using HTML and CSS/Bootstrap

Since the app will be run on a webserver it is necessary to know HTML to be able to create the webpages for the user to interact with. Since my client mentioned they would like a clear and easy to use design so I decide to use Bootstrap. Bootstrap will allow the webapp to be ran across multiple platforms including phones and desktops alike. It will provide a good-looking design and add functionality with tables and a navbar. This should make the application easy to use allow it to carry out its task efficiently.

WMI

WMI or Windows Management Interface provides a tool which allows management information to be shared with management applications. It allows admins to access and monitor systems either locally and remotely.

Design

Introduction

Server Management System.

* Web User interface with different levels of access
* Get server details such as CPU usage
* Manage Updates and servers
* Report servers which are not running.

Web User interface with different levels of access

* The system will provide information on the servers in the network such as HDD space and CPU usage. It will also list updates available which is important in a corporate setting due to vulnerabilities needing to be patched quickly.
* Different user accounts will have more access than others. For example, the Admin account will have access to all servers, but a normal user account may only be able to check servers which aren’t critical to the business. Only IT staff will have accounts.
* Additional servers may need to be added through the web app but the initial setup is planned to be with a CSV file which has all the server attributes already listed.

Get server details such as CPU usage.

* The system will provide the user with information which will involve HDD capacity, specs and CPU usage (Maybe in a graph).
* This will work by using python modules which get information from the servers. This data will then be shown to the user in real/near to real time which can be used to quickly analyse servers.

Manage updates and servers

* One of the primary functions of this system will be to alert the IT staff that updates are available. This can be done via email notification and on the web app. If it is possible a button to upgrade the server from the web app would save time. However, a button to open a VNC client to connect to the server would also be a good idea.

Report servers which are not running

* In a business environment it is important that faults are found quickly so they can be dealt with. When a server is seen to be having problems or has gone down an email notification will be sent. This will alert staff of the problem and allows them to start fixing the problem as soon as possible.

**Pages**

Homepage

The home page will act as the first page users will see when they enter the website. Users will be prompted to Login to gain further access.

Login/Register

Unlike most systems the Administrator will be the only user that can create accounts. Users do not have access to the register page for security reasons. Only created and authorised users can enter the website. The login page consists of a Username Login form and password form while also containing a remember me button for ease of use. When a new user is registered the Admin will have to enter an email, username and password.

Admin Page

The Admin page will control most of the options on the site. It has links to change the WMI domain user, register a new user and add new servers to the system. This page can only be viewed by the authorised user since it contains information and access that needs to be secure.

Add Servers Page

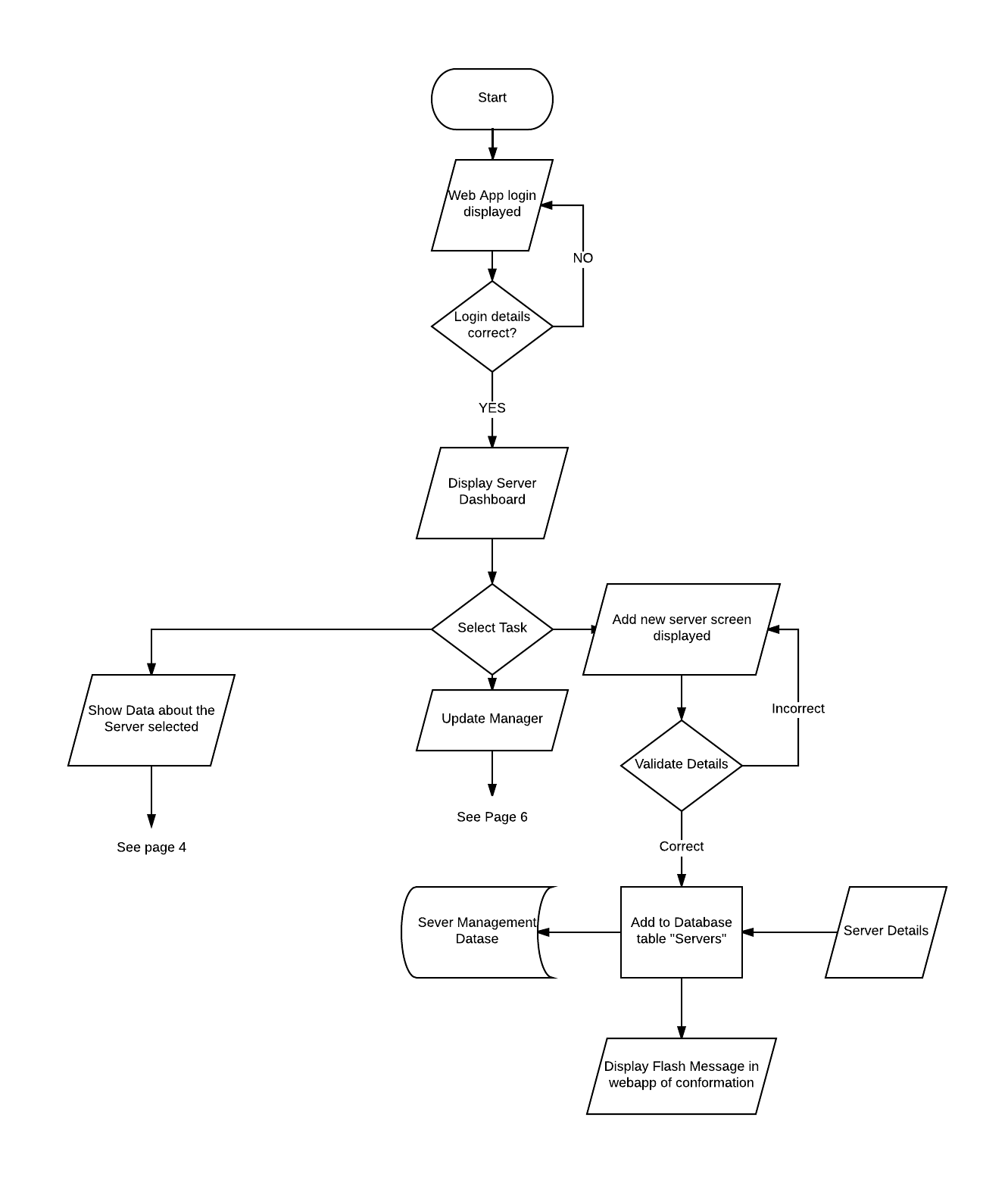
The add servers page is the first step in setting up the functionality of the app. The Admin is required to enter a nickname for the server, so it can be easily identified afterwards. An IP Address is also required which is important since it will be used to connect to WMI. A few other options are required to make it distinguishable from other servers, these options contain the Primary Role (For example, Primary Domain Controller), a secondary role which can be blank if not required. A commission Date is also required to keep track of servers and lets IT staff know how old each server is which is useful for routine maintenance. The last option is the make which makes it easy to search for drivers online for the server by knowing the make.

Stats Page

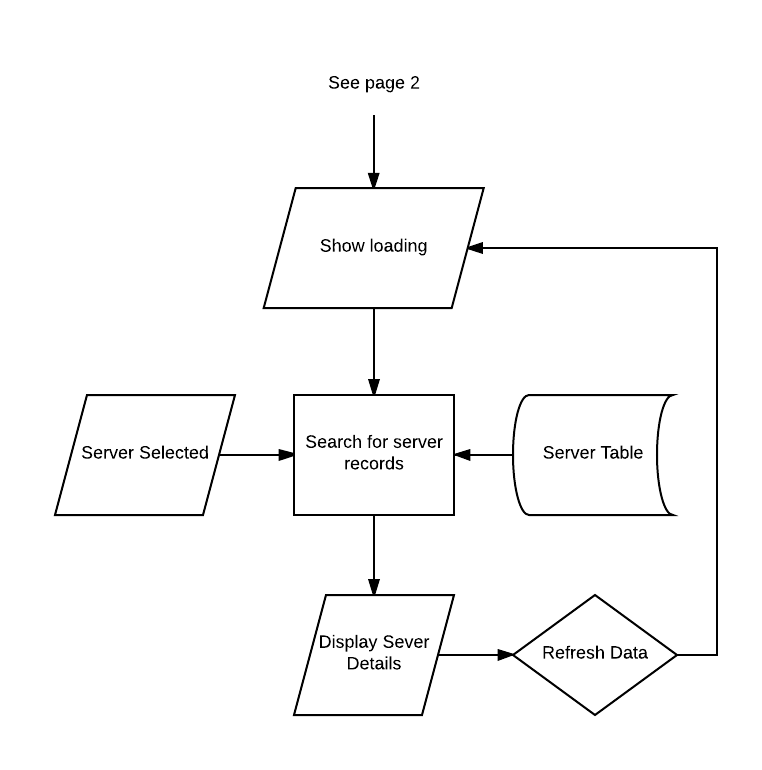
The stats page isn’t overly important but provides a way for getting live stats on a given system. It uses a form to capture the IP Address and uses the WMI domain user to communicate with the server and find live stats such as ram usage, CPU load and other useful information.

Servers Page

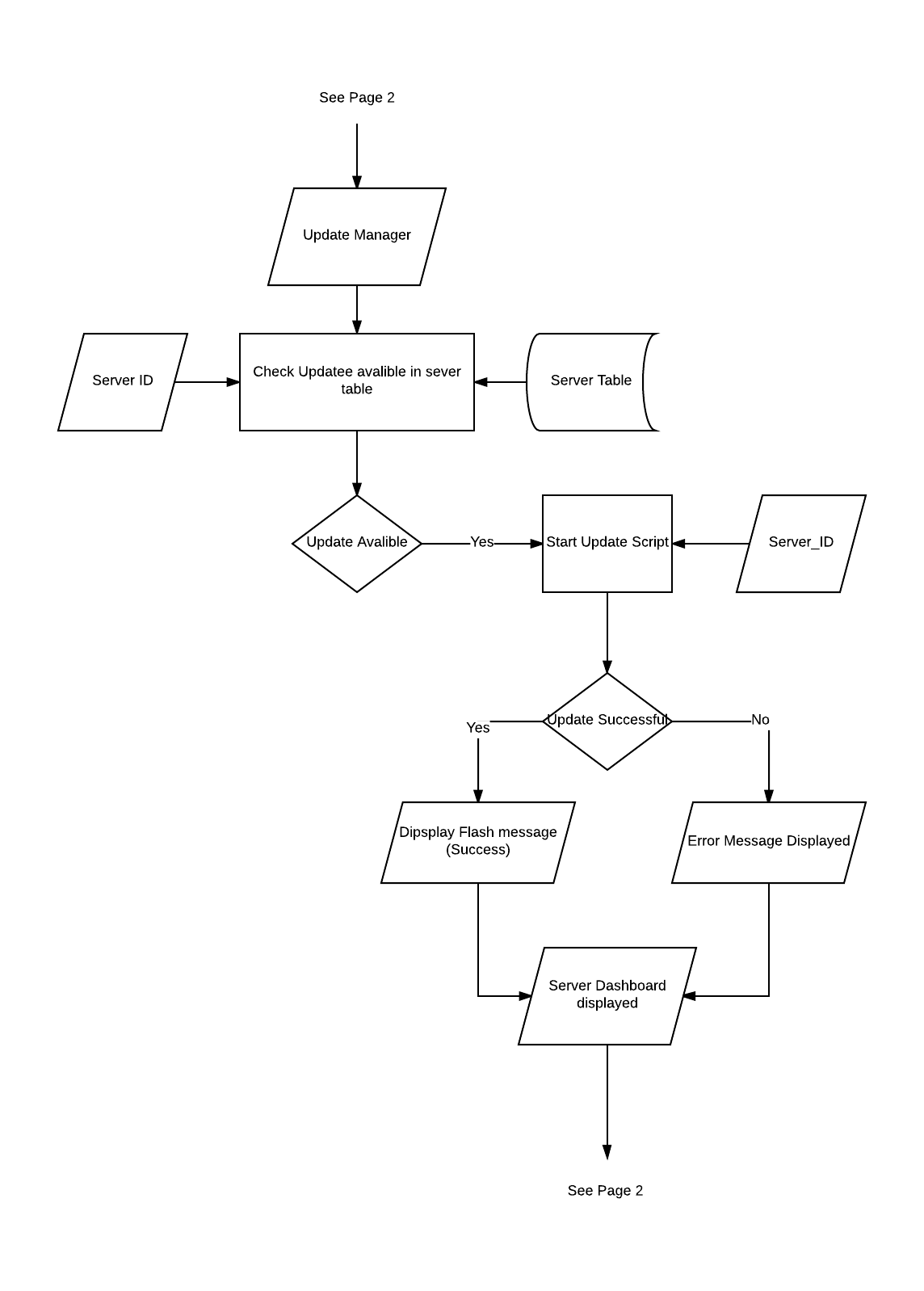
The servers page is the main dashboard for the system. It works with WMI and the database to provide the latest stats collected. It displays each server on the dashboard in a clean and easy to understand way.

Flowcharts

Show data about the server….

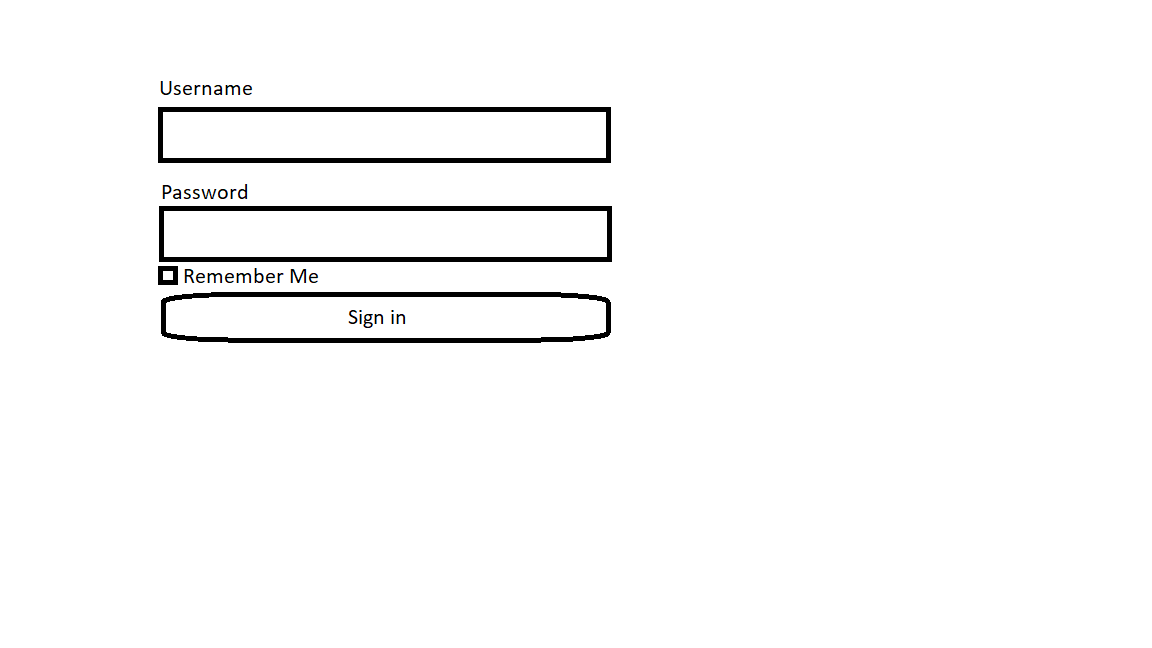


Update Manager

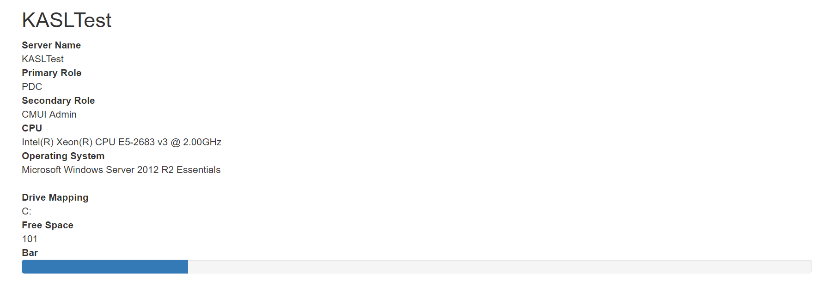


User Interface Design (Human-Computer Interface HCI)

Login Form.

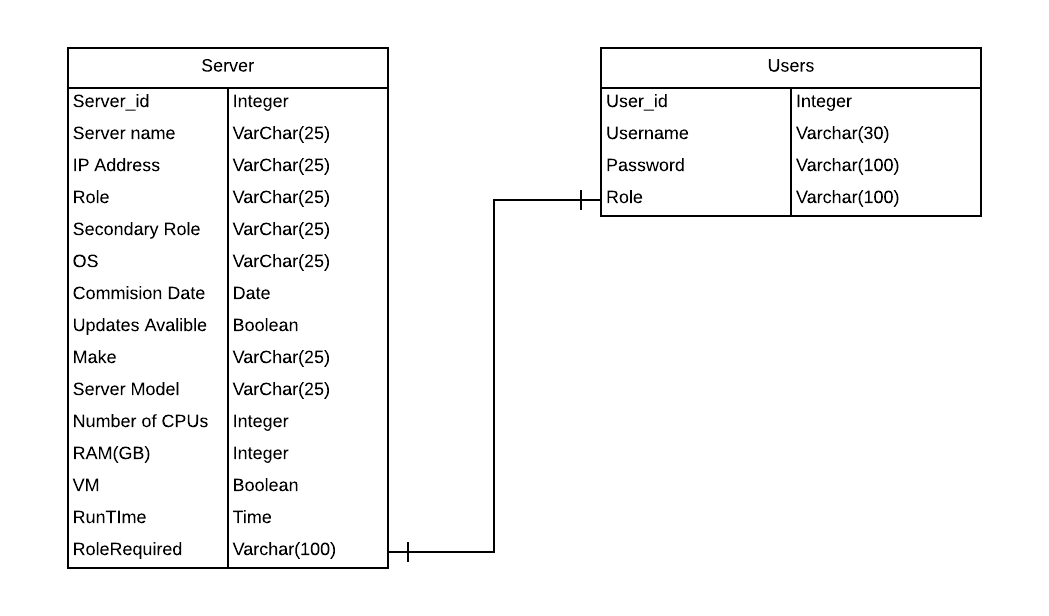
The login form uses Bootstrap classes to create a clean user interface. WT-Forms was used to create the form fields which allows validation to be implemented to the form. For the Login a register forms I used SQL-Alchemy query instead of SQL this was to make the login the most secure element of the system and easily allows the passwords to checked against the stored password hash. However, the rest of the system does use normal SQL queries.

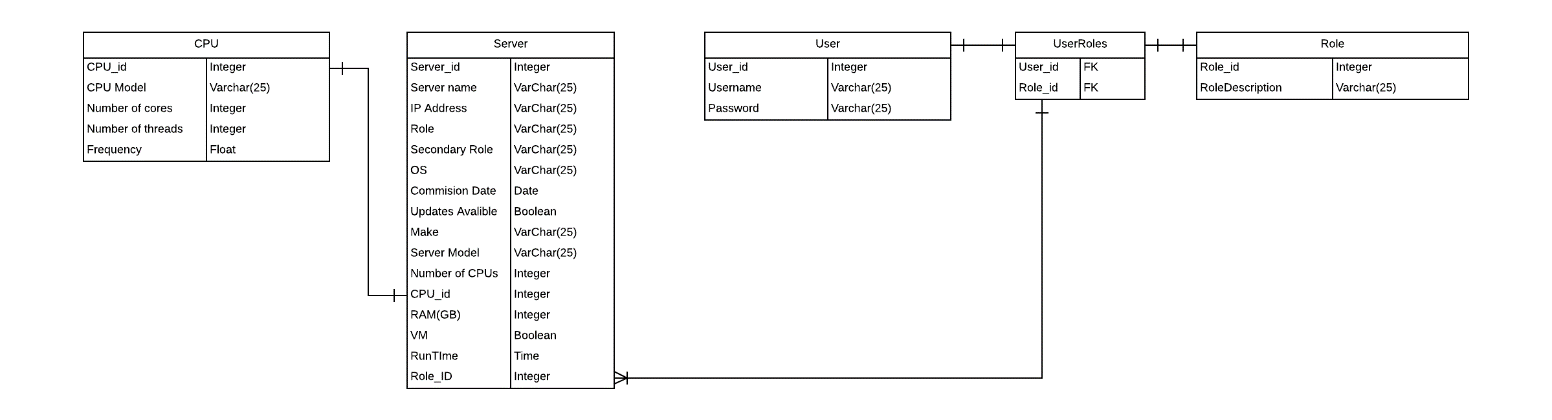
Servers Page

This is a mock-up in bootstrap what the server’s dashboard might look like. It shows information from across three tables in an easy to read format. This makes it easy to check the status of a given system.

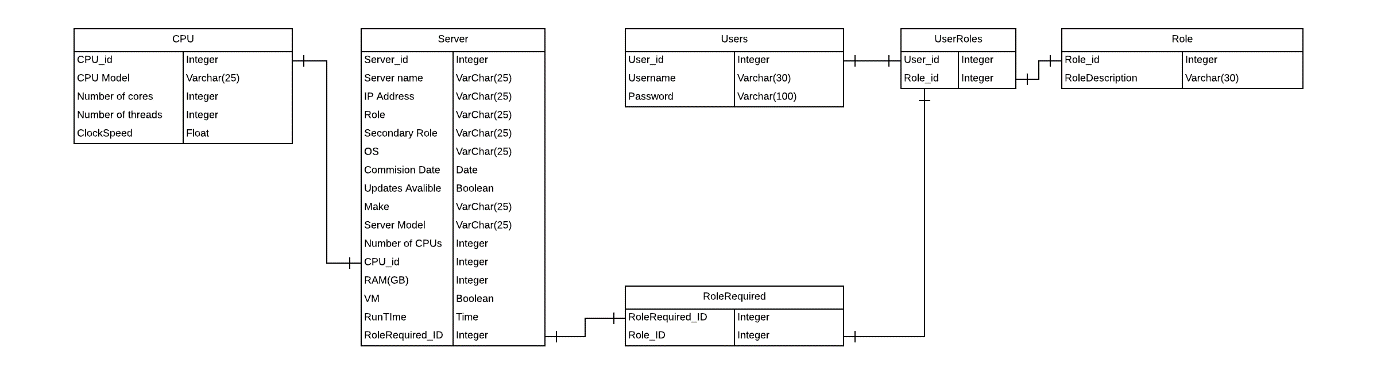
Normalisation

My database design started with just a servers and users table. This is UNF since there would be many repeating factors in the database.



I then moved to 2NF where I stopped repeating factors in the Users table and with CPU specs such as cores and clock speed.

The final step was to move to 3NF where no factors are repeated in database. I added the RoleRequired table to do this.



Hardware Specification

The client already has a large number of servers and the plan will be to run the application on a virtual server. This will allow the application to be used across the network which is important if multiple users need to use the information. Also by having it run on a virtual server it will be up 24/7 which is important if updates need to be run overnight or users are investigating problems.

Security and Integrity of System and Data

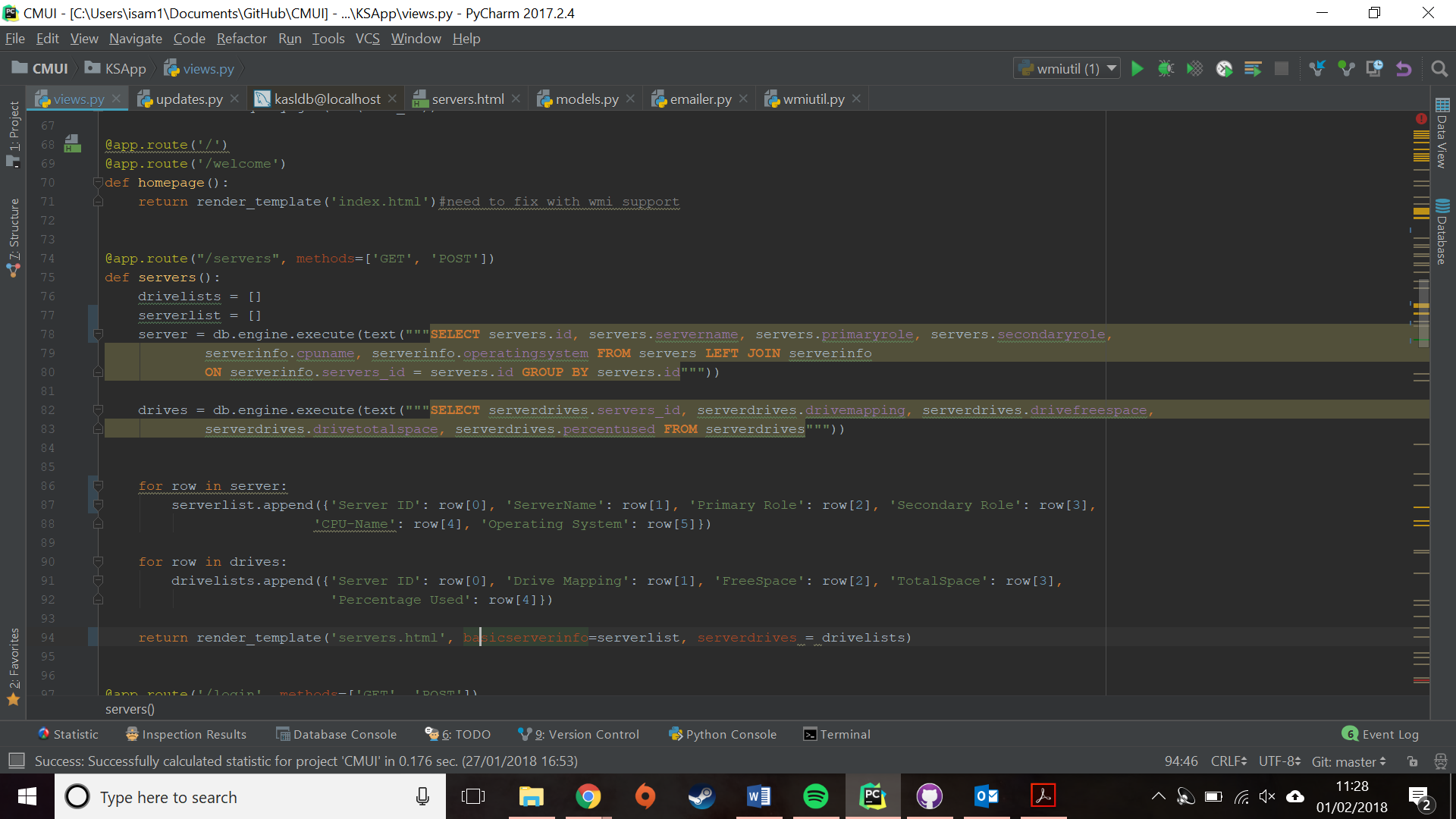
I plan on using a module in python called pycrypto and encrypt sensitive data such as passwords with SHA256. I did develop a hashing program and hash table using my own algorithm but for security reasons I will use SHA256 since the system will be used in a corporate environment and information needs to be secure.

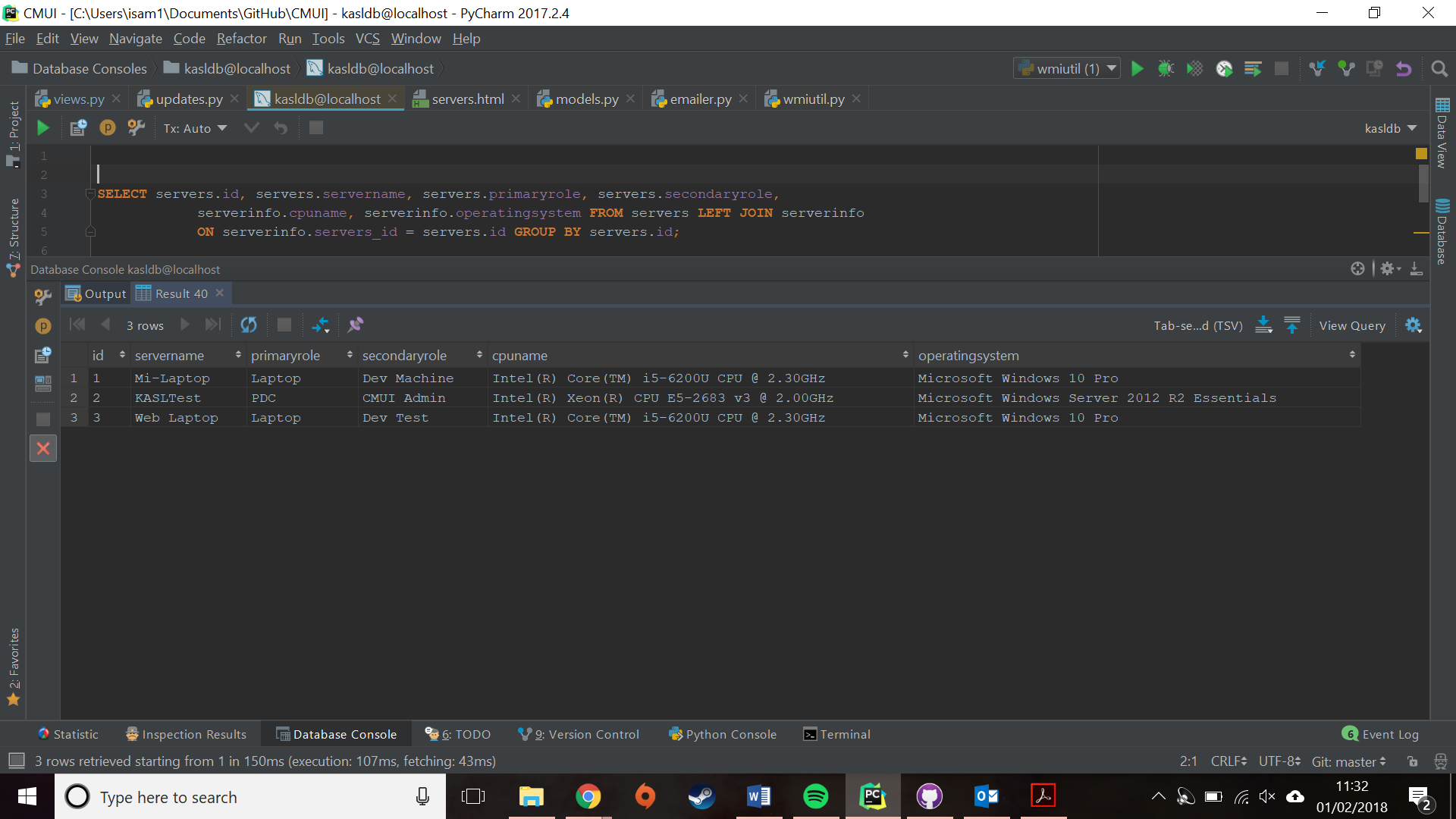
I have also created my own encryption to encrypt and decrypt domain details easily since it wouldn’t be good to store them in plain text, but they need to be used regularly.

class vc:  
  
 def \_\_init\_\_(self, text, key):  
 self.text = text  
 self.key = key  
  
 def VernamCipher(self, text, key):  
 result = ""  
 ptr = 0  
 for char in text:  
 result = result + chr(ord(char) ^ ord(key[ptr]))  
 ptr = ptr + 1  
 if ptr == len(key):  
 ptr = 0  
 return result

in depth look into the pages.

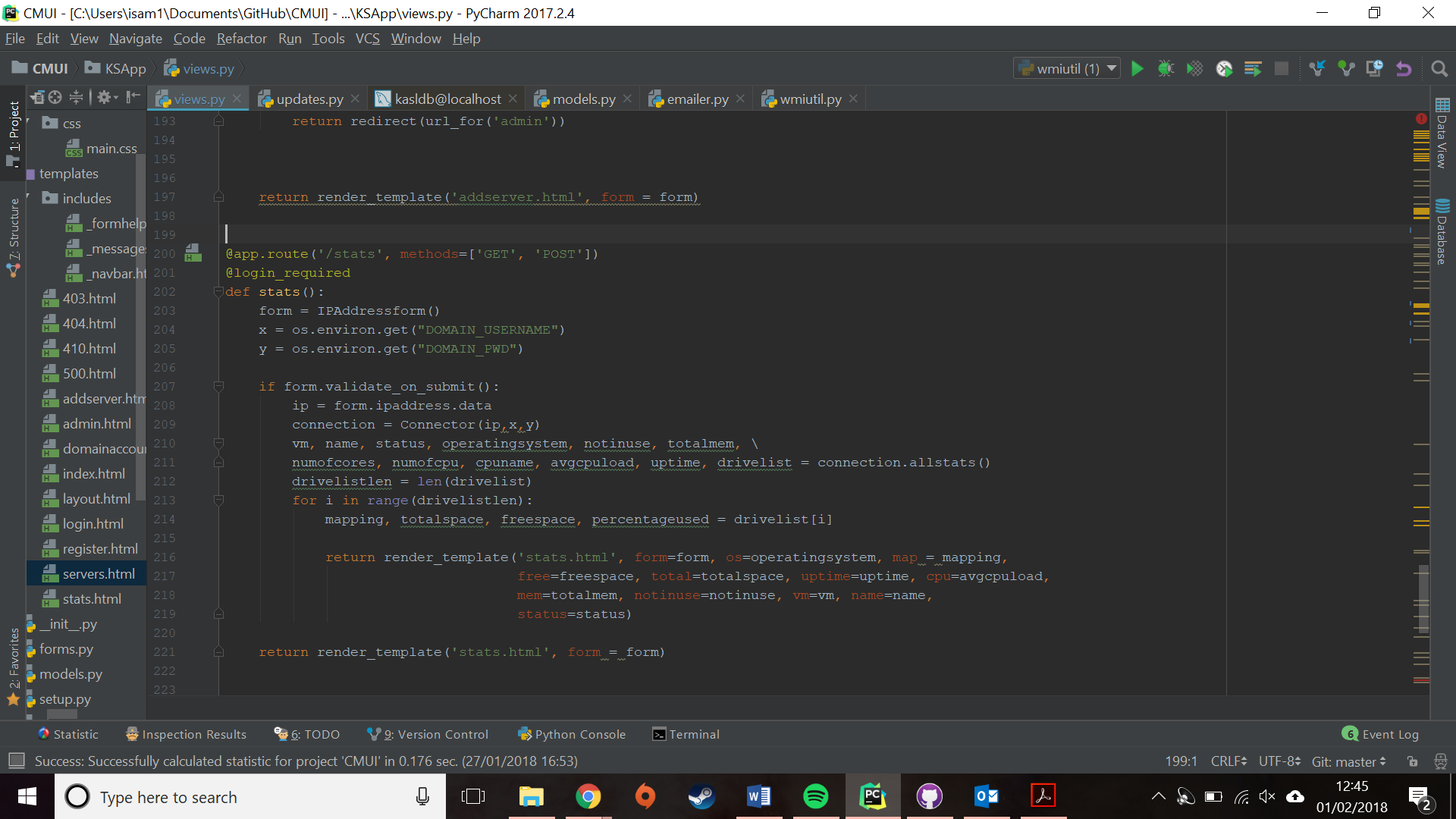
Servers Page

The servers page is made up of many components that work together including HTML, CSS/Bootstrap, Python/Flask and SQL.



When the page is loaded two lists are created (Drivelist that contains the drive information and Serverlist which contains info on each server). First information about each server is fetched from the database using SQL.

The statement (SELECT servers.id,servers.servername, servers.primaryrole, servers.secondaryrole, serverinfo.cpuname, serverinfo.operatingsystem FROM servers LEFT JOIN serverinfo ON serverinfo.servers\_id = servers.id GROUP BY servers.id) will select entities from both the servers and serverinfo tables via a JOIN on the foreign key servers\_id which matches with the ID in servers.. As seen above it returns all three current systems in the database. To make this useable in the HTML template I used a for loop. For each row in Server means for each result in the SQL query. After this we take that single row and append it to the list we created at the start. However, to make it easier to interact with we use a dictionary as a KEY:VALUE pair. The system then repeats this process for the drives and finally pass the lists of dictionaries to the template. It is then passed to the HTML and each server has its information displayed in a table.

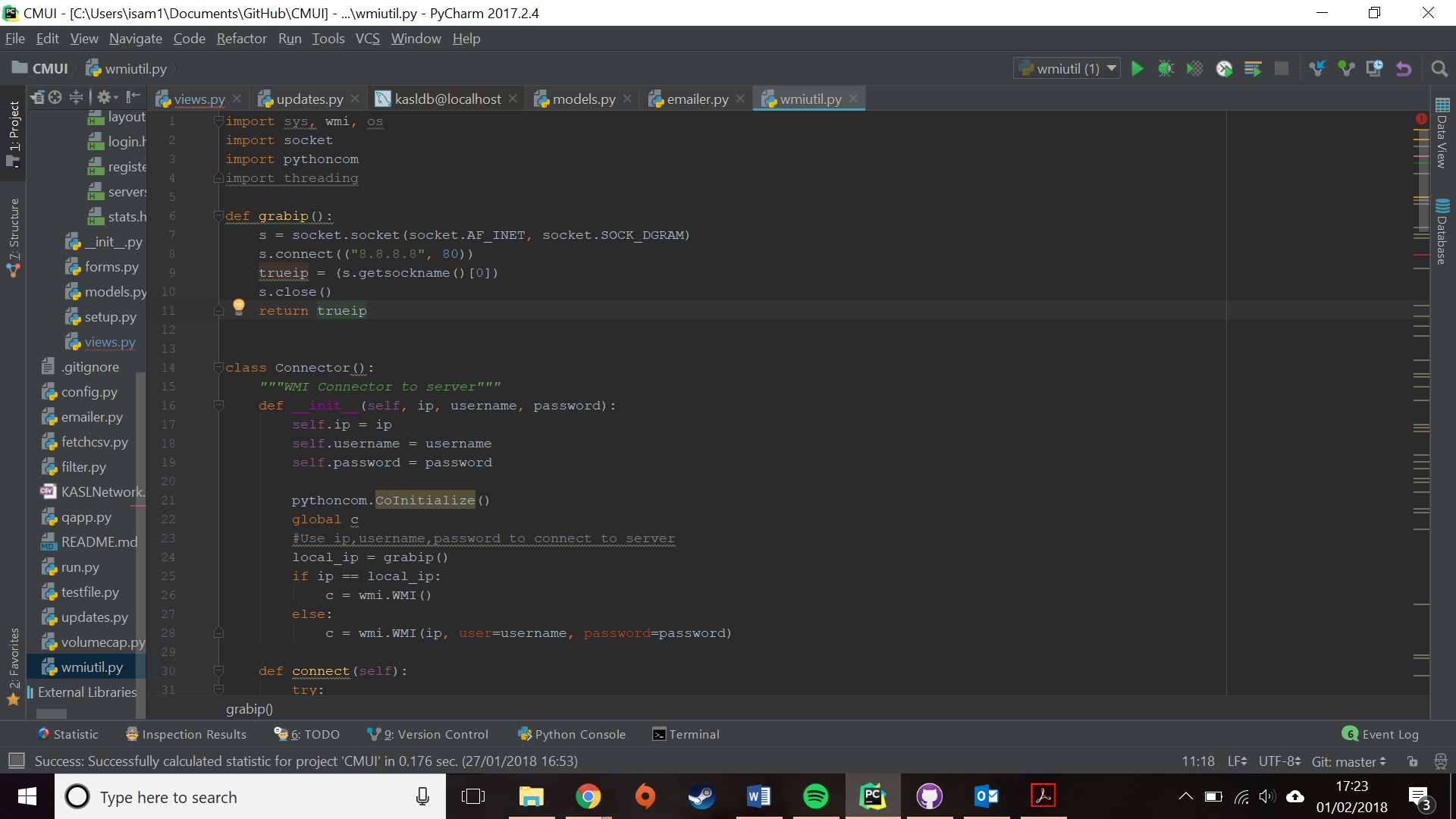
Stats Page

The stats page accepts an IP Address which is validated via the form. Using this IP and the domain credentials it connects to the server via WMI using the wmiutil script I created. Using the allstats function we fetch all the needed variables and pass them to the HTML template. The only variable that isn’t directly passed if the drivelist. To get the drives we get the length of the list and run a for loop in the range of the list. \*The drivelist code isn’t done yet

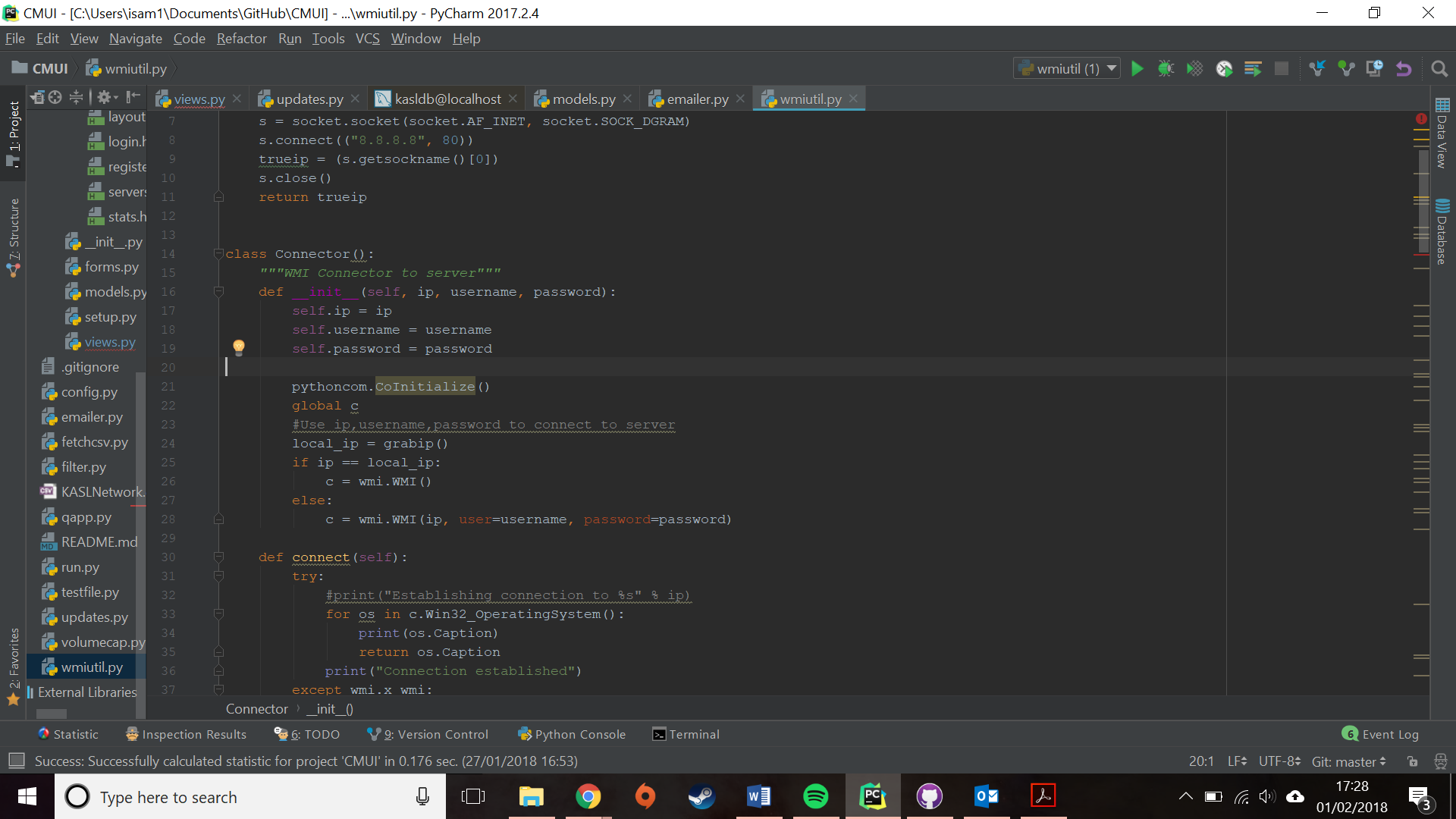
Add servers

WMIUTIL

One of the most important parts of the program is wmiutil. The connector class is imported into the main views file to interact with WMI. Firstly, we need an IP, username and password to access WMI. The problem with this is if the webserver is run on one of the servers what needs to be monitored then it will not work due to the way local WMI sessions work. A local session doesn’t need a IP, Username and password. To combat this, I created the grab IP function. This uses the socket module and connects to googles main DNS servers (8.8.8.8) it then returns the socket name which in this case will be the local IP for the machine and returns it in the variable “trueip”.



The next step is to establish the connection to WMI. When we create the connector class we create the global variable “c”. We find the IP of the local machine with grabip() and then compare it to the IP address that has been passed to the connector class. If they match c = wmi.WMI() if not c = wmi.WMI(ip, user, pass). wmi.WMI() creates a local session to interact with.



The WMI class contains many methods which access different elements of the server and returns variables. The one I will focus on is the allstats() method. This method is a combination of all the methods and is used in the servers view.

**Technical Solution**

https://github.com/mahoney777/CMUI

|  |  |  |
| --- | --- | --- |
| Page Name | Technical Skills | Page Number |
| Homepage | The homepage is a basic HTML file which extends both the navbar and the layout html files from the includes folder. It uses flask to route the page and render the template.  When the webapp is first created it runs the function createAdminAccount which creates a randomly generated password and writes it to a text file and stores it in the database, so the user can login as the Admin. | Page 24 & 25 |
| Login | The login page uses forms to collect user data and queries the database using sql-alchemy and compares the password collected in the form to the hashed password in the database. | Page 26 |
| Addserver | The addserver page allows the user to add a new server to the database so it can be displayed in the app. It gets the domain username and password from the environmental variables set in the adddomainaccount page. These details and the ip address from the form are passed to the connector I created for WMI. It then uses WMI to get stats from the server and adds it to the database. It uses an SQL query where it fetches the IP from the table. | Page 27 |
| Stats | The stats page works similar to adding a server. It uses the same domain details and the WMI connector to get the most up to date stats. Instead of writing it to the database it is temporary stored and passed to the HTML. | Page 29 |
| Servers | The servers page uses the information in the database which was gathered from WMI. These details are selected from multiple tables through SQL queries. The data is then displayed in HTML and uses part of Jinja2 to create a for loop which cycles through the SQL data and puts it in a table. | Page 31 |
| Emailer | The emailer program adds the ability to send email notifications. It uses the gmail SMTP server and a gmail account to send emails to all users in the contacts file.  The email webapp page allows users to add names and emails to the text file and remove them if needed. The contents of the file are read and printed at the top of the email page. | Page 32 |
| Reload Server Stats | Reload server stats refreshes all the information on the servers. It uses WMI and selects all IP Addresses in the database and collects the latest information. | Page 33 |
| WMI | WMI uses the wmiutil script I created to get information on servers. Before it tries to access the server, it checks the IP to make sure its not the local one. To do this I used sockets and connected to 8.8.8.8 which is googles DNS server. After this it tests the connection and if this doesn’t work it prints an error. To get the stats it uses different WMI classes to get certain details and assigns them to variables before returning them to the main program. | Page 34 |
| Models | Models are the source of information about your data and contains attributes about the database before it is created in a database. I defined my models and when the app is run it creates the database and relationships from these. | Page 35 |
| Init and run | Init is where the flask app is created and all the config information is set. Run is then called to start the app. | Page 36 |
| Fetch CSV | Fetch CSV was used in an early prototype of the app where the servers were imported from a csv file. All this does is checks if the file exists and if not, it asks for another file. After it reads the file and returns it. | Page 37 |
| Error Handling | I have added error handlers for all the common HTTP error codes such as 404,403,410 and 500. If this happens it just redirects the user to the correct error page. | Page 38 |
| Vernam Cipher | The Vernam Cipher uses an encryption key to encrypt and decrypt messages. It is used to keep the domain details safe. | Page 39 |

**Before first request**

Before the first request the app needs to set some things up. An Admin account and the details output into a txt file so the admin can login and create accounts.

Views.py

@app.before\_first\_request  
def createAdminAccount():  
 try:  
 print("Checking for Admin account...")  
 pw = ''.join(random.sample((string.ascii\_uppercase + string.digits), 9))  
 hashed\_password = generate\_password\_hash(pw, method='sha256')  
 new\_user = Users(username='Admin', email='Admin@cmui.co.uk', password=hashed\_password, urole="Admin")  
 db.session.add(new\_user)  
 db.session.commit()  
 print("------------------")  
 print("New Admin account generated")  
 print("Username = Admin, Password = "+pw)  
 print("Check CMUI-Login.txt for details")  
 print("------------------")  
 f = open("CMUI-Login.txt", "w+")  
 f.write("Username = Admin Password = %s" % pw)  
 f.close()  
 except:  
 print("Admin account exists")

**Layout**

Every page uses a base layout template which is extended in the html files.

Layout.html

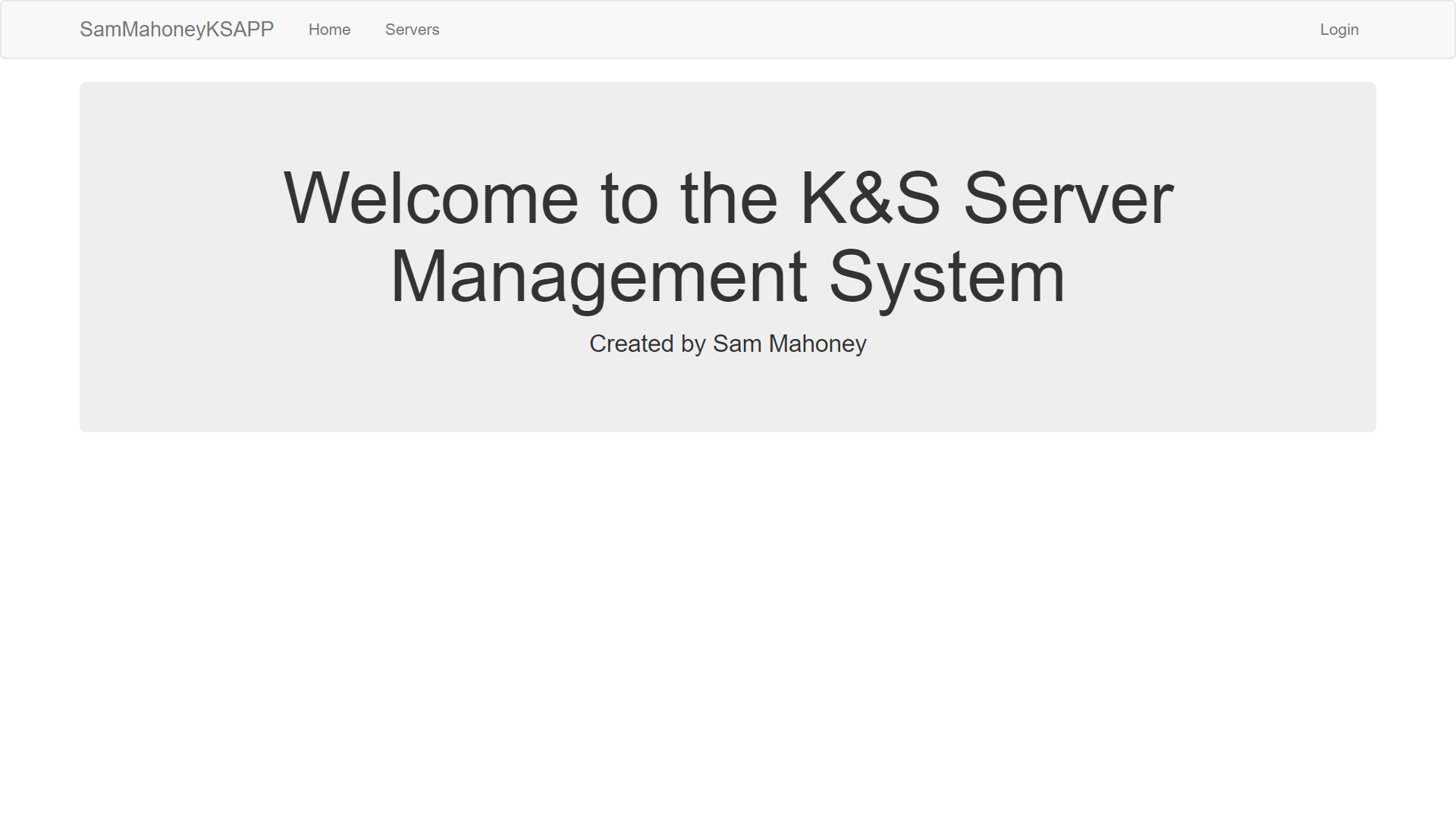
<!DOCTYPE html>  
<html lang="en">  
 <head>  
 <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">  
 <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">  
 <link rel="stylesheet" href="static/css/main.css">  
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>  
 <meta charset="UTF-8">  
 <title>KSAPP</title>  
 </head>  
 <body>  
 {% include 'includes/\_navbar.html' %}  
 <div class="container">  
 {% include 'includes/\_messages.html' %}  
 {% block body %}{% endblock %}  
 </div>  
  
 <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>  
 </body>  
</html>

**NavBar**

<nav class="navbar navbar-default">  
 <div class="container">  
 <div class="navbar-header">  
 <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navbar" aria-expanded="false" aria-controls="navbar">  
 <span class="sr-only">Toggle navigation</span>  
 <span class="icon-bar"></span>  
 <span class="icon-bar"></span>  
 <span class="icon-bar"></span>  
 </button>  
 <a class="navbar-brand" href="#">SamMahoneyKSAPP</a>  
 </div>  
 <div id="navbar" class="collapse navbar-collapse">  
 <ul class="nav navbar-nav">  
 <li><a href="{{ url\_for('homepage') }}">Home</a></li>  
 <li><a href="{{ url\_for('servers') }}">Servers</a></li>  
 </ul>  
  
 <ul class="nav navbar-nav navbar-right">  
 {% if current\_user.is\_authenticated %}  
 <li><a href="{{ url\_for('admin') }}"> Admin </a></li>  
 <li><a href="{{ url\_for('logout') }}">Logout</a></li>  
 {% else %}  
 <li><a href="{{ url\_for('login') }}">Login</a></li>  
 {% endif %}  
  
 </ul>  
 </div><!--/.nav-collapse -->  
 </div>  
 </nav>

The navbar is one of the main ways the user navigates the webapp. When the user isn’t logged in the navbar only displays options to access the homepage and servers page as well as a button to the login page.

**Homepage**



{% extends 'layout.html' %}  
  
{% block body %}  
 <div class="jumbotron text-center">  
 <h1>Welcome to the K&S Server Management System</h1>  
 <p class='lead'>Created by Sam Mahoney</p>  
 </div>  
{% endblock %}

The home page is very simple and just consists of a bootstrap jumbotron which tells the user the name of the system.

**Login**

{% extends 'layout.html' %}  
{% import "bootstrap/wtf.html" **as** wtf %}  
  
{% block body %}  
<div class="container">  
  
 <form class="form-signin" method="POST" action="/login">  
 <h2 class="form-signin-heading">Please sign in</h2>  
 {{ form.hidden\_tag() }}  
 {{ wtf.form\_field(form.username) }}  
 {{ wtf.form\_field(form.password) }}  
 {{ wtf.form\_field(form.remember) }}  
 <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>  
 </form>  
  
</div> <!-- /container -->  
{% endblock %}

@app.route('/login', methods=['GET', 'POST'])  
def login():  
 form = LoginForm()  
  
 if form.validate\_on\_submit():  
 user = Users.query.filter\_by(username=form.username.data).first()  
 if user:  
 if check\_password\_hash(user.password, form.password.data):  
 login\_user(user, remember=form.remember.data)  
 return redirect(url\_for('admin'))  
  
 flash('Wrong email or password')  
 return render\_template('login.html', form=form)  
 return render\_template('login.html', form=form)

**Register**

Once the admin has logged in they can register a new user via the register page.

register.html

{% extends 'layout.html' %}  
{% import "bootstrap/wtf.html" **as** wtf %}  
  
{% block body %}  
<div class="container">  
  
 <form class="form-signin" method="POST" action="/register">  
 <h2 class="form-signin-heading">Register</h2>  
 {{ form.hidden\_tag() }}  
 {{ wtf.form\_field(form.username) }}  
 {{ wtf.form\_field(form.email) }}  
 {{ wtf.form\_field(form.password) }}  
 {{ wtf.form\_field(form.confirm) }}  
  
 <button class="btn btn-lg btn-primary btn-block" type="submit">Sign Up</button>  
 </form>  
  
</div> <!-- /container -->  
{% endblock %}

Views.py

@app.route('/register', methods=['GET', 'POST'])  
@login\_required  
def register():  
 form = RegisterForm()  
  
 if form.validate\_on\_submit():  
 hashed\_password = generate\_password\_hash(form.password.data, method='sha256')  
 new\_user = Users(username=form.username.data, email=form.email.data, password=hashed\_password, urole="user")  
 db.session.add(new\_user)  
 db.session.commit()  
  
 return redirect(url\_for('login'))  
  
 return render\_template('register.html', form=form)

**Add Server**

Addservers.html

{% extends 'layout.html' %}  
{% import "bootstrap/wtf.html" **as** wtf %}  
  
 {% block body %}  
 <div class="jumbotron text-center">  
 <h1>Welcome to the addserver page</h1>  
 <p class='lead'>WIP</p>  
 </div>  
  
  
  
 <div class="container">  
  
 <form class="form-signin" method="POST" action="/addserver">  
 <h2 class="form-signin-heading">Add Server</h2>  
 {{ form.hidden\_tag() }}  
 {{ wtf.form\_field(form.servername) }}  
 {{ wtf.form\_field(form.ipaddress) }}  
 {{ wtf.form\_field(form.primaryrole) }}  
 {{ wtf.form\_field(form.secondaryrole) }}  
 {{ wtf.form\_field(form.commission) }}  
 {{ wtf.form\_field(form.make) }}  
 <button class="btn btn-lg btn-primary btn-block" type="submit">Add Server</button>  
 </form>  
  
  
</div> <!-- /container -->  
 <h1>Please submit the form</h1>  
 {% endblock %}

Views.py

@app.route('/addserver', methods=['GET', 'POST'])  
@login\_required  
def addserver():  
 form = AddServerForm()  
 if form.validate\_on\_submit():  
 x = os.environ.get("DOMAIN\_USERNAME")  
 y = os.environ.get("DOMAIN\_PWD")

v = vc(y, encryption\_key)  
 y = v.VernamCipher(y, encryption\_key)

z = form.ipaddress.data  
 server\_info = Connector(z, x, y)  
 vm, name, status, operatingsystem, notinuse, totalmem, \  
 numofcores, numofcpu, cpuname, avgcpuload, uptime, drivelist = server\_info.allstats()  
  
 new\_server = Servers(servername=form.servername.data, ipaddress=form.ipaddress.data,  
 primaryrole=form.primaryrole.data, secondaryrole=form.secondaryrole.data,  
 commission=form.commission.data, make=form.make.data)  
 db.session.add(new\_server)  
 print("Adding Basics")  
 time.sleep(2)  
 db.session.commit()  
 print("Added!")  
 time.sleep(0.5)  
 ipselect = db.engine.execute(text("SELECT id FROM SERVERS WHERE ipaddress = :IPA"), IPA=form.ipaddress.data)  
 for row in ipselect:  
 print(row)  
 serverid = row[0]  
 new\_serverinfo = serverinfo(servers\_id=serverid, operatingsystem=operatingsystem, cpuload=avgcpuload,  
 ramnotinuse=notinuse, totalram=totalmem, status=status,  
 cpuname=cpuname, numofcores=numofcores, numofcpu=numofcpu, uptime=uptime)  
 db.session.add(new\_serverinfo)  
 print("Adding info...")  
 time.sleep(2)  
 db.session.commit()  
 print("Added!")  
  
 drivelistlen = len(drivelist)  
 for i in range(drivelistlen):  
 mapping, totalspace, freespace, percentageused = drivelist[i]  
 new\_serverdrives = serverdrives(servers\_id = serverid, drivemapping=mapping, drivefreespace = freespace,  
 drivetotalspace = totalspace, percentused=percentageused)  
 db.session.add(new\_serverdrives)  
 db.session.commit()  
  
 return redirect(url\_for('admin'))  
  
 return render\_template('addserver.html', form = form)

(Forms.py)

class AddServerForm(FlaskForm):  
 servername = StringField('Server Name', validators=[InputRequired(), Length(min=2, max=80)])  
 ipaddress = StringField('IP Address', validators=[InputRequired(), Length(min=4, max=80)])  
 primaryrole = StringField('Primary Role', validators=[InputRequired(), Length(min=1, max=200)])  
 secondaryrole = StringField('Secondary Role', validators=[Length(min=1, max=200),Optional()])  
 commission = DateField("Commission Date", format='%d/%m/%Y', validators=[Optional()])  
 make = StringField('Manufacturer (HP)', validators=[InputRequired(), Length(min=1, max=100)])

**Live Stats**

Stats.html

{% extends 'layout.html' %}  
{% import "bootstrap/wtf.html" **as** wtf %}  
  
 {% block body %}  
 <div class="jumbotron text-center">  
 <h1>Welcome to the stats page</h1>  
 <p class='lead'>WIP</p>  
 </div>  
  
  
  
 <div class="container">  
  
 <form class="form-signin" method="POST" action="/stats">  
 <h2 class="form-signin-heading">Server Stats</h2>  
 {{ form.hidden\_tag() }}  
 {{ wtf.form\_field(form.ipaddress) }}  
 <button class="btn btn-lg btn-primary btn-block" type="submit">Get Live Server Stats</button>  
 </form>  
  
  
  
  
 <div class="container">  
 <div class="spinner">  
 <div class="rect1"></div>  
 <div class="rect2"></div>  
 <div class="rect3"></div>  
 <div class="rect4"></div>  
 <div class="rect5"></div>  
 </div>  
 </div>  
  
 <script>  
  
 $("button").click(**function**() {  
 **var** $btn = $(**this**);  
 $btn.button('loading');  
 setTimeout(**function** () {  
 $btn.button('reset');  
 }, 8000);  
 });  
  
 </script>  
  
  
  
  
  
  
</div> <!-- /container -->  
 <table border = 1>  
 {% if form.validate\_on\_submit() %}  
  
 <dl class="dl-horizontal">  
 <dt>Operating System</dt>  
 <dd> {{ os }} </dd>  
 <dt>Disk Space</dt>  
 <dd> {{ map }} has {{ free }} GB of {{ total }}GB of free space</dd>  
 <dt>Up time</dt>  
 <dd> {{ uptime }} Days</dd>  
 <dt>CPU</dt>  
 <dd>The CPU load is {{ cpu }} %</dd>  
 <dt>Total Mem</dt>  
 <dd>This server has {{ mem }} GB of RAM </dd>  
 <dt>Mem</dt>  
 <dd>{{ notinuse }} GB of RAM is not in use</dd>  
 <dt>VM</dt>  
 <dd> {{ vm }} </dd>  
 <dt>Name</dt>  
 <dd> {{ name }} </dd>  
 <dt>Status</dt>  
 <dd> {{ status }} </dd>  
 </dl>  
  
 {% endif %}  
 </table>  
 <h1>Please submit the form</h1>  
 {% endblock %}

Views.py

@app.route('/stats', methods=['GET', 'POST'])  
@login\_required  
def stats():  
 form = IPAddressform()  
 x = os.environ.get("DOMAIN\_USERNAME")  
 y = os.environ.get("DOMAIN\_PWD")

v = vc(y, encryption\_key)  
 y = v.VernamCipher(y ,encryption\_key)

if form.validate\_on\_submit():  
 ip = form.ipaddress.data  
 connection = Connector(ip,x,y)  
 vm, name, status, operatingsystem, notinuse, totalmem, \  
 numofcores, numofcpu, cpuname, avgcpuload, uptime, drivelist = connection.allstats()  
 drivelistlen = len(drivelist)  
 for i in range(drivelistlen):  
 mapping, totalspace, freespace, percentageused = drivelist[i]  
  
return render\_template('stats.html', form=form, os=operatingsystem, map = mapping, free=freespace, total=totalspace, uptime=uptime, cpu=avgcpuload, mem=totalmem, notinuse=notinuse, vm=vm, name=name, status=status)  
  
return render\_template('stats.html', form = form)

**Servers**

Servers.html

{% extends 'layout.html' %}  
{% import "bootstrap/wtf.html" **as** wtf %}  
  
{% block body %}  
 <div class="page-header">  
 <h1>Server Dashboard <small>Find info on servers</small></h1>  
 </div>  
 {% for info **in** basicserverinfo %}  
 <div class="container">  
 <h2>{{ info['ServerName'] }}</h2>  
 <table class="table table-bordered dl-horizontal">  
 <dt>Server Name</dt>  
 <dd> {{ info['ServerName'] }}</dd>  
 <dt>Primary Role</dt>  
 <dd> {{ info['Primary Role'] }}</dd>  
 <dt>Secondary Role</dt>  
 <dd> {{ info['Secondary Role'] }}</dd>  
 <dt>CPU</dt>  
 <dd> {{ info['CPU-Name'] }}</dd>  
 <dt>Operating System</dt>  
 <dd> {{ info['Operating System'] }}</dd>  
 </table>  
 </div>  
 {% for drives **in** serverdrives %}  
 {% if drives['Server ID'] == info['Server ID'] %}  
 <div class="container">  
 <table class="table table-bordered dl-horizontal">  
 <dt>Drive Mapping</dt>  
 <dd> {{ drives['Drive Mapping'] }} </dd>  
 <dt>Free Space</dt>  
 <dd> {{ drives['FreeSpace'] }} </dd>  
 <dt>Bar</dt>  
 <div class="progress">  
 <div class="progress-bar" role="progressbar" aria- valuenow="70"  
 aria-valuemin="0" aria-valuemax="100" style="width:{{ drives['Percentage Used'] }}%">  
 {{ drives['Percentage Used'] }}%  
 </div>  
 </div>  
 </dd>  
 </table>  
 </div>  
  
 {% endif %}{% endfor %}{% endfor %}  
  
{% endblock %}

Views.py

@app.route("/servers", methods=['GET', 'POST'])  
def servers():  
 drivelists = []  
 serverlist = []  
 server = db.engine.execute(text("""SELECT servers.id, servers.servername, servers.primaryrole, servers.secondaryrole, serverinfo.cpuname, serverinfo.operatingsystem FROM servers LEFT JOIN serverinfo   
ON serverinfo.servers\_id = servers.id GROUP BY servers.id"""))  
  
drives = db.engine.execute(text("""SELECT serverdrives.servers\_id, serverdrives.drivemapping, serverdrives.drivefreespace, serverdrives.drivetotalspace, serverdrives.percentused FROM serverdrives"""))  
  
  
 for row in server:  
 serverlist.append({'Server ID': row[0], 'ServerName': row[1], 'Primary Role': row[2], 'Secondary Role': row[3],'CPU-Name': row[4], 'Operating System': row[5]})  
  
 for row in drives:  
 drivelists.append({'Server ID': row[0], 'Drive Mapping': row[1], 'FreeSpace': row[2], 'TotalSpace': row[3],'Percentage Used': row[4]})  
  
return render\_template('servers.html', basicserverinfo=serverlist, serverdrives = drivelists)

**Domain Account**

Domainaccount.html

{% extends 'layout.html' %}  
{% import "bootstrap/wtf.html" **as** wtf %}  
  
 {% block body %}  
 <div class="jumbotron text-center">  
 <h1>Add a Domain Account</h1>  
 <p class='lead'>WIP</p>  
 </div>  
  
  
  
 <div class="container">  
  
 <form class="form-signin" method="POST" action="/domainaccount">  
 <h2 class="form-signin-heading">Add Domain Account</h2>  
 {{ form.hidden\_tag() }}  
 {{ wtf.form\_field(form.username) }}  
 {{ wtf.form\_field(form.password) }}  
 {{ wtf.form\_field(form.confirm) }}  
  
 <button class="btn btn-lg btn-primary btn-block" type="submit">Add Domain account</button>  
 </form>  
  
</div> <!-- /container -->  
 <h1> {{ data }}</h1>  
 <h1>Please submit the form</h1>  
 {% endblock %}

Views.py

@app.route('/domainaccount', methods=['GET','POST'])  
@login\_required  
def domainaccount():  
 form = add\_domain\_account()  
  
 if form.validate\_on\_submit():  
 v = vc(form.password.data, encryption\_key)  
 y = v.VernamCipher(form.password.data, encryption\_key)  
 os.environ["DOMAIN\_USERNAME"] = form.username.data  
 os.environ["DOMAIN\_PWD"] = y  
 #export password to env variable  
 print(os.environ.get('DOMAIN\_USERNAME'))  
 print(os.environ.get('DOMAIN\_PWD'))  
 return render\_template('domainaccount.html', form = form)

Forms.py

class add\_domain\_account(FlaskForm):  
 username = StringField('Domain Username', validators=[InputRequired(), Length(min=4, max=80)])  
 password = PasswordField('Domain password', validators=[InputRequired(), Length(min=8, max=80),  
 EqualTo('confirm', message='Passwords do not match')])  
 confirm = PasswordField('Confirm Password')

**Emailer**

Emailer.html

{% extends 'layout.html' %}  
{% import "bootstrap/wtf.html" **as** wtf %}  
{% block body %}  
  
  
  
  
 <div class="container">  
 <h3>Current Users who receive emails: </h3>  
 {% for i **in** contactlist %}  
 <p>{{ i }}</p>  
 {% endfor %}  
 </div>  
  
  
 <div class="container">  
 <form class="form-signin" method="POST" action="/emailer">  
 <h2 class="form-signin-heading">Add Email Address to receive notifications</h2>  
 {{ addform.hidden\_tag() }}  
 {{ wtf.form\_field(addform.name) }}  
 {{ wtf.form\_field(addform.emailAddress) }}  
  
 <button class="btn btn-lg btn-primary btn-block" type="submit">Add Email Address</button>  
 </form>  
 </div>  
  
  
 <div class="container">  
 <form class="form-signin" method="POST" action="/emailer">  
 <h2 class="form-signin-heading">Remove Email Address from contacts</h2>  
 {{ removeform.hidden\_tag() }}  
 {{ wtf.form\_field(removeform.emailAddress) }}  
  
 <button class="btn btn-lg btn-primary btn-block" type="submit">Remove This Email Address</button>  
 </form>  
 </div>  
  
  
{% endblock %}

Views.py

app.route('/emailer', methods=['GET', 'POST'])  
@login\_required  
def emailer():  
  
 contactlist = []  
  
 emailaddform = emailform()  
 emailremoverform = emailremoveform()  
  
 if request.method == 'GET':  
 f = open("contacts.txt", "r")  
 for line in f:  
 contactlist.append(line)  
 print(contactlist)  
  
 if request.method == 'POST' and emailaddform.validate\_on\_submit():  
 name = emailaddform.name.data  
 emailaddress = emailaddform.emailAddress.data  
 f = open('contacts.txt', 'a')  
 f.write(name +' - '+ emailaddress +' \n')  
 f.close()  
 return redirect(url\_for('emailer'))  
  
 if request.method == 'POST' and emailremoverform.validate\_on\_submit():  
 removeaddress = emailremoverform.emailAddress.data  
 print(removeaddress)  
 f = open('contacts.txt', 'r')  
 lines = f.readlines()  
 f.close()  
 f = open('contacts.txt', 'w')  
 for line in lines:  
 if removeaddress not in line:  
 f.write(line)  
 f.close()  
 return redirect(url\_for('emailer'))  
  
  
  
 return render\_template('emailer.html', addform=emailaddform, removeform=emailremoverform, contactlist=contactlist)

forms.py

class emailform(FlaskForm):  
 name = StringField('Name', validators=[InputRequired(), Length(min=3, max=80)])  
 emailAddress = StringField('Email Address', validators=[InputRequired(), Email(message='Invalid email'), Length(max=50), ])

emailer.py

import smtplib  
from email.mime.multipart import MIMEMultipart  
from email.mime.text import MIMEText  
from string import Template  
  
  
def emailer():  
 fromaddr = "cmuiemailer@gmail.com"  
 toaddr = "info@cmui.co.uk"  
 msg = MIMEMultipart()  
 msg['From'] = fromaddr  
 msg['To'] = toaddr  
 msg['Subject'] = "Test mail"  
  
 body = "CMUI test"  
 msg.attach(MIMEText(body, 'plain'))  
  
 server = smtplib.SMTP('smtp.gmail.com', 587)  
 server.starttls()  
 server.login(fromaddr, "Zebra100")  
 text = msg.as\_string()  
 server.sendmail(fromaddr, toaddr, text)  
 server.quit()  
  
  
  
  
def get\_contacts(filename):  
 names = []  
 emails = []  
 with open(filename, mode='r', encoding='utf-8') as contacts\_file:  
 for a\_contact in contacts\_file:  
 names.append(a\_contact.split()[0])  
 emails.append(a\_contact.split()[1])  
 print(names, emails)  
 return names, emails  
  
  
get\_contacts('contacts.txt')

**Reload server stats**

Views.py

def reloadserverstats():  
 serverIP = []  
 x = os.environ.get("DOMAIN\_USERNAME")  
 y = os.environ.get("DOMAIN\_PWD")  
 y = VernamCipher(y, encryption\_key)  
 print(x, y)  
  
 servers = db.engine.execute(text("""SELECT ipaddress FROM servers"""))  
  
 #runtime needs to be added  
 for ip in servers:  
 serverIP.append(ip)  
 reloader = Connector(ip, x, y)  
 vm, name, status, operatingsystem, notinuse, totalmem, \  
 numofcores, numofcpu, cpuname, avgcpuload, uptime, drivelist = reloader.allstats()  
 drivelistlen = len(drivelist)  
 for i in range(drivelistlen):  
 mapping, totalspace, freespace, percentageused = drivelist[i]  
  
 db.engine.execute(text("""UPDATE serverinfo SET serverinfo.operatingsystem = :OS, serverinfo.ramnotinuse =   
 :notinuse, serverinfo.totalram = :totalmem, serverinfo.numofcores = :cores, serverinfo.numofcpu = :cpu,  
 serverinfo.cpuname = :cpuname, serverinfo.cpuload = :cpuload WHERE servers\_id IN   
 (SELECT servers.id FROM servers WHERE ipaddress = :ip);"""))  
 db.engine.execute(text("""UPDATE serverdrives SET serverdrives.drivemapping = :map, serverdrives.drivetotalspace,  
 serverdrives.drivefreespace WHERE servers\_id IN (SELECT servers.id FROM servers WHERE ipaddress = :ip); """))

**WMI Module**

wmiutil.py

import sys, wmi, os  
import socket  
import pythoncom  
import threading  
  
def grabip():  
 s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)  
 s.connect(("8.8.8.8", 80))  
 trueip = (s.getsockname()[0])  
 s.close()  
 return trueip  
  
  
class Connector():  
 *"""WMI Connector to server"""* def \_\_init\_\_(self, ip, username, password):  
 self.ip = ip  
 self.username = username  
 self.password = password  
  
 pythoncom.CoInitialize()  
 global c  
 #Use ip,username,password to connect to server  
 local\_ip = grabip()  
 if ip == local\_ip:  
 c = wmi.WMI()  
 else:  
 c = wmi.WMI(ip, user=username, password=password)  
  
 def connect(self):  
 try:  
 #print("Establishing connection to %s" % ip)  
 for os in c.Win32\_OperatingSystem():  
 print(os.Caption)  
 return os.Caption  
 print("Connection established")  
 except wmi.x\_wmi:  
 print("Your Username and Password of " + getfqdn(ip) + " are wrong.")  
 print("error")  
  
  
 def diskspace(self):  
 for d in c.Win32\_LogicalDisk():  
 a = d.FreeSpace  
 freespace = round((int(a) / 2 \*\* 30), 2)  
 b = d.Size  
 totalspace = round((int(b) / 2 \*\* 30), 2)  
 drive = d.Caption  
 print("The drive %s has %s GB free space out of its %s GB capacity" % (drive, freespace, totalspace))  
 return drive, freespace, totalspace  
  
  
 def get\_uptime(self):  
 uptime = int([uptime.SystemUpTime for uptime in c.Win32\_PerfFormattedData\_PerfOS\_System()][0])  
 uptimedays = round(((uptime / 3600) / 24), 2)  
 print("The Server has been up for %d days" % (uptimedays))  
 return uptimedays  
  
 def get\_cpu(self):  
 cpuload = [cpu.LoadPercentage for cpu in c.Win32\_Processor()]  
 avgcpuload = int(sum(cpuload) / len(cpuload)) # avg all cores/processors! Change to per core?  
 print("The CPU load is %d percent" % (avgcpuload))  
 numofcores = ([cpu.NumberOfLogicalProcessors for cpu in c.Win32\_ComputerSystem()][0])  
 numofcpu = ([cpu.NumberOfProcessors for cpu in c.Win32\_ComputerSystem()][0])  
 cpu\_name = ([cpu.Name for cpu in c.Win32\_Processor()][0])  
 return avgcpuload, numofcores, numofcpu, cpu\_name  
  
  
 def totaltestmem(self):  
 memt = ([mem.TotalVisibleMemorySize for mem in c.Win32\_OperatingSystem()][0])  
 a = memt  
 totalmem = round((int(a) / 2 \*\* 20), 2)  
 #This is my own code that returns the RAM. 7.9 on laptop is correct  
 print("This server has %d GB of total RAM" % (totalmem))  
 return totalmem  
  
 def notinusemem(self):  
 numem = ([mem.FreePhysicalMemory for mem in c.Win32\_OperatingSystem()][0])  
 a = numem  
 notinuse = round((int(a) / 2 \*\* 20), 2)  
 print("%d GB of RAM is not in use" % (notinuse))  
 return notinuse  
  
  
  
  
  
 def sysinfo(self):  
 #This uses the Win32 ComputerSystem Class  
 #This return the model name but calls a VM a VM instead of a model name  
 vm = ([system.Model for system in c.Win32\_ComputerSystem()][0])  
 name = ([system.Name for system in c.Win32\_ComputerSystem()][0])  
 status = ([system.Status for system in c.Win32\_ComputerSystem()][0])  
 return vm, name, status  
  
  
 def allstats(self):  
 drivelist = []  
 #Computer info  
 vm = ([system.Model for system in c.Win32\_ComputerSystem()][0])  
 ####need to change this  
 name = ([system.Name for system in c.Win32\_ComputerSystem()][0])  
 status = ([system.Status for system in c.Win32\_ComputerSystem()][0])  
 os = ([system.Caption for system in c.Win32\_OperatingSystem()][0])  
 #mem stats  
 notinuse = round((int(([mem.FreePhysicalMemory for mem in c.Win32\_OperatingSystem()][0])) / 2 \*\* 20), 2)  
 totalmem = round((int(([mem.TotalVisibleMemorySize for mem in c.Win32\_OperatingSystem()][0])) / 2 \*\* 20), 2)  
 #cpu stats  
 numofcores = ([cpu.NumberOfLogicalProcessors for cpu in c.Win32\_ComputerSystem()][0])  
 numofcpu = ([cpu.NumberOfProcessors for cpu in c.Win32\_ComputerSystem()][0])  
 cpu\_name = ([cpu.Name for cpu in c.Win32\_Processor()][0])  
 avgcpuload = int(sum([cpu.LoadPercentage for cpu in c.Win32\_Processor()]) /  
 len([cpu.LoadPercentage for cpu in c.Win32\_Processor()])) # avg all cores/processors! Change to per core?  
 #other stats  
 uptime = round(((int([uptime.SystemUpTime for uptime in  
 c.Win32\_PerfFormattedData\_PerfOS\_System()][0])) / 3600 / 24), 2)  
  
 for drive in c.Win32\_LogicalDisk():  
 try:  
 drivename = drive.caption  
 totalspace = round((int(drive.Size) / 2 \*\* 30), 2)  
 freespace = round((int(drive.FreeSpace) / 2 \*\* 30), 2)  
 usedspace = totalspace - freespace  
 percentused = round((usedspace / totalspace \* 100), 2)  
 singledrive = drivename, totalspace, freespace, percentused  
 drivelist.append(singledrive)  
 except TypeError:  
 pass  
  
 return vm, name, status, os, notinuse, totalmem, numofcores, numofcpu, cpu\_name, avgcpuload, uptime, drivelist  
  
  
  
def runtime():  
 uptime = round(((int([uptime.SystemUpTime for uptime in  
 c.Win32\_PerfFormattedData\_PerfOS\_System()][0])) / 3600 / 24), 2)  
  
  
  
def servertest():  
 p = Connector("192.168.31.86", "CMUIAdmin", "Admin2017")

**Database Models**

Models.py

from KSApp import db  
from flask\_login import UserMixin  
from sqlalchemy.orm import relationship  
  
class Users(UserMixin, db.Model):  
 \_\_tablename\_\_ = 'Users'  
 id = db.Column(db.Integer,primary\_key=True)  
 username = db.Column(db.String(80),unique=True)  
 password = db.Column(db.String(200))  
 email = db.Column(db.String(256),unique=True)  
 urole = db.Column(db.String(80))  
  
  
 def \_\_init\_\_(self,username,password,email,urole):  
 self.username = username  
 self.password = password  
 self.email = email  
 self.urole = urole  
  
  
 def get\_id(self):  
 return self.id  
 def get\_username(self):  
 return self.username  
 def get\_urole(self):  
 return self.urole  
  
  
  
  
  
class Servers(db.Model):  
 \_\_tablename\_\_ = 'servers'  
 id = db.Column(db.Integer, primary\_key=True)  
 servername = db.Column(db.String(200), nullable=False, unique=True)  
 ipaddress = db.Column(db.String(80), nullable=False)  
 primaryrole = db.Column(db.String(200), nullable=False)  
 secondaryrole = db.Column(db.String(200), nullable=True)  
 commission = db.Column(db.Date, nullable=True)  
 make = db.Column(db.String(100), nullable=True)  
 serverinfo = relationship("serverinfo", uselist=False, backref="servers")  
 serverdrives = relationship("serverdrives", uselist=False, backref="servers")  
  
 def \_\_init\_\_(self, servername, ipaddress, primaryrole, secondaryrole, commission, make):  
  
 self.servername = servername  
 self.ipaddress = ipaddress  
 self.primaryrole = primaryrole  
 self.secondaryrole = secondaryrole  
 self.commission = commission  
 self.make = make  
  
 def get\_id(self):  
 return self.id  
  
 def get\_servername(self):  
 return self.servername  
  
 def get\_ipaddress(self):  
 return self.ipaddress  
  
  
class serverinfo(db.Model):  
 \_\_tablename\_\_ = 'serverinfo'  
 id = db.Column(db.Integer, primary\_key=True)  
 servers\_id = db.Column(db.Integer, db.ForeignKey("servers.id"))  
 operatingsystem = db.Column(db.String(200), nullable=True)  
 cpuload = db.Column(db.Integer, nullable=True)  
 totalram = db.Column(db.Integer, nullable=True)  
 ramnotinuse = db.Column(db.Integer, nullable=True)  
 status = db.Column(db.String(100))  
 cpuname = db.Column(db.String(200))  
 numofcores = db.Column(db.Integer)  
 numofcpu = db.Column(db.Integer)  
 uptime = db.Column(db.Integer)  
  
  
 def \_\_init\_\_(self, servers\_id, operatingsystem, cpuload, ramnotinuse,  
 totalram, status, cpuname, numofcores, numofcpu, uptime):  
  
 self.servers\_id = servers\_id  
 self.operatingsystem = operatingsystem  
 self.cpuload = cpuload  
 self.ramnotinuse = ramnotinuse  
 self.totalram = totalram  
 self.status = status  
 self.cpuname = cpuname  
 self.numofcores = numofcores  
 self.numofcpu = numofcpu  
 self.uptime = uptime  
  
 def get\_id(self):  
 return self.id  
  
  
  
class serverdrives(db.Model):  
 \_\_tablename\_\_ = "serverdrives"  
 id = db.Column(db.Integer, primary\_key=True)  
 servers\_id = db.Column(db.Integer, db.ForeignKey("servers.id"))  
 drivemapping = db.Column(db.String(200), nullable=True)  
 drivefreespace = db.Column(db.Integer, nullable=True)  
 drivetotalspace = db.Column(db.Integer, nullable=True)  
 percentused = db.Column(db.Integer, nullable=True)  
  
  
 def \_\_int\_\_(self, servers\_id, drivemapping, drivefreespace, drivetotalspace, percentused):  
  
 self.servers\_id = servers\_id  
 self.drivemapping = drivemapping  
 self.drivefreespace = drivefreespace  
 self.drivetotalspace = drivetotalspace  
 self.percentused = percentused

**Init and running the app**

\_\_Init\_\_.py

import os  
from flask import Flask  
from filter import getsevername  
from flask\_sqlalchemy import SQLAlchemy  
from flask\_login import LoginManager  
from flask\_bootstrap import Bootstrap  
  
  
  
app = Flask(\_\_name\_\_)  
app.config.from\_object('config')  
  
  
app = Flask(\_\_name\_\_)  
app.config['SQLALCHEMY\_DATABASE\_URI'] = 'mysql://mahoney:Yb1dAI3vna3xTAMuJNOV@localhost:3306/kasldb'  
db = SQLAlchemy(app)  
app.config['SECRET\_KEY'] = 'hfoafoafodnpnad'  
bootstrap = Bootstrap(app)  
login\_manager = LoginManager()  
login\_manager.init\_app(app)  
login\_manager.login\_view = 'login'  
login\_manager.login\_message = 'Please Login'  
  
  
  
  
  
from KSApp.models import Users, Servers, serverinfo  
  
db.create\_all()  
  
db.session.commit()  
  
  
from KSApp import views

run.py

from KSApp import app  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True, use\_reloader=True)

**Checking the CSV file exists**

Fetchcsv.py

import csv  
from os.path import exists  
  
FILE\_NAME = 'KASLnetwork.csv'  
  
def fetch(fn):  
 if exists(fn):  
 print(fn, "Already exists")  
 else:  
 fn = str(input("Input file name (Network.csv): "))  
 if exists(fn):  
 print(fn, "now exists")  
  
  
def read\_data():  
 fetch(FILE\_NAME)  
 with open(FILE\_NAME, 'r') as rf:  
 return rf.read()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 read\_data()

**Error Handling**

@app.errorhandler(404)  
def page\_not\_found(e):  
 return render\_template('404.html'), 404  
  
@app.errorhandler(403)  
def page\_not\_found(e):  
 return render\_template('403.html'), 403  
  
@app.errorhandler(410)  
def page\_not\_found(e):  
 return render\_template('410.html'), 410  
  
@app.errorhandler(500)  
def page\_not\_found(e):  
 return render\_template('500.html'), 500  
  
@login\_manager.user\_loader  
def load\_user(user\_id):  
 try:  
 return Users.query.get(int(user\_id))  
 except exc.InvalidRequestError:  
 print("Error...")

**Vernam Cipher**

class vc:  
  
 def \_\_init\_\_(self, text, key):  
 self.text = text  
 self.key = key  
  
 def VernamCipher(self, text, key):  
 result = ""  
 ptr = 0  
 for char in text:  
 result = result + chr(ord(char) ^ ord(key[ptr]))  
 ptr = ptr + 1  
 if ptr == len(key):  
 ptr = 0  
 return result

**Testing**

Since the system is going to be used in a corporate environment it needs to be thoroughly tested and ready to be deployed. King & Shaxson have sent me a powerful workstation computer to test the system. I will be using Microsoft’s Hyper-V to virtualise a domain with two computers and two servers. The computers will have different domain rights to test the security of the system and the servers will be running Server 2012 R2 and Server 2016 to test compatibility.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Description | Data | Expected Result | Pass/fail |
| 1 | Entering correct details into stats page | 192.168.31.16 | **Operating System**  Microsoft Windows 10 Pro  **Disk Space**  C: has 84.77 GB of 236.55GB of free space  **Up time**  15.6 Days  **CPU**  The CPU load is 65 %  **Total Mem**  This server has 7.9 GB of RAM  **Mem**  1.55 GB of RAM is not in use  **VM**  TM1613  **Name**  MAHONEY-MIBOOK  **Status**  OK | Pass |
| 2 | Entering Invalid details into stats page | TestString | Redirect to error page | Pass |
| 3 | Entering valid login details | Username : Admin  Password:…. | Redirect to welcome page with access to the admin page | Pass |
| 4 | Entering invalid login details | Username: Username123  Password:  Password123 | Flash message “Wrong username or password” | Pass |
| 5 | Can users enter the admin page without logging in. | URL: 127.0.0.1:5000/admin | URL redirects to:  127.0.0.1:5000/login?next=%2Fadmin | Pass |
| 6 | Can a user that has logged on access the admin page | URL:  127.0.0.1:5000/admin | URL:  127.0.0.1:5000/admin | Pass |
| 7 | Can a user add a server | Server Name: CMUITESTSERVER2  IP Address :  192.168.31.87  Primary Role:  Web Server  Secondary Role:  MySQL Server  Commission Date:  26/03/2015  Manufacturer: HP | Redirect to admin page and server listed on the servers page with up to date statistics. | Pass |
| 8 | Can a user add the wrong info to add server without it crashing | Server Name: Test  IP Address :  8.8.8.8  Primary Role:  Test  Secondary Role:  (left blank)  Commission Date:  27/03/2016  Manufacturer: ? | Redirects to error page and server isn’t added to servers page. | Pass |
| 9 | Use incorrect date format in add servers | Commission Date:  Test | Form isn’t submitted without valid date | Pass |
| 10 | Adding Email to contacts file | Name – Test  Email Address – test@cmui.co.uk |  | Yes |
| 11 | Remove an email from contacts file | Email Address – test@cmui.co.uk |  | Yes |
| 12 | Add a domain account to the system | Domain Username – CMUIAdmin  Password – Admin2017 | Output in console with CMUIAdmin and the encrypted password. | Yes |
| 13 | Trying to add a domain account with the wrong password confirmed | Domain Username- CMUIAdmin  Password – Admin2017  Confirm Password – Admin2018 |  | Yes |
| 14 | Trying to use WMI with the wrong domain details. | Username – CMUIAdmin  Password- Admin2019  Accessing 192.168.31.86 in the stats page. | Console outputs access denied WMI | Yes |

**Evaluation**

**Introduction**

1. Create a login system for users and redirecting them to the home page.
2. Create a page where users can add servers to the database
3. Create a page where users can see the latest server information
4. Automate gathering information from servers
5. Allow users to add and remove emails from a file
6. Encrypt domain login details and decrypt them and use them to access WMI
7. Send email notifications if a server has a problem
8. Create a page where users can input an IP and get the latest stats on that machine
9. Automatically generate and admin account and password when the app is first run

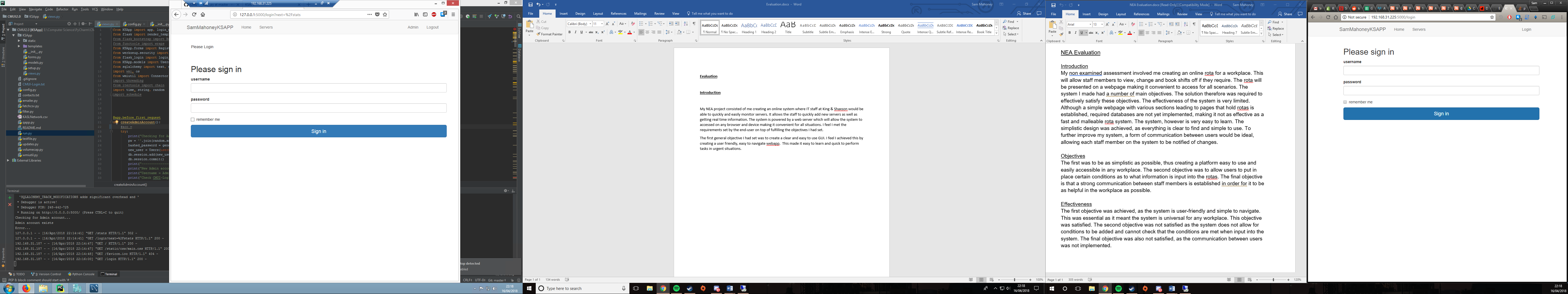
My NEA project consisted of me creating an online system where IT staff at King & Shaxson

would be able to quickly and easily monitor servers. It allows the staff to quickly add new servers as well as getting real time information. The system is powered by a web server which will allow the system to accessed on any browser and device making it convenient for all situations. I feel I met the requirements set by the end-user on top of fulfilling the objectives I had set.

My first objective was to provide a login system to the webapp. This was also achieved as there is a database which stored hashed passwords and a clear login page for users.

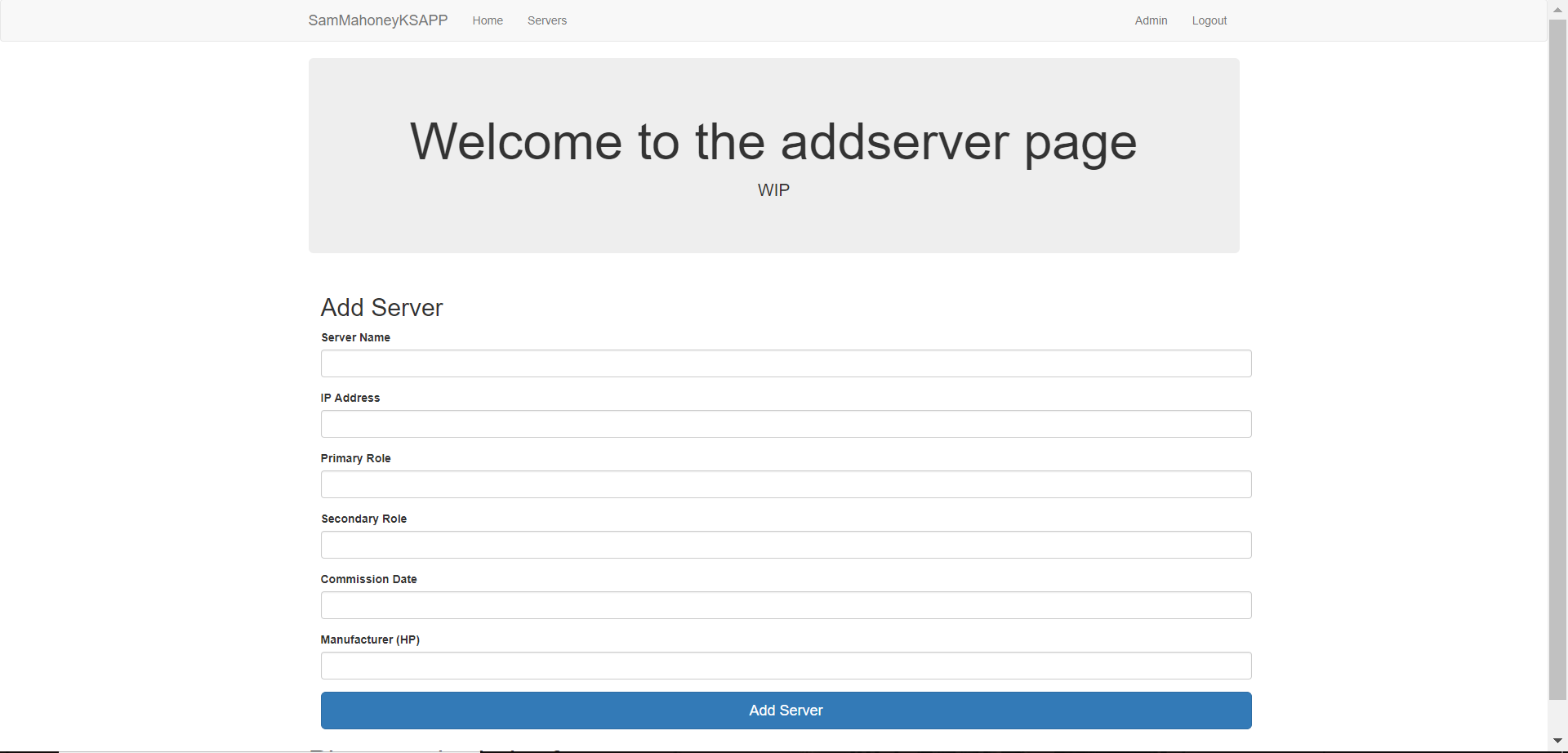


The code above shows how the login system is programmed. It uses the username from the form and compares the password hash with the one in the database. To keep the app secure an admin account is generated when the app is first initialised and the password and username appear on the console so the admin can login and create account. This means that the register page isn’t open to anyone to create an account.

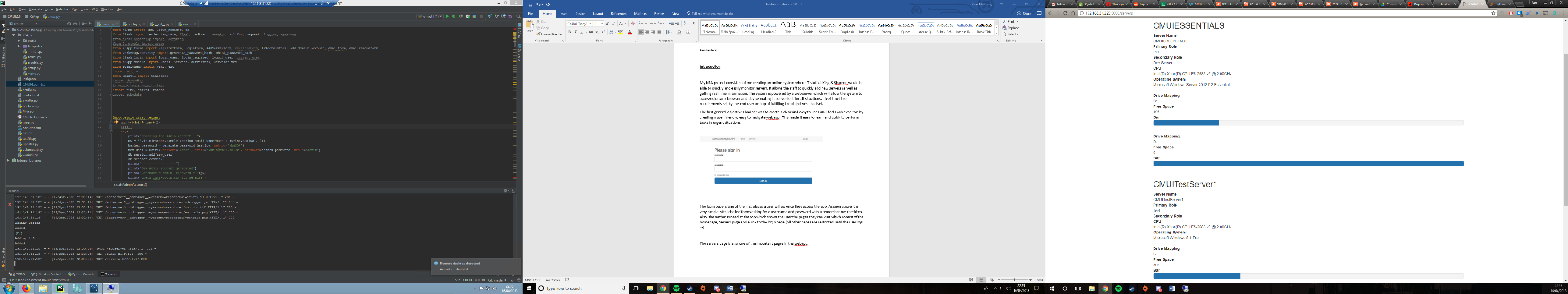


The login page is one of the first places a user will go once they access the app. As seen above it is very simple with labelled forms asking for a username and password with a remember me checkbox. Also, the navbar is need at the top which shows the user the pages they can visit which consist of the homepage, Servers page and a link to the login page (All other pages are restricted until the user logs in).

My second objective was to create a page where users can add servers to the database. This was meet with the create of the addservers page on the webapp as well as using WMI to gather the required information and adding it to the database.



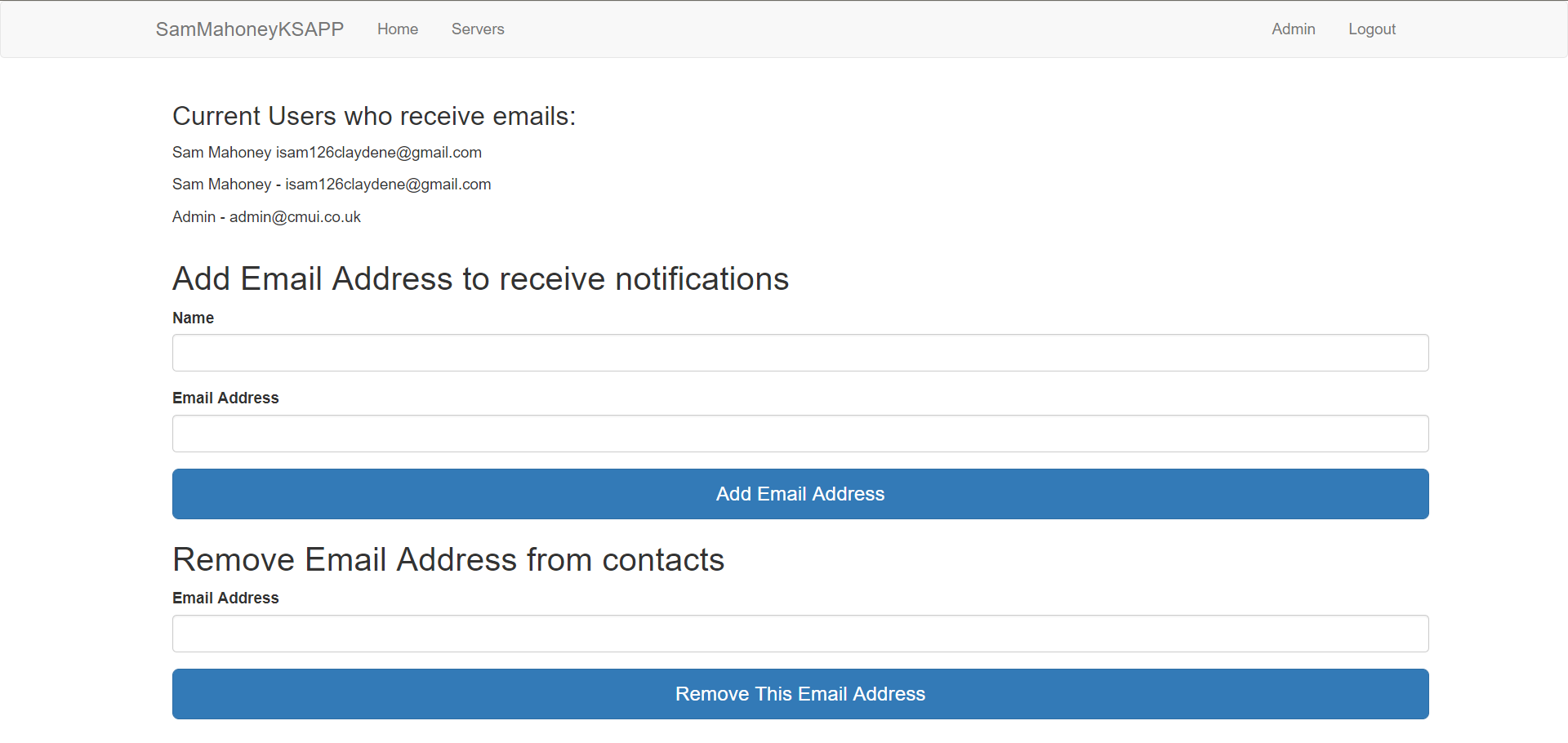
My third objective was to create the servers page where users can see the latest information on servers.

The servers page is also one of the important pages in the webapp. This is where staff can see the stats for all the servers added to the system.

As you can see in the picture it clearly shows different servers as well as key information that may be required. This makes it simple to pick out key information for specific servers.

My fourth objective was to automate tasks the IT staff did manually. I also achieved this objective since I have created a system which allows the staff to bypass remoting into a system to gain information and instead have it all available in the webpage. A quote from Bert was “This system allows us to be more efficient in doing day to day tasks and reduces the time it takes since we do not need to VNC into machines to check simple details”

My fifth objective was to allow users to add emails to a file and remove them after if they need to. I achieved this by creating a webpage where users can add email addresses to the file or remove them. The contents of the file are listed at the top of the webpage and updates when it has been changed.



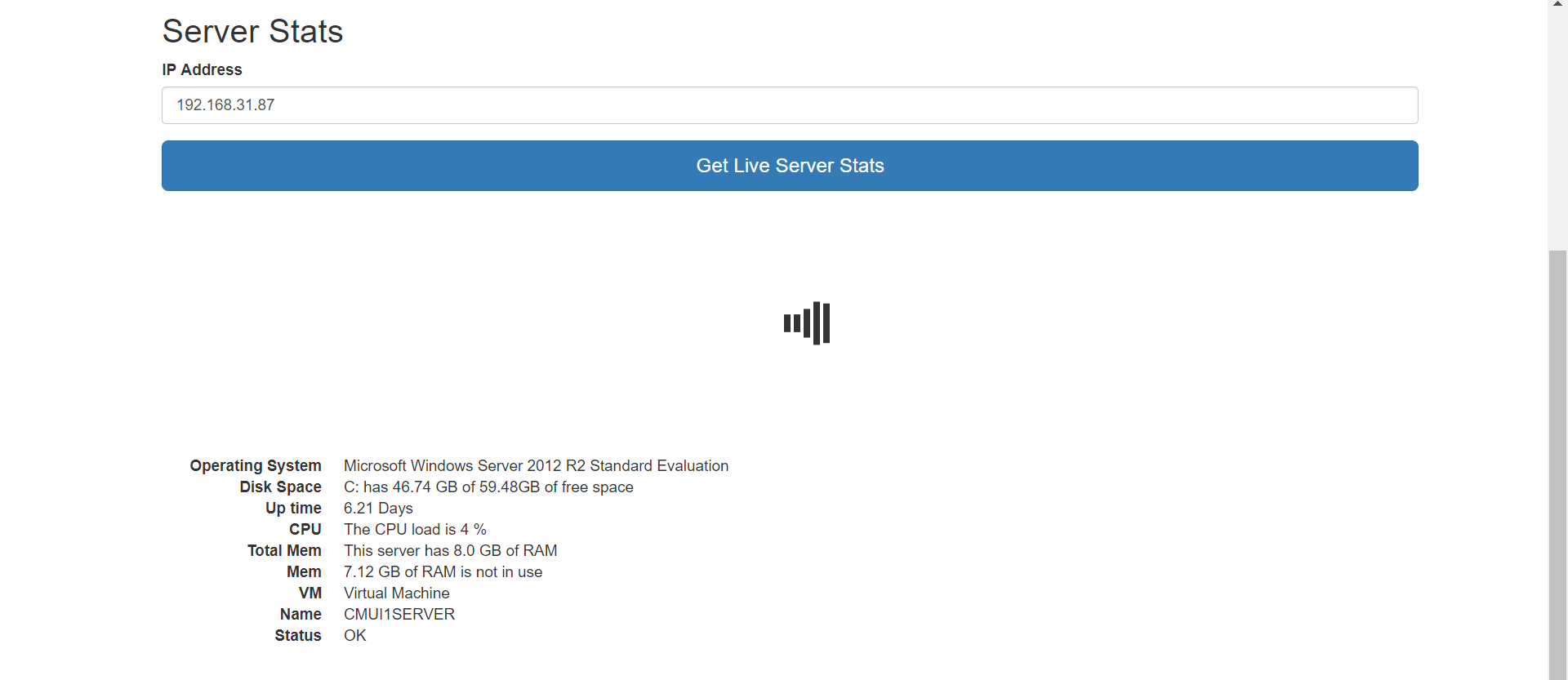
My sixth objective was to create my own encryption for the domain details. I did this by using a VernamCipher which I created in a separate class, so the function could be called throughout the webapp. The code for this is below.

class vc:  
  
 def \_\_init\_\_(self, text, key):  
 self.text = text  
 self.key = key  
  
 def VernamCipher(self, text, key):  
 result = ""  
 ptr = 0  
 for char in text:  
 result = result + chr(ord(char) ^ ord(key[ptr]))  
 ptr = ptr + 1  
 if ptr == len(key):  
 ptr = 0  
 return result

When the username and password aren’t being used they are encrypted stored in environmental variables in the windows operating system making them hard to find. The encryption key can be changed in the views file.

My seventh objective was to send email notifications when there is a problem with the server. This is done when the server is fetched from WMI and if the status doesn’t come back as OK the system will send an email containing the status.

My eighth objective was to create a page where users can enter an IP Address and get the live stats for that server. To achieve this I used the domain details as well as WMI to gather the information and send the stats back to the webserver to be displayed.



My final objective was to automatically generate and admin account and password when the app is first run. To achieve this I used the @app.before\_first\_request function in flask and added a try and expect statement to make sure the account doesn’t already exist. A random password is generated using the .join function in python and is then added to the database after being encrypted. The results are also added to a txt file so the user can login to the admin account and register users.

**Improvements**

This system will continue to be worked on and improved over the coming months. Some improvements would be to add a search function on the server’s page so it’s easier to find individual machines quicker. Also, graphs will eventually be added which will show past data and any anomalies in the servers. Finally, I believe that it would be possible to add a function that would allow the servers to be updated by more research is needed in the WIN32api and WIN32com and how it can communicate with WMI. The staff at King & Shaxson said it would be very useful to be able to update servers via a webapp since everything needs to be kept up to date and secure. It also means that it is easier for them to multitask and can spend more time improving infrastructure in the growing company.