**Monitor Lizard:**

**A Simple Monitoring Application**

**Final Project Report**

**Requirements, Design,**

**Implementation/Testing &**

**Installation/Delivery**

**Michael DeHaan LLC.**

**CSC 492 Team 33:**

**Creighton DeVal**

**Soumya Gade**

**Matthew Leonard**

**Orion Qin**

**Rose Zendels**

**North Carolina State University**

**Department of Computer Science**

**May 7, 2021**

## Executive Summary

*Author(s): Soumya Gade & Creighton DeVal*

*Editor(s): Orion Qin, Michael Leonard, & Rose Zendels*

Our team has constructed a simple monitoring platform that will be able to observe and display various metrics including CPU utilization, disk space, free memory, uptime, and statistics on both running processes and containers. This application is capable of monitoring multiple servers across a wide area. The goal of this project is to demonstrate the development of a monitoring system while providing in-depth experience with the components involved.

Our monitoring application consists of five main components. There is a host daemon, which is installed on each monitored device and is responsible for sending reports containing host metrics to the management server. This transition occurs in the message bus, which is a pipeline between this host daemon and the processing daemon. The processing daemon runs on the management server and is responsible for verifying reports and publishing them to the database. Once reports are on the database a user can authenticate on the web application to view host statistics and alerts. The alert daemon manages these alerts and will eventually be capable of sending either SMS or email notifications depending on the user preferences.

During initial iterations our team gathered a more concrete understanding of the requirements and began to solidify our design. We constructed an architectural diagram of the system that details the various technologies that will be used in the application as well as the way in which the components are expected to fit together. The team also designed a database schema and testing plan comprised of several end to end tests for quality assurance. We created several mockups for the web app and improved them based on feedback provided by our sponsor.

Currently, our web application has a login page with basic authentication. Logging in to the system will leave you at the dashboard, which contains information about which hosts you are monitoring and highlights alerts related to these hosts. Navigating to a particular host will allow you to see graphs with date specific report data about each collected metric. While logged in you can use the Django admin tool to set up alert rules which allow you to customize which alerts you are interested in. Our alert daemon is capable of providing in-app notifications of alerts as well as sending out email notifications, but does not yet send SMS notifications.

## Project Description

### Sponsor Background

*Author: Creighton DeVal*

*Editor(s): Soumya Gade*

Michael DeHaan is the founder and former CTO of Ansible, an open source software provisioning, configuration management, and application deployment tool. Ansible sold to Red Hat in 2015 and is currently in use by companies such as Twitter, eBay, Verizon, NASA, and Electronic Arts. DeHaan has also developed storage software for IBM, and video management and systems management solutions for local startups. He recently created SourceOptics, a tool which provides analytics, reports, and graphs on how software engineering teams and projects operate. In addition, DeHaan is a technical consultant for Section 4 of senior design at NCSU.

Sponsor Contact:

Michael DeHaan (Technical Consultant): mpdehaa2@ncsu.edu.

### Problem Statement

*Author: Rose Zendels*

*Editor(s): Soumya Gade*

Professional deployments often include dozens, if not hundreds of different hosts and processes, and those hosts have physical and virtual resources whose usage and status can reflect valuable information about the deployment. For example, if a host’s CPU usage unexpectedly spikes, it could indicate a potential issue with the software running on it. Every host has access to its own metrics, and knowledge about its processes, but it is difficult to keep track of all of the useful data in an organized fashion. Monitoring is about sending those metrics to a separate monitoring server to check if the host is functioning within acceptable levels. We want the server to notify users of any suspected problems found from host statistics and provide them with enough information so they can track down and solve them.

Many companies create their own monitoring software in-house, but there are also many publicly available monitoring tools on the market with complicated designs and features. Rather than build off of those, we developed a simple monitoring application that can meet all the core demands of the required functionality and be built from the ground up to be extensible enough to be worked on by future senior design teams. Our project is primarily a learning platform that can be used as a testing ground for new ideas, concepts, and implementations in a much easier way than the large, complicated proprietary alternatives.

### Project Goals & Benefits

*Authors: Soumya Gade & Matthew Leonard*

*Editor(s): Orion Qin*

The main goal of this project is to construct a monitoring application that replicates interesting features from similar tools on the market today such as Prometheus and Nagios. Admins will be able to configure alert rules such that they will receive an email notification when the threshold for a certain metric for a host is passed. Admins will be able to further examine the anomaly in the web application as well as view live updates of various metrics for each host. They can get a quick overview of all that is going on in the system by studying the different graphs on the web app such as the metric or alerts by tag graphs. Admins can also assign any number of hosts to any number of tags depending upon how best they see fit.

Our simple monitoring application will benefit people on production teams working on professional deployments which are typically rather large. These users will be made aware when hosts are not functioning within acceptable levels with the alerting capabilities our application will provide. Our system will also help with the aforementioned problem of organization in a large deployment environment as admins will be able to group hosts by tag. The application we have built will also benefit future senior design teams as they can use it as an arena for implementing new ideas and concepts related to monitoring. This will be possible because we have focused on building the core functionality which allows others a lot of room to add on features and enhancements in various areas of the application.

### Development Methodology

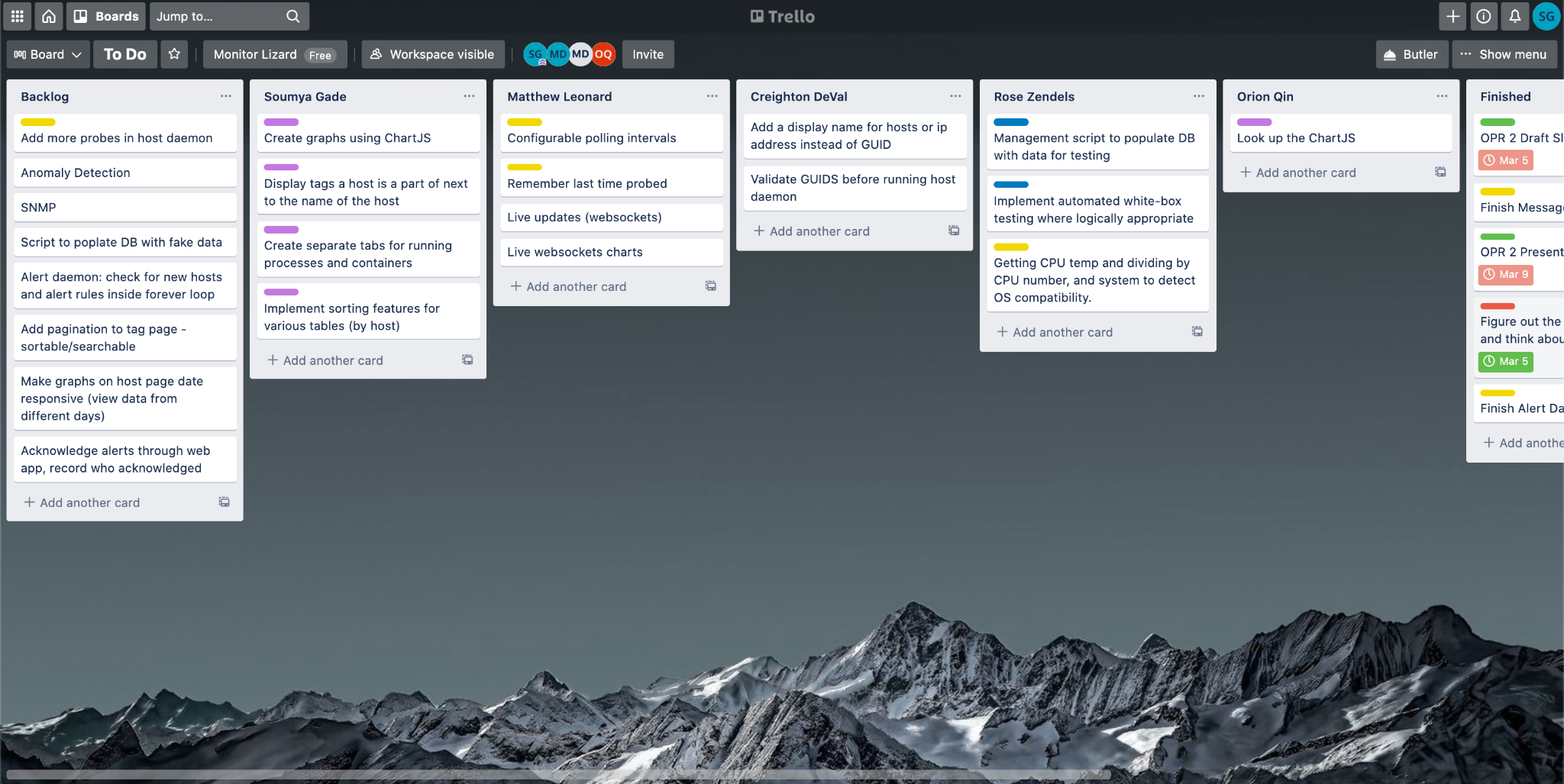
*Author: Soumya Gade*

*Editor(s): Creighton DeVal*

Our project consists of several main components - a host daemon, processing daemon, alert daemon, message bus, and web application. When brainstorming a development approach, we determined that we could initially build these components independent of each other and then work on integrating them together to come up with a functional system. As a result, our team has chosen to use an agile, iterative approach to development and project management. We plan to spend initial iterations developing each of these components before moving on to tackle integrations and various feature enhancements. Thus, successful completion of one iteration should allow for a smooth transition into the next.

Our iteration process is approximately 3-4 weeks long and includes phases such as requirements, development, testing, delivery, and feedback. We define the requirements for iterations based on the project backlog and sponsor feedback. As seen in Figure 1.0, we have a Trello board with a list of tasks that each team member is currently responsible for as well as a list of backlog items that have yet to be assigned. We design and develop software based on the defined requirements, integrate and deliver the working iteration into production after it has gone through substantial testing, and accept sponsor feedback which we work into the requirements of our next iteration. We have formally defined these iterations in the “Implementation” section of this report.

Due to the complexity of our project, we decided to use NC State’s enterprise GitHub as a version control system for branching, pull requests, and code reviews. In our repository, we have created separate branches corresponding to the different components of our project (i.e. web application, email alerts, etc.). Lightweight code reviews are done via screen sharing on Google Meets and more extensively by unit testing and pair programming. Reviewers who have been assigned to a pull request are responsible for making sure that the new code follows coding and documentation best practices as well as taking care to highlight any potential issues with regards to merging the branch to the larger codebase (master).



**[Figure 1.0, Our Trello Board during Iteration 2. This shows tasks we have completed (Finished), tasks that are in progress, and tasks that had not yet been started (Backlog). Each team member has a list with tasks that they have been assigned & expected to update this board as they make progress.]**

We have regular team meetings twice a week that involve tasks such as pair programming, addressing sponsor feedback, and task planning. In addition, we meet with our sponsor, Michael DeHaan, every Thursday afternoon from 12:50-1:50PM. During sponsor meetings we showcase our work from the previous week, and field questions and suggestions from Michael. By meeting with him on a weekly basis, we are able to obtain feedback and criticism frequently. This creates an Agile development environment that is conducive to meeting our sponsor’s needs.

## Challenges & Resolutions

*Author: Orion Qin*

*Editor(s): Creighton DeVal*

We want Monitor Lizard to be able to scale vertically like other monitoring apps, which means that it should be able to process a large, constant inflow of reports. Our app should also be able to scale horizontally across many web servers.

Our app must be extensible. The database structure and architecture will accommodate many different metrics such as CPU usage or RAM usage. In the future, it should be easy for extra functionality to be integrated into the system. A good example of this is the design of our probe system. New host probes can be added to the probe folder which will automatically add them to the list of recorded metrics.

Moreover, our app must present data in a digestible and informative way. The dashboard does this with a pie chart listing all of the host tags and a table containing alerts . Hosts that need your attention (to address an alert) are highlighted. A user can navigate to a page with more detailed information about host metrics. The user can toggle between different graphs, choose how many reports are displayed, and change the date range.

## Resources Needed

*Author: Matthew Leonard*

*Editor(s): Soumya Gade & Rose Zendels*

Each of our team members will need a development environment, test deployment environment, and testing environment. This can be accomplished by a single computer with each of the architecture components installed on it, and certain services mocked. The applications needed include a supervising application to run the monitoring and worker daemons, message bus, web server, mock classes of the external alert services, and PostgreSQL.

### External Dependencies, Interfaces and Libraries

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Version** | **Purpose** | **Status** | **License** |
| Python3  (with pip) | 3.9.1 | Programming language for our backend. | Obtained | PSF License Agreement |
| Django | 3.1.6 | Python-based web framework used for creating models for the database & templates for the web application. Constraint by our sponsor. | Obtained | BSD |
| Bootstrap | 4.6.0 | Framework used for styling on the web application. | Obtained | MIT License |
| PostgreSQL | 12.0 | Database to store data from the Django models. Constraint by our sponsor. | Obtained | PostgreSQL |
| ChartJS | 2.9.4 | JS framework used for drawing the pie & line charts on the dashboards. | Obtained | MIT License |
| AlpineJS | 2.8.0 | JS framework used for toggling the metric graphs on the host pages. | Obtained | MIT License |
| RabbitMQ | 3.8.12 | Message bus along which metrics will be sent as reports. | Obtained | Mozilla Public License |
| psutil | 5.80 | Library that measures metrics such as disk & CPU usage within Python. | Obtained | BSD |

## Requirements

### Glossary

*Author: Creighton DeVal*

*Editor(s): Soumya Gade*

|  |  |
| --- | --- |
| **Host/Monitoring Daemon** | Process run by host machines that is responsible for collecting data about that host and sending reports containing that data to the central server. |
| **Processing Daemon** | Process run by the management server that is responsible for filtering report data and publishing to the database. |
| **Message Bus** | Tunnel between host & processing daemons. |
| **Alert Daemon** | Responsible for creating & managing alerts. |
| **Metric** | Something that can be measured on a host. This can be anything from the percentage of disk used, to the exact byte count of disk used, to a list of disks in the system. |
| **Host** | Computer that has registered via a host\_daemon. Identified by their guid as any other value could be subject to change. |
| **Host Tag** | A collection of hosts. Hosts can be in multiple tags. The two primary purposes of tags are to make it easier to find hosts, and to allow users to assign the same alert rule to many hosts in a way that makes sense. |
| **Report** | A report is a measurement of one metric, on one host, at a point in time. The report would also contain the value of the metric. |
| **User** | A user that can receive alerts. |
| **Team** | Collection of users that are all subscribed to the same alerts and will be notified when an alert rule has been triggered. |
| **Alert Rule** | A rule that designates when alerts should be thrown. The operator and threshold together determine when a host triggers the alert rule. For example, the operator “min”, the threshold “80”, and the metric “cpu\_usage” means the alert will be triggered when any host in the host\_tag reports a cpu\_usage below a value of 80. |
| **Alert** | Notification sent to users when an Alert Rule has been triggered. The alert contains all of the information necessary for a user to investigate and diagnose the reason for the alert. The user can add notes to the alert, and acknowledge the alert. An acknowledgement means the alert has been investigated. |

### Overall View

*Author: Matthew Leonard*

*Editor(s): Creighton DeVal*

The system runs on two different kinds of machines, Hosts and Management Servers. Hosts run the Monitoring Daemon, and the Management Servers run instances of the Web Application, Processing Daemon, and Alert Daemon. Hosts must register with the Management Server to begin sending reports. Hosts send their Globally Unique Identifier (GUID) with Metrics in JSON over the Message Bus to the management server as a Report. The Management Server’s Processing Daemon listens on the other end of the Message Bus to verify and persist the Reports to PostgreSQL.

Users must authenticate to begin using the web application. Once they do so, they may modify Tags and create or modify Alert Rules. An Alert Rule can be placed on a Tag to notify users when any Host in that Tag has passed a certain threshold. Users can place themselves in a team and add that team to the Alert Rule. Once the Alert Daemon notices an Alert Rule has been tripped, it will notify any users on the team by the users notification settings.

### Functional Requirements

*Author: Matthew Leonard*

*Editor(s): Creighton DeVal*

1. Web app must display host metrics and associations
   1. The user must be able to view a list of all the hosts in the system, and what tag they are in.
   2. The user must be about to view graphs of statistics about hosts over time.
2. Connectivity
   1. Each host shall send reports to the management server via the message bus.
   2. The management server shall have a daemon that processes reports from the hosts and saves them into the database.
3. Registration
   1. New hosts must have a unique GUID
4. Alerts
   1. Users can customize which alerts they receive by selecting a particular host tag.
   2. Teams can subscribe to alert rules from a particular host tag
   3. Users can choose whether to be notified about alerts via either SMS or email.
   4. Users can acknowledge alerts
   5. Users can add comments to alerts.
5. Tags
   1. The user must be able to change host grouping by modifying their host tag
6. Authentication
   1. Users must authenticate before viewing or changing data.

### Constraints

*Author: Matthew Leonard*

*Editor(s): Creighton DeVal*

1. All applications must use Python and the Django framework.
2. The models must be stored in a PostgreSQL database.
3. The web application must be able to run on multiple servers simultaneously.
4. The front end application should be implemented without the use of heavyweight frameworks such as Angular or React. Instead, it will use ChartJS and AlpineJS.

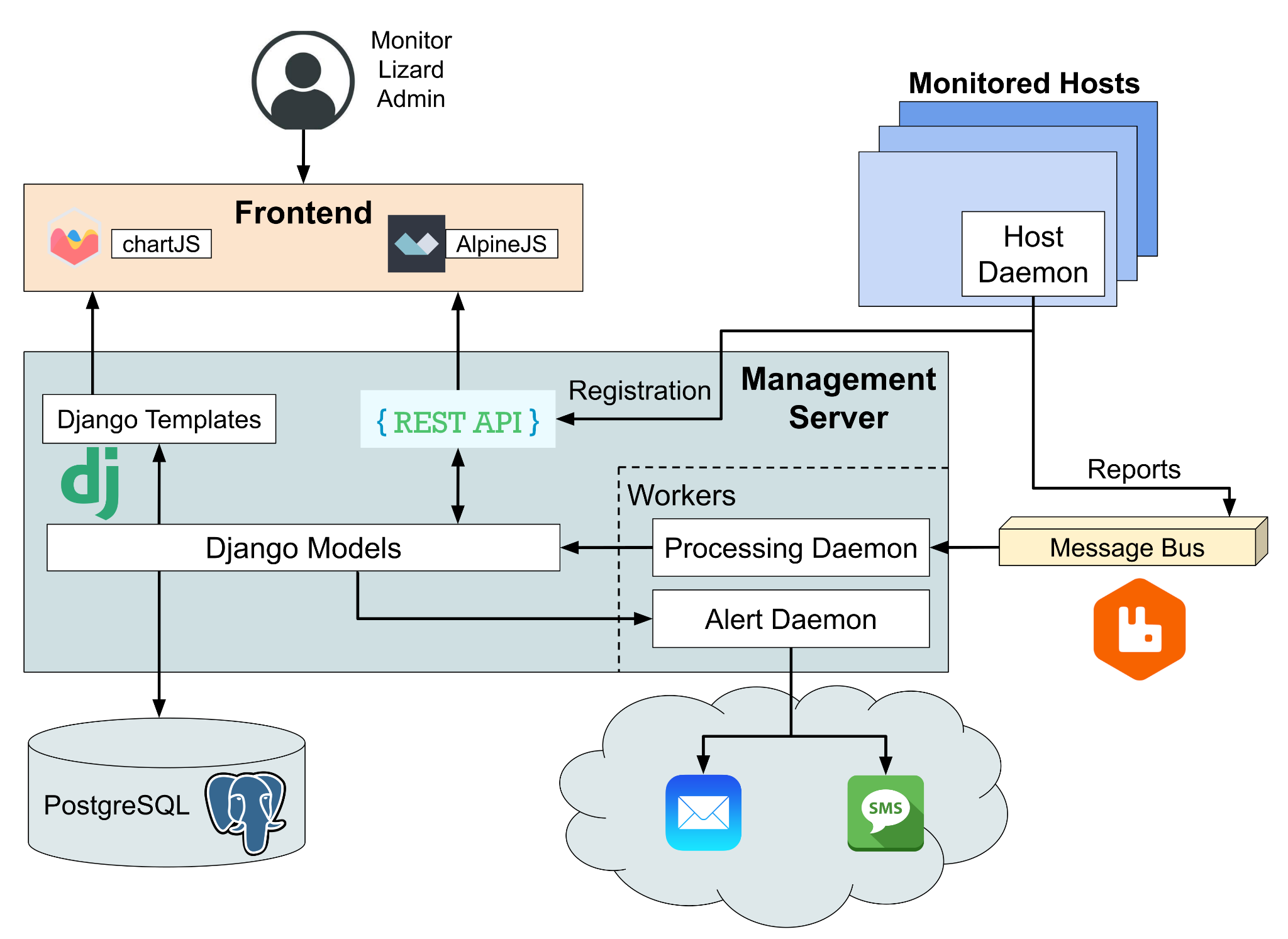
## Design

### High-Level Design

*Author: Creighton DeVal*

*Editor(s): Soumya Gade*

Our monitoring application consists of five main components. There is a host daemon, which is installed on each monitored device and is responsible for sending reports containing host metrics to the management server. This transition occurs in the message bus, which is a pipeline between this host daemon and the processing daemon. The processing daemon runs on the management server and is responsible for verifying reports and publishing them to the database. Once reports are on the database a user can authenticate on the web application to view host statistics and alerts. The alert daemon manages these alerts and is capable of sending either SMS or email notifications depending on the user preferences. Figure 1.1 demonstrates the way in which all of these components are connected.



**[Figure 1.1, High-Level System Architecture diagram for our project which shows all of the technologies being used and the way in which the components are connected.]**

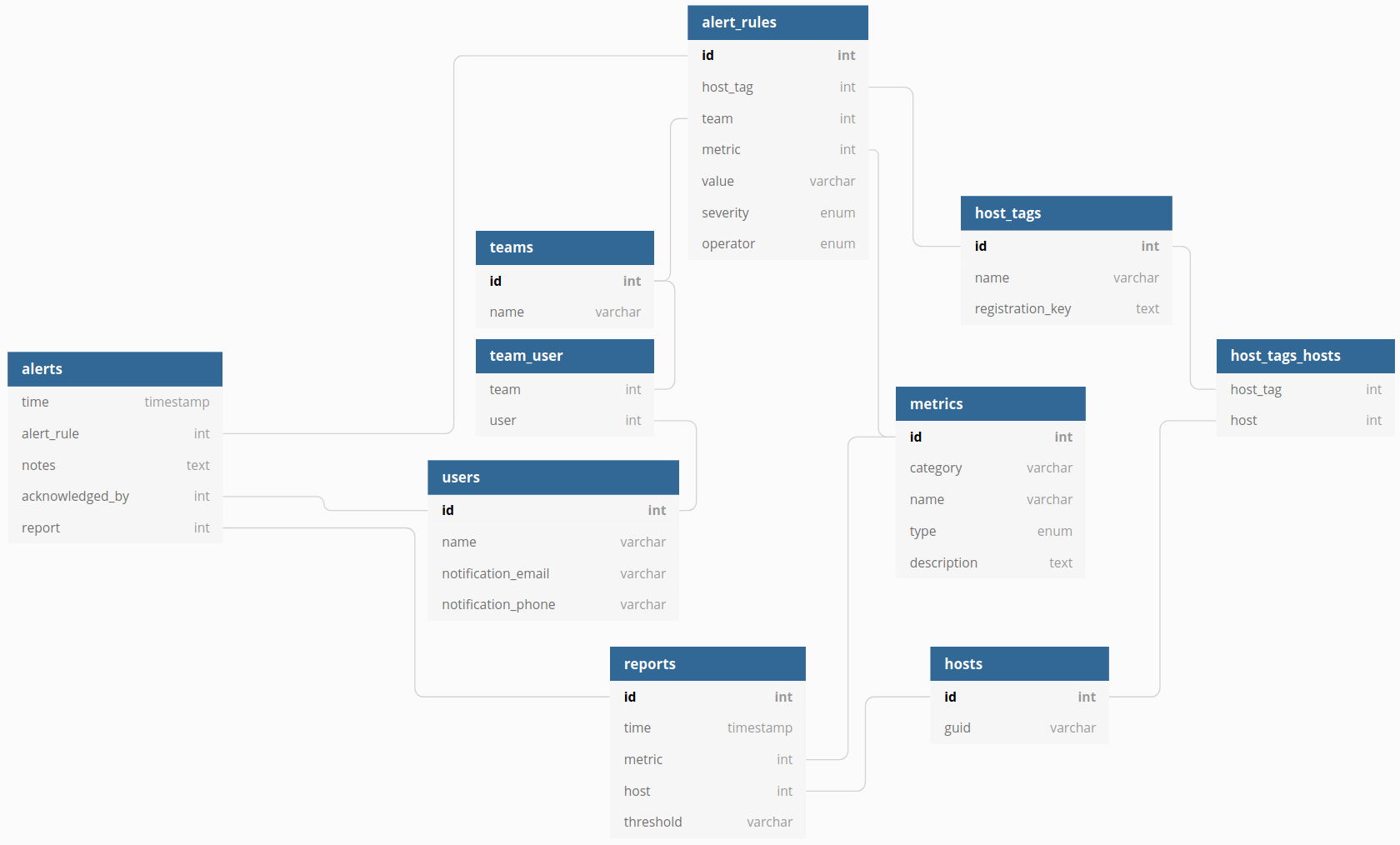
### Low-Level Design

*Author: Matthew Leonard*

*Editor(s): Creighton DeVal*

#### Database Design

The database is designed around being able to monitor Hosts with Alert Rules, so we will explain the database starting with the alert\_rules table as can be seen at the top of Figure 2.1. The Alert Rule model stored in the alert\_rules table represents a rule for the Alert Daemon on when to send Alerts and notifications. Each Alert Rule watches all the Hosts in a Host Tag, which is a collection of Hosts. Each Alert Rule will Alert and notify all the users in a Team, which is a collection of Users. Each Alert Rule watches one Metric on each Host, which can be something like cpu temperature, ram usage, or disk usage, and sorted into categories, and given types for display and descriptions. The Alert Rule will watch the Metric on every Host in the Host Tag until the system receives a Report from one of the Hosts with a value that trips the Alert Rules threshold and operator (ex. 90% cpu usage would trip an 80% maximum, 10% free disk space would trip a 15% minimum). Once an Alert Rule is tripped, it will send a notification to every User in the Team watching it, and save an Alert to the alerts table, which points back to information about where and how the alert was tripped. The Alert also can be acknowledged\_by a user to “dismiss” the alert from the UI, and users can write notes on the alert.



**[Figure 2.1, Database Design]**

#### Host Daemon

The Host Daemon starts by ensuring the Host is registered with the server. First it checks if a GUID is saved on the Host, and if it is, checks if the GUID is valid, then validates the GUID by checking it with the REST API. If the GUID is absent, or invalid, it attempts to register the Host with a registration key. If the Host is not able to register, the daemon exits. If the Host is successfully registered, it saves the GUID to a file and continues. The Host Daemon then scans the host\_plugins directory for any probe plugins. Every probe plugin provides a list of probes, with their metric information, measure functions, and polling frequency. Once it has started, the Host Daemon regularly checks the polling frequencies and the last time it sent each metric to see if it should measure and send the metric as a report. The Host Daemon runs until it encounters an error or is interrupted.

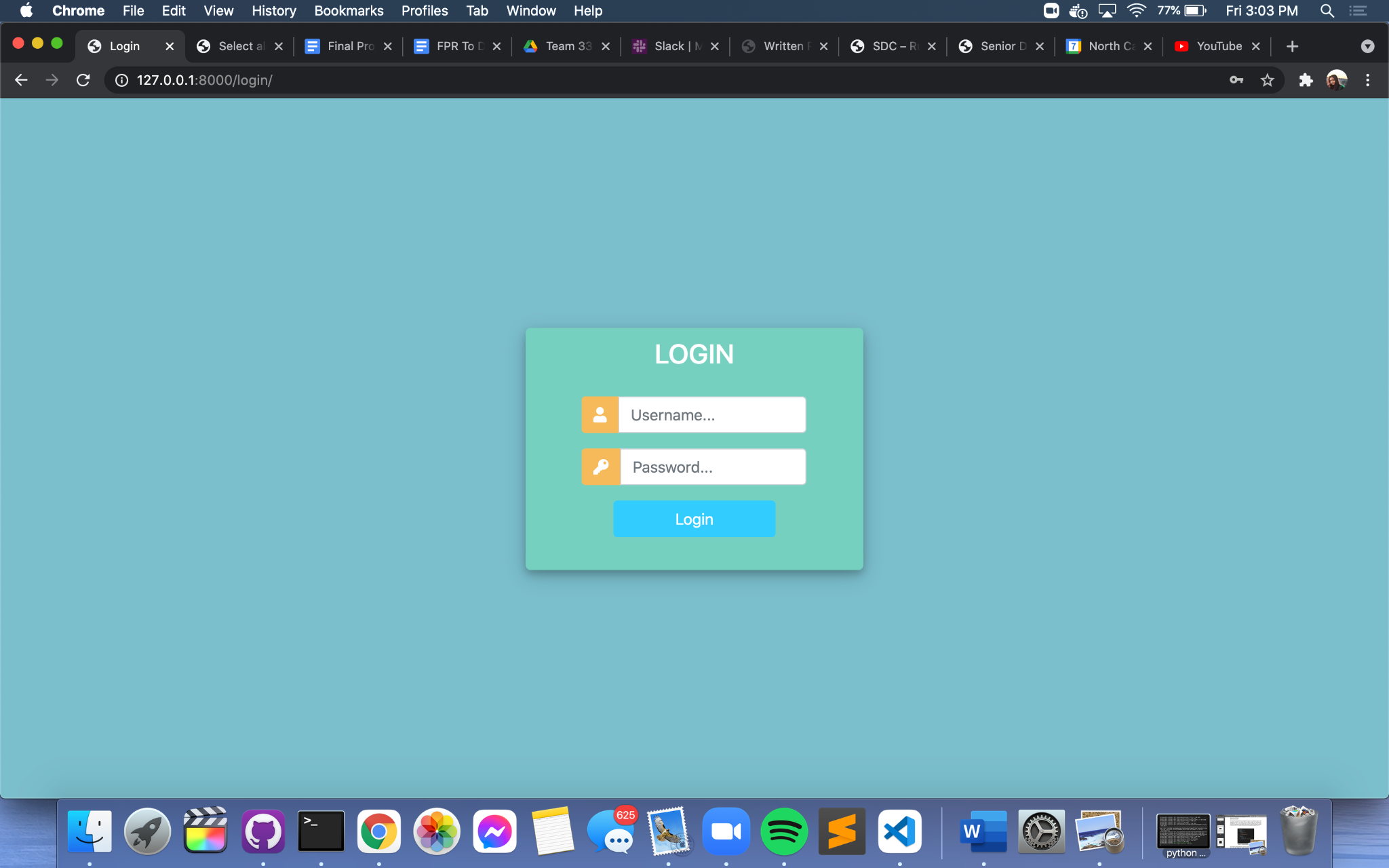
### GUI Design

*Author: Soumya Gade*

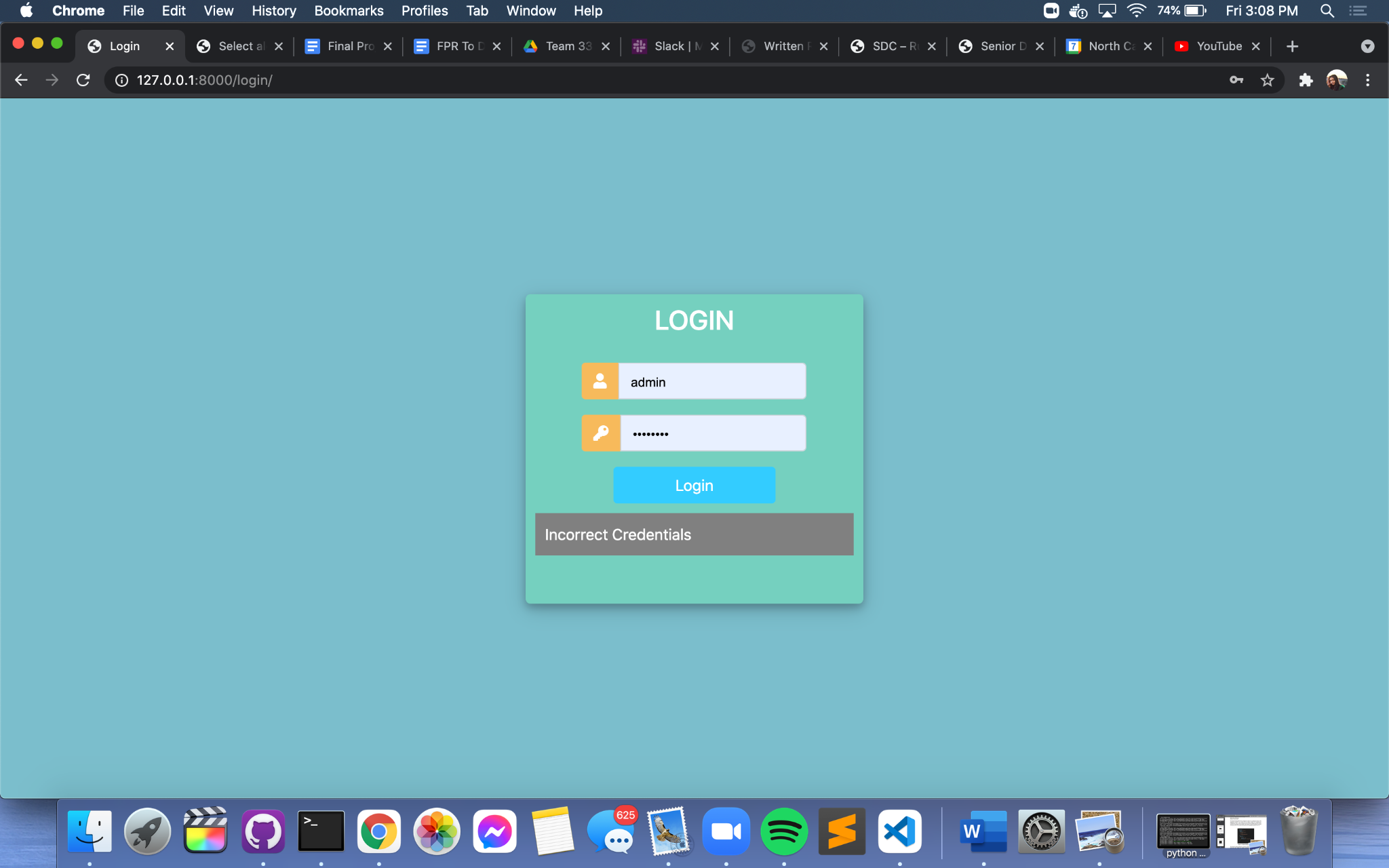
*Editor(s): Orion Qin*

We’ve designed our web application using a combination of Django, Bootstrap, ChartJS, and AlpineJS. Django has allowed us to avoid having to use dependency heavy frameworks which has allowed for the construction of a lightweight application. We used Bootstrap for the styling and coloring of the web pages and ChartJS to construct the various pie charts and line graphs on the dashboard and host pages. AlpineJS was used to create the functionality for being able to toggle between metric graphs on the host page.

As can be seen in Figure 4.1, we have a login page where admins will need to enter their username and password and be authenticated in order to view the rest of the application. This is also the page that admins will be taken to upon logging out of the application. If an admin enters an incorrect username and/or password, an error message reading “Incorrect Credentials” is displayed as can be seen in Figure 4.2. The user is unable to login to the application and will need to try again with inputting the correct credentials.



**[Figure 4.1, Login Page that is displayed when the application is first run and also the page that the admin is directed to when he/she logs out.]**

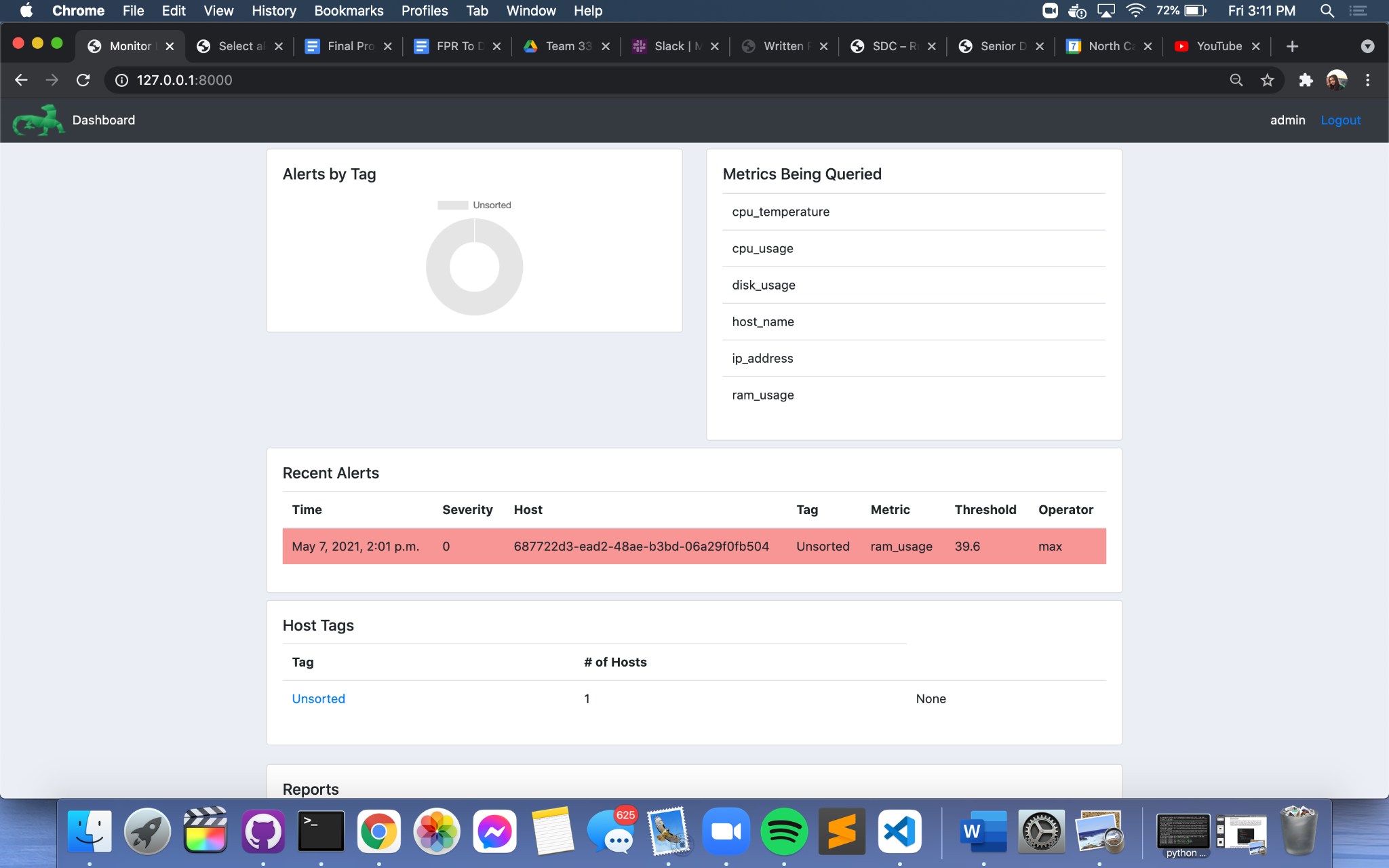


**[Figure 4.2, Login Page that shows error message when incorrect username and/or password is entered.]**

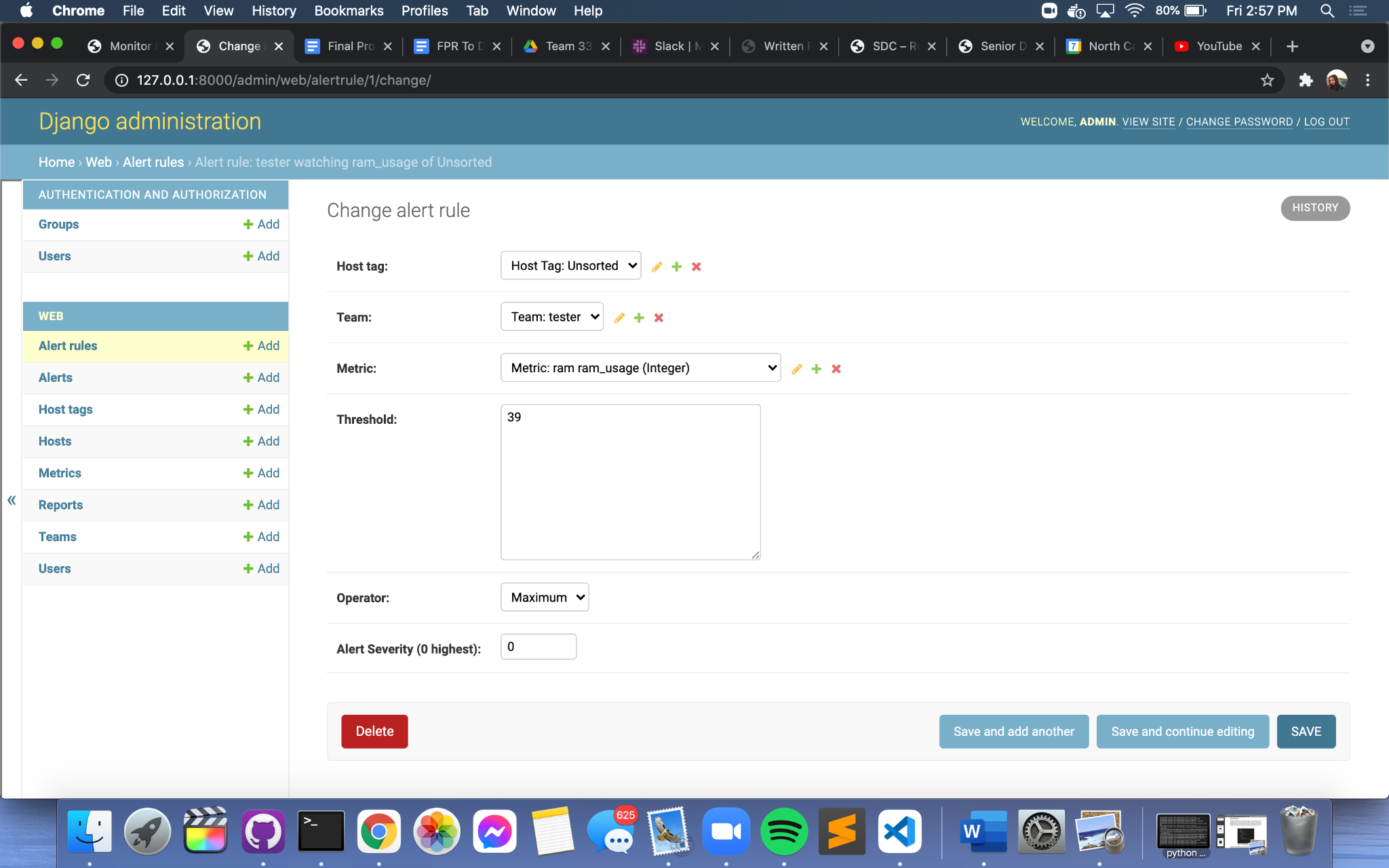
Upon a successful login, the admin is taken to the view seen in Figure 4.3, or the Dashboard page. This page is intended as a way for the admin to view a summary of all that is currently going on in the system. There is a chart in the upper left hand corner that shows the number of alerts each tag has associated with it. For example, in Figure 4.3 it can be seen that the Unsorted tag currently has one alert tied to it. The table to the right shows the various metrics that are currently being queried and returned as part of the reports being sent in. The Recent Alerts table shows all of the alerts that have been generated and not yet acknowledged by an admin. If the severity is zero, or the most concerning, the table row is highlighted in red to bring attention to the severity level. In Figure 4.3 it can be seen that the alert was generated because the ram usage went above the maximum threshold of 39.6. The guid of the host and the tag that it is a part of are also shown as part of the alert information.

In order to create an alert rule, the admin will need to go to the Django Admin page as seen in Figure 4.3.1. This page was not developed by our team, but is rather an in-built feature of Django. In Figure 4.3.1 we are creating the alert rule that triggered the alert shown in the Recent Alerts table of Figure 4.3. An alert rule requires a host tag, team, metric, threshold value, operator, and alert severity level to be assigned to it.

On the Dashboard Page in Figure 4.3, an admin is also able to view the Host Tags table, which shows the various tags which are currently in the system as well as the number of hosts that are currently assigned to that tag. When a tag in this table is clicked, the user will be taken to a page for that tag such as the one shown in Figure 4.5.

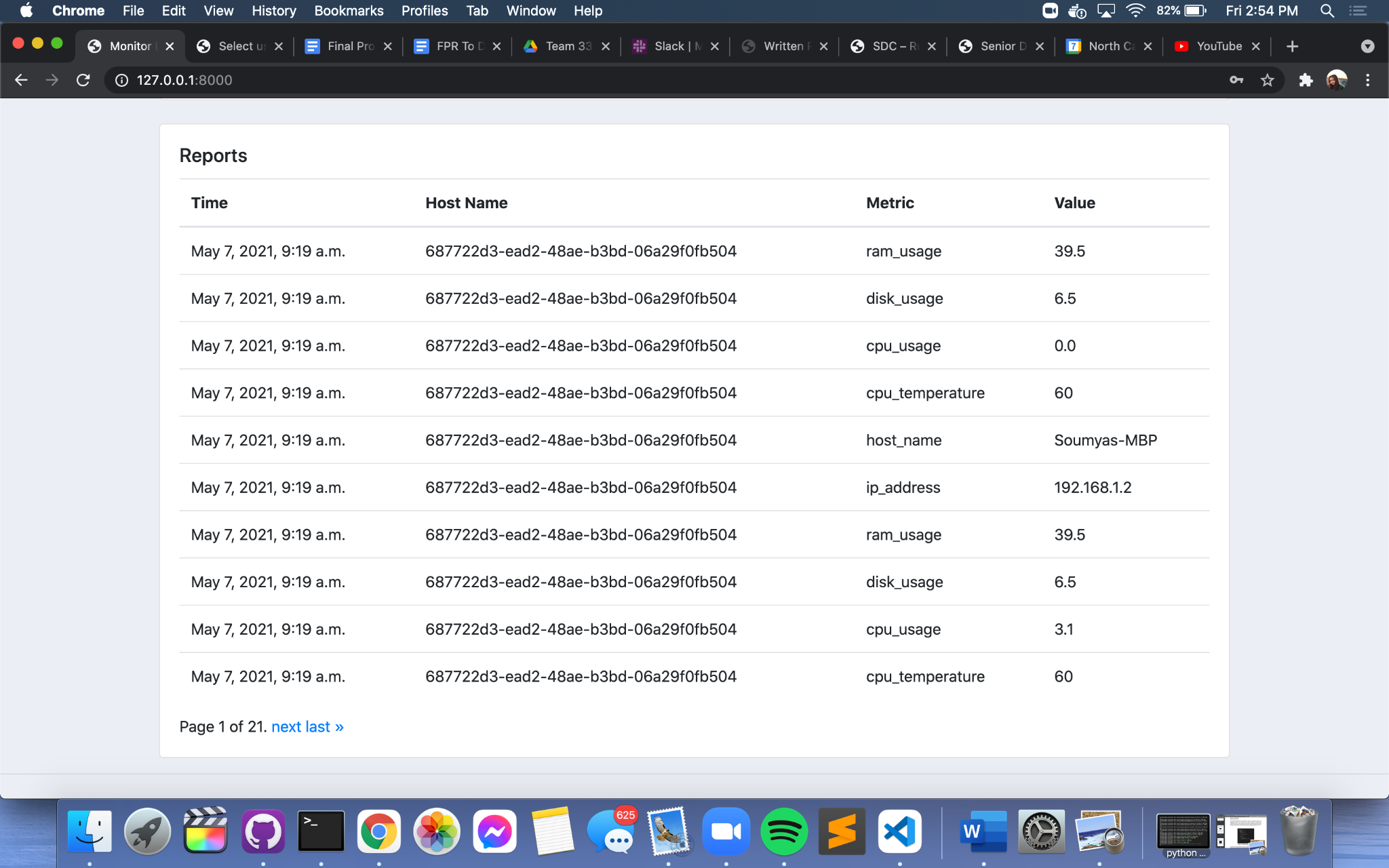


**[Figure 4.3, Dashboard Page that the admin lands on upon logging in. Currently one alert in the Unsorted tag that has been tripped because the ram usage of the host went above the value of 39.6. It can also be seen that there is currently one tag in the system (Unsorted) with one host attached to it.]**

****

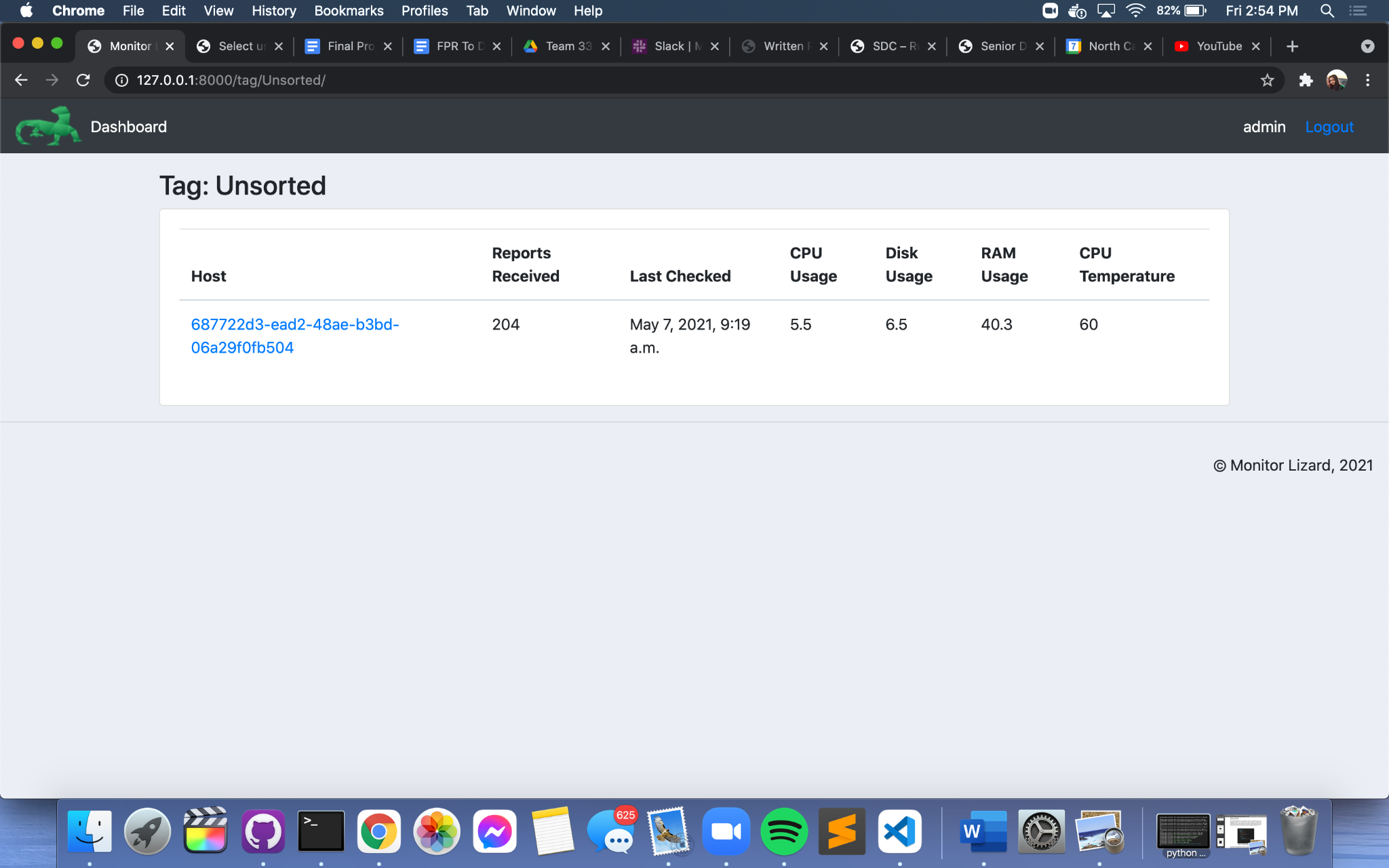
**[Figure 4.3.1, Demonstrates the creation of an alert rule in Django Admin. In order to create an alert rule, the admin must assign a host tag, team, metric, threshold value, operator, and alert severity. After this rule has been saved (and if the alert daemon is running), an alert will be triggered if this threshold is crossed.]**

As seen in Figure 4.4, the lower half of the Dashboard page has a table that shows all of the reports that have been generated in the system thus far. The admin can move between pages in the table to view various reports. The host guid, metric, and value that was received in each report are being displayed in this table view.



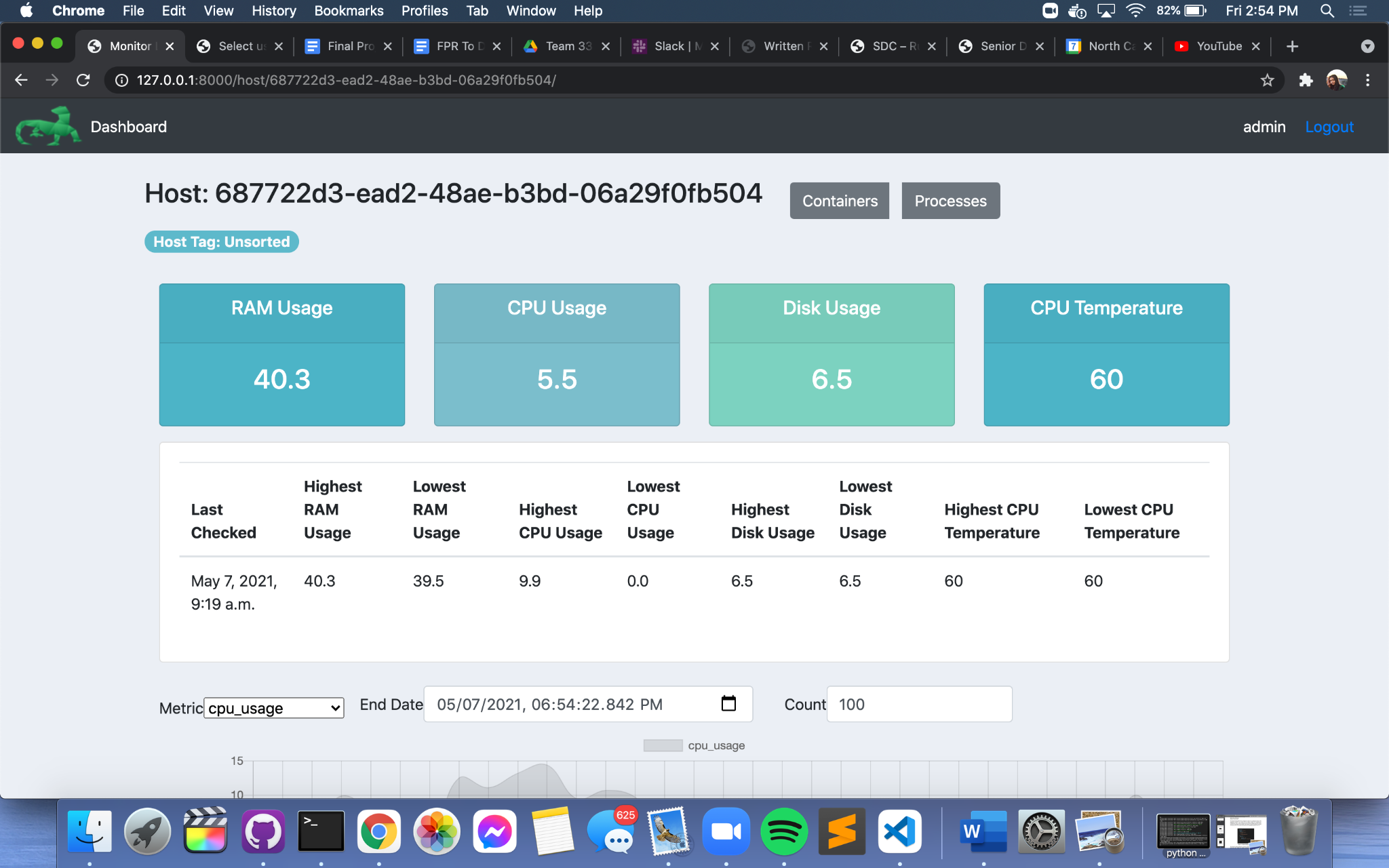
**[Figure 4.4, Lower half of the Dashboard Page that shows all of the reports that are currently in the system. Admin can move back and forth between the various pages in this table.]**

As seen in Figure 4.5, the page for a Tag has a table with a list of all the hosts currently associated with the tag. In this specific example, the Unsorted Tag has only one host as part of it. The admin is able to view the number of reports received for this host (by guid) in addition to the last received value of each of the four metrics - CPU usage, disk usage, RAM usage, and CPU temperature. When the guid of a host in this table is clicked, the admin is taken to a page for that host such as the one seen in Figure 4.6.



**[Figure 4.5, Tag Page for the Unsorted tag (this page is displayed when the name of a tag is clicked from the Host Tags table on the Dashboard page). Admin is able to view all of the hosts which are a part of this tag.]**

The host page displays the guid of the host at the top of the view as the main identifier. There are also tabs to show the running processes and containers, but these have not yet been implemented at the time of writing and only show hard coded information in the tables. There is also a list of all the tags the host is currently a part of at the top of the page. In Figure 4.6 it can be seen that this particular host is only a part of the Unsorted tag. The four large colored boxes show the last recorded values of each of the four metrics. The table below these boxes shows the highest and lowest recorded value of each of the metrics received thus far from the reports that have been coming into the system. These values are dynamic and change each time the page is refreshed and more reports have been received.

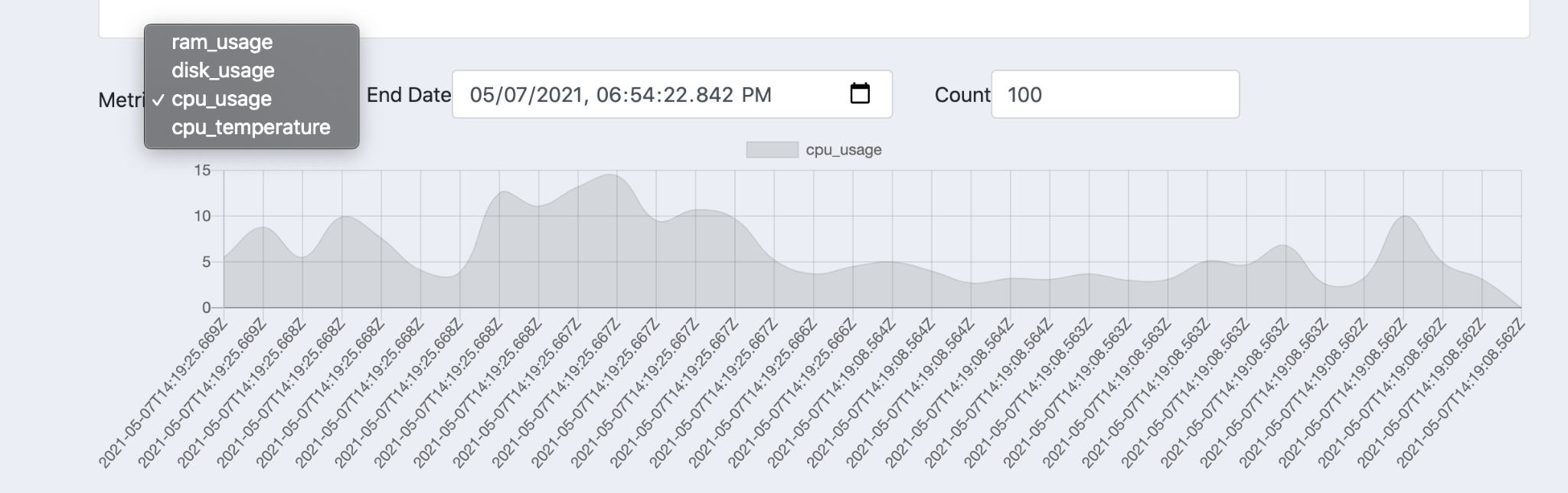
****

**[Figure 4.6, Host Page that is displayed when a host is clicked from the table on a Tag page. Hosts are referenced by their guid & the admin can view various statistics on the metrics being queried.]**

The bottom half of the page for a host has graphs of the various metrics as can be seen in Figure 4.7. These graphs show the value of the metric as it changes over time based on the new reports that are coming in. Each of these line graphs was made using ChartJS while the functionality for toggling between graphs was made possible with AlpineJS. Admins can also change the number of reports that are being shown on the graphs or focus on a specific period of time by changing the end date of the reports being displayed. Figure 4.8 illustrates the toggling functionality as the admin can elect to either view the graph for CPU usage, disk usage, RAM usage, or CPU temperature. Although we initially had each of these graphs displayed at one time on the host page, we believed the toggle functionality would make items more visually appealing for users due to the page becoming significantly less cluttered in nature.

****

**[Figure 4.7, Lower half of the Host Page with the metric graphs. Currently viewing the CPU usage graph with a total of 100 reports being displayed on the graph.]**

****

**[Figure 4.8, Shows how the graphs can be toggled to view the graph for a different metric. This functionality was made using AlpineJS. Can also change the number of reports on the graph and the end date.]**

## Implementation

*Author(s): Soumya Gade & Orion Qin*

*Editor(s): Matthew Leonard*

### Iteration Definitions

**Pre-Phase: January 25th-February 16th**

The pre-phase describes work our team needed to complete prior to implementing the functional requirements of this project. Tasks for this phase include creating an initial system architecture design and gathering more information on the requirements from our sponsor. As we designed our system architecture and created wireframes in Figma, we realized that we would need to use technologies with which our team was not familiar such as Django and ChartJS. Thus, this phase also consisted of researching these technologies through the creation of a simple “Hello World” project that allowed our team to become more familiar with these tools.

**Iteration 1: February 16th-March 11th**

In Iteration 1, our team was able to design and to implement the database for the system. Each member of the team experimented on the technologies that they are going to use, including the web application, database creation, plugin system and email-sending system. Later, we were able to build a skeleton webapp that uses Django authentication [FR 6.1] and displayed the layout of the website based on the wireframes that we had created in the pre-phase. At the end of this iteration, we were able to implement the message bus and read in system metrics [FR 2.1 & 2.2] and insert them into the database only with a valid GUID [FR 3.1] that registered through the host daemon.

**Iteration 2: March 11th-March 30th**

In Iteration 2, our team spent most of the time on integrating the whole system together. We connected the web application and the database. Our team was able to display live data on the website and update to receive the newest reports upon refreshing the webpage. The web application is now able to list all of the hosts in the system as well as the tag(s) that they are a part of [FR 1.1]. We also figured out how to use ChartJS and AlpineJS to display graphs of statistics [FR 1.2]. Some of the web page layout was also adapted to change based on the feedback from our sponsor. The backend functionality at this stage works completely and the hosts are able to register with the server and store themselves in the database. We also started to implement the alert daemon at the end of this iteration.

**Iteration 3: March 30th-April 13th**

In Iteration 3, our team finished integrating the web application to the database and adding visualizations and graphs using ChartJS. We used real time data that will update upon refreshing the page. We made the tables and graphs customizable depending on the range of the reports pulled from the user. The alert daemon was fully integrated into the system as well with the two most useful operators [FR 4.1-4.5]. The team also has started implementing the testing focused on the UI and API.

**Post-Phase: April 13th-April 30th**

All of the required functionality that our sponsor requested was completed during Iterations 1, 2, and 3. In the final two weeks of the project, or the “post phase”, our team integrated the email notification system and added split settings for the application. We also spent a good bit of time on designing the poster and practiced our presentation for the Posters & Pies Day. Before the presentation day, we managed to improve the testing coverage for the application. We gave demos of the finished product to our sponsor, and allowed him to make suggestions and provide feedback about the application as a whole.

### Current Status

We are currently finished with the pre-phase and Iterations 1 through 3. Work is ongoing in the post-phase at the time of writing. Currently, we have finished all the key requirements of this project and are working on fixing small bugs and creating extensive documentation to make this project easier for future teams to understand and expand upon.

On the web application, the admin is able to view all of the reports that are coming in from the message bus and daemons. The admin is also able to view the values of the metrics in these reports, the guid of the host, and the tag that the host is a part of. The admin can also create alert rules using Django Admin which, when tripped, will send an email notification with details of the alert and the value at which it was triggered.

### Security Considerations

Confidentiality is maintained as users are required to be logged in in order to view the pages of the web application with all of the reports that are coming in and information about the hosts. We used the Django authentication features for ensuring that users are being authenticated. All users must use valid credentials to login to the application and to monitor the reports that are being generated at the host. To maintain integrity, admins will only be able to interact with the user interface of the web application, and will not have access to the code itself.

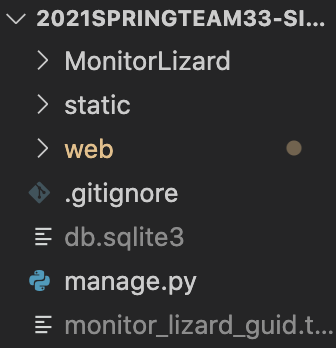
Hosts save the GUID in the hosts file system, and are responsible for protecting it. The GUID is used to identify the host that sent the report, hence, anyone with access to the GUID could impersonate a host. Because of the integration of the email system, we also used split settings to protect personal email credentials from being published on Github. Availability is achieved as the admin is able to view all of the reports that are being collected as well as the information in those reports. Privacy is achieved as admins are able to view and modify the settings on which alert rules are tripped through Django Admin. They are also able to add users and teams by utilizing the Django Admin feature. Thus, the admin is able to understand which metrics are being tracked and which threshold levels are being applied.

### Project Folder Structure

As seen in Figure 5.0, the project is structured into three main directories - “MonitorLizard”, “static” and “web”. “MonitorLizard” is the directory containing project level configuration files, most of which were provided by Django upon creation of the project. For example, in this directory there is the urls.py file that can be configured to route URLs to views.

In the “static” directory, there are two subfolders, “css” and “images”. CSS has some styling used for the logo displayed on the web application, while images has the lizard image itself. The “web” directory is the app that our project essentially is building the web application off of. This directory has the models.py file upon which migrations are run and database tables are being built. There are also various subdirectories including “controllers”, “host\_plugins”, “management”, “migrations”, and “templates/web”.

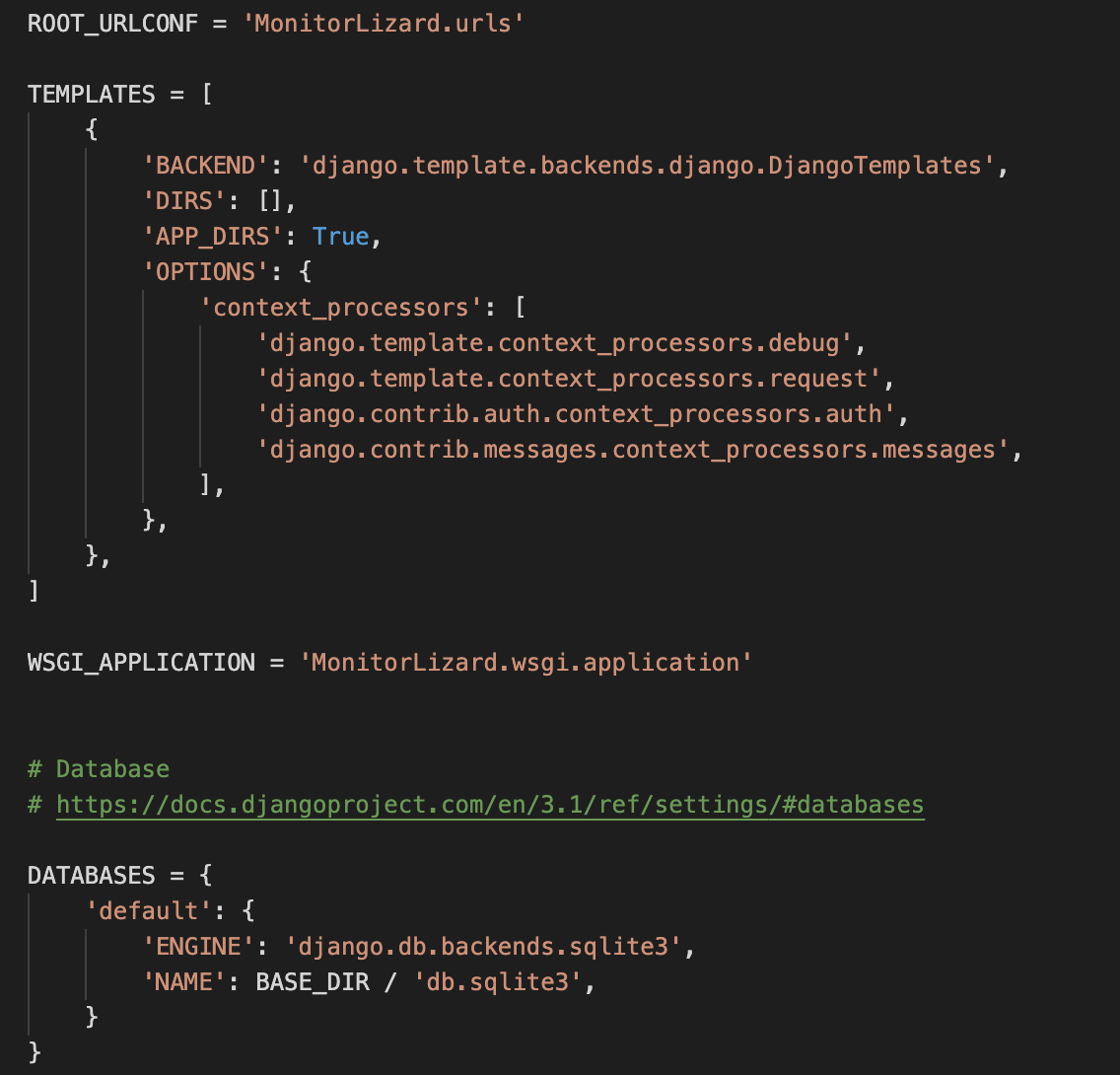
The “templates/web” subdirectory has all of the HTML files for the various pages of the web app. The “management” subdirectory includes the python scripts for the alert, host, and processing daemons. The scripts for the various metric plugins (CPU, disk, & RAM) are included in the “host\_plugins” subdirectory.



**[Figure 5.0, Project root level directory structure.]**

### Project Configuration and Settings

Most of the default project configuration provided by Django was kept as can be seen in the settings.py file in Figure 5.1. By default, the configuration uses the SQLite database. SQLite is included in Python, so there was no further installation necessary in order to support the database. The INSTALLED\_APPS module in settings.py contains various apps by default which we used in our project. For example, it comes with django.contrib.admin which is the admin site through which we were able to view all of the information in the database. It also comes with django.contrib.auth which is the authentication system that came in handy for our login and logout functionality. However, we did add ‘web.apps.WebConfig’ to INSTALLED\_APPS so that the code we wrote for the web application will be considered by Django.



**[Figure 5.1, Project settings.py default configuration]**

## Test Plan & Results

*Author: Rose Zendels*

*Editor(s): Matthew Leonard*

Django has built in libraries for creating robust test suites that we have used for some unit testing of the frontend, and we have a suite of manual tests to be performed for frontend components and system integration. The majority of our project has revolved around establishing and linking together multiple simple programs with minimal logic within each component, so frontend testing is minimal, but as of writing this we have 78% test coverage on the web application. At our current stage of development, these tests revolve around checking responses and statuses from web endpoints and the host api, however in the future they can be expanded to other areas of the program. We found that due to the simple logic within components, there was minimal need for comprehensive white box unit tests, and so our testing strategy will primarily focus on ensuring proper functionality within ideal conditions through end to end testing.

This is the current reported coverage when running our automated test suite:

Name Stmts Miss Cover

---------------------------------------------------------------

MonitorLizard\\_\_init\_\_.py 0 0 100%

MonitorLizard\settings.py 20 0 100%

MonitorLizard\urls.py 3 0 100%

manage.py 12 2 83%

web\\_\_init\_\_.py 0 0 100%

web\admin.py 10 0 100%

web\apps.py 4 0 100%

web\controllers\graph.py 23 15 35%

web\controllers\host\_endpoints.py 20 5 75%

web\controllers\views.py 141 51 64%

web\migrations\0001\_initial.py 6 0 100%

web\migrations\0002\_auto\_20210302\_1333.py 4 0 100%

web\migrations\\_\_init\_\_.py 0 0 100%

web\models.py 89 10 89%

web\tests.py 46 0 100%

web\urls\api.py 3 0 100%

web\urls\ui.py 3 0 100%

---------------------------------------------------------------

TOTAL 384 83 78%

Here are the remaining manual black box tests we have performed on our software:

### UI Testing on currently implemented frontend components

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| testUnauthorizedLogout | Prerequisite: User is logged out.  1. User attempts to access /logout/ page. | User is redirected to /login/ | Console reports 302 response, redirected back to the /login/ page |
| testUnauthorizedTagAccess | Prerequisite: User is logged out.  1. User attempts to access /tag/ page. | User is redirected to /login/ | Console reports 302 response, redirected to /login/?next=/tag/ |
| testUnauthorizedHostAccess | Prerequisite: User is logged out.  1. User attempts to access /host/ page. | User is redirected to /login/ | Console reports 302 response, redirected to /login/?next=/host/ |
| testIncorrectCredentials | Prerequisite: User is logged out.  1. User navigates to /login/ page.  2. User inputs credentials that do not reflect any administrative user. | User is notified that the credentials are invalid | Console reports 200 response but not redirected, gray box appears below login button with “Incorrect Credentials” |
| testValidLogin | Prerequisite: User is logged out.  1. User navigates to /login/ page  2. User inputs valid credentials. | User is given access to the Monitor Lizard dashboard | Console reports a 302 response, then a 200 response, redirected to the home page of the dashboard |
| testLoginLoggedIn | Prerequisite: User is logged in.  1. User attempts to access /login/ page. | User is not brought to login page, as they are already logged in | Console reports a 302 response,then a 200 response, redirected to the home page of the dashboard |
| testTagAccess | Prerequisite: User is logged in.  1. User navigates to home page  2. User selects a host tag from list. | User is brought to page for that tag, listing available hosts within it. | User is correctly brought to the page corresponding to that host tag, which lists all available hosts. Data seems to be incorrect for recent reports from each host, however. |
| testHostAccess | Prerequisite: User is logged in.  1. User navigates to host tag page  2. User selects a host from list. | User is brought to the page for that host and given metric data about that host. | User is correctly brought to the page corresponding to that  host. There is information displayed about the host, however the maximum and minimum values  seem to not be reporting correctly. |
| testContainersAccess | Prerequisite: User is logged in.  1. User navigates to a host page  2. User selects the “Containers” button. | User is brought to page with information about all monitored containers on that host. | Correctly redirected to containers page for the host and shown placeholder data. |
| testProcessesAccess | Prerequisite: User is logged in.  1. User navigates to a host page  2. User selects the “Processes” button. | User is brought to page with information about all monitored processes on that host. | Correctly redirected to processes page for the host and shown placeholder data. |
| testLogout | Prerequisite: User is logged in.  1. User is logged in and selects the “Logout” button on the page. | User is logged out of the service. | Console reports a 302 response, then a 200 response, redirected to the login page with no more access to other pages. |

It can be seen through these results that our frontend UI is linked together. The majority of pages show live data, however a few pages hold static information (tabs for running processes and containers) or have minor bugs in the information displayed.

### End to End Tests

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| testDB | 1. User creates and saves a new host tag through the web application. | A new row is added to the host\_tags table. | This test can only be performed through Django Admin, rather than the Monitor Lizard application. A new row is properly added to the host tag table, but it links to an error page if it does not have any hosts within it. |
| testRegister | Prerequisite: A host tag and registration key have been created.  1. A host registers with the registration key of a host tag through the web application. | A new host is created. That host is associated with the host tag. The server responds with the new hosts guid. | The new host is properly created with a brand new guid, and the host creates a guid file. It is placed in the Unsorted host tag. If that tag does not exist or the guid file already exists, an error will be returned instead. |
| testMessage | Prerequisite: A host has registered.  1. A host sends a report with their guid to the message bus. | The report appears in the message queue | The message bus passes the report along to the processing daemon. |
| testProcess | Prerequisite: A host has sent a report to the message bus.  1. The processing daemon processes reports in the bus. | The report appears in the database. | The report is processed by the daemon and then added to the database, as long as the host is in the database. |
| testCreateUser | 1. A user creates a Monitor Lizard user through the web application and sets valid addresses for all notification fields. | The user appears in the database | This test can only be performed through Django Admin, rather than the Monitor Lizard application, but users can be created properly through it. |
| testCreateTeam | Prerequisite: A user has been created  1. A user creates a team through the web application and sets a user on it. | The team appears in the database. | This test can only be performed through  Django Admin, rather than the Monitor Lizard application, but teams can be created properly through it as long as they have one user. |
| testJoinTeam | Prerequisite: A team has been created  1. A user joins a new user object to an existing team. | The user is associated with the team in the database | This test can only be performed through Django Admin, rather than the Monitor  Lizard application, but users can be added to teams. |
| testRegisterAlertRule | Prerequisite: A host tag has been created. A team has been created.  1. A user registers an alert rule for the host tag that has been created to notify the team they created. | The alert rule appears in the database. | This test can only be performed through Django Admin, rather than the Monitor Lizard application, but alert rules can be created properly through it. |
| testTripAlertRule | Prerequisite: An alert rule has been registered, which will notify a team, which has at least one user which has set both their notification addresses. A report has been processed which will trip the alert rule.  1. The alert daemon scans recent reports for tripped alert rules. | The alert is created.  Notifications are sent to the user. | Alerts are properly created when their rule is tripped. These alerts are displayed on the Monitor Lizard Dashboard, and if emails are configured correctly they get sent out. |
| testNoteAlert | Prerequisite: An alert has been created in the system.  1. A user makes notes on the alert. | The notes appear on the alert. | Notes can be added to alerts through Django Admin, however they do not appear on the frontend application. |
| testResolveAlert | Prerequisite: An alert has been created in the system.  1. A user acknowledges the alert. | The alert shows the user acknowledged it. The alert disappears from “unresolved alerts” counts. | Alerts can be acknowledged to alerts through Django Admin, however this has no effect on the  frontend and they remain in the alerts table. |
| testServerFailure | Prerequisite: Program is running.  1. User manually shuts off management server and all associated components | Hosts continue to send messages to the message queue  that remain until the processing daemon is started. | Messages will collect in the message bus, and are processed  after the processing daemon and server are started back up again. |

## Suggestions for Future Teams

*Author: Soumya Gade*

*Editor(s): Rose Zendels*

There are many areas in which future teams can expand upon this project whether it be with regards to the web application or the various daemons as outlined below:

* **Alert Acknowledgement**: alert acknowledgement from the web application itself would be beneficial as admins would be able to acknowledge that they have received and reviewed an alert which would allow future alerts to come in if the rule is tripped once again. Record which user acknowledged the alert.
* **Tag Configuration**: allow admins to create new tags and assign hosts to those tags from the web application itself as opposed to through Django Admin. It would also be nice if admins are able to remove hosts from tags.
* **Search**: add search bar or some form of search functionality that allows admins to search for specific hosts or tags. Would be useful in a professional deployment where there is a lot of data & information in the system.
* Add more probes to the host daemon to be able to query for more metrics.
* **Anomaly Detection**: add more operators on which alerts can be tripped as opposed to only the minimum and maximum as there currently are.
* **Reporting Rollups**: delete reports after time passes to lower data granularity & avoid database overfilling. Retain hourly reports after a day, daily reports after a month, etc.
* **Host Names**: display host names as opposed to guids in various places on the web application (on the host page, reference to the host on tags page, etc.).
* **Pagination**: add pagination to all of the tables on the different pages of the web app. There is currently only pagination for the Reports table on the Dashboard page.
* Make tables **sortable** and **searchable** by different columns. For instance, add functionality to be able to sort by tag, host name, or metric value.
* Make metric graphs on the host page **date responsive** (elect to view data from different days, weeks, or months). Change formatting of the timestamp on the x axis.
* On the Recent Alerts table, have the threshold column list the actual value on which the alert was tripped in addition to the threshold value itself. Might want to format the columns in this table to be more readable to the admin.
* For Recent Alerts, change severity to a warning icon (two levels) as opposed to highlighting the table row in red as this sort of coloring might be a bit confusing when there aren’t that many alerts currently in the system.
* **Sockets**: use web sockets to get live data on the graphs without having to refresh the page each time. Will need to look into how to do this with chartJS.
* Add SMS support for the alert daemon so that SMS messages are sent when alerts are tripped, similar to how there are currently emails being sent for tripped alerts.
* Create tabs for the running processes and containers on the host page. Change the values in the tables for the processes and containers to be real data as opposed to hard coded values as are currently in the system.