Data Prioritization Manager
Final Project Report

Requirements, Design, Implementation & Testing

LAS 1

CSC 492 Team 27

Seth Emory, William Flathmann, Theodore Japit, Oliver Liang, Dion Ybanez

North Carolina State University
Department of Computer Science

4/28/2023

Executive Summary

Author(s): Dion Ybanez, William Flathman
Reviewer(s)/Editor(s): Dion Ybanez, William Flathman, Theodore Japit

The Laboratory for Analytical Sciences (LAS) is a joint partnership between select research institutions and the National Security Agency to combine academia, industry, and governmental knowledge to advance national security tradecraft. They process large amounts of data and are seeking a way to effectively organize it, so they have tasked us with developing a data prioritization manager.

Our solution is a web-based full-stack configuration management system based on a REST-API backend with a SQL server backing the entire stack. This system is created using React.JS, Django REST Framework, and PostgreSQL.

Our team has managed to successfully implement and thoroughly test all of our original requirements. Our backend unit tests and frontend acceptance tests are all passing. In this iteration, we successfully implemented a login page, the functionality for exporting configuration files, the page for importing/exporting configuration files, and the pages for viewing individual rules and buckets. We are now working on cosmetic improvements for the frontend.

Project Description

Author(s): Dion Ybanez, William Flathman
Reviewer(s)/Editor(s): Dion Ybanez, William Flathman, Xiaochun Liang, Theodore Japit, Seth Emory

Sponsor Background

The sponsor for this project is the Laboratory for Analytical Sciences (LAS). LAS is an organization located on NCSU's campus and is a partnership between the National Security Agency (NSA) and select research institutions. They seek to combine governmental knowledge with the expertise of academia and industry to further their technology and tradecraft through research specifically designed to benefit certain missions pertaining to national security.

Stephen Williamson
NC State Centennial Campus
Poulton Innovation Center
1021 Main Campus Drive
Raleigh, NC 27606
sbwilli3@ncsu.edu

Sean Lynch
NC State Centennial Campus
Poulton Innovation Center
1021 Main Campus Drive
Raleigh, NC 27606
sclynch@ncsu.edu

Aaron Wiechmann
NC State Centennial Campus
Poulton Innovation Center
1021 Main Campus Drive
Raleigh, NC 27606
awiechm@ncsu.edu

Problem Description

Our sponsor's parent organization processes incredibly large volumes of data every day and this leads to them having difficulty organizing and managing it all in a reasonable way. Additionally, they seek to manage the amount of data passing through their system so as not to overload it. For this reason, they are trying to create an outside system to their information pipeline to group data into buckets based on certain rules and the priority of data, so that only the important data is kept and organized. These buckets will each have a certain topic, say a certain nation-state, a hacker group, or an organization of note. Each of these buckets will have rules that will determine the data that gets categorized into them.

We were tasked not with the creation of the system to organize the data, but rather a web-based configuration manager that will manage the configuration of these buckets. Our task was to design and construct a web-based system that can load in existing rule and bucket configurations as well as allow users to view, modify, and create new rules and view their associations with buckets. The rules being loaded in will be a combination of pre-existing rules created by users and rules generated by machine systems tasked with tracking user activity in the main data silos.

Proposed Solution & Project Goals/Benefits

The major goal of this project was to help solve LAS' problem of data prioritization. We hoped to help develop a more efficient means of storing and filtering data. In order to do this, we created an application that serves as a data management system.

In order to achieve this goal, the web application that has been developed will allow for the configuration of the buckets and rules that LAS is using to sort and store the data that they are receiving. This will be a major benefit for the LAS in reducing time required to process such massive amounts of data that would otherwise be unorganized.

Resources Needed

Author(s): Theodore Japit
Reviewer(s)/Editor(s): Dion Ybanez

For the development of our system, we made use of the following technologies. Along with each of them, we have also marked down the purpose for each of the technologies that we used. It is important to note when considering our system, it is supplementary to a bigger overall system. This is mentioned because any configuration file that will be imported into our system will need to be in a .csv format and the same goes for any system that uses our Data Prioritization manager. Any external system accessing ours will need to have functionality for processing our exported .csv file.

Name	Purpose	Status	Version	Licensing
Python	Language used in our backend	Have access	3.1.1	Open Source
Django REST Framework	Framework for our backend	Have access	3.14	Open Source
PostgreSQL	Relational database management system	Have access	15.2	Open Source

React.js	Library for building our frontend UI	Have access	18.2.0	Open Source

Risks & Risk Mitigation

Author(s): Theodore Japit
Reviewer(s)/Editor(s): Dion Ybanez

The main risk that we were worried about was the possibility of requirements to change. LAS made the project seemingly very open-ended for us with ambiguity around the exact requirements. As we progressed with the project, LAS gave us feedback which helped us work toward their envisioned end product as we got a better understanding of it. We kept our project flexible and our code clean, readable, and documented so we could easily make later changes as needed.

We were also concerned with roadblocks in productivity and progress due to the unclear requirements that we were provided. We proactively raised questions to LAS in Slack as they arose and assigned each team member tasks and regularly checked in with each other. This enabled us to work more productively on the project in between meetings.

As we worked on the frontend, another risk was discovered: the potential for a user to upload a malicious file. Another risk was possible performance issues in our database as making GET calls for all the lists can take a long time. The performance risk was mitigated by using caching.

While we did not consider it as heavily due to being outside the scope of the project, security was also a risk that existed.

Development Methodology

Author(s): Theodore Japit
Reviewer(s)/Editor(s): Dion Ybanez

Our team adopted an iterative process for this project. Going with an iterative process helped us to plan and envision the development of the project and set goals, giving us an idea of what exactly we should have completed by specific deadlines. This aspect of an iterative approach also helped with tracking our progress and communicating with our sponsor any questions or issues that we ran into. We conducted sponsor meetings every Thursday alternating between in-person and remote, and planned each of our iterations to last around a month each. We planned for our iterations to last around a month each.

In the beginning we had divided up the workload into three main parts, the frontend, the backend, and the database. We initially had one person working on the database, one on the frontend, and then three on the backend. We divided it in this manner because we concluded we needed a functional backend before making any substantial work on a frontend. The idea is to get the backend working quickly so more effort can be put to the frontend. After the backend was completed, backend members switched to frontend development.

In our development, we created branches for different components and features in our GitHub repository, and worked on them individually. This way, we could test each functionality we intended to add to the project to ensure it is working before merging it with the main branch. When merging branches, pull requests were checked and approved by a different member from the one who posted the pull request. We agreed to have our code pushed to the main branch at least 48 hours before each iteration deadline to give us ample time to check for bugs and fix them.

Our standards for communication were for members to respond to messages within 12 hours and say something in advance if there was a time they would not be available. If we ran into a disagreement between two members, we planned to have both sides put forward their reasoning and then have the rest of the team vote on which side to go with.

System Requirements

Author(s): William Flathmann, Seth Emory, Theodore Japit, Xiaochun Liang, Dion Ybanez

Reviewer(s)/Editor(s): Dion Ybanez

Overall View

Our system is a web-based application that will be accessible by users that have either read or write access [DS1]. We just have a basic implementation for users and roles that later developers can add onto and fully flesh out.

We have a simple system that allows users to login with a username and password; each user has one of the simplified roles of read or write. Users that hold read access can only view the entire system and see the different buckets, bucket information, rules, rule information, and their respective settings. Users with write access can view the entire system the same way one with read access can and also edit the configuration of the system. When the user is finished, they can export the configuration of the system.

Ultimately, this system is a configuration manager for a method of storing data. The data is stored through the use of buckets, which are objects that store data based on the rules associated with them. The storage of data is outside of our system; the system we are creating only deals with the configuration of the buckets and rules. These buckets are defined by a name, a data size, a data size limit per 24 hours, max rule duration limit, associated rules along with their priorities, and an age off policy for rules within them. Buckets are created in the database by an admin.

The rules in our system configure the type of data that is stored inside the buckets. A rule is defined by a unique identifier/name, the bucket(s) they are associated with, the type of rule (IP only or IP-port pair), their priority, their status (active or inactive), their set active time, and the source it came from. Rules can be imported from a CSV file or created by a user on the system. Rules are associated with buckets. Depending on the user of the system, the rules can be created, modified, or deleted.

The main goal of this system is to either take an existing configuration or start from scratch to modify or create the configuration of the buckets and their rules respectively. The configuration develops a more efficient means of storing and filtering data by using bucket and rule combinations to prioritize storing certain data over other data.

Functional Requirements

FR1 User Interface

- FR1.1. On startup, user can login with a username and password
- FR1.2. User can import machine-generated rules from a csv file
- FR1.3. User can create rules
    - FR1.3.1 System will verify rule is valid
- FR1.4. User can view buckets as a list
    - FR1.4.1. User can view rules for each bucket
    - FR1.4.2. User can view other bucket data
- FR1.5. User can edit an existing rule
- FR1.6. User can delete a rule
    - FR1.6.1 User can delete a rule from one or all buckets.
        - FR1.6.1.1 Rule is removed when no longer associated with any buckets
- FR1.7. User can export a bucket and rule config file
- FR1.8. User can view Rules as a list
- FR1.9. User can view Individual bucket
- FR1.10 User can view Individual rules

FR2 Rule

- FR2.1. Can be generated by machine tools
- FR2.2. Can be created by users

FR3 Bucket

- FR3.1. Buckets will be imported from the database

Non-Functional Requirements

- NFR1. The system will be able to handle machine-generated input

- NFR2. The system will be able to handle user supplied input
    - NFR2.1 Input includes text, number, and IP address formatted in single or CIDR notation
- NFR3. Input like ports and IP addresses must be valid
- NFR4. The UI should be responsive (define what responsive means here)
- NFR5. Rules will contain data
    - NFR5.1. Rules will have a name
    - NFR5.2. Rules will have an ID
    - NFR5.3. Rule will hold the bucket(s) they are associated with
    - NFR5.4. Rule will have type (IP-only, IP-port pair)
    - NFR5.5. Rule will have a priority
    - NFR5.6. Rule will have a status of active/inactive
    - NFR5.7. Rule will store time of being set to active (in UNIX format)
    - NFR5.8. Rule will store its source (RADs, TLeaves, etc.)
    - NFR5.9. Rule will store the IP or IP port pair it represents
- NFR6. Buckets will contain certain information
    - NFR6.1. Buckets will contain name
    - NFR6.2. Buckets will contain ID
    - NFR6.3. Buckets will contain data size
    - NFR6.4. Buckets will contain data size limit per 24 hours
    - NFR6.5. Buckets will contain max rule duration limit
    - NFR6.6. Buckets will contain associated rules and priorities
    - NFR6.7. Buckets will contain age-off policy configuration
- NFR7. Users will contain certain information
    - NFR7.1 Users will contain id
    - NFR7.2 Users will contain name
    - NFR7.3 Users will contain type
- NFR8. The UI should contain some landing page for overview information
    - NFR8.1 The landing page should tell the user how many rules and buckets are in the system
    - NFR8.2 The landing page should have a graph to show active vs inactive rules
    - NFR8.3 The landing page should have a visualization to show how many rules are in each bucket
    - NFR8.4 The landing page should have a heat map to show the locations that the rules are tagging through their IPs

Constraints

- C1. The system must be a stand-alone application
- C2. The system must operate on commodity hardware
- C3. The system must be accessible via standard modern browsers

Design

Author(s): William Flathmann
Reviewer(s)/Editor(s): Dion Ybanez, Theodore Japit, Seth Emory

High-Level Design

From a high level, our design consists of three main components: a frontend, a backend, and a database.

The frontend is built in React.JS and it allows users to log into the system and view the information about rules and buckets in the system. Rules and buckets can be both viewed as a list and as individual items with more specific details displayed. If the user has write permission, they can also manage rules and import and export configuration files.

The backend, built in Django REST Framework, facilitates the communication between the database and the frontend through the models.py, serializers.py, and views.py files. The backend also contains the implementation for the functionality to import and export configuration files through the frontend.

PostgreSQL is our relational database management system that stores the data tables for users, buckets, rules, and rule-bucket associations.

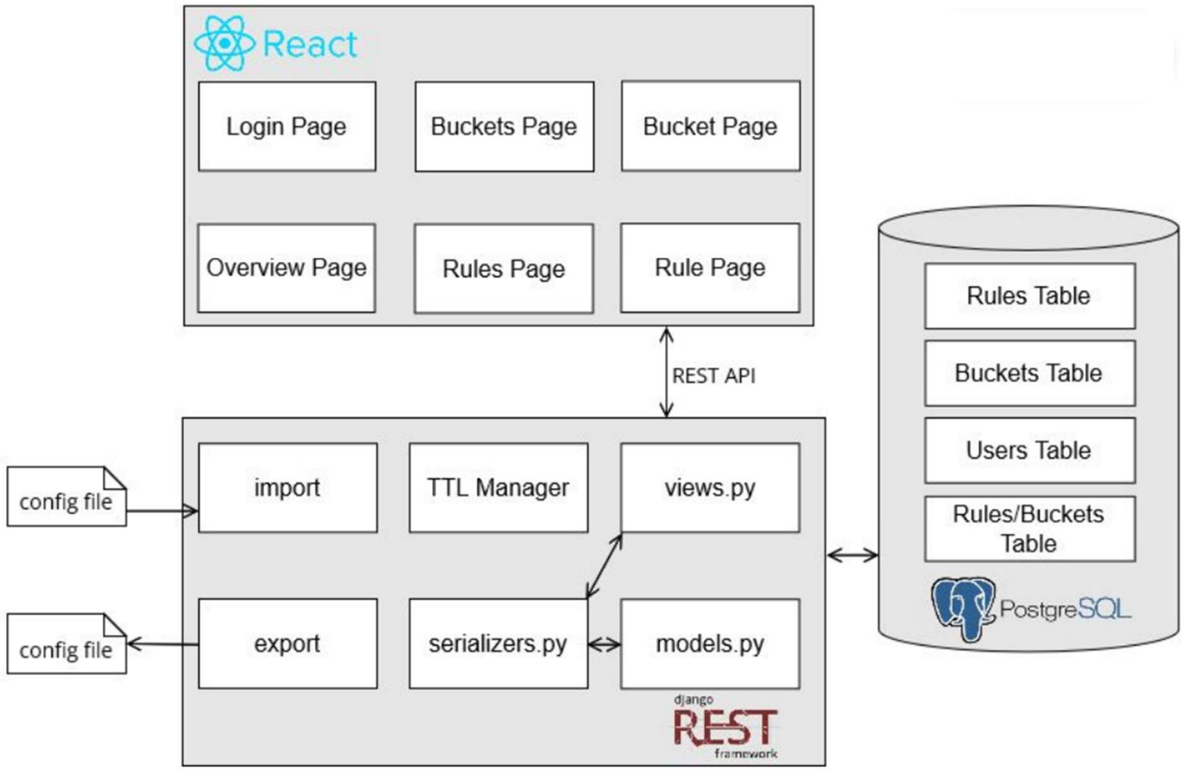


fig 1. General System Diagram

Low-Level Design

Frontend

For this frontend portion, we will be sharing screenshots from our working system and describing the purpose for each section using these screenshots for reference.

Upon start-up, the user will see a login page. Users either have read-only or write permissions. If a user does not log in, they cannot gain any access to our system until they do login.

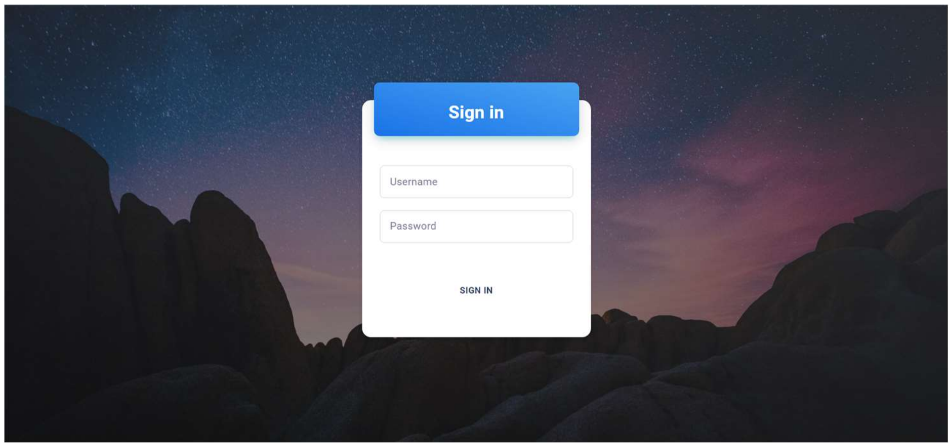


Fig 2.1: Login Page

After the user logs in, they are taken to the landing page that displays the number of buckets and rules. There are also many graphs to display more information about the rules and buckets like a pie chart showing the percentages of active and inactive rules, a box chart showing the amount of rules in buckets compared to other buckets, and a heat map at the bottom showing the locations of the IPs on the world map.

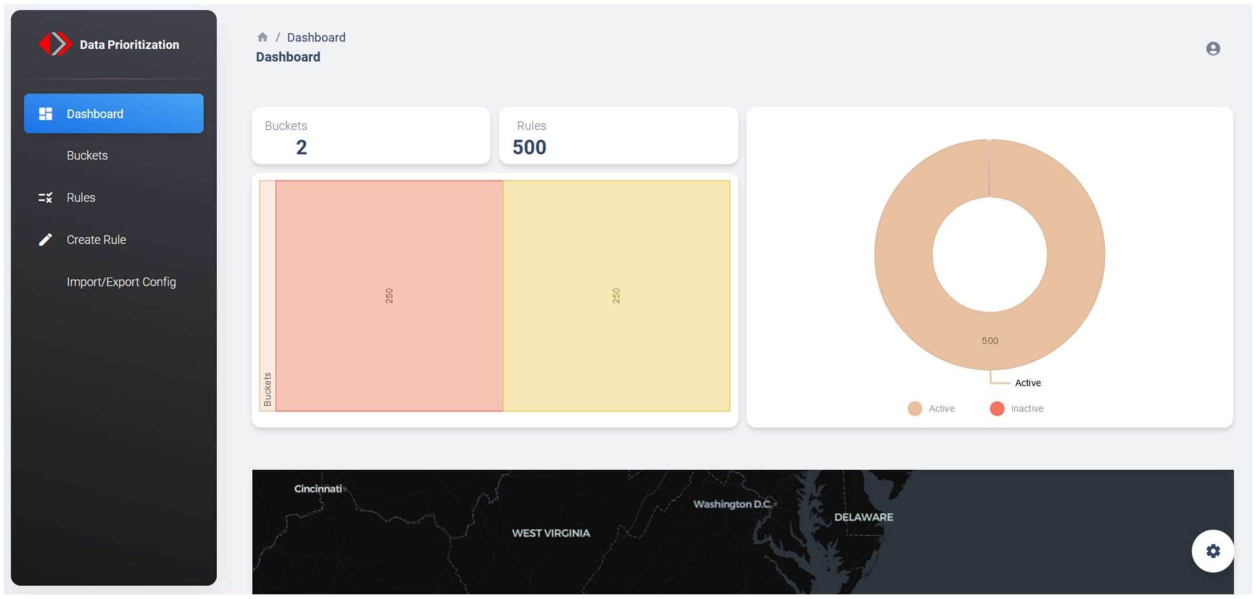
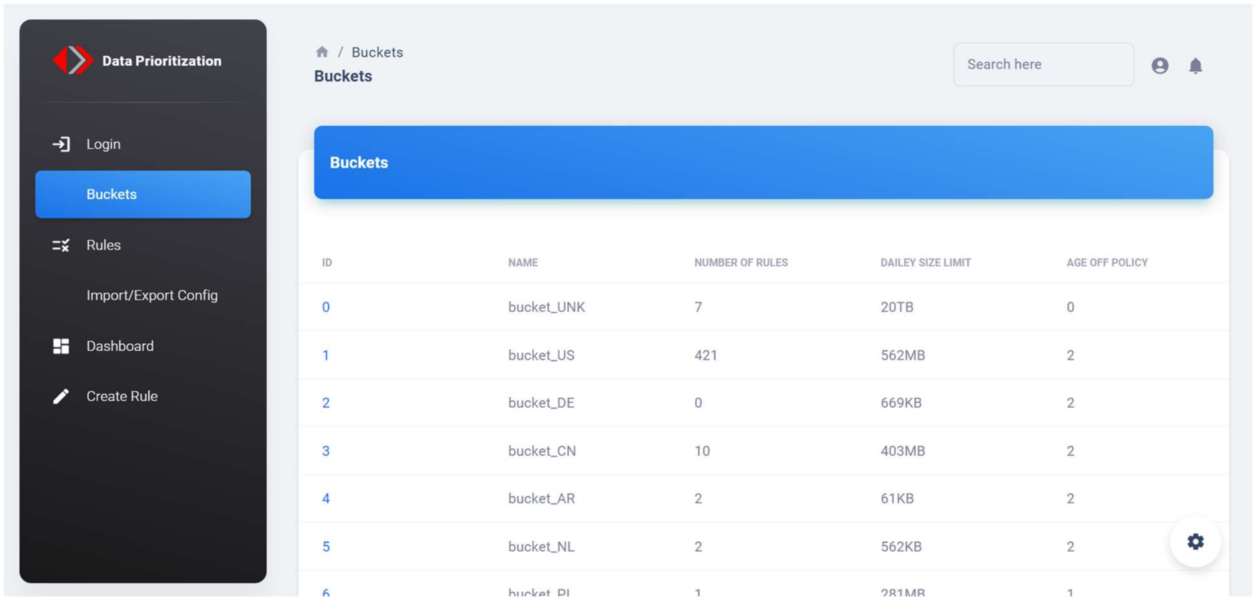


Fig 2.2: Landing Page

On the left side of the page is a menu with links to pages for list of buckets, list of rules, import/export config file, and rule creation. Clicking on the "Buckets" page will lead the user to a page that shows a list of the buckets with each bucket's ID, name, number of associated rules, daily size limit, and age off policy.

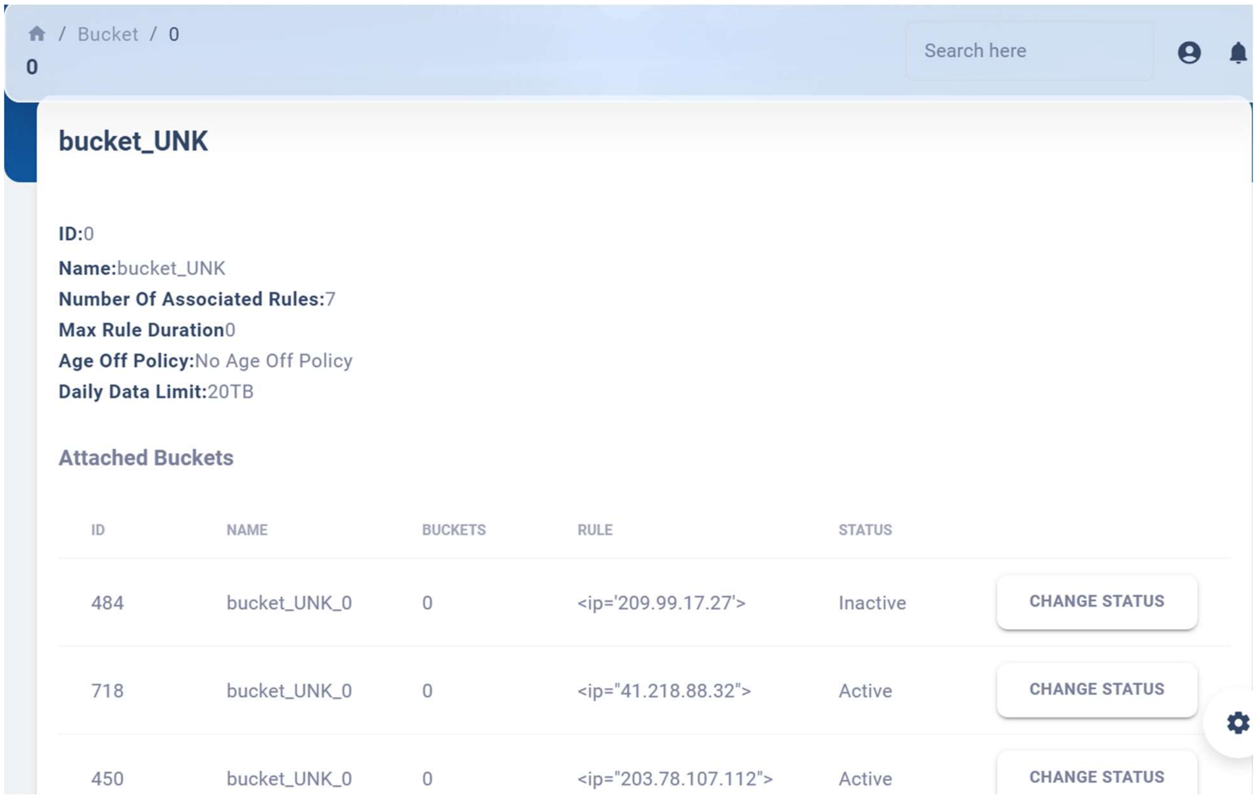


ID	NAME	NUMBER OF RULES	DAILEY SIZE LIMIT	AGE OFF POLICY
0	bucket_UNK	7	20TB	0
1	bucket_US	421	562MB	2
2	bucket_DE	0	669KB	2
3	bucket_CN	10	403MB	2
4	bucket_AR	2	61KB	2
5	bucket_NL	2	562KB	2
6	bucket_PL	1	281MB	1

Fig 2.3 Buckets Page

After the user clicks on a bucket, they will be taken to this expanded bucket view. They can see all the details about the bucket and a list of their rules. There is no edit functionality for the bucket, as buckets are not intended to be edited. The only edit functionality on this page is the

Change Status button on each row of a rule; this button will change the rule's status from active to inactive or inactive to active.


Bucket / 0
0

Search here

bucket_UNK

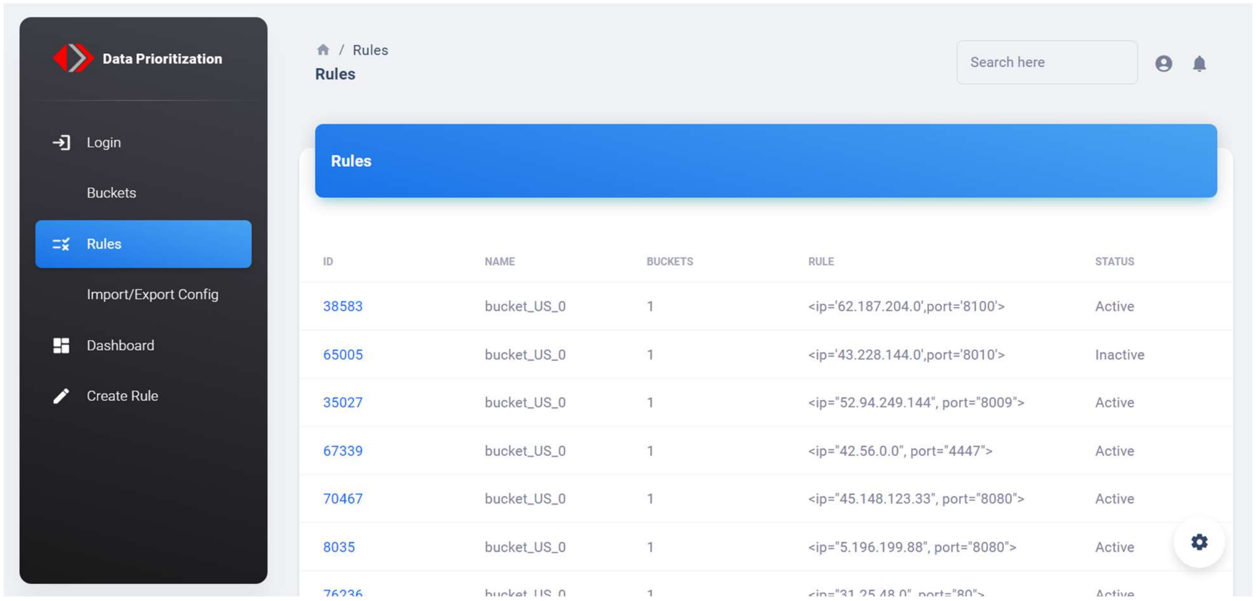
ID:0
Name:bucket_UNK
Number Of Associated Rules:7
Max Rule Duration0
Age Off Policy: No Age Off Policy
Daily Data Limit:20TB

Attached Buckets

ID	NAME	BUCKETS	RULE	STATUS	
484	bucket_UNK_0	0	<ip='209.99.17.27'>	Inactive	CHANGE STATUS
718	bucket_UNK_0	0	<ip="41.218.88.32">	Active	CHANGE STATUS
450	bucket_UNK_0	0	<ip="203.78.107.112">	Active	CHANGE STATUS

Fig 2.4 Single Bucket View Page

When a user clicks on the "Rules" page, they see a list of all the rules with the ID, name (which is set to an associated bucket), number of associated buckets, type (IP/IP-port), and status (active/inactive) of each rule.



Rules
Rules

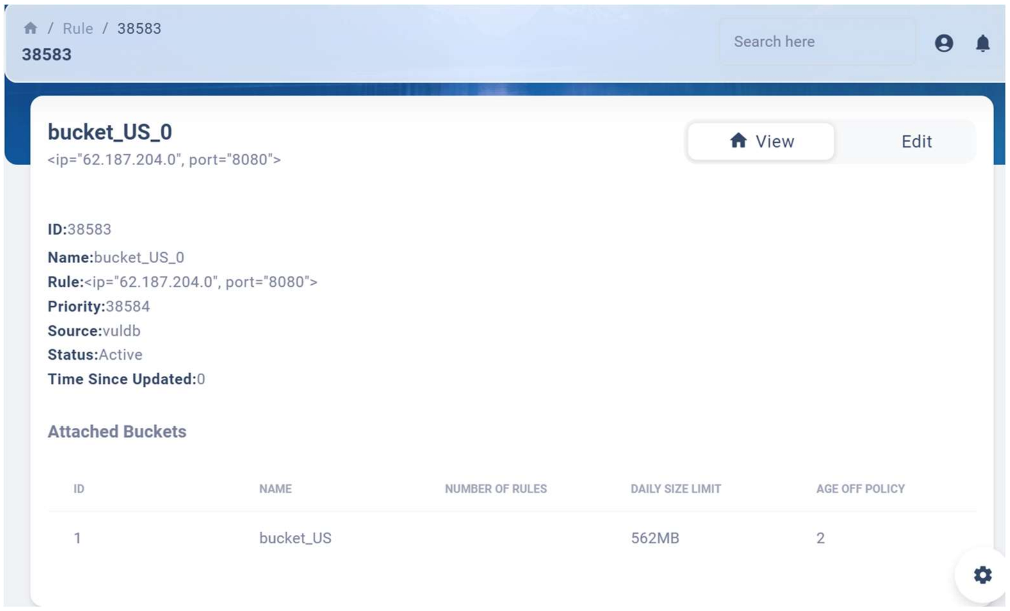
Search here

Rules

ID	NAME	BUCKETS	RULE	STATUS
38583	bucket_US_0	1	<ip='62.187.204.0',port='8100'>	Active
65005	bucket_US_0	1	<ip='43.228.144.0',port='8010'>	Inactive
35027	bucket_US_0	1	<ip="52.94.249.144", port="8009">	Active
67339	bucket_US_0	1	<ip="42.56.0.0", port="4447">	Active
70467	bucket_US_0	1	<ip="45.148.123.33", port="8080">	Active
8035	bucket_US_0	1	<ip="5.196.199.88", port="8080">	Active
76236	bucket_US_0	1	<ip="31.25.48.0", port="80">	Active

Fig 2.5 Rules Page

When a user clicks on a rule ID, they will then be taken to this expanded rule view. From here, they can see the rule name, all its details, and the buckets that it is associated with.



bucket_US_0
<ip="62.187.204.0", port="8080">

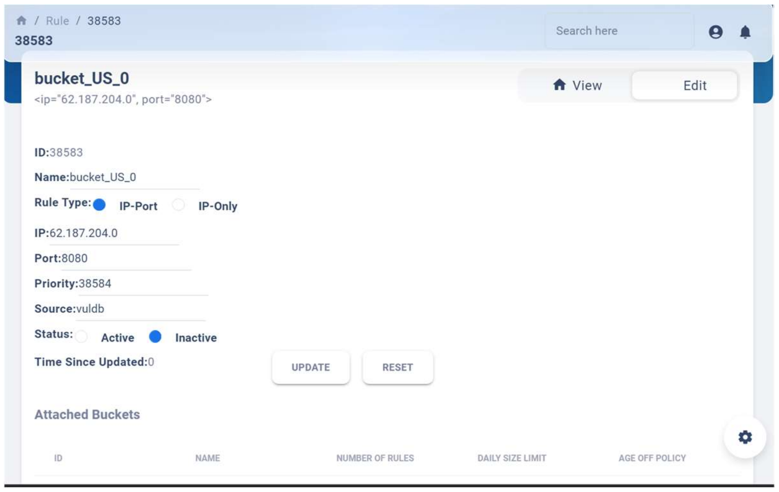
ID:38583
Name:bucket_US_0
Rule:<ip="62.187.204.0", port="8080">
Priority:38584
Source:vuldb
Status: Active
Time Since Updated:0

Attached Buckets

ID	NAME	NUMBER OF RULES	DAILY SIZE LIMIT	AGE OFF POLICY
1	bucket_US		562MB	2

Fig 2.6 Single Rule View Page

Switching to the "Edit" tab allows the user to edit the rule. Name, IP/Port, priority, source, and active/inactive status can all be changed. The user enters the desired new information on a form then hits the update button to submit the change. There is also a reset button to reset the form to its original state.



bucket_US_0
<ip="62.187.204.0", port="8080">

ID:38583
Name:bucket_US_0
Rule Type: ● **IP-Port** ○ **IP-Only**
IP:62.187.204.0
Port:8080
Priority:38584
Source:vuldb
Status: ○ **Active** ● **Inactive**
Time Since Updated:0

Attached Buckets

ID	NAME	NUMBER OF RULES	DAILY SIZE LIMIT	AGE OFF POLICY

Fig 2.7 Rule Edit Tab

Next, when the user clicks the "Import/Export Config" page, they are taken to a page where they can either import a set of rules or download the rules and buckets currently in use as CSV files.

Configuration files can be imported by either dragging a CSV file into the designated space or browsing the user's file directory to select the CSV file to import.

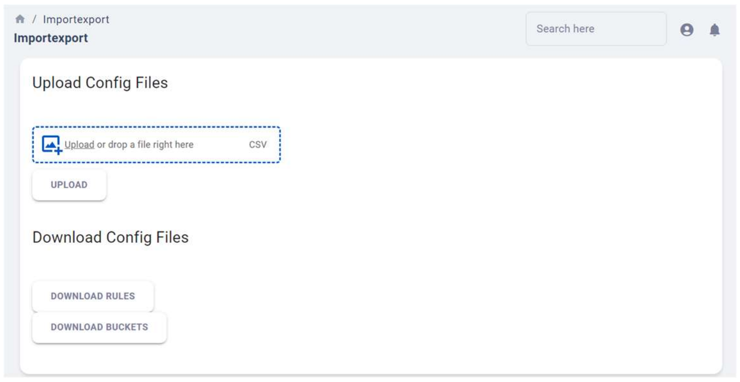


Fig 2.8 Import/Export Page

Finally, the "Create Rule" page allows the user to create a new rule to add to the system. To add an associated bucket, they can select from a dropdown of the buckets in the system.

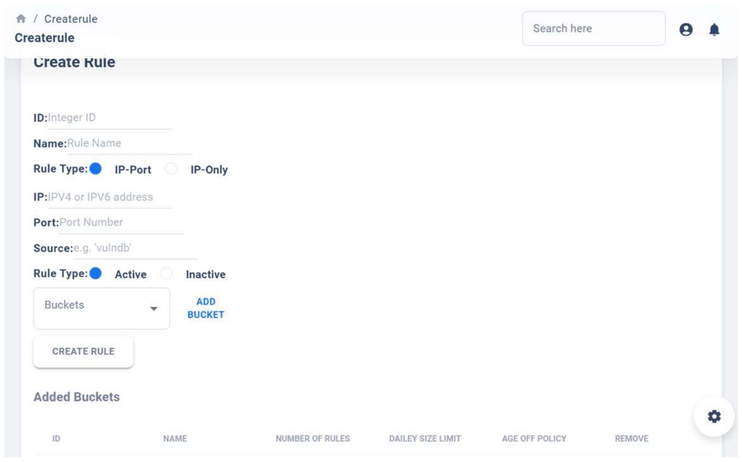


Fig 2.9 Create Rule Page

Backend

Note: In the current project plan, buckets and users are not created by the system and will be manually input into the database. Our system will only interact with them. This may change in future iterations.

There are many files that are auto-generated by the Django REST Framework and not touched in the backend, but the main files created by us are the Django views file, the serializers file, and the models file. There is also a file that contains the tests for the API endpoints.

The models.py file will contain model classes for users, buckets, and rules, defining the data fields and behaviors for each of them. These models can be thought of as the object classes for the types of data stored in our database.

The serializers.py file will contain serializer classes for each of the models. The serializers are what convert the models into JSON to be sent through the view classes to the frontend. They define what attributes of each model class will be sent through the view classes.

The views.py file will contain views for each of the models. It will have two views for each of the models: one for a single object and one for all of the objects. These classes will contain the GET, PUT, POST, and DELETE REST functions.

1. allUsers: Fetch the list of all users using a GET request.
2. user: Fetch a specific user when checking for authentication using a GET request.
3. allBuckets: Fetch a list of all of the buckets using a GET request.
4. bucket: Interact with a specific bucket using a GET request to pull the bucket's information and a PUT request to remove rules from the bucket.
5. allRules: Fetch a list of all of the rules using a GET request.
6. rule: Interact with a specific rule using GET, PUT, and DELETE requests to extract the rule's information, edit the rule, and delete the rule respectively.
7. authentication: used to verify if a username and password pair match the database, can use PUT
8. import_rules_and_buckets: Used to ingest data into the system, supports PUT requests
9. export_rules_and_buckets: Used to exfil data from the system, supports GET requests
10. rule_exists: used to check if a rule exists, supports GET requests

The view classes will create API endpoints as follows:

/users:
GET: Request is a simple fetch for the list of users. Returns JSON data with HTTP response 200 if users exist, or an empty JSON list with HTTP response 200 if no users exist.

/user/{int:id}:
GET: Request is a simple fetch with the integer ID of a specific user. Returns JSON data with HTTP response 200 if a user with the ID exists, or an HTTP response of 404 if it does not exist.

api/buckets:
GET: Request is a simple fetch for the list of buckets. Returns JSON data with HTTP response 200 if buckets exist, or an empty JSON list with HTTP response 200 if no buckets exist.

bucket/{int:id}:
GET: Request is a simple fetch with the integer ID of a specific bucket. Returns JSON data with HTTP response 200 if a bucket with the ID exists, or an HTTP response of 404 if it does not exist.
PUT: Request is an HTTP payload of the new bucket data as JSON. The server will return HTTP response 200 if the JSON payload is valid and the data is able to be saved, and an HTTP 400 error message if not.

rules:
GET: Request is a simple fetch for the list of rules. Returns JSON data with HTTP response 200 if rules exist, or an empty JSON list with HTTP response 200 if no rules exist.

rule/{int:id}:
GET: Request is a simple fetch with the integer ID of a specific rule. Returns JSON data with HTTP response 200 if a rule with the ID exists, or an HTTP response of 404 if it does not exist.
PUT: Request is an HTTP payload of the new rule data as JSON. The server will return HTTP response 200 if the JSON payload is valid and the data is able to be saved, and an HTTP 400 error message if not.
DELETE: Request contains the integer ID of the rule to delete. If the rule exists and is able to be deleted, an HTTP response 200 is returned. If the deletion is unsuccessful, HTTP response 404 is returned.

export/rules
GET: request is a simple fetch with the value “rules” attached to the request, the endpoint will package the buckets existing in the database and return them as a csv file downloadable by the user.

export/buckets
GET: request is a simple fetch with the value “buckets” attached to the request, the endpoint will package the buckets existing in the database and return them as a csv file downloadable by the user.

import
POST: request contains a csv file containing the data, if the data is of the correct format of a rule or bucket list, it is read in and a response of 200 is returned, else 400

Login
POST: verifies the password and username sent in the request with what is existing in the database, if it matches return 200 and permissions, else return 400

RuleExists/<int:pk>/
GET: checks to see if a rule matching the id pk exists in the database, returns 200 if it does, 400 otherwise

Database

This project is using PostgreSQL to store and manage data, and will contain 4 types of data: **user**, **bucket**, **rule,** and **rule-bucket associations**.

Users have a name, ID, password, and whether they are read or write.

Buckets have a name, ID, data capacity, max data size per 24 hours, max rule duration limit, a list of rules and priorities associated with it, and its assigned age-off policy configuration.

Rules have a name, ID, bucket list, type (IP/IP-port pair), priority, status (active/inactive), the time that they were last set as active, and the source of the rule.

Rule-Bucket Association each entry stores an id for the association, the rule id, and a bucket id.

Example data that would be in an input/output file for our system (CSV File):

```
# Users
id,name,role
1,UserA,write
2,UserB,read
3,UserC,write
```

id	name	role
1	UserA	write
2	UserB	read
3	UserC	write

```
# Buckets
name,id,size,daily-size-limit,max-rule-duration,rules,age-off-policy
Bucket Awesome,1,90,20,30,<1>,none
Bucket Wonderful,2,1000,100,30,<1>,individual
Bucket Crazy,3,10,3,30,<2,3>,bucket
```

name	id	size	daily size limit	max rule duration	rules	age off policy
Bucket Awesome	1	90	20	30	<1>	none
Bucket Wonderful	2	1000	100	30	<1>	individual
Bucket Crazy	3	10	3	30	<2,3>	bucket

# Rules
Name,id,buckets,type,priority,status,active time,source,ip
Rule A,1,<1,2>,ip-only,1,active,20,RADs,<127.0.0.1>
Rule B,2,<3>,ip-port,2,inactive,10,TLeaves,<127.0.0.2,80>
Rule C,3,<3>,ip-port,3,inactive,5,TLeaves,<192.1.2.3,100>
Rule D,4,<1,2,3>,ip-only,4,active,30,RADs,<192.1.2.1>

name	id	buckets	type	priority	status	active time	source	ip
Rule A	1	<1,2>	ip-only	1	active	20	RADs TLeaves	<127.0.0.1>
Rule B	2	<3>	ip-port	2	inactive	10	TLeaves	<127.0.0.2,80>
Rule C	3	<3>	ip-port	3	inactive	5	TLeaves	<192.1.2.3,100>
Rule D	4	<1,2,3>	ip-only	4	active	30	RADs	<192.1.2.1>

Implementation
Author(s): William Flathmann
Reviewer(s)/Editor(s): Dion Ybanez, Theodore Japit

Iteration Definition & Current Status

- Iteration 0
    - start date: Jan. 19, 2023
    - end date: Feb. 1, 2023
    - Features Implemented: no real functional requirements, setting up all the technologies to allow for start of interation1
- Iteration 1
    - start date: Feb 2, 2023
    - end date: Mar 1, 2023
    - Features Implemented:
        - FR 1.2 A user can import machine generated rules from a csv file
        - FR 1.4 A user can view buckets as a list
        - FR 1.8. User can view rules as list
        - FR 1.9. User can view individual bucket
        - FR 1.10. User can view individual rule
        - FR3.1 buckets can be imported from the database
        - NFR5 what a bucket contains
        - NFR6 what a rule contains
        - NFR7 What a user contains
- Iteration 2
    - start date: Mar 2, 2023
    - end date: Apr 1, 2023
    - Features implemented:
        - FR1.2. User can import machine-generated rules from a csv file
        - FR1.3. User can create rules
            - FR1.3.1 System will verify rule is valid
        - FR1.4. User can view buckets as a list
            - FR1.4.1. User can view rules for each bucket
            - FR1.4.2. User can view other bucket data
        - FR1.8. User can view Rules as a list
- Iteration 3
    - start date: Apr 2, 2023
    - end date: Apr 30, 2023
    - Features implemented:
        - FR1.1. On startup, user can login with a username and password
        - FR1.3. User can create rules
            - FR1.3.1 System will verify rule is valid

- FR1.5. User can edit an existing rule
- FR1.6. User can delete a rule
    - FR1.6.1 User can delete a rule from one or all buckets.
        - FR1.6.1.1 Rule is removed when no longer associated with any buckets
- FR1.7. User can export a bucket and rule config file
- FR1.9. User can view Individual bucket
- FR1.10 User can view Individual rules
- FR2.1. Rules can be generated by machine tools
- FR2.2. Rules can be created by users
- NFR1. The system will be able to handle machine-generated input
- NFR2. The system will be able to handle user supplied input
    - NFR2.1 Input includes text, number, and IP address formatted in single or CIDR notation
- NFR3. Input like ports and IP addresses must be valid
- NFR4. The UI should be responsive

Security Considerations

Currently, for our implementation, we do not have many security concerns. For the final implementation, we planned to only go as far as defining who can do what in the system. That is, we have users who are able to read, and those who are able to read/write.

These users are authenticated through fairly basic password storage techniques of hashing and salting the password strings and comparing them to a given password from a user. Having the authentication gives us integrity, meaning those who are not authorized to edit the data should not be able to edit the data. Confidentiality is not a concern in our current system outside of the user authentication data which will be handled by the given system.

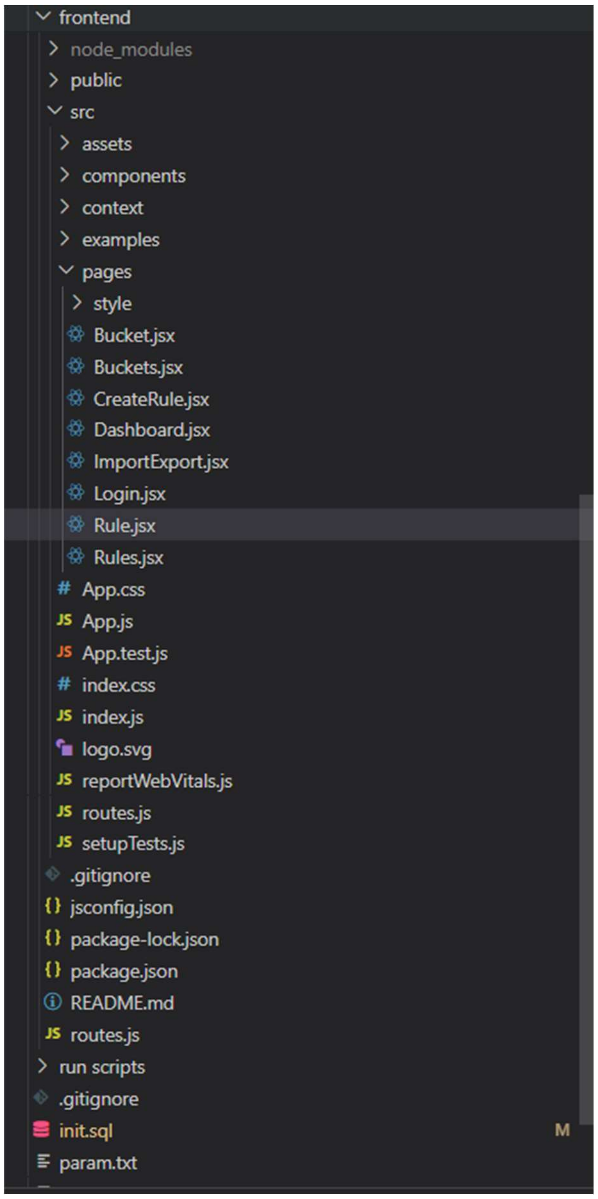
For availability, the system is predominantly run locally, so its availability is not able to be interrupted unless the physical system is attacked, and in that case, there are larger things to worry about.

As for accountability, we did not have any plans to implement a system to track which users make which changes, though this can be added into the system if needed. Lastly, privacy is not a concern outside of user data, as everything else will be viewable to everyone.

Project Folder Structure

The project is set up in two major folders: the frontend folder and the backend folder. For this reason, we will go over each of these folders separately.

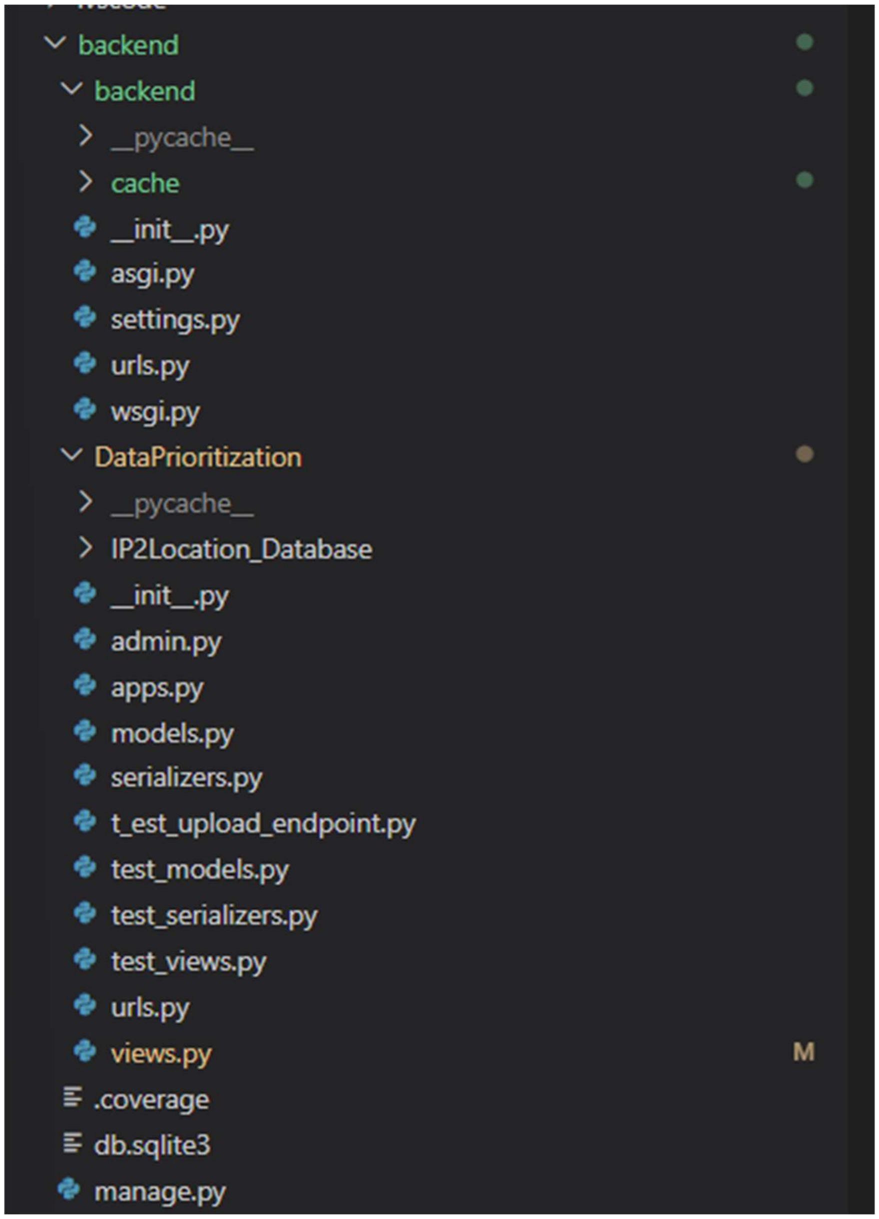
The frontend folder, as the name suggests, holds all the files that pertain to the frontend code. There are certain autogenerated files and folders, such as all of /public. The other folder /src contains all of the code for our frontend web page. The main part of the folder contains the general routing stuff for the web page, while the /pages folder contains the code pertaining to the individual pages on our frontend web page.



```
frontend
  node_modules
  public
  src
    assets
    components
    context
    examples
    pages
      style
      Bucket.jsx
      Buckets.jsx
      CreateRule.jsx
      Dashboard.jsx
      ImportExport.jsx
      Login.jsx
      Rule.jsx
      Rules.jsx
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    reportWebVitals.js
    routes.js
    setupTests.js
  .gitignore
  jsconfig.json
  package-lock.json
  package.json
  README.md
  routes.js
run scripts
.gitignore
init.sql                                   M
param.txt
```

fig 3.1: Frontend file structure

Next, there is the backend folder. This folder also contains some general auto-generated files. This folder contains two other folders, which are another backend folder and a DataPrioritization folder. The internal backend folder contains some of the higher level parts of the Django backend, such as configurations. The DataPrioritization folder contains all of the files that define the specific parts of our backend models and their connection with the server.



```
backend
  backend
    __pycache__
    cache
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
  DataPrioritization
    __pycache__
    IP2Location_Database
    __init__.py
    admin.py
    apps.py
    models.py
    serializers.py
    t_est_upload_endpoint.py
    test_models.py
    test_serializers.py
    test_views.py
    urls.py
    views.py                                M
  .coverage
  db.sqlite3
  manage.py
```

fig 3.2 backend file structure

Any folders present in either of the screenshots that were not mentioned are inconsequential and auto-generated by React or Django.

Project Configuration/Settings

At this moment, the main configuration file is the configuration file for the backend Django REST Framework, which is located in “backend\backend\settings.py”. This contains all of the installed softwares on the backend, the connection type, and connection details for the backend database. The frontend doesn’t have too much configuration, as it just calls the open backend server for data. There is also some minor configuration involved with setting up the environments, but this is done through the standard pyenv library. The needed libraries are kept track of through a package.json file for node.js libraries, and a requirements.txt file for python libraries.

Testing

Author(s): Seth Emory
Reviewer(s)/Editor(s): Dion Ybanez

Overall View

For this system, we performed unit tests and black box system tests. The data prioritization system is mainly just a backend and a web frontend, so we tested all of the backend with automated unit tests and the frontend with black box tests. The file with the test dataset will hold data for three buckets and seven rules and can be seen below.

Buckets:

id	name	size	age_off
0	bucket_awesome	90MB	0
1	bucket_wondeful	1G	30
2	bucket_crazy	10MB	30

Rules:

id	bucket	rule	priority	status
0	bucket_awesome	<rule:ip-only; ip=127.0.0.1>	1	1

1	bucket_awesome	<rule:ip-pair; ip=127.0.0.2, port=8080>	2	1
2	bucket_awesome	<rule:ip-only; ip=0.0.0.0/0>	-	0
3	bucket_wondeful	<rule:ip-only; ip=9.9.9.9>	-	0
4	bucket_wonderful	<rule:ip-only; ip=4.4.4.4>	1	1
5	bucket_wonderful	<rule:ip-only; ip=0.0.0.0/0>	-	0
6	bucket_crazy	<rule:ip-only; ip=0.0.0.0/0>	1	1

Unit Testing

The unit testing in the system will be testing the backend. Specifically, the models, views, serializers, and import/export pieces of the backend. Models are tested for the creation and usage of the different bucket, rule, and user models to ensure that they behave as expected. Views are tested to ensure that the API endpoints are running correctly. The serializers are tested to make sure that the data being run through is converted correctly. Finally, the import/export portion is tested to ensure that the system is getting the correct data from the file and can generate a proper file from the data the system holds.

The frontend web application piece of the system is tested through our blackbox acceptance test plan rather than automated unit tests.

We initially decided to write our unit tests in Pytest-Django and also use it for the code coverage report, but we decided to use Django's built in testing tools. When testing for code coverage of the system, our goal was to reach 70% coverage across all metrics, so 70% coverage for line, method, statement, and branch.

Unit test results:

- Model: 95% coverage, 4/4 tests passing
- Serializer: 100% coverage, 41/41 tests passing
- View: 100% coverage, 10/10 tests passing

Acceptance Testing

Here is the entire table of black box test cases for the acceptance testing. These test the system functionality from the perspective of the user that is interacting with the frontend of the system. It covers the import/export of the configuration, rule creation, rule editing, rule deletion, user permissions, and different invalid cases.

When performing these tests, we assume that the user will be using the provided “sample_test_file.xlsx” file and that the database will already have the sample buckets “bucket awesome,” “bucket wonderful,” and “bucket crazy” added, but with no rules associated.

To run these tests, the users will need to turn on the servers and navigate to the webpage.

Some of these tests can be tested independently, but they are written to be performed in sequence, so rules 1-13 should be performed one after the other. Tests 14-17 are not reliant on the previous.

We managed to get all of our acceptance tests in our original test plan to pass.

Test ID	Steps	Expected Results	Actual Results
Test1 LoginWithRead	1) Navigate to the website 2) Login with the username “username” and password “password” 3) Click the Rules button in the navigation tab 4) Navigate to the first rule in the list and check for an add rule button	The user should be taken to the home screen after login. When going to the rules page, the user should be able to see a page for where the rules would be but there should be no add button for the user.	User is taken to the home screen after login. When going to the rules page, the user sees a page for where the rules would be, with no add button.
Test2 LoginWithWrite	1) Navigate to the website 2) Login with the username “” and password “” 3) Click the Rules button in the navigation tab 4) Navigate to the first rule in the list and check for an add rule button	The user should be taken to the home screen after login. When they navigate to the rules page, they should be able to see the add rule button.	The user is taken to the home screen after login. When they navigate to the rules page, they can see the add rule button.

Test3 ImportRules	1) The user clicks the “import/export” button in the navigation menu 2) When the user is taken to the new page, they will click on the import button. 3) From there, they will select the “sample_test_file” file to import into the system.	The system now should have 7 rules. There should be 4 active rules and 3 inactive rules. Bucket awesome will have 3 rules, bucket wonderful will have 3 rules, and bucket crazy will have 1 rule. Bucket awesome has the rules with 127.0.0.1 IP, 127.0.0.2:8080 IP/port, and 0.0.0.0/0 IP/port.	The system has 7 rules. 4 are active, and 3 are inactive. Bucket awesome has 3 rules, bucket wonderful has 3 rules, and bucket crazy has 1 rule. Bucket awesome has rules with 127.0.0.1 IP, 127.0.0.2:8080 IP/port, and 0.0.0.0/0 IP/port
Test4 ExportRules	1) The user clicks the “import/export” button in the navigation menu 2) When the user is taken to the new page, they will click on the export button. 3) The user will be prompted to give a name for the file and location of export, give the name “test_export” and select the desktop.	The file named “test_export” should be on the user’s desktop. When comparing the input file that was used for test3 ImportRules, the two files should be the same.	The file named “test_export” is on the user’s desktop. When comparing the input file that was used for test3 ImportRules, the two files are the same.
Test5 ViewListOfBuckets	1) The user clicks the “Buckets” button in the navigation menu 2) On this page, the user should be able to see all the buckets in the system and a brief summary of their information.	The user should see 3 buckets: bucket awesome, bucket wonderful, and bucket crazy. The user should see that bucket awesome has 3 rules. They should see that bucket wonderful has 3 rules. Finally they should see that bucket crazy has 1 rule.	The user sees 3 buckets: bucket awesome, bucket wonderful, and bucket crazy. The user sees that bucket awesome has 3 rules, bucket wonderful has 3 rules, and bucket crazy has 1 rule.
Test6 ViewBucketDetails	1) While the user is at the Buckets overview page, they select bucket wonderful, by clicking on it 2) The user should now be on the individual bucket page and should now look at the details they see.	The user should see that the bucket has a size of 1G and an age off policy of 30 days. That bucket should have the rules with IPs of 9.9.9.9, 4.4.4.4, and 0.0.0.0/0.	The user sees that the bucket has a size of 1G and an age off policy of 30 days. That bucket has the rules with IPs of 9.9.9.9, 4.4.4.4, and 0.0.0.0/0.
Test7 AddRuleToBucket	1) While looking at the details for the bucket, click the “Add Rule”	The user should be taken back to the bucket detail page.	The user is taken back to the bucket detail page. When looking at

	button 2) Add this info to the rule: Bucket: wonderful Rule: <rule:ip-only; ip=6.6.6.6> Priority: 4 Status: 1 3) Click “Finish”	When looking at the rules that are attached, you should see the rule that the user just created.	the rules that are attached, the rule the user created is shown.
Test8 EditRuleAssociatedToBucket	1) While on the details page for the buckets, click the edit button for the rule that was just created. 2) Change the IP for this rule to be 7.7.7.7 3) Click finish	When the user is taken back to the details page, they should see that the rule that is priority 4 now has an IP of 7.7.7.7 instead of an IP of 6.6.6.6 like it was before.	When the user is taken back to the details page, they see that the rule that is priority 4 now has an IP of 7.7.7.7 instead of an IP of 6.6.6.6 like it was before.
Test9 ChangeRulePriority	1) While on the details page for the buckets, find the rule that is priority 4. 2) Click and drag that rule to the top.	The rule with IP 7.7.7.7 should be at priority 1. Rule with IP 4.4.4.4 should be in priority 2.	The rule with IP 7.7.7.7 is at priority 1. Rule with IP 4.4.4.4 is in priority 2.
Test10 MakeRuleInactive	1) While on the bucket details page, the user will click the edit button next to the rule with IP 7.7.7.7 2) They will then select the deactivate button. 3) Take note of the rule status for the results 4) The user will go to the home page by clicking the home button in the navigation menu. 5) Check the summary details on the home page	When the user checks the summary details on the homescreen, they should see that the number of inactive rules is 3. When the user goes back to the bucket details page for bucket wonderful, they should see that the rule with IP 7.7.7.7 is no longer active.	When the user checks the summary details on the homescreen, they see that the number of inactive rules is 3. When the user goes back to the bucket details page for bucket wonderful, they see that the rule with IP 7.7.7.7 is no longer active.
Test11 MakeRuleActive	1) From the bucket details page of bucket wonderful, click the edit button for the inactive rule with IP 7.7.7.7 2) Select the active button for the rule 3) Take note of the status of the rule while on the bucket details page	The user should see that the home page now is reporting that there are 5 active rules instead of 4. Also, when the user looks at the bucket details page for bucket wonderful, they should see there are x active rules in the bucket.	The user sees that the home page now is reporting that there are 5 active rules instead of 4. Also, when the user looks at the bucket details page for bucket wonderful, they see there are x active rules in the bucket.

	4) The user will go to the home page by clicking the home button in the navigation menu. 5) Check the summary details on the home page		
Test12 DeleteRuleFromBucket	1) From the bucket details page the user will click on the delete button for the rule with IP 7.7.7.7 2) Click the confirm button for the rule deletion.	The page should update for the user. After it does, the user should no longer see the rule with IP 7.7.7.7 on the bucket details page. In the bucket details page, there should be 3 rules associated with the bucket.	The page updates for the user. After it does, the user no longer sees the rule with IP 7.7.7.7 on the bucket details page. In the bucket details page, there are 3 rules associated with the bucket.
Test13 DeleteRuleFromSystem	1) The user selects the “rules” button in the navigation menu 2) On the rules overview page, the user clicks the delete button for the rule with IP 0.0.0.0/0. 3) Confirm the deletion	The rule is deleted from the rules overview page. Also, when looking at buckets awesome, wonderful, and crazy; the rule with IP 0.0.0.0/0 should not appear in their rule lists anymore.	The rule is deleted from the rules overview page. Also, when looking at buckets awesome, wonderful, and crazy; the rule with IP 0.0.0.0/0 does not appear in their rule lists anymore.
Test14 InvalidLogin	1) The use navigates to our system 2) The user enters “password” for the username and “username” for the password	The user should not be able to login because some credentials are invalid. The user should also be prompted to try again.	The user cannot login because some credentials are invalid. The user is also prompted to try again.
Test15 ImportInvalidFile	1) The user clicks the “import/export” button in the navigation menu 2) When the user is taken to the new page, they will click on the import button. 3) From there, they will select the “invalid_import_file” file to import into the system	The user should be given an error message saying “Invalid file” and taken back to the import/export page.	The user is given an error message saying “Invalid file” and taken back to the import/export page.
Test16	1) While looking at the	There should be an	There is an error when

AddWithInvalidIP	details for the bucket, click the “Add Rule” button 2) Add this info to the rule: Bucket: wonderful Rule: <rule:ip-only; ip=1..1> Priority: 4 Status: 1 3) Click “Finish”	error when the user submits this information to be added as a rule. They will get an error message being told that they have an invalid IP. They will stay on the add page and be allowed to edit the info to try again.	the user submits this information to be added as a rule. They will get an error message being told that they have an invalid IP. They will stay on the add page and be allowed to edit the info to try again.
Test17 AddWithInvalidPort	1) While looking at the details for the bucket, click the “Add Rule” button 2) Add this info to the rule: Bucket: wonderful Rule: <rule:ip-port; ip=1.1.1.1/1.2> Priority: 4 Status: 1 3) Click “Finish”	There should be an error when the user submits this information to be added as a rule. They will get an error message being told that they have an invalid port. They will stay on the add page and be allowed to edit the info to try again.	There is an error when the user submits this information to be added as a rule. They will get an error message being told that they have an invalid port. They will stay on the add page and be allowed to edit the info to try again.

Task Plan

Author(s): Xiaochun Liang
Reviewer(s)/Editor(s): William Flathman, Theodore Japit

Coursework

Task description	**Owner**	**Due Date**	**Status**
Team Ground Rules	All	1/19	Complete
Sponsor Meeting	All	1/19	Complete
Draft of Initial Sponsor Meeting Agenda	All	1/23	Complete
Conduct First Official Sponsor Meeting	All	1/25	Complete
Sponsor Meeting	All	1/26	Complete
Rough Draft of System Requirements & Preliminary Design	All	1/27	Complete
Individual Logs/Peer Evals	All	2/1	Complete
Sponsor Meeting	All	2/2	Complete
System Requirements & Preliminary Design-Beta	All	2/3	Complete
OPR 1	Dion	2/3	Complete

Sponsor Meeting	All	2/9	Complete
Rough Draft of System Requirements & Preliminary Design	All	2/10	Complete
Task Planning	All	2/10	Complete
Sponsor Meeting	All	2/23	Complete
Draft OPR 2	Oliver, Theodore	2/23	Complete
Interim Project Report	All	2/24	Complete
Individual Logs/Peer Evals	All	3/1	Complete
Sponsor Meeting	All	3/2	Complete
Sponsor Meeting	All	3/9	Complete
OPR 2	Oliver, Thedore	3/9	Complete
Sponsor Meeting	All	3/23	Complete
Rough Draft of Posters & Pies Poster	All	3/24	Complete
Draft OPR 3 Slides	Seth, William	3/28	Complete
Sponsor Meeting	All	3/30	Complete
Individual Logs/Peer Evals	All	4/1	Complete
Sponsor Meeting	All	4/6	Complete
OPR 3	Seth, William	4/6	Complete
Sponsor Meeting	All	4/13	Complete
Sponsor Meeting	All	4/20	Complete

Implement

Task description	**Owner**	**Due Date**	**Status**
Iteration 0		1/31	Complete
Go though project description	All		
Set up the testing environment on local	All		
Backend draft models.py	Dion		

Backend draft view.py	Oliver		
Draft app.js and test	Seth		
Backend draft serializers.py	Theodore		
Set up database configuration	William		
Iteration 1		2/28	Complete
Go through Django tutorial to learn Django	Oliver, Dion	2/2	Complete
Backend models file	Dion	2/7	Complete
Backend views file	Oliver	2/7	Complete
Frontend pages and testing	Seth	2/7	Complete
Backend serializers file	Theodore	2/7	Complete
Test database with rules, buckets, and users	William	2/13	Complete
Organize backend and merge it together	Theodore, Oliver, Dion	2/14	Complete
Finalize the frontend to work with React	Seth	2/14	Complete
Link backend and front end and database	All	2/16	Complete
Set up database import/export	William	2/19	Complete
Black box test draft for frontend	Seth	2/20	Complete
Bug fixing for merging things together	All	2/28	Complete
Iteration 2		3/31	Complete
Read input config file	William	3/24	Complete
Functionality for adding and removing rules in buckets	Oliver	3/24	Complete
Set up time manager	William	3/24	Complete
Rule and bucket list pages	Seth	3/24	Complete
Iteration 3		4/30	Complete
Output config file functionality	William	4/21	Complete
Login page	Theodore	4/21	Complete
Individual rule and bucket pages	Seth	4/21	Complete

Import/export page	Oliver	4/21	Complete

Suggestions for Future Teams

Author(s): Seth Emory
Reviewer(s)/Editor(s): William Flathman, Xiaochun Liang

In this report, we have discussed a lot about the system we created and the processes/design decisions that we used to create this system. Now we would like to discuss a few bugs that should be addressed,some aspects that were not finished completely, and how we see the system progressing past our group working on the project.

First we can talk about bugs that we have noted within the system. There is one with the caching system for the data. There are some instances where this does not work properly, especially during the ingestion of importing files. Ultimately in the future it would be nice to see the need for caching be removed or minimized through optimization of the backend. The other bug that we have noticed is the rule not being properly updated when editing a rule multiple times without refreshing the page. Currently there is a bandage fix on the edit rule page to just force a page refresh after saving the change,but this is only a workaround.

In regards to the aspects that were not finished, the time tracking/rule age off system was not worked on, also the capability for sorting and filtering tables needs to be implemented on the tables for the rules and buckets. Other than specific features or aspects of the system, there just needs to be general optimization mainly within the views file in the backend. The main issue that needs to be optimized is regarding how data is being given to the frontend. Buckets only store the association data in their object, so in order to also get the rule object, there are currently individual rule calls for every ID that the bucket holds. This would be fine if there were only a few rules but this will ultimately be a system with hundreds to thousands of rules. This optimization would ultimately lead to a much more efficient system and could also reduce the need for caching.

In the future, we see this system eventually becoming integrated with larger systems so that the import and export functions can work without having to manually download files and move them to other systems. When needing to add or change any of our codebase, please make sure to reference the developers guide. This includes in heavy detail specifics of our code and where and how you would need to change it. Good luck future team with your project; hope you have a good semester.

William Flathman, Team Lead, wjflathm@ncsu.edu
Seth Emory, Lead Frontend Developer, smemory@ncsu.edu, 706-699-1628

Theodore Japit, Backend Developer & Recorder, tjapit2@ncsu.edu,
Oliver Liang, Xiaochun Liang, Backend Developer, xliang8@ncsu.edu
Dion Ybanez, Backend Developer, dybanez@ncsu.edu