Data Prioritization
Interim Project Report

Preliminary Requirements, Design, Implementation & Testing

LAS

CSC 492 Team 27

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

North Carolina State University
Department of Computer Science

2/24/2023

Note: Please revise date with each submission

Executive Summary

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

This project is sponsored by the Laboratory for Analytical Sciences (LAS), a joint partnership between select research institutions and the National Security Agency (NSA) to combine academia, industry, and governmental knowledge to advance national security tradecraft. This parent organization processes large amounts of data, and thus is seeking a better system to manage this data. We are not working on the actual management system, but a configuration manager to keep track of the configuration of this manager.

Our solution is to create a web-based full-stack configuration management system based on a REST-API backend with a SQL server backing the entire stack. This system will be created using React.JS, Django REST Framework, and PostgreSQL. Currently, we have put together the majority of the requirements and design, though we see both of those as living documents.

In terms of implementation, we have created all three components of the stack to some extent. The backend is much more fleshed out than the frontend, due to its importance in frontend development. As of now, we are able to load data into the database, and have it pass through the backend to be served to the frontend through GET requests.

Our next steps are to finish out the backend, and then pivot most of our efforts to the frontend. We have not set up much testing yet, but we have put together a testing plan, included later in this document, that we intend to implement soon.

Project Description

Author(s): Dion Ybanez, William Flathman, Xiaochun Liang
Reviewer(s)/Editor(s): Dion Ybanez, William Flathman, Xiaochun Liang

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 which bucket.

We have been tasked not with the creation of the system to organize the data, but a web-based configuration manager that will manage the configuration of these buckets. Our task is to design and construct a web-based system that can load in existing rule and bucket configurations, and 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 user generated rules 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 is to help solve LAS' problem of data prioritization. We hope to help develop a more efficient means of storing and filtering data. In order to do this, we will create an application that serves as a data management system.

An application like this could be complicated. However, in order to solve this and make the creation of the application simpler, we have decided to use Django REST Framework. Django REST Framework is a toolkit that reduces the amount of code needed to be written for REST interfaces. It is also customizable and supports lots of databases, which includes PostgreSQL, that we are also using.

Specifically, the project is a system that allows users to manage rules, which are represented by an IP or IP-port pair. The project will use buckets and associate rules with buckets. The basic functionality we are aiming for is for the user to be able to import a set of buckets and rules associated with it. Inside buckets users can manage rules by adding, deleting, and deactivating. Finally, users can export a config of updated buckets.

Resources Needed

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

Name	Purpose	Status	Version	Licensing
Python	Language for the project	Have access	3.1.1	Creative commons
djangorestframework	Framework for our backend	Have access	3.14	Creative commons
PostgreSQL	Relational database management system	Have access	15.1	Creative commons
React.js	Library for building our frontend UI	Have access	18.2.0	Creative commons

Risks & Risk Mitigation

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

One major risk is changing requirements. LAS seems to have this project quite open-ended for us and the exact requirements are not completely clear. The requirements of the project will be changing over time by decisions and feedback from LAS, which will help us get a better understanding of what their envisioned end product will look like. Our mitigation methods for this are to keep our project flexible and to keep our code clean, readable, and documented so that we can more easily make changes to it if needed.

Another risk is productivity and teamwork. Specifically, due to the unclear requirements of the project, we had some concerns about potential blocks in productivity and progress if we do not understand what is expected from us. We will be proactive in raising any questions to our sponsor in Slack as they arise, and also assigning tasks to each team member and checking in, so we can be more productive in our work on the project in between meetings instead of waiting for the next meeting to discuss.

Another potential risk is security. However, while security is definitely a risk present in the project, it is outside the scope of what is asked of us, and thus we are not considering it as heavily as the other risks.

Development Methodology

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

Our team is adopting an iterative process for this project. Going with an iterative process helps 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 can also help us with tracking our progress and communicating with our sponsor any questions or issues that we run into. We are conducting sponsor meetings every Thursday alternating between in-person and remote. We are planning for each of our iterations to last around a month each.

In our development, we are creating branches for different components and features in our GitHub repository, and working on them individually so we can test each functionality we intend to add to the project to ensure it is working before merging it with the main branch. When we are merging branches, we are having pull requests checked and approved by a different member from the one who posted the pull request.

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

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]. Currently, there does not need to be an in-depth implementation for users and roles, we just need to make sure to include the hooks that later developers could add onto and fully flesh out.

We will just have a simple system that will allow users to login with username and passwords; each of these users will have one of the simplified roles of read or write. Users that hold read access will only be able to view the entire system and see the different buckets, bucket information, rules, rule information, and their respective settings. Users with the write access will be able to view the entire system exactly how the user with right access can, but the write access will also allow the user to edit these configurations. When the user is finished, they will be able to export the configuration of the system.

Ultimately, this system will be a configuration manager for a method of storing data. The way that data will end up being stored is through the use of buckets. Buckets are just objects that will be holding data, but 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 will be 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 the bucket age off policy. Buckets will not be created in our system. They will be created through an admin creating them directly inside of our database. Users will then be able to view the buckets and add/edit/delete the rules that are associated with the buckets.

The rules in our system will be used to configure the type of data that will be stored inside the buckets. A rule will be defined by a unique identifier/name, the bucket(s) they are associated with, the type of rule, their priority, their status of active or inactive, their set active time, and the source it came from. Rules can be generated from the input in a source file, or be created by a user on the system. Rules will be associated with buckets of data. 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 and to create or modify the configuration of the buckets and their rules. The purpose of this would be to use the configuration to develop a more efficient means of storing and filtering data. This is achieved by using the bucket and rule combination to prioritize certain data above 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

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
- C4. The system must be web-based application

Design

Author(s): William Flathmann
Reviewer(s)/Editor(s): Dion Ybanez

High-Level Design

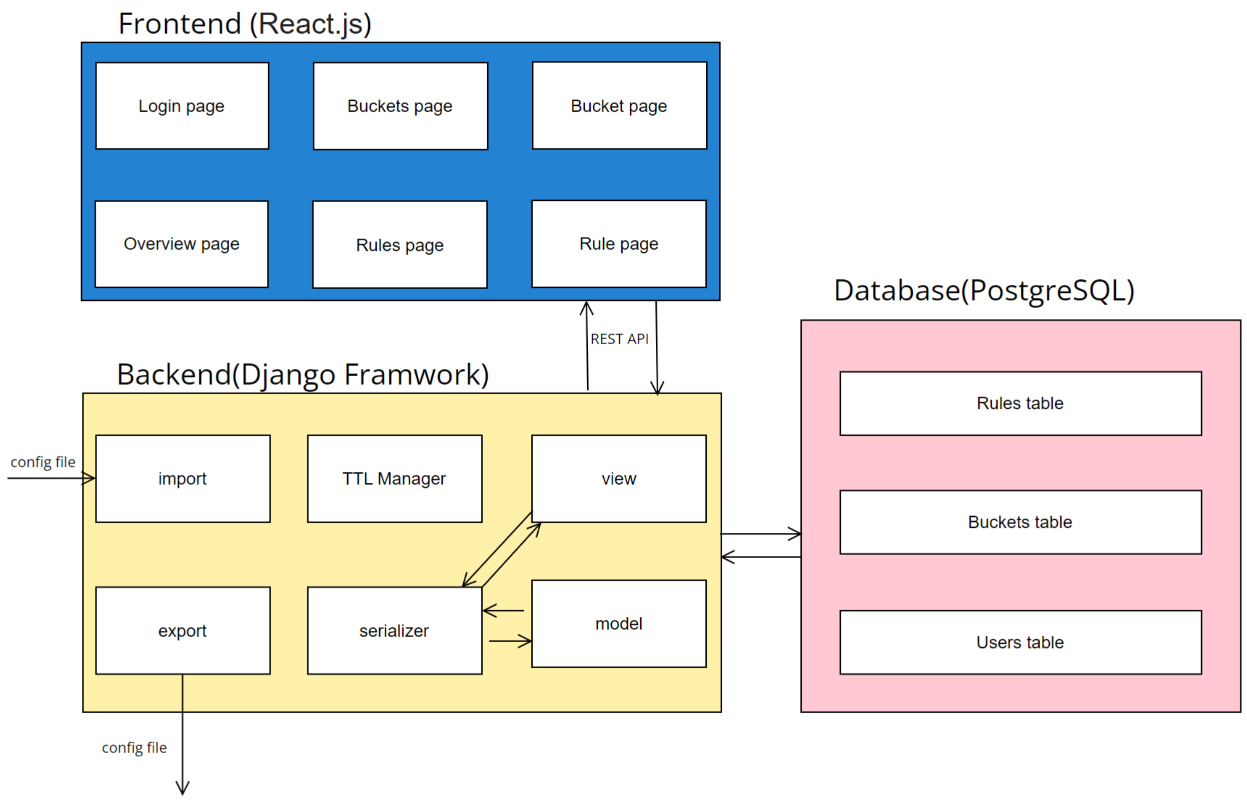


fig 1. General System Diagram

From a high level, our design will consist of three main components. It will have a frontend UI component built in React.JS that will be used to allow users to interact with the rules and buckets in the system. It will also contain a backend built-in Django Rest Framework (DRF) that will facilitate the communication between the backend database and the frontend UI. Lastly, there will be a backend PostgreSQL database that will store the data for users, buckets, and rules.

Low-Level Design

Frontend

Upon start-up, the user will see a login page. Users are either read-only or non-read-only.

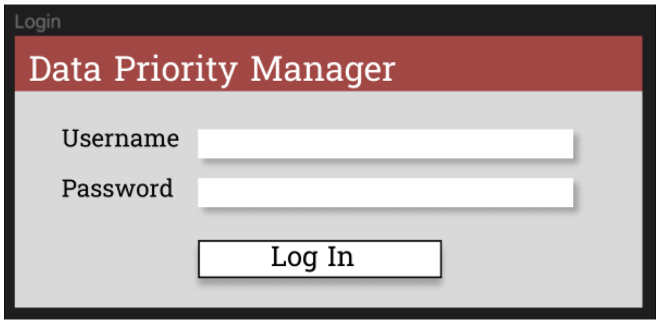
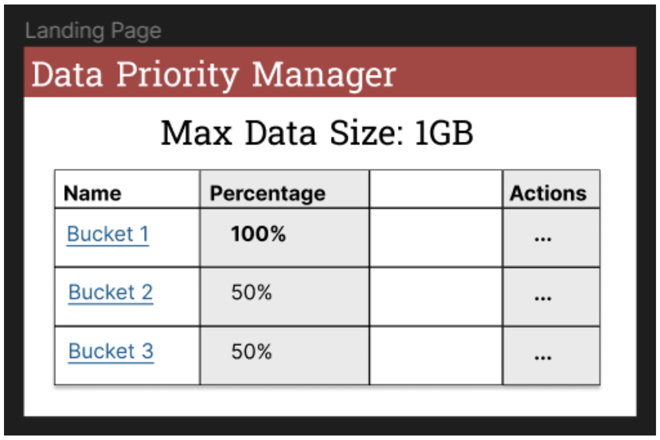


Fig 2.1: Login Page

After the user logs in, they are taken to the landing page that displays all of the buckets. From there, they can see a little bit of the details for the buckets. They can also click on the buckets to see more details and the rules that are associated.

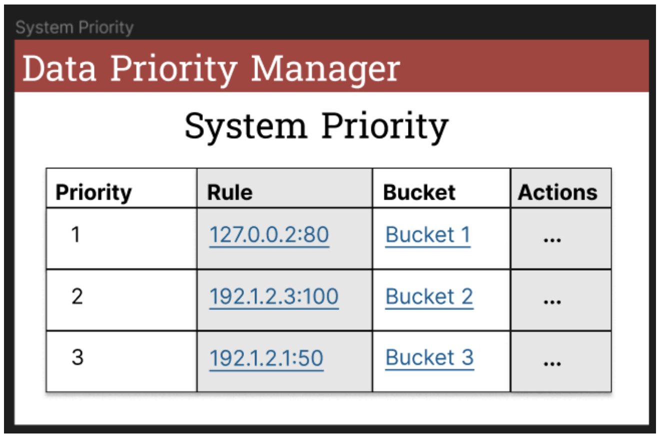


Max Data Size: 1GB

Name	Percentage		Actions
Bucket 1	**100%**		...
Bucket 2	50%		...
Bucket 3	50%		...

Fig 2.2: Bucket List Page

Viewing rules by system priority is another option. It will show each rule in the system, ordered by their priority, and include which bucket they are associated with. Through this view, you can select the individual rules to see their information, or select the bucket they are associated with.

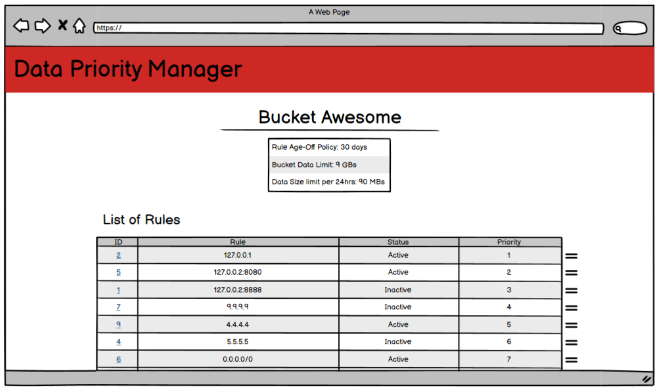


System Priority

Priority	Rule	Bucket	Actions
1	127.0.0.2:80	Bucket 1	...
2	192.1.2.3:100	Bucket 2	...
3	192.1.2.1:50	Bucket 3	...

Fig 2.3 System Priority List

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. They can see the rules and how they are all prioritized. They can also click on the individual rules to see their full details and edit them if the user has edit capabilities.



Bucket Awesome

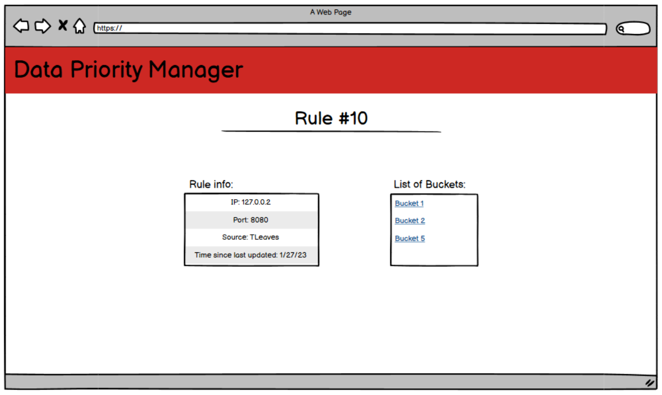
Rule Age-Off Policy: 30 days
Bucket Data Limit: 9 GBs
Data Size limit per 24hrs: 90 MBs

List of Rules

ID	Rule	Status	Priority
2	127.0.0.1	Active	1
5	127.0.0.2:8080	Active	2
1	127.0.0.2:8888	Inactive	3
7	9.9.9.9	Inactive	4
9	4.4.4.4	Active	5
4	5.5.5.5	Inactive	6
6	0.0.0.0/0	Active	7

Fig 2.4 Single Bucket View Page

When a user clicks on the rule 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. The user would also be able to edit the rule from this page.



Rule #10

Rule info:
IP: 127.0.0.2
Port: 8080
Source: TLeaves
Time since last updated: 1/27/23

List of Buckets:
Bucket 1
Bucket 2
Bucket 5

Fig 2.5 Single Rule View Page

Backend

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

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

The model.py file will contain model classes for the Users, the Buckets, and the Rules. The models can be thought of as the object classes for each of the three things. They will define how they work and what data they carry

The serializer file will contain serializer classes for each of the models. The serializers are what convert the models into text to be sent on through the view classes to the frontend. They will define what attributes from the list of attributes of each of the classes will be passed to the view classes, this will likely be all attributes.

The view file will contain views for each of the models. It will have two views for each of the models; one to interact with a single object, and one to view all of the objects. These classes will contain the REST functions for get, put, post, and delete.

1. allUsers: This will be used to fetch lists of all of the users, it will only contain get requests
2. user: this will be used to fetch specific users when checking for authentication, it will only have a get
3. allBuckets: this will be used to fetch a list of all of the buckets, it will only contain a get
4. bucket: This will be interact with specific buckets, it will contain a get and a put (the put is for removing rules from the bucket)
5. allRules: this will be used to fetch all of the rules, it will only have a get
6. rule: this will be used to interact with specific rules. It will contain, get, put, and delete

The view classes will create API endpoints as follows:

api/users:
GET: request will be a simple fetch with the integer ID, will return data as JSON with HTTP response 200 if data exists, if there is no data to return, it will return an empty JSON list with HTTP response 200

api/user/{int:id}:
GET: request will be a simple fetch with the integer ID of the user, will return data as JSON if data exists, if the data does not exist it will return an HTTP response of 404

api/buckets:
GET: request will be a simple fetch with the integer ID, will return data as JSON with HTTP response 200 if data exists, if there is no data to return, it will return an empty JSON list with HTTP response 200

api/bucket/{int:id}:

GET: request will be a simple fetch with the integer ID, will return data as JSON with HTTP response 200 if data exists, if the data does not exist it will return an HTTP response of 404

api/rules:
GET: request will be a simple fetch with the integer ID, will return data as JSON with HTTP response 200 if data exists, if there is no data to return, it will return an empty JSON list with HTTP response 200

api/rule/{int:id}:
GET: request will be a simple fetch with the integer ID of the rule, will return data as JSON with HTTP response 200 if data exists, if the data does not exist it will return an HTTP response of 404
PUT: request will be an http payload of the rule data as JSON and the server will return 200 if the JSON payload is valid and the data is able to be saved, otherwise it will return an HTTP 400 error message
DELETE: request will contain the DELETE request and the integer ID of the rule to delete, if the rule exists and is able to be deleted a HTTP response 200 will be returned, otherwise a 404 will be returned if the data cannot be found.

Database

This project is using PostgreSQL to store and manage data, and will contain 3 types of data: **user**, **bucket**, and **rule**.

Users have a name, ID, 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, IP/IP-port pair, priority, active/inactive, the time that they were last set as active, and the source of the rule.

Example input/output (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>

GUI Design
The project does not have a GUI.

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

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: ongoing
    - Features Implemented (or to be 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
- Future iterations will implement more parts, but we have not predefined what the iterations will contain.

Security Considerations

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

These users will be authenticated through existing Django authentication systems which will give us our identification & authentication component. 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.

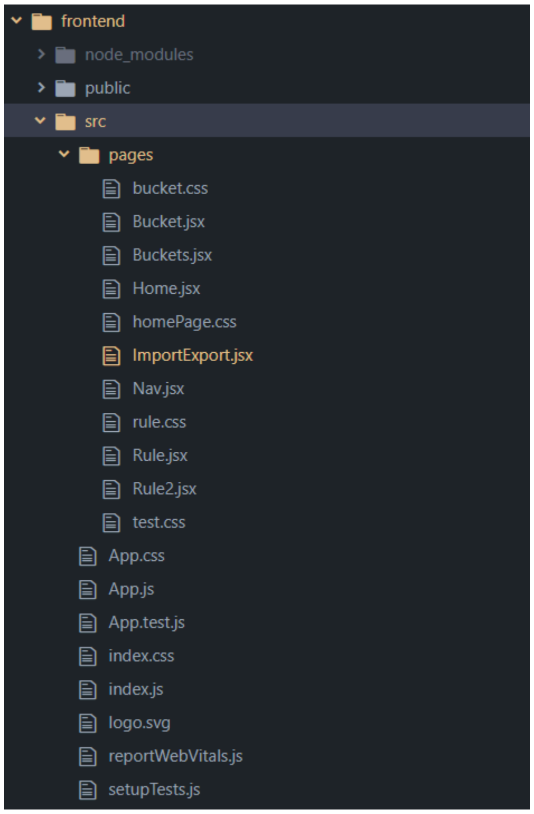
For availability, the system will be predominantly run locally, so its availability will not be able to be interrupted unless the physical system is attacked, and in that case, there are larger things to worry about.

As for accountability, we do 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, I will separately talk about each of these folders.

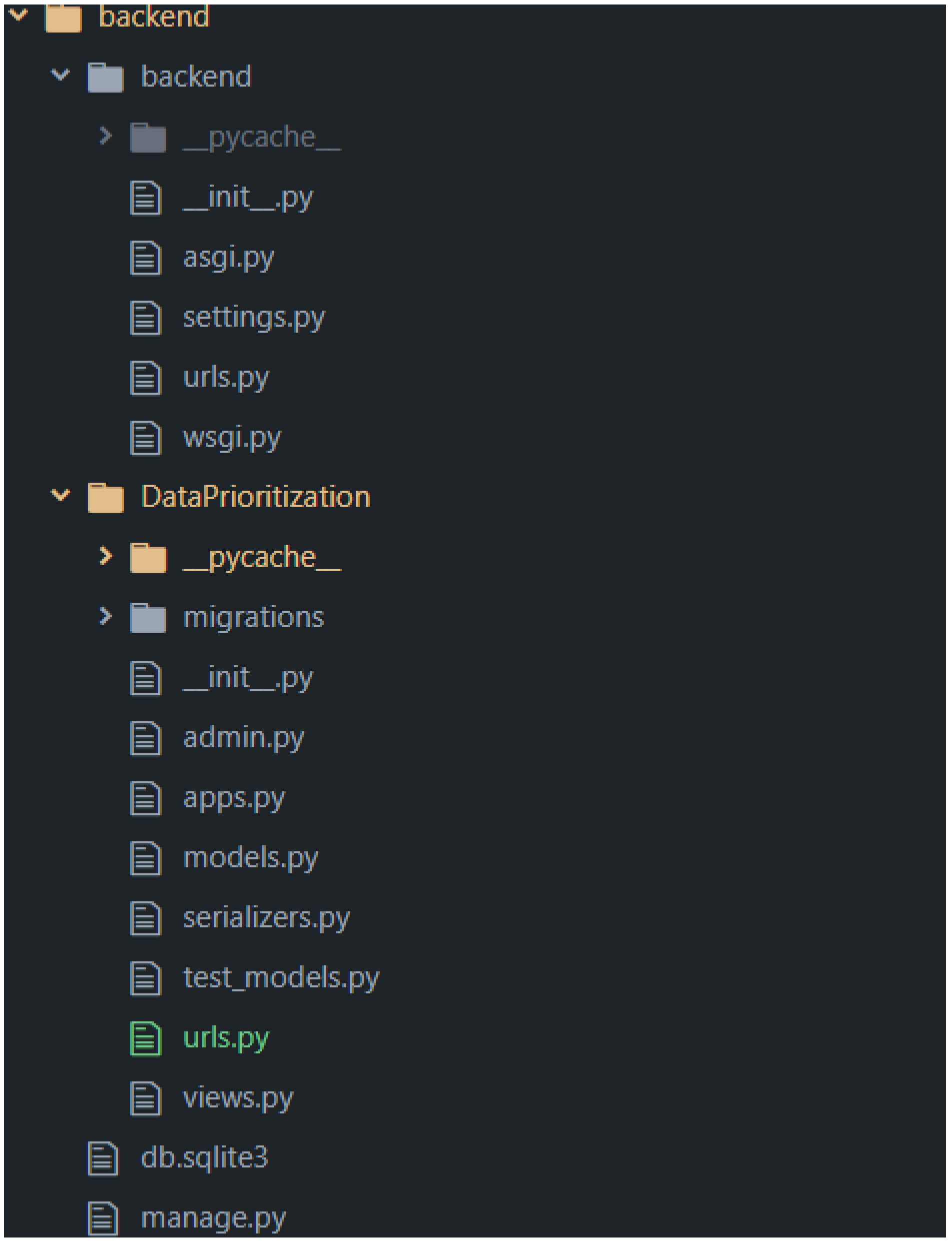
The frontend folder, as the name suggests, holds all the files that pertain to the frontend code. There are certain autogenerated files and folders, such as all of /public. The other folder /src contains all of the code for our frontend web page. The main part of the folder contains the general routing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



```
frontend
  node_modules
  public
  src
    pages
      bucket.css
      Bucket.jsx
      Buckets.jsx
      Home.jsx
      homePage.css
      ImportExport.jsx
      Nav.jsx
      rule.css
      Rule.jsx
      Rule2.jsx
      test.css
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    reportWebVitals.js
    setupTests.js
```

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__
        __init__.py
        asgi.py
        settings.py
        urls.py
        wsgi.py
    DataPrioritization
        __pycache__
        migrations
        __init__.py
        admin.py
        apps.py
        models.py
        serializers.py
        test_models.py
        urls.py
        views.py
    db.sqlite3
    manage.py
```

fig 3.2 backend file structure

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

Project Configuration/Settings

At this moment in our system, 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, the tests that will be performed are unit tests and black box system testing. The Data Prioritization system is mainly just a backend and a web frontend, so we can test all of the backend with automated unit tests and the frontend with the black box testing. 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, this will be testing the models, views, serializers, and import/export pieces of the backend. For models, they are being tested for the creation and usage of the different bucket, rule, and user models. For views, these are tested to ensure that the API endpoints are running correctly. The serializers are being tested to make sure that the data being run through is converted correctly. Finally, the import/export portion is being 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 will not be tested through automated unit tests. Instead, the frontend is being tested through our blackbox acceptance test plan.

Since the backend of the system is written in Python and using the Django REST Framework, it has been decided that the unit tests are being run using Pytest-Django. Pytest-Django is also being used for the code coverage report. When testing for code coverage of the system, the goal has been set to reach 70% coverage across all metrics, so 70% coverage for line, method, statement, and branch.

Since the first iteration is not complete yet, the unit tests have not been run, so there are no test reports to show here. There are no results to talk about yet, since tests have not been run. The lack of test results means that we need to finish up the little bit of implementation needed before we can start the unit testing.

Acceptance Testing

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

When performing these tests, we are assuming 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.

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.	
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.	

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.	
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.	
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.	
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.	
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.	

	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.	
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.	
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.	
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.	
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.	

	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.	
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.	
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.	
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.	
Test16	1) While looking at the	There should be an	

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.	
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.	

Task Plan

Author(s): <xiaochun Liang>
Reviewer(s)/Editor(s): <William Flathman>

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
Rought 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
OPR1	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	In progress
Interim Project Report	All	2/24	In progress
Individual Logs/Peer Evals	All	3/1	Not started
Sponsor Meeting	All	3/2	Not started
Sponsor Meeting	All	3/9	Not started
OPR 2	Oliver, Thedore	3/9	Not started
Sponsor Meeting	All	3/23	Not started
Rough Draft of Posters & Pies Poster	All	3/24	Not started
Draft OPR 3 Slides	seth, william	3/28	Not started
Sponsor Meeting	All	3/30	Not started
Individual Logs/Peer Evals	All	4/1	Not started
Sponsor Meeting	All	4/6	Not started
Final Presentation	seth, william	4/6	Not started
Sponsor Meeting	All	4/13	Not started
Sponsor Meeting	All	4/20	Not started

Implement

Task description	**Owner**	**Due Date**	**Status**
Iteration 0		1/31	Complete
Go thought project description	All		
set up the testing envirment on local	All		
Backend draft models.py	Dion		
Backend draft view.py	Oliver		
Draft app.js and test	seth		
Backend draft serializers.py	Theodore		
Database config set up	william		

Iteration 1		2/28	In progress
Going though django tutorial to learn django	Oliver,Dion	2/2	Complete
models	Dion	2/7	Complete
view	Oliver	2/7	Complete
frontend page and testing and	seth	2/7	Complete
serializers	Theodore	2/7	Complete
Testing database with rule, bucket and user.	william	2/13	Complete
Organize backend and merge it together	backend team	2/14	Complete
Finalize the front end to work with react	seth	2/14	Complete
Link backend and front end and database	All	2/16	Complete
Database import outport set up	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	Not started
read input config file			
add,remove,deleted rule in bucket			
set up time manager			
Iteration 3		4/30	Not started
output config file			
TBD			

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, Xiaochun Liang, Backend Developer, xliang8@ncsu.edu,980-255–1639
Dion Ybanez, Backend Developer, dybanez@ncsu.edu