NJIT ROOMFINDER

SYSTEM DOCUMENTATION

Created By:

Maciej Jedryczka	**……**	**30%**
John Rafael De Los Reyes	**……**	**30%**
Darian Moore	**……**	**15%**
Hima Patel	**……**	**10%**
Bhargav Jillepalli	**……**	**15%**

Contents

Section 1 – Project Description

1.1 Project

NJIT RoomFinder

1.2 Description

RoomFinder is a software as a service that provides the user with an up-to-date feed of all the usable NJIT classrooms. It will be using official data provided by NJIT and its associated sites that list where and when the course sections are being held. Currently, the only way to figure out if a classroom is available is by either physically going to the classroom or by searching a long course list that states when a section meets and its location. This project will provide a service to make it easier for students to find a room because of the lack of convenience that exists for them to check for unused classrooms.

Section 2 – Overview

2.1 Purpose

This project is a showcase of system integration. By fully utilizing what has been taught throughout the Information Technology course load, this project combines all the skills and knowledge necessary to create a network of computers to work together to host a software as a service. The challenge imposed is getting all the subsystems and components to work together so that it can be one large system.

2.2 Scope

The scope of the project encompasses the planning, work and the outline of what needs to be done.

2.2.2 Goals and Objectives

- A front end system that interacts with the user via HTTP and communicates with the messaging system.
    - Have 3 instances of messaging
- A messaging system (RabbitMQ) that brokers the exchange of information between the front end and the back end systems.
    - Have 2 instances of front end
- A database system (PostgreSQL, MariaDB, MySQL) that the back end uses for the storage of

information. All stateful information should be stored in a Cloud (Azure, AWS, GCP) / VMWare.
    - Have 2 instances of database
- A back end system that gathers information from your data sources, populates the database, and interacts with the messaging system.
    - Have 1 instance of backend
- These four systems will be working with each other to provide a registration and authentication system for your users.
    - Consider the password as hash (at the least) not at clear text.
    - User requests should be performed as requested by the end user.

2.2.3 Resources

- The resource used for this project is based on NJIT's schedule builder. From obtaining the CSV file used from this site, it gives all the classes and its information all in one file. This site only provides the scheduling within the next semester since it is a planning site. To access previous semesters, it can be accessed through the wayback machine.

  https://uisapppr3.njit.edu/scbldr/
- A backup source is with NJIT's course schedule web page that provides all the courses organized by their majors/subjects. The data can be accessed through a spreadsheet which is less ideal.

  https://generalssb-prod.ec.njit.edu/BannerExtensibility/customPage/page/stuRegCrseSched

2.2.3 System Addresses

Nodes	Public IPv4	Private IPv4	Zerotier
Frontend	137.184.133.3	10.116.0.2	192.168.194.50
Messaging	137.184.72.83	10.116.0.6	192.168.194.51
Backend1	137.184.72.83	10.116.0.3	
Backend2	68.183.124.145	10.116.0.5	
Database	157.230.217.246	10.116.0.4	

```
# Load balancer Port = Frontend:7007
# Site Port = Frontend/Messaging:7009
```

2.2.4 Diagrams

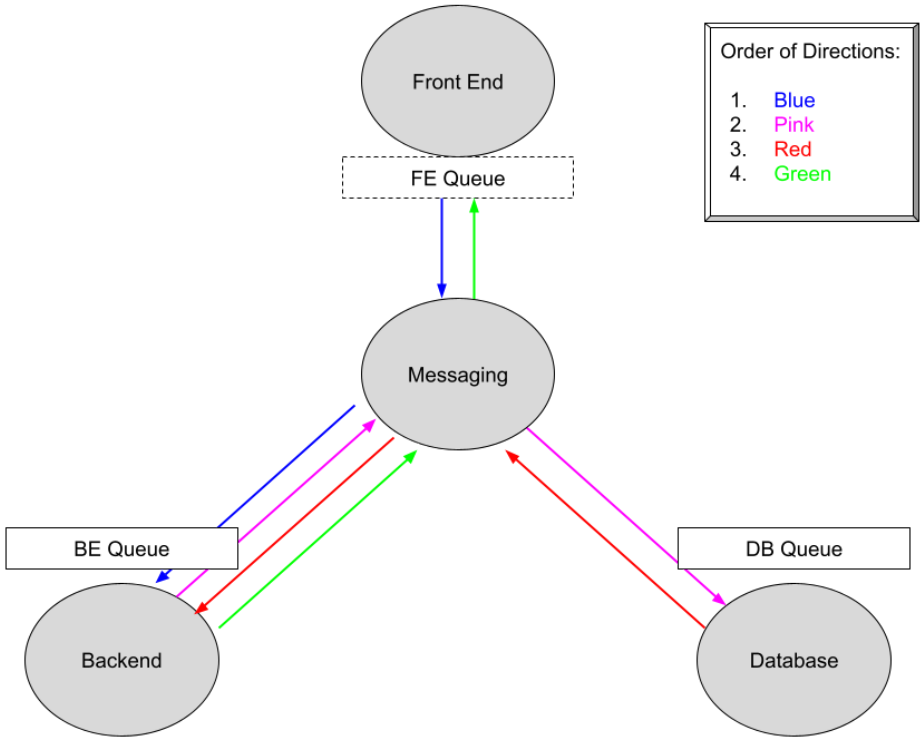


Figure 2.2.4-1: Simplified Outline of the transfer of data in the system

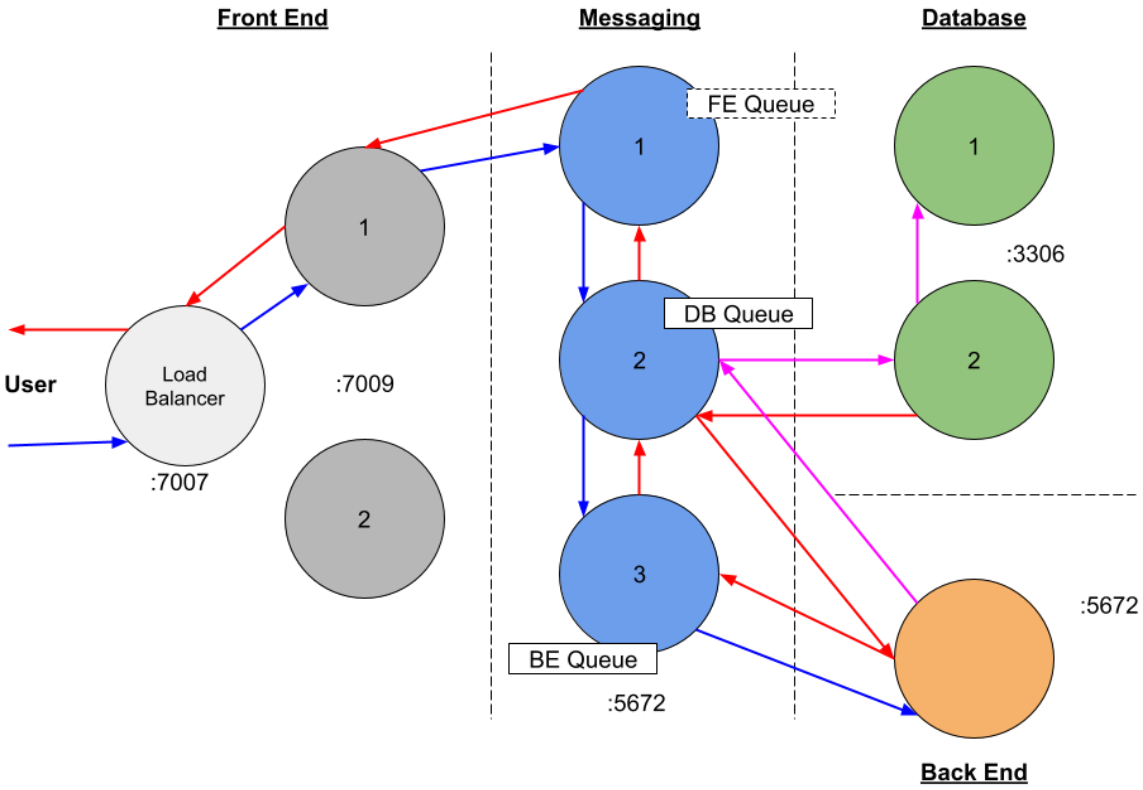


Figure 2.2.4-2: More detailed overview of the transfer of data.

1. User connects to the webpage via the load balancer and is assigned a web server to communicate with for the duration of the session defined until the user disconnects from the website.
2. When a user sends a request for data to the database. It will first create a temporary queue named after a randomly generated cookie that is also kept in the message as a return address.
3. Once the message reaches the Backend queue via the Messaging cluster, it will be scanned and filtered for any anomalous or malicious data.
4. It will then be redirected to a random Database node via the Messaging cluster.
5. The Database will process the request and send back a response message destined for the user.
6. Message will arrive at Backend, scanned and filtered, and then returned to Frontend using the return address set when the message was first created.

Section 3 – Cloud Hosting

3.1 Purpose

This project utilizes virtual machines in order to make servers for a specific role of each group member. These virtual machines can be created locally in a person's computer, or it can be all created within the cloud. We have decided to use cloud hosting so we can move the resources from our local machines to the cloud to make it less of a hassle on our own machines. By doing this in the cloud, any group member can access the virtual machines from any device as long as they have the login credentials.

3.2 Digital Ocean

What is DigitalOcean used for? DigitalOcean is a cloud hosting provider that offers cloud computing services and Infrastructure as a Service (IaaS). Through this provider, you can deploy, manage, and scale applications on the Virtual Private Cloud (VPC).

3.2 Screenshots

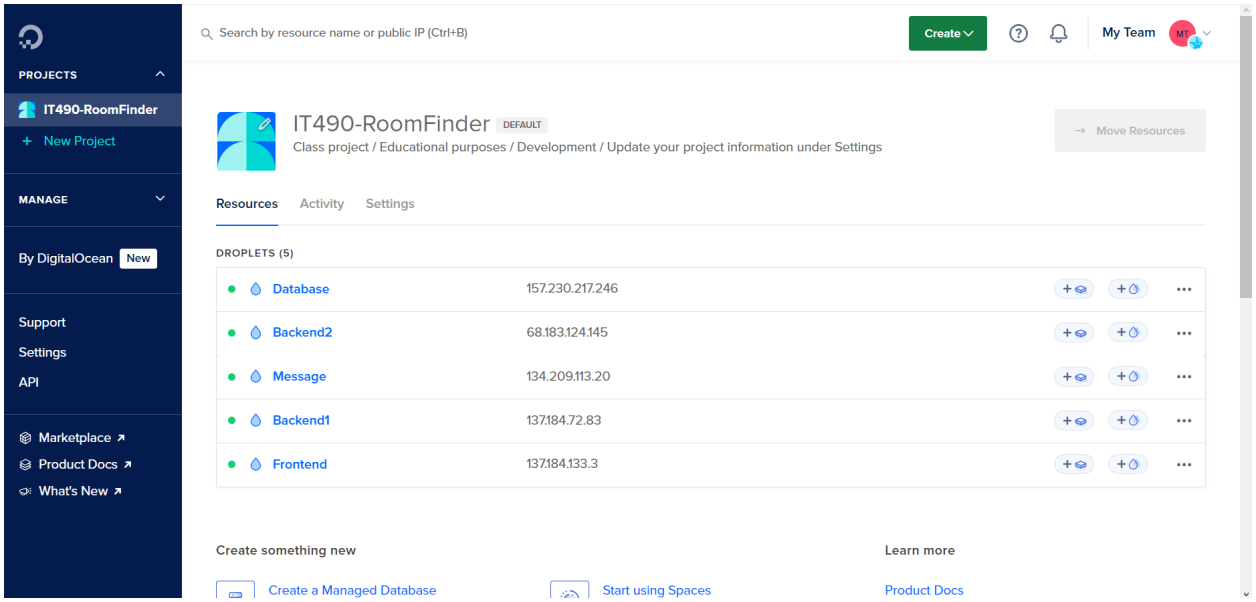


Figure 3.3-1: The dashboard of Digital Oceans

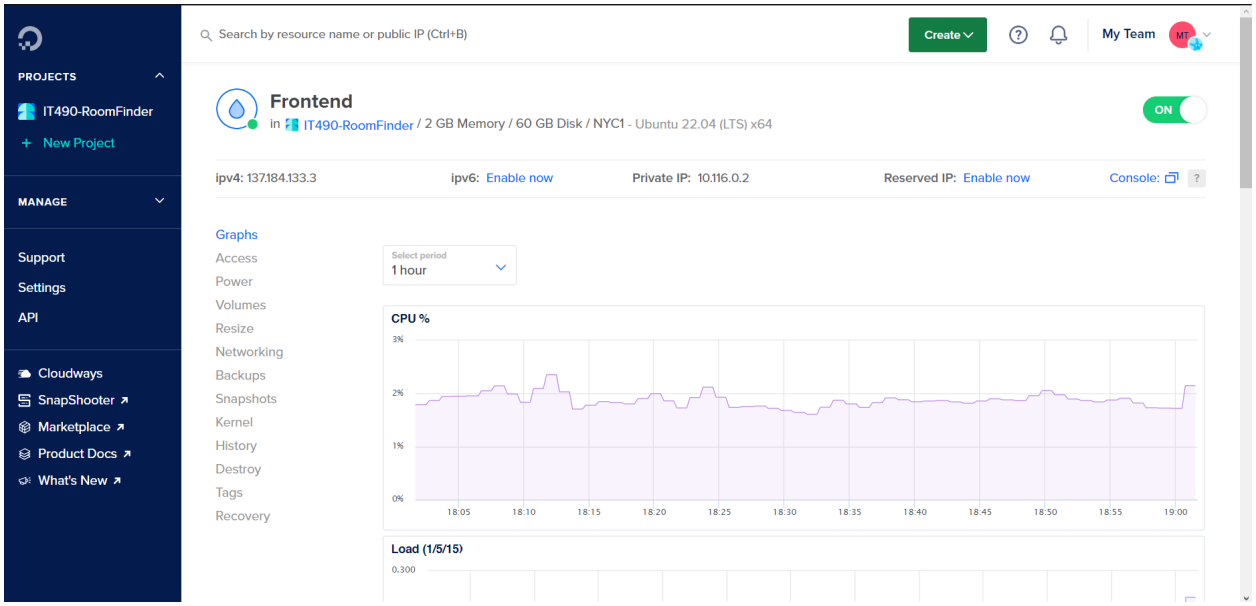


Figure 3.3-2: Frontend VM dashboard that shows system resources being used.

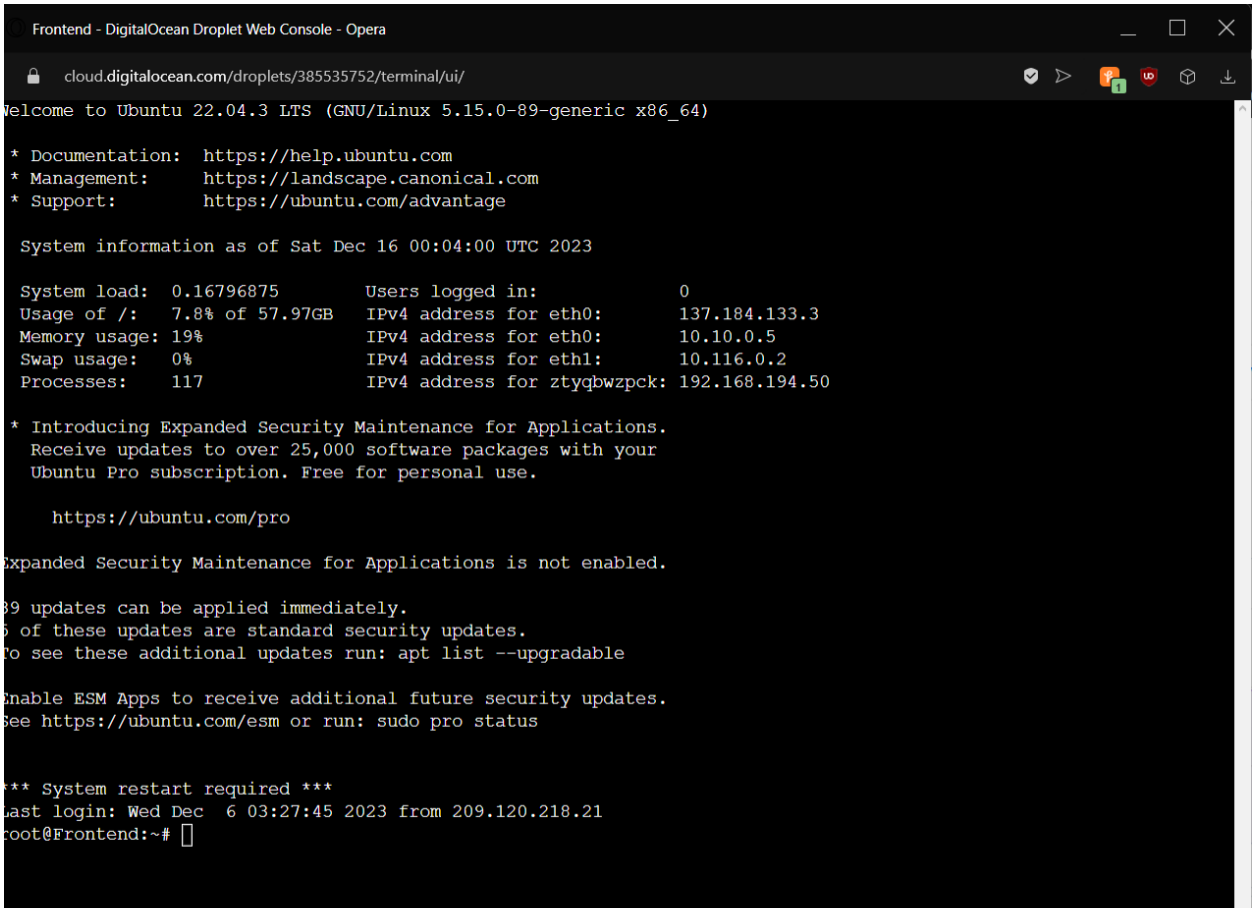


Figure 3.3-3: Frontend console provided by Digital Ocean.

Section 4 – Virtual Network

4.1 Purpose

In order for the virtual machines to be able to talk and communicate with each other, they must be in the same network. This can be achieved by using a Virtual Local Area Network (VLAN) so that the virtual machines can be able to connect and ping to each other.

4.2 ZeroTier

ZeroTier is a software-defined networking (SDN) platform that enables the creation of virtual local area networks (VLANs) over the Internet. It allows devices, regardless of their physical location, to connect to a virtual network as if they were on the same local network.

4.3 Screenshots

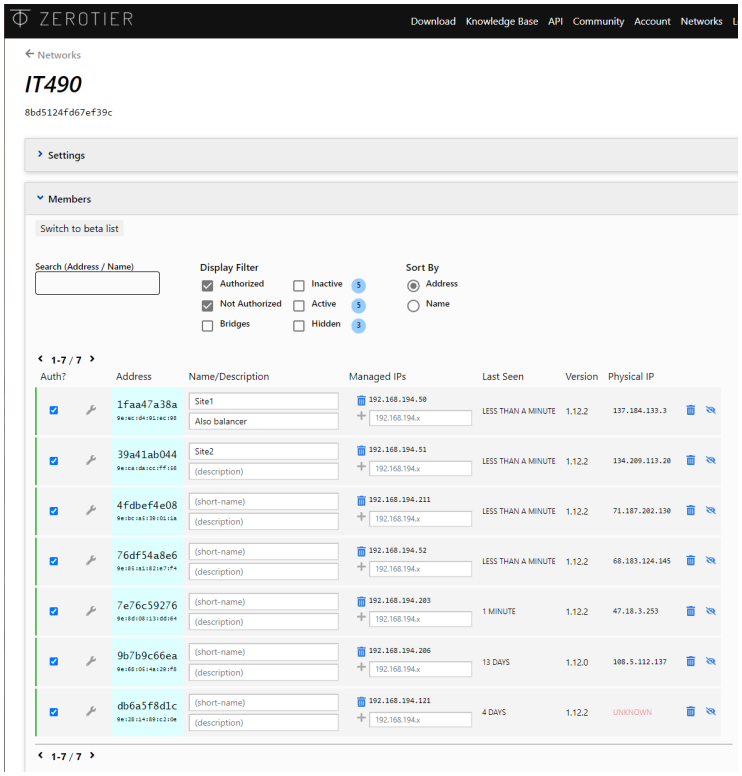


Figure 4.3-1: Zerotier Dashboard that shows all the systems under the same VLAN.

Section 5 – Frontend

Frontend is the development of the graphical user interface of a website, through the use of HTML, CSS, and JavaScript, so that users can view and interact with that website.

5.1 Application Framework

Frontend is created using multiple applications in order for our system to work properly.

5.1.1 Applications Used With

- ***5.1.1.1: PM2***
    - Process Manager 2, is a popular and widely used process manager for Node.js applications. Its primary purpose is to simplify the deployment and management of Node.js applications, ensuring that they run reliably and efficiently in production environments.
    - **Important Commands:**
        - pm2 start proc.json -- watch : what this argument lets us do is when there's a change in the files, it will restart the server. This will overtake the start command.
        - pm2 stop RoomFinder : find every process in pm2 with the name “RoomFinder” and shut it down. This is needed to stop the server for maintenance.
        - pm2 restart RoomFinder: restarts the server.js process
        - pm2 reload RoomFinder: reload the RoomFinder process with 0 downtime
        - pm2 reload RoomFinder --watch: reload RoomFinder process with 0 downtime and enable *watch*
        - pm2 monitor: open the console logs of the currently running process in PM2
- ***5.1.1.2: NodeJS***
    - Node.js is an open-source, server-side JavaScript runtime environment that allows developers to build and run server-side applications using JavaScript.
    - This is the framework being used for this project.
- ***5.1.1.3: Webpage Engine - EJS***
    - Embedded Javascript Templating (EJS) is a templating engine used by Node.js. Template engine helps to create an HTML template with minimal code. Also, it can inject data into an HTML template on the client side and produce the final HTML. EJS is a simple

templating language that is used to generate HTML markup with plain JavaScript. It also helps to embed JavaScript into HTML pages.

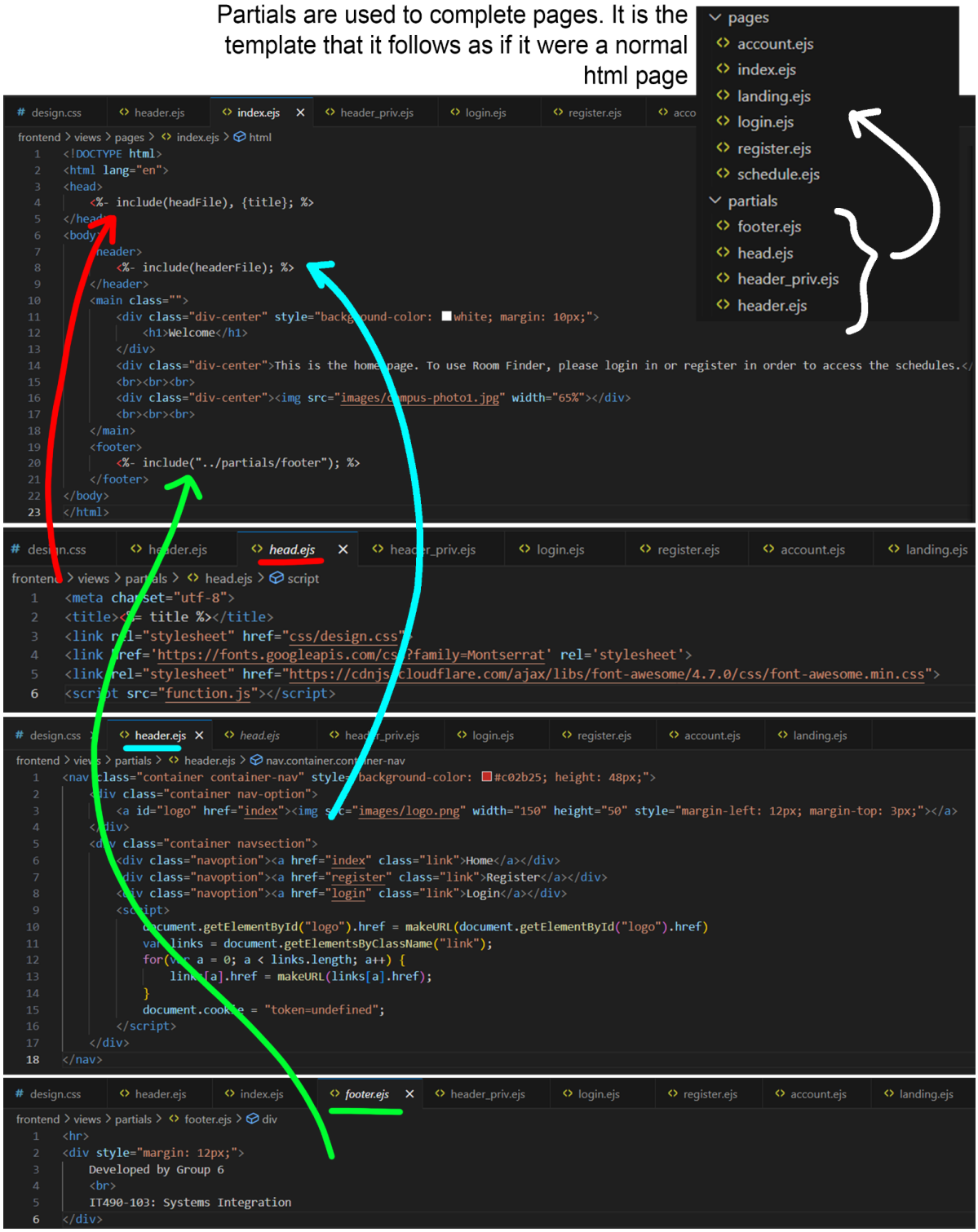


Figure 5.1.1.3-1: Shows the makeup of the index page using EJS templating

5.2 Files and Code Structure

5.2.1 Promises

A promise is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value. Promises provide a way to work with asynchronous code in a more structured and readable manner.

5.2.2 Redirection

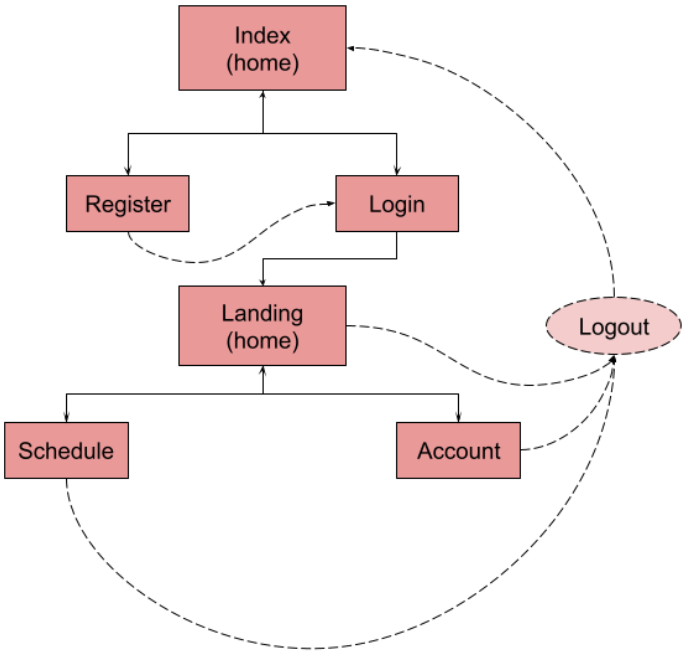
- ***5.2.2.1: Tokens - Java Web Tokens - (JWTs)***
    - A Java Web Token (JWT) is a compact, URL-safe means of representing claims between two parties. It is often used for authentication and information exchange in web development. The claims in a JWT are encoded as a JSON object, and the token is usually digitally signed to ensure its integrity and authenticity

- ***5.2.2.2: Privileged Headers***
    - A privileged header in the terms of this project means a set of elements within the header that denotes whether or not the user is logged in. In a scenario where the user is not logged in. It would show the *Home*, *Register*, and *Login* links. And when the user is logged in, they will see *Home*, *Account*, and *Schedule* links.

5.3 Website Architecture

The structural design and organization of a website. It encompasses the way in which web content is arranged, navigated, and presented to users.

- This diagram presents the web pages as rectangle boxes. The oval button represents a button that can be used on any logged in page.
- The arrows give a direction in where the user can move from page to page.



5.3.1 Template Building

Template Building refers to the use of EJS. Instead of using .html files, .ejs replaces the files in order to allow for the use of template building. By looking at figure 5.1.1.3-1, you can see how the section of code has <%- include( ); %> . This will have a variable inside that references the .ejs file that continues the html code. Essentially this separates a portion of the webpage code so that it can allow for an easier change or update to multiple pages.

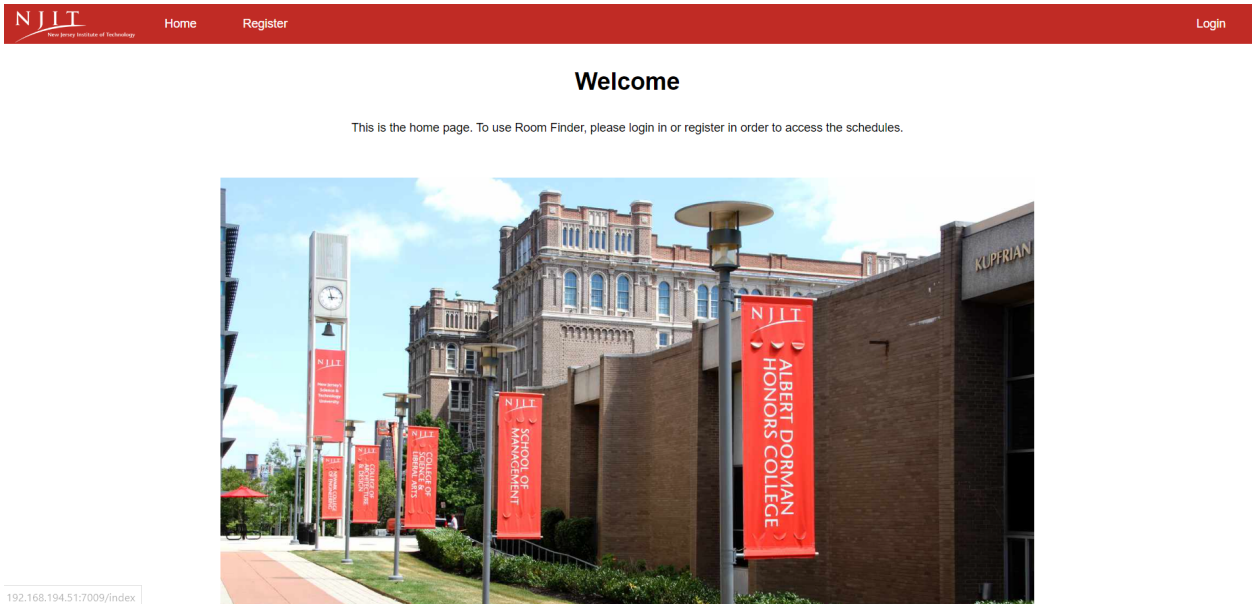
```
<head>
    <%- include(headFile); %>
</head>
```

```
<meta charset="utf-8">
<title><%= title %></title>
<link rel="stylesheet" href="css/design.css">
<link href='https://fonts.googleapis.com/css?family=Montserrat' rel='stylesheet'>
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">
<script src="function.js"></script>
```

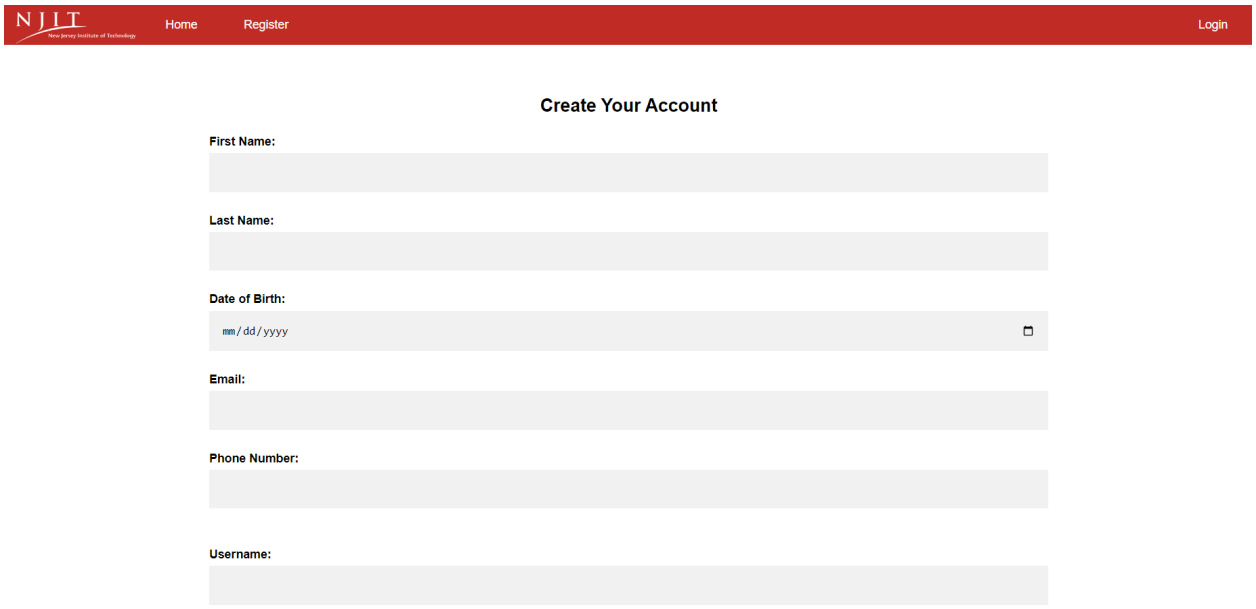
5.3.2 Navigation

The navigation bar is quite simple. There are only 3 options to choose from when the user is not logged in: Home, Register, and Login. Home encompasses the index.ejs page, Register is register.ejs, and login is login.ejs page.

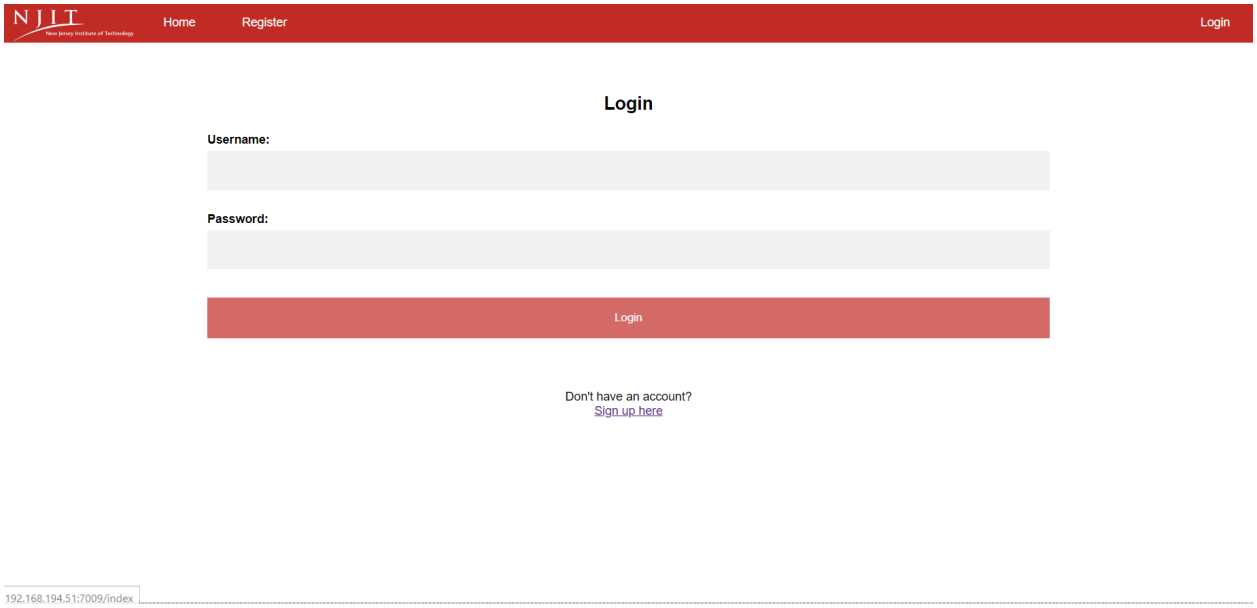
Home is a basic welcome page. Its main use is to differentiate when a user is logged in or not. It will also tell people that in order to view the schedules they need to use.



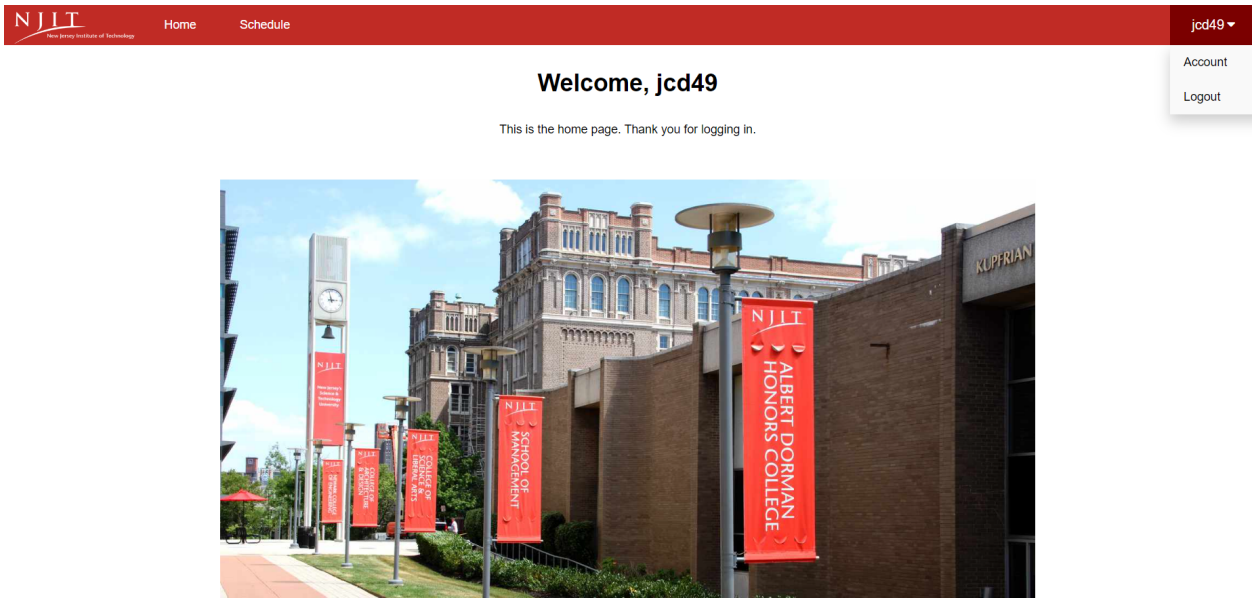
Register is where the user will enter their information to create an account. This has all the basic information needed when creating an account.



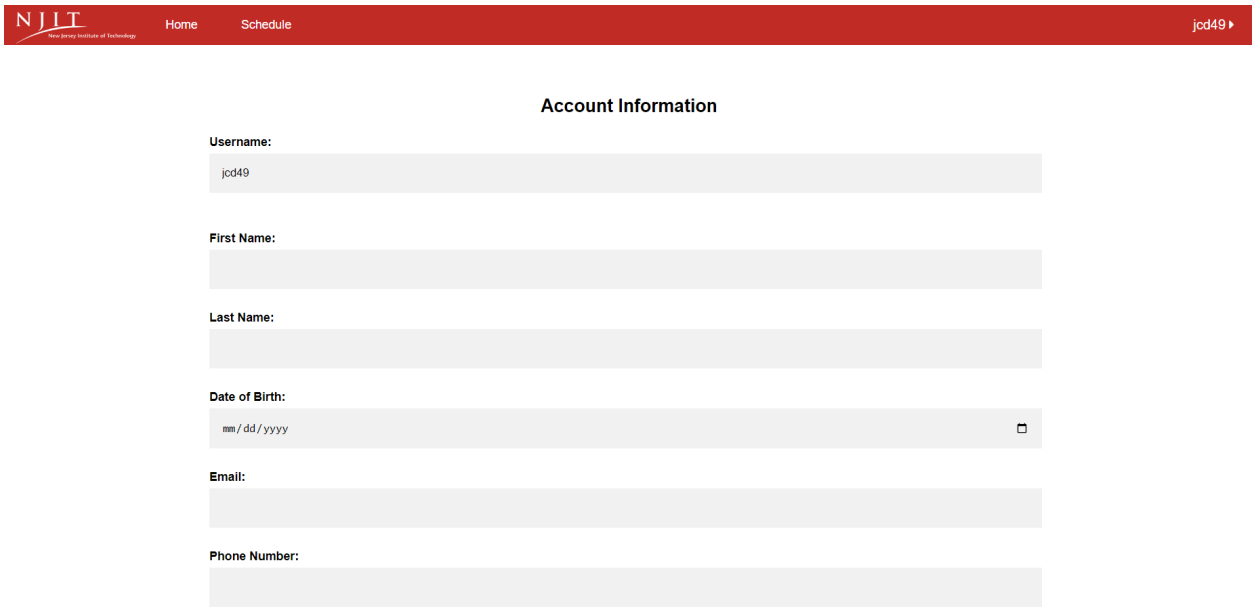
Login is where those who already made an account will go. Once the user logs in, it will change the navigation bar to use the header_priv.ejs file which is where privileged users will get an updated navbar.



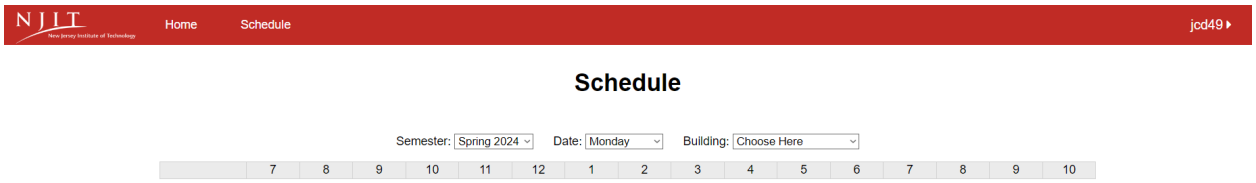
Landing page is the home page where privileged users will be redirected to. This will replace the Home page link on the nav bar. This will also change the Register option to Schedule. And Login will be replaced with the user's username that acts as a dropdown menu. The dropdown menu will give the option for the Account page and the Logout button.

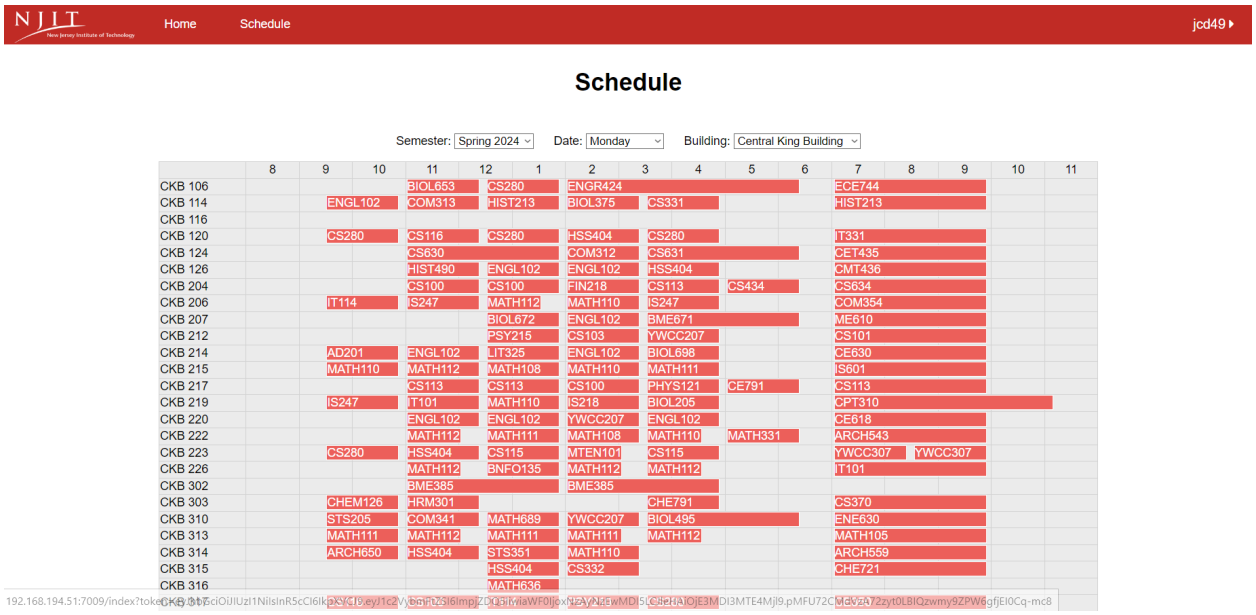


Account page is where the user can update all their information except for the username. The username input will be restricted from being edited. It will use security questions to confirm change to the account information.



Schedule page starts off as an empty page. Once the user selects the semester, date and building, the table will be populated with class information.





5.3.3 Scheduling

The main functionality of the software is to display the scheduling information in a table that gives all the rooms available in the building.

- ***5.3.3.1: Table Generation***
    - Table generation is based on the data saved onto the database from data scraping the course information. Essentially, the for loop will take in the course's specific information each time and fill it in the right spot accounting for room number and time. A more detailed explanation of the code will be provided in Section 7 of Backend.

5.4 Security

Security is important in any scenario where it involves a user's data. Security can be done throughout multiple areas in the system, but Front End's security will be through input validation.

5.4.1 Input Validation

Input validation is used in the Register, Login and Account page. The validation consists of html input validation when it comes to have required input boxes, patterns to follow, and password confirmation comparisons.

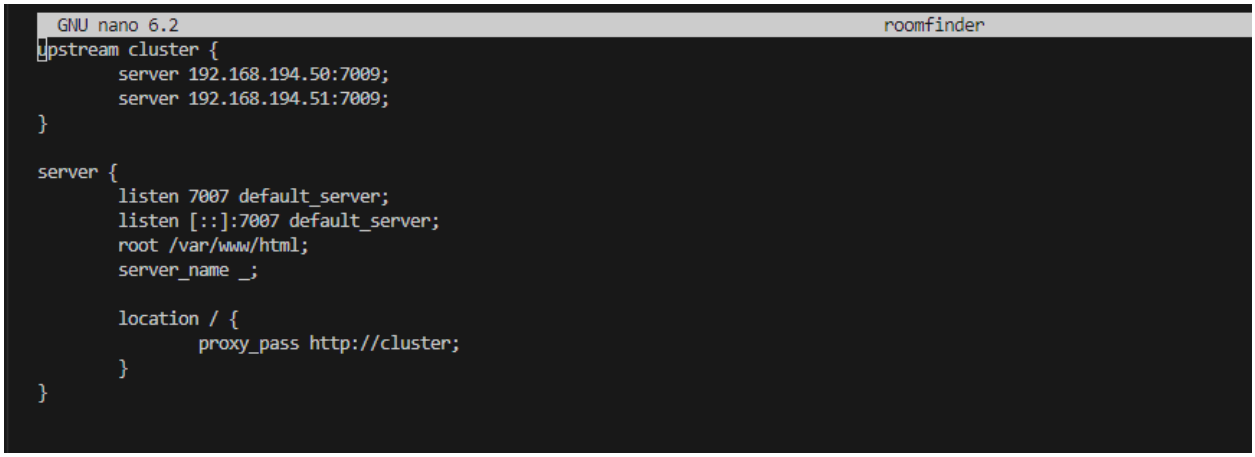
5.5 Load Balancing

Load balancing is a technique used to distribute incoming network traffic across multiple servers to ensure no single server is overwhelmed, thereby improving reliability and availability of a web application.

5.4.1 Nginx

Nginx, a popular web server and reverse proxy server, is often used as a load balancer due to its efficiency, flexibility, and scalability.

- The configuration file for nginx is located at /etc/nginx/sites-available/RoomFinder
  This shows how load balancing is taken care of using the two nodes of server 192.168.194.50 and 192.168.194.51 (That is the IP of).



Section 6 – Messaging

6.1 Clustering

Clustering in Messaging works by having one or more nodes connecting to one primary node. The only specialty of having a node being the primary node is that every other node will need to copy its stored cookie. During a clustering operation, each node can see every other node as well as their queues, exchanges, and system resources.

6.2 Queues

A queue is a collection of messages in the form of a stack. First In, First Out.

- ### 6.2.1 Quorum Queue
    - A quorum queue is a relatively new type of replication queue in which it introduces the concept of achieving a majority agreement among a set of nodes. Note: when a program tries to initialize a queue on a node, it may be ignored if the queue already exists. An error if a program tries to redefine a queue with different attributes.

    - #### 6.2.2.1 Layout
        - Frontend: When a user sends a request into the messaging cluster, typically destined for the *Database* queue, the web server creates a new quorum queue with a 15 second expiration date. After, the web server will expect a response and promptly closes the channel that was used to connect to that queue; allowing it to die.
        - Backend: Backend attempts to initialize a *Backend* queue and a *Database* queue in preparation to transmit messages to via the messaging cluster. Any message that's decoded and found to have a keyword related to *Request* in the action attribute, of the json object that is stored in the message, will be sent to the *Database* queue. And if a keyword related to *Response* is found, it will redirect the message back to the *Frontend* queue that the user generated.
        - Database: On this node, the *Database* and *Backend* queue is defined again and awaits messages to arrive in the *Database* queue. When a message is received, it will generate a response to the request and send it off back to the *Backend* queue.

Section 7 – Backend

7.1 Code Structure

7.1.1 Asynchronous

Programming asynchronously on the backend node allows for higher throughput rates opposed to its synchronous counterpart. In the figure below, this setup allows for any errors that happen to occur during execution to loop back and restart the process automatically; preventing any downtime. Another way

asynchronous programming is used is when executing commands related to messaging functionality. In this project when we try to send a message, the program will wait for a response before continuing on to the next line, thus allowing commands that require hold of the interpreter to run beside asynchronous functions.

```
#Main Function
async def main() -> None:
    flag = True #Connect flag. Used to decide if program should reconnect to another
    another node
    while flag:···

    #Used to run the asynchronous flow for the program to follow
    try:
        await asyncio.Future()
    finally:
        await connection.close()


#Start
if __name__ == "__main__":
    asyncio.run(main())
```

Figure 7.1.1-1

7.1.2 Messaging Connection

Messaging connection is determined by feeding AMQP urls from a list into a connection object from Pika. Connecting to a node is done inside of a while loop that contains a try-except pair. Until a connection is established and the commands that proceed have successfully executed, it will keep attempting to connect to a random url from the server list.

- *7.1.2.1: Failover*
    - When a connection fails midway through processing a message, our connection object has features to prevent data loss and allows the server to pick up where it left off. Of course after a connection fails, it will trigger an error, telling the interpreter to restart the try-except pair and subsequently attempt a random url in the server list.

7.1.3 Redirection

The role of the Backend node is to redirect messages to their respective destinations and to validate any input coming from the user.

- *7.1.3.1: Frontend to Database & Vice Versa*

  When a user requests data or requests to edit their data in the database, everything goes through the Backend first. Same with any response that the user expects from the database. Once a user's message is received by the Database, the message will be converted into a request and sent back through Backend and to the user.

- *7.1.3.1: Input Validation*

  Information passed from the user side is by far one of the biggest threats to the entire system. Users attempting to violate the website via SQL injection raises the possibility of them accessing resources they should not be accessing and or the ability to run their own code to compromise the system running the script. By scanning and filtering messages sent by the user to Backend, we can do all that we can to protect the project's most valuable assets, the Database.
  On the returning side of the project, the threat can be equally as dangerous. In the scenario that a user somehow manages to enter code into the database, any user that wishes to interact with said database can be vulnerable to whatever that code may do.

7.1.4 Code Explanation

- *7.1.4.1: download_data.py*

```
courseURL = "https://uisapppr3.njit.edu/scbldr/include/datasvc.php?p=/" #URL used to scrap the
data directly from the website
urlretrieve(courseURL, "courses.json") #Download the data file and name it as courses.json

#Open courses.json and store the values into a variable
with open("courses.json", "r") as f:
    lines = f.readlines()

#Remove some characters encapsulating the data so we can parse it
with open("new_courses.json", "w") as f:
    lines[-1].strip("\n")
    f.write(lines[-1].replace("define(", "")[:-2])

#Reopen and parse with temporary name
f = open("new_courses.json")
courseJSON = hjson.load(f)
```

This section of code downloads the data file from the URL, processes, and converts it to dictionary objects.

```
newCourse = {} #Initialize a dictionary to store the reformatted data into
for course in courseJSON["data"]: #Access the data atribute of the og dict
    for sec in course[3:]: #Go into the section of the dictionary that has the data we actually
    want
        secClasses = sec[-1] #Get the last thing in the list which is where the class meet times
        exist as well as other information
        for secClass in secClasses:
            start = secClass[1] // 60 + (secClass[0] - 2) * (24 * 60) #Convert the start time of
            the class from seconds of the week to minutes of the week and start from sunday
            end = secClass[2] // 60 + (secClass[0] - 2) * (24 * 60) #Convert the end time of the
            class from seconds of the week to minutes of the week and start from sunday

            #Add an entry into newCourses is that courses doesn't exist yet
            #Add the schedule data into that course into it as well
            if secClass[-1] not in newCourse:
                newCourse[secClass[-1]] = [(course[0], course[1], start, end)]
            else:
                newCourse[secClass[-1]].append(tuple((course[0], course[1], start, end)))

del newCourse[" "] #Delete online classes that were added
```

Here, we're running through the data provided by NJIT. The stored data is formatted by having each key being a course at NJIT and the value in the pair is the information of every section of the course. By taking only what we need, we create a new dictionary object to store the key, being the room number, and section information as the value. Section information meaning course number, course name, start, and end times. Of course, since we want to provide our users with information about rooms inside of NJIT, we omit all of the information stored in " " (what NJIT names the rooms as).

```
#New dictionary used to send to the messaging server and then database
courseMessage = {}
courseMessage["data"] = newCourse #Store the formatted data into the data attribute of the dict
courseMessage["term"] = courseJSON["term"].replace(" ", "") #Add the term of the semester to the
term attribute

#Reformat and add the time the data was updated originally from the website to the update
attribute of the message
courseMessage["update"] = datetime.strptime(" ".join(courseJSON["update"].replace(",", "").split
(" ")), '%a %b %d %Y %H:%M').strftime('%Y-%m-%d %H:%M:%S')

#Store the dict into a new file
with open("courses_processed.json", "w") as f:
    json.dump(courseMessage, f)

#Delete unnecessary files
os.remove("new_courses.json")
os.remove("courses.json")
```

We create a new dictionary object that will work as the message to be sent to the Database. We store the processed information in the “data” field, the term/semester name in the “term” field, and an acceptable format of the last update of the data in the “update” field. Then we finish off by saving the new dictionary to a file and removing unnecessary files.

- ***7.1.4.2: send_data.py***

```
#Run the script to download the data if it doesn't exist
if not os.path.isfile("courses_processed,json"):
    os.system("python3 download_data.py")

#Open and parse the course schedule data
fileJson = json.load(open("courses_processed.json"))


messJson = {"action": "buildingDataUpdate", "body": json.dumps(fileJson)} #Cram the course data in a message JSON object the schedule data from the file

```

This code checks if a file containing course schedule data exists. If not, it attempts to download the data using an external script. Then, it opens and parses the downloaded or existing JSON file and prepares a new JSON object (messJson) with information about the action (building data update) and the JSON data from the file.

```
# IP addresses for servers in the messaging cluster
servers = [
    "amqp://backend:lin67^&User@10.116.0.2:5672",
    "amqp://backend:lin67^&User@10.116.0.6:5672",
    "amqp://backend:lin67^&User@10.116.0.3:5672"
]

#Get a random ip address from the messaging server list
def getNode():
    server = servers[random.randint(0, len(servers) - 1)]
    print("Selected: ", server)
    return server

#A function used to send messages to the messaging cluster taking a channel object, message, and target queue name
async def pubMessage(channel, body, route):
    print("Sent: ", route)
    await channel.default_exchange.publish(
        Message(body=json.dumps(body).encode()),
        routing_key = route
    )

```

This code provides utility functions for interacting with a messaging cluster. The getNode function allows for selecting a random messaging server from the configured list, and the pubMessage function facilitates sending messages with specified content and routing information to the messaging cluster.

```python
#Main Function
async def main() -> None:
    flag = True #Connect flag. Used to decide if program should reconnect to another another node
    while flag:
        try:
            userConnection = await connect(getNode()) #Create a connection of type, robust, and select a random node
            userChannel = await userConnection.channel() #Create a channel for that connection
            backendQ = await userChannel.declare_queue("Backend", durable=True, arguments={"x-queue-type": "quorum"}) #Create a database queue
            databaseQ = await userChannel.declare_queue("Database", durable=True, arguments={"x-queue-type": "quorum"}) #Create a backend queue

            await pubMessage(userChannel, messJson, "Database") #Send the data loaded from the file to the database
            await userConnection.close() #Close the connection
            flag = False #If all goes, well, then there is no need to replay this. Allowing normal crashes to occur else where
        except:
            print("Error, connecting to new node")

    #Used to run the asynchronous flow for the program to follow
    try:
        await asyncio.Future()
    finally:
        await userConnection.close()


if __name__ == "__main__":
    asyncio.run(main())
```

The script connects to a messaging node, creates channels and queues, sends a message to a specified queue ("Database"), and then closes the connection. It handles errors during the connection process and attempts to connect to a new node if an error occurs.

- *7.1.4.3: backend.py*

```python
#Function to run when the backend recieves a message
async def onMessage(message: AbstractIncomingMessage) -> None:
    async with message.process():
        messageBody = message.body.decode("utf-8")


        messJson = json.loads(messageBody) #Parses the message it recieved
        if type(messJson) == str:
            messJson = json.loads(messJson) #Parses it again because the same method can do multiple things. In this case it goes from """ to "" to the JSON object

        userConnection = await connect_robust(getNode()) #Create a new connection to the messaging cluster
        channel2 = await userConnection.channel() #Creates a new channel for that connection

        action = messJson["action"]

        #Ungodly set of if statements that redirect the messages to their respective destinations basedd
        #If it has Req, send to database. If it has Res, send to user's temporary queue
        if "loginReq" == action:
            await pubMessage(channel2, messJson, "Database")
        elif "loginRes" == action:
            await pubMessage(channel2, messJson, messJson["userID"])

        elif "registerReq" == action:
            await pubMessage(channel2, messJson, "Database")
        elif "registerRes" == action:
            await pubMessage(channel2, messJson, messJson["userID"])

        elif "scheduleReq" == action:
            await pubMessage(channel2, messJson, "Database")
        elif "scheduleRes" == action:
            await pubMessage(channel2, messJson, messJson["userID"])

        elif "editReq" == action:
            await pubMessage(channel2, messJson, "Database")
        elif "editRes" == action:
            await pubMessage(channel2, messJson, messJson["userID"])
        #If the action is not known, catch it and do nothing
        else:
            print("Bad Action: ", action)

        await userConnection.close() #Close connection
        await asyncio.sleep(1) #Delay becuase it doesn't work without it
```

This function processes incoming messages, routes them based on the specified actions, and interacts with a messaging cluster using an asynchronous programming model.

7.2 Getting Course Data

The goal of this project is to provide a service to any user that wishes to use it. For this project our service depends on the source of data, which is NJIT itself. (https://uisapppr3.njit.edu/scbldr/)

7.2.1 Downloading

- ***7.2.1.1: Source***
    - NJIT Schedule Builder is the perfect source for this project because it has everything we would need inside a single file that it freely serves. (https://uisapppr3.njit.edu/scbldr/?datasvc=)

- ***7.2.1.2: Formatting***
    - Although NJIT was kind enough to freely provide this data file, it does come with some problems. For one, it was made to work with their framework, not our's. To fix that, we're using Python to remove the characters that encapsulate and to parse everything into a dictionary object so we can easily adjust the formatting for our purposes.

7.2.2 Processing

We find that the data format of the school seems inefficient, almost messy. But they make it work. For our database to read and understand the data we're feeding it, we want to reformat the data into course schedule per classroom, instead of course per classroom.

- ***7.2.2.1: Reformatting***
    - Going through every course in the dataset, we take note of the course, classroom, start time, and end time. Example: {roomNumber: [courseCode, courseName, start, end]}. And we keep appending if we come across any duplicate roomNumbers.
    - At the end of the json, we add the date that the data was originally updated by NJIT as well as the semester name.

7.2.1 Uploading

The processed data is then sent to Database via the Messaging cluster. The theoretical size limit for a message is 2GB and the most we could ever send will never exceed 10MB.

Section 8 – DataBase

8.1 Overview

Describe the data contained in databases and other shared structures between domains or within the scope of the overall project architecture

8.1.1 MariaDB

MariaDB is an open-source relational database management system (RDBMS) that is designed to be highly compatible with MySQL. After issues involving poor connectivity between our VM's, our team decided to migrate over to MariaDB instead.

8.2 Table Design

```
MariaDB [(none)]> SHOW COLUMnS FROM users FROM RoomFinder;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| fname    | varchar(255) | YES  |     | NULL    |       |
| lname    | varchar(255) | YES  |     | NULL    |       |
| dob      | date        | YES  |     | NULL    |       |
| email    | varchar(255) | YES  |     | NULL    |       |
| phone    | varchar(255) | YES  |     | NULL    |       |
| username | varchar(255) | NO   | PRI | NULL    |       |
| password | varchar(255) | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
7 rows in set (0.003 sec)
```

```
MariaDB [(none)]> SHOW COLUMNS FROM Spring2024 FROM RoomFinder;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| room  | varchar(255) | NO   | PRI | NULL    |       |
| data  | longtext    | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
2 rows in set (0.004 sec)

MariaDB [(none)]>
```

8.3 Code Structure

8.3.1 Asynchronous

Programming asynchronously on the backend node allows for higher throughput rates opposed to its synchronous counterpart. In the figure below, this setup allows for any errors that happen to occur during execution to loop back and restart the process automatically; preventing any downtime. Another way asynchronous programming is used is when executing commands related to messaging functionality. In this project when we try to send a message, the program will wait for a response before continuing on to

the next line, thus allowing commands that require hold of the interpreter to run beside asynchronous functions.

```
#Main Function
async def main() -> None:
    flag = True #Connect flag. Used to decide if program should reconnect to another
    another node
    while flag:···

    #Used to run the asynchronous flow for the program to follow
    try:
        await asyncio.Future()
    finally:
        await connection.close()


#Start
if __name__ == "__main__":
    asyncio.run(main())
```

Figure 8.3.1-1

8.3.2 Function Calls

1.
```
#Grab a random ip address from the servers list
def getNode():
        server = servers[random.randint(0, len(servers) - 1)]
        print("Selected: ", server)
        return server
```

‘getNode’ - The getNode function selects a random server IP address from the ‘servers’ list

2.
```
#A function used to send messages to the messaging cluster taking a channel object, message, and target queue na
async def pubMessage(channel, body, route):
        print("Sent To: ", route)
        await channel.default_exchange.publish(
                Message(body=json.dumps(body).encode()),
                routing_key = route
        )
```

‘pubMessage’ - The pubMessage function is used to publish a message to the messaging cluster. It accepts as parameters a channel object, message body, and target queue name.

```
        )

#Function to run when the backend recieves a message
async def onMessage(message: AbstractIncomingMessage) -> None:
        async with message.process():
                messageBody = message.body.decode("utf-8")


                messJson = json.loads(messageBody) #Parses the message it recieved
                if type(messJson) == str:
                        messJson = json.loads(messJson) #Parses it again because the same method can do multiple things.

                userConnection = await connect_robust(getNode()) #Create a new connection to the messaging cluster
                channel2 = await userConnection.channel() #Creates a new channel for that connection

                action = messJson["action"]

                #Ungodly set of if statements that redirect the messages to their respective destinations basedd
                #If it has Req, send to database. If it has Res, send to user's temporary queue
                if "loginReq" == action:
                        await pubMessage(channel2, messJson, "Database")
                elif "loginRes" == action:
                        await pubMessage(channel2, messJson, messJson["userID"])

                elif "registerReq" == action:
                        await pubMessage(channel2, messJson, "Database")
                elif "registerRes" == action:
                        await pubMessage(channel2, messJson, messJson["userID"])

                elif "scheduleReq" == action:
                        await pubMessage(channel2, messJson, "Database")
                elif "scheduleRes" == action:
                        await pubMessage(channel2, messJson, messJson["userID"])

                elif "editReq" == action:
                        await pubMessage(channel2, messJson, "Database")
                elif "editRes" == action:
                        await pubMessage(channel2, messJson, messJson["userID"])
                #If the action is not known, catch it and do nothing
                else:
                        print("Bad Action: ", action)

                await userConnection.close() #Close connection
                await asyncio.sleep(1) #Delay becuase it doesn't work without it
```

3. ‘onMessage’ - The onMessage function is a callback function that is called when a message is received by the backend. It processes the incoming message, determines the action type, and routes it to the appropriate destination in the messaging cluster based on the action type.

8.4 Table Update

The process of updating is dependent on where the data arrives from. From the Database perspective, a message will arrive in its queue and it will process it based on the action. A request from the Frontend would eventually run a function in the Database script. These functions include *addUser* and *editUser*. A request from the Backend would run the *updateCourses*. The Database is also dependent on the Backend for making sure there isn’t anything malicious in the messages it's sending.

8.5 Clustering

Clustering for this project relies on Galera-4, a MariaDB extension. Galera-4 lets us automate a cluster at the same time allowing us to change every behavior of it. In our case we’ve configured Galera-4 in a Master-Master mode and we can easily add/drop by shutting down the server on the primary node and editing the configuration that lives on the same node.

8.5.1 Master-Master

Master-Master works by having any change committed to one database occur to every other database in the cluster. Not by sending the entire database on change, but instead only sending the differences.

8.5.2 Failover

When one node happens to go online from either a crash or done manually, nothing will happen to the other nodes. Once that node comes online again, it will request any changes from the cluster in order to be up to date and verify its contents. This behavior is based on a majority vote. If the majority of databases have the same contents, then those databases will be used as a reference to update or repair a database.

Section 9 - Setting Up

Installation for Messaging is by far the most difficult. But at the same time not hard at all. This should tell you that the rest of the installation guides for the rest of the nodes should be a walk in a park.

9.1 Frontend

Npm Installation:

1. Download and import the Nodesource GPG key
   ```
   sudo apt-get update
   sudo apt-get install -y ca-certificates curl gnupg
   sudo mkdir -p /etc/apt/keyrings
   curl -fsSL https://deb.nodesource.com/gpgkey/nodesource-repo.gpg.key | sudo gpg
   --dearmor -o /etc/apt/keyrings/nodesource.gpg
   ```
2. Create Deb repository
   ```
   NODE_MAJOR=20
   echo "deb [signed-by=/etc/apt/keyrings/nodesource.gpg]
   https://deb.nodesource.com/node_$NODE_MAJOR.x nodistro main" | sudo tee
   /etc/apt/sources.list.d/nodesource.list
   ```
3. Run Update and Install
   ```
   sudo apt-get update
   sudo apt-get install nodejs -y
   ```
4. Upload frontend folder to /home from Github repository
   When inside the frontend folder run this to validate the packages that are used in the server.
   ```
   npm init -y
   ```

Pm2 Installation:

1. Install
   ```
   npm install pm2 -g
   ```

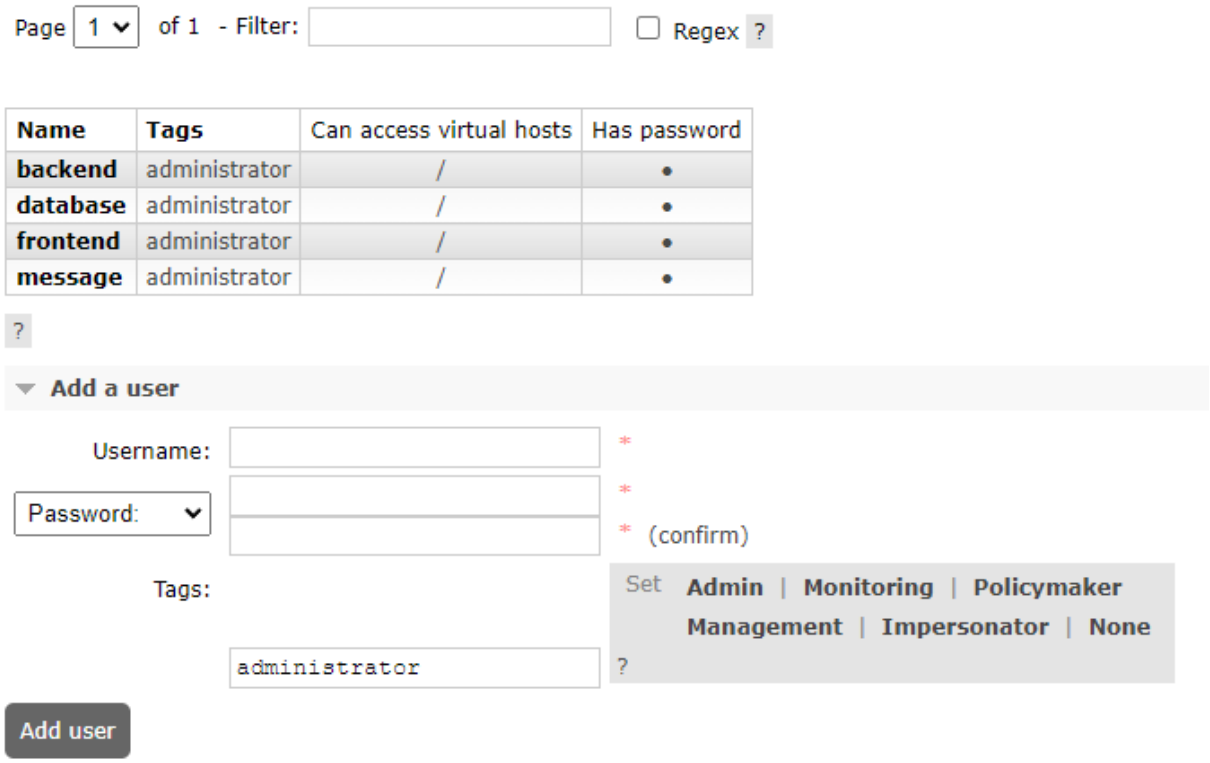
Nginx Installation:

1. Install
   sudo apt update
   sudo apt install nginx
2. Add configuration file
   nano /etc/nginx/sites-available/roomfinder
   Paste in code from nginx configuration file from Github

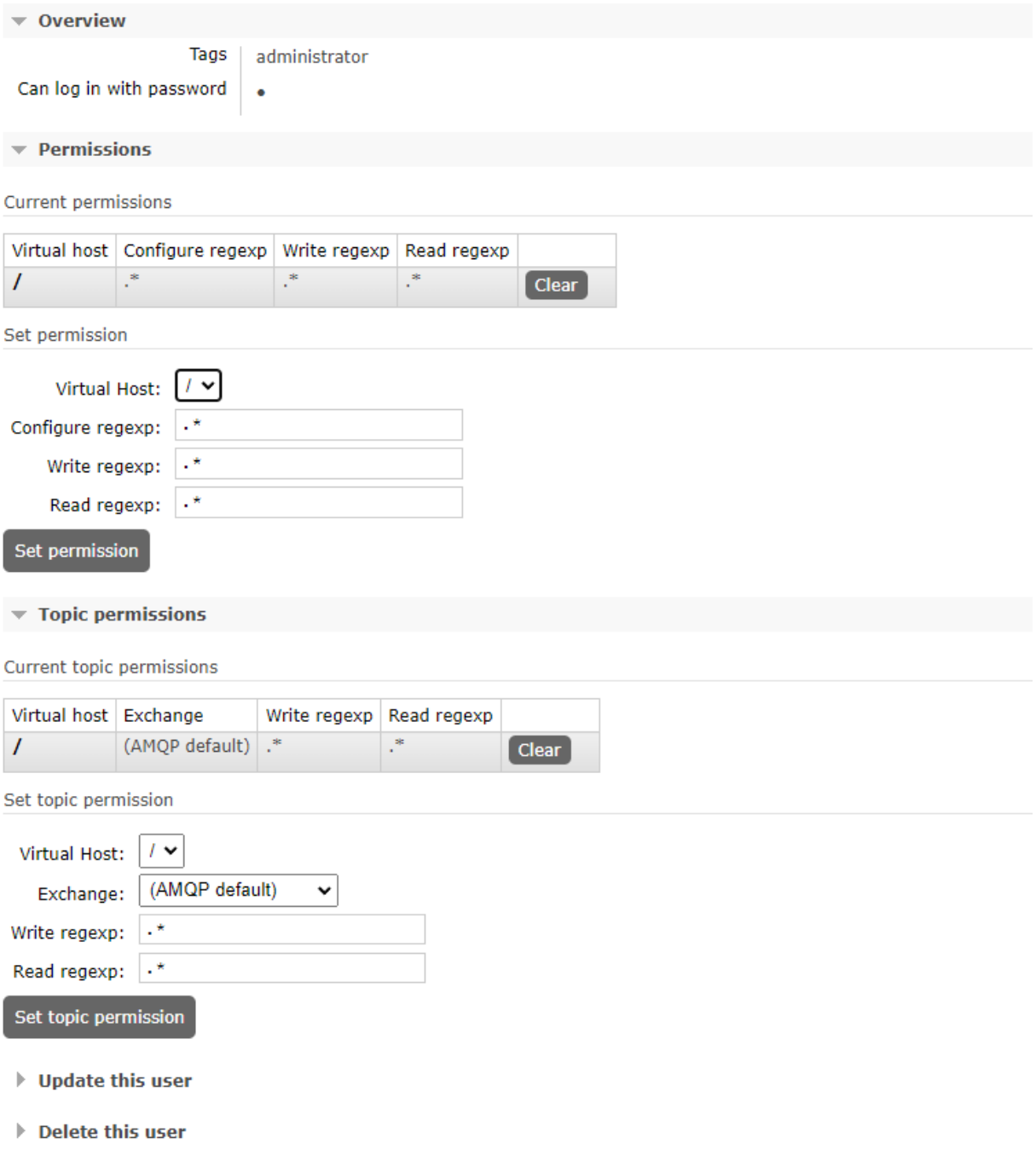
9.2 Messaging

Rabbitmq Installation:

1. Download and run the Rabbitmq quickstart.sh file
   chmod +x quickstart.sh
   .quickstart.sh
2. Start the server
   systemctl start rabbitmq-server
3. Enable management plugin
   rabbitmq-plugins enable rabbitmq_management
4. Create users with privileges
   Login to your web interface using localhost:15672
   Login using username: guest and password: guest
   Add users for Frontend, Backend, Messaging, and Database using the figure below as a guide.



Cast the highest permissions/privleges to all of these users



5. Clustering

After running the quickstart scripts on all Messaging nodes(no need to create users and permissions), Grab the primary node's cookie in:

/var/lib/rabbitmq/.erlang.cookie

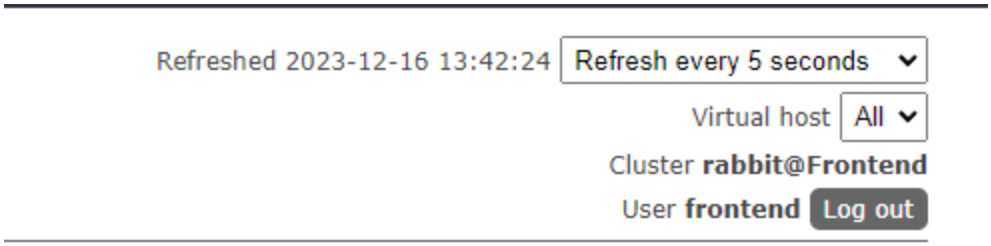
Take that cookie and overwrite the non-primary node's cookie

After, restart and stop the app on the non-primary nodes:

systemctl restart rabbitmq-server

rabbitmqctl stop_app

Take the hostname of the primary node from the top. An example:



Attempt to join that primary node

rabbitmqctl join_cluster rabbit@Frontend

Start the app on the non-primary nodes

rabbitmqctl start_app

Check the status of the cluster

rabbitmqctl cluster_status

9.3 Backend

Download the Backend folder from the Github repository to /home

1. Install python

   sudo apt update

   sudo apt install python3
2. Install requirements

   pip install -r requirements.txt

9.4 Database

Download the Database folder from the Github repository to /home

1. Install python

   sudo apt update

   sudo apt install python3
2. Install requirements

   pip install -r requirements.txt

Database and Galera 4

1. Import MariaDB repository

   sudo apt-get install apt-transport-https curl

   sudo mkdir -p /etc/apt/keyrings

   sudo curl -o /etc/apt/keyrings/mariadb-keyring.pgp 'https://mariadb.org/mariadb_release_signing_key.pgp'
2. Edit mariadb sources file

nano /etc/apt/sources.list.d/mariadb.sources

3. Paste sources into that file

   # MariaDB 11.2 repository list - created 2023-12-16 18:50 UTC
   # https://mariadb.org/download/
   X-Repolib-Name: MariaDB
   Types: deb
   # deb.mariadb.org is a dynamic mirror if your preferred mirror goes offline. See https://mariadb.org/mirrorbits/ for details.
   # URIs: https://deb.mariadb.org/11.2/ubuntu
   URIs: https://ftp.osuosl.org/pub/mariadb/repo/11.2/ubuntu
   Suites: jammy
   Components: main main/debug
   Signed-By: /etc/apt/keyrings/mariadb-keyring.pgp

4. Install MariaDB-Server

   sudo apt-get update
   sudo apt-get install mariadb-server

5. Repeat this process for all nodes that are involved with Database Clustering

Clustering

1. Install Galera-4

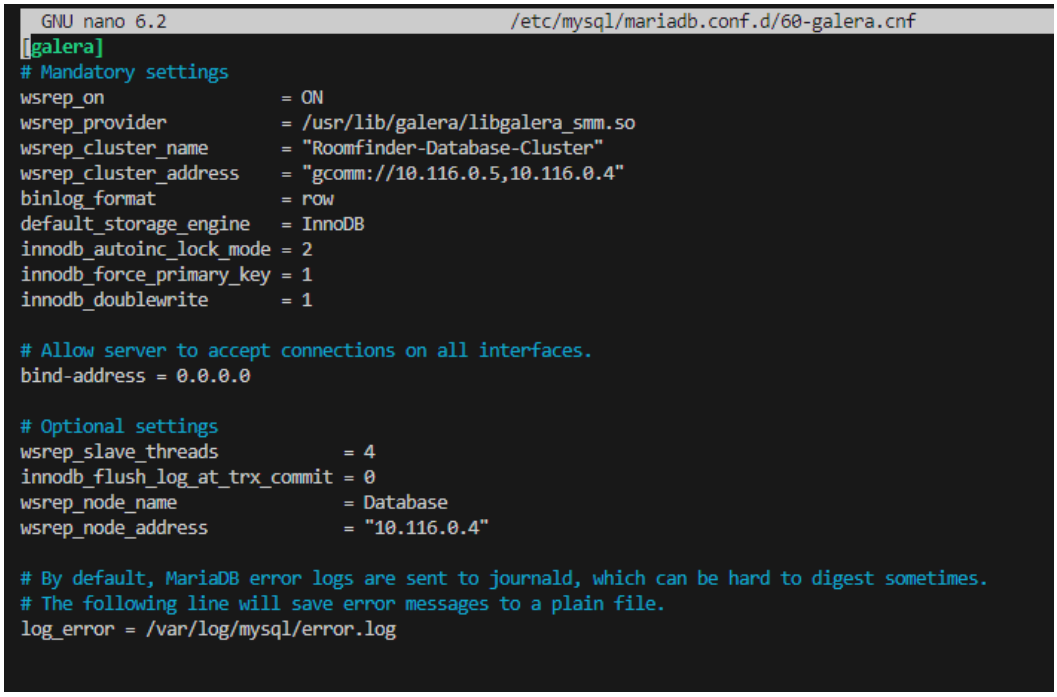
   sudo apt install galera-4

2. Edit cluster config file

   sudo nano /etc/mysql/mariadb.conf.d/60-galera.cnf

3. Paste in the cluster configuration file contents into the file

   Example:



4. Allow mysqld plugin to be allowed through ubuntu's AppArmor
   cd /etc/apparmor.d/disable/
   sudo ln -s /etc/apparmor.d/usr.sbin.mysqld
   sudo systemctl restart apparmor
5. Start and check the cluster
   sudo systemctl stop mariadb
   sudo galera_new_cluster
   mysql -u root -p
   show status like 'wsrep_cluster_size';
6. To add or drop nodes from the cluster, follow the guide in the reference for Galera 4 installation

Section 10 - Known Bugs and Issues

Scalability Concerns:

- Address potential scalability issues as the user base grows, ensuring the system can handle increased loads.

Cross-Browser Compatibility:

- Test and resolve any compatibility issues with different web browsers to ensure a consistent user experience.

Data Security Measures:

- Enhance data security measures to protect sensitive user information and ensure compliance with data protection regulations.

User Feedback Mechanism:

- Establish a system for users to provide feedback on room conditions, equipment functionality, and overall satisfaction.

Localized Language Support:

- Add support for multiple languages to accommodate a diverse user community within the school.

System Downtime Mitigation:

- Develop strategies to minimize system downtime during maintenance and updates to avoid disruptions for users.

Training and Onboarding for Users:

- Address any issues related to user onboarding and training to ensure a smooth adoption process for new users.

User Authentication and Authorization Issues:

- Review and strengthen the authentication and authorization mechanisms to prevent unauthorized access.

User Interface Accessibility:

- Continuously evaluate and enhance the accessibility of the user interface for individuals with disabilities.

Software Integration Challenges:

- Resolve any challenges related to integrating Roomfinder with existing school management

software or other relevant systems.

User Data Privacy Compliance:

- Regularly audit and update privacy measures to align with evolving data protection laws and regulations.

Performance Optimization:

- Identify and optimize any performance bottlenecks, ensuring the system operates efficiently under varying loads.

Support for Different School Configurations:

- Address issues related to schools with unique configurations or structures to ensure Roomfinder's adaptability.

Cross-Platform Compatibility:

- Ensure the software is compatible with various operating systems and devices to provide a seamless experience for all users.

User Interface Consistency:

- Maintain consistency in the user interface design to prevent confusion among users and improve overall usability.

Section 11 – References

Any documents which would be useful to understand this design document or which were used in drawing up this design.

Installing frontend stuff

Npm - (https://github.com/nodesource/distributions)

Nginx -

(https://www.digitalocean.com/community/tutorials/how-to-install-nginx-on-ubuntu-20-04)

Installing messaging stuff

Rabbitmq - (https://www.rabbitmq.com/download.html)

For database stuff

MariaDB - (https://www.linuxbabe.com/mariadb/install-mariadb-10-5-ubuntu)

Galera 4 - (https://www.linuxbabe.com/mariadb/galera-cluster-ubuntu)

Section 12 – Glossary

Glossary of terms / acronyms

1. **Front End** - The user interface and user experience component of a software application that interacts directly with users also known as the "client-side" of a software application. It is in

charge of informing users and facilitating their interaction with the application.

2. **Back End** - The back end of a software application, also known as the "server-side," is in charge of data storage, business logic, and server-side operations.
3. **Messaging** - Messaging is a communication mechanism that allows different software components, systems, or processes to exchange data, commands, or events.
4. **Database** - A database is a structured collection of data that is organized in such a way that efficient data storage, retrieval, and manipulation is possible.
5. **Node** - A node is typically a device or point of connection within a network.
6. **JWTs (Java Web Tokens)**- A compact, URL-safe means of representing claims between two parties. Often used for authentication and information exchange in web development.
7. **EJS (Embedded JavaScript)**- A templating language that helps embed JavaScript into HTML pages.
8. **Promises**- Objects that represent the eventual completion or failure of an asynchronous operation and its resulting value. They provide a structured and readable way to work with asynchronous code.
9. **Redirection**-
    - **Tokens** - JWTs: Java Web Tokens used for authentication and information exchange.
    - **Privileged Headers**- Elements within the header indicating whether the user is logged in.
10. **Website Architecture**- The structural design and organization of a website, encompassing how web content is arranged, navigated, and presented to users.
11. **Template Building (EJS)**- Refers to the use of EJS (Embedded JavaScript) for building templates, allowing for easier changes or updates to multiple pages.
12. **Navigation Bar**- The bar containing navigation options for users, typically displaying links like Home, Register, and Login.
13. **Scheduling**- The main functionality of the software to display scheduling information in a table, showing available rooms in a building.
14. **Input Validation**- Security measure used in the Register, Login, and Account pages to ensure required input boxes, patterns, and password confirmation comparisons.

15. **Load Balancing**- A technique distributing incoming network traffic across multiple servers to improve reliability and availability of a web application.
16. **Nginx**- A popular web server and reverse proxy server, often used as a load balancer due to its efficiency, flexibility, and scalability.
17. **Clustering**- Connecting nodes in a messaging system, allowing them to share information, queues, and system resources.
18. **Queues**- Collections of messages in the form of a stack, typically following a First In, First Out (FIFO) order.
19. **Quorum Queue**- A type of replication queue that introduces the concept of achieving a majority agreement among a set of nodes.
20. **Asynchronous Programming**- Programming technique allowing for higher throughput rates and automatic error handling.
21. **Galera-4**- A MariaDB extension enabling cluster automation and changes in behavior.
22. **Scalability Concerns**- Addressing potential issues related to accommodating increased loads as the user base grows.
23. **Cross-Browser Compatibility**- Testing and resolving compatibility issues with different web browsers to ensure a consistent user experience.
24. **Data Security Measures**- Enhancing measures to protect sensitive user information and ensure compliance with data protection regulations.
25. **User Feedback Mechanism**- Establishing a system for users to provide feedback on room conditions, equipment functionality, and overall satisfaction.
26. **Localized Language Support**- Adding support for multiple languages to accommodate a diverse user community.
27. **System Downtime Mitigation**- Developing strategies to minimize system downtime during maintenance and updates.
28. **Training and Onboarding for Users**- Addressing issues related to user onboarding and training to ensure a smooth adoption process for new users.
29. **User Authentication and Authorization Issues**- Reviewing and strengthening mechanisms to prevent unauthorized access.

30. **User Interface Accessibility**- Continuously evaluating and enhancing the accessibility of the user interface for individuals with disabilities.
31. **Software Integration Challenges**- Resolving challenges related to integrating Roomfinder with existing school management software or other relevant systems.
32. **User Data Privacy Compliance**- Regularly auditing and updating privacy measures to align with evolving data protection laws and regulations.
33. **Performance Optimization**- Identifying and optimizing performance bottlenecks to ensure efficient system operation under varying loads.
34. **Support for Different School Configurations**- Addressing issues related to schools with unique configurations or structures to ensure Roomfinder's adaptability.
35. **Cross-Platform Compatibility**- Ensuring the software is compatible with various operating systems and devices.
36. **User Interface Consistency**- Maintaining consistency in the user interface design to prevent confusion among users and improve overall usability.