Major Project - Team 4

Akanksha Arun - aa2013

Eshaan Mathur - em919

Eoin O'Hare - eso25

Sidhu Arakkal - ssa150

Mohammad Awais Zubair - maz106

Github Link:
https://github.com/radical-teach/major-project-group-4

Problem Statement

Clubbing in times of COVID! YelpHelp is a company that specializes in recommending bars based on quality and occupancy. YelpHelp approaches you to help design a system that will allow them to recommend a bar that is both cool (to be seen!), as well as secure (not above a given occupancy or density factor). Obviously, everyone can't go to the highest-rated bar, as it would be unsafe. And no one wants to go to the safest bar, i.e, which by definition is the bar where no one else goes! In other words, if there are too many people it's unsafe; if there are too few people at the bar, it's uncool. YelpHelp!

Sprint 1

Overview

In this sprint we went over the requirements (functional and non functional) as a team and made a list of us cases that we would be needing for this project. We also implemented the subsystems to store bar information, as well as functionality including adding bars, deleting bars, and getting bar data.

Last Commit: https://github.com/radical-teach/major-project-group-4/pull/10

Requirements Engineering

Non-Functional Requirements

ID	*Requirement and Description*
NF1	Capability to handle large number of requests (300+)
NF2	Capacity to store large amounts of bar data (100+)
NF3	User friendly I/O interface to interact with data
NF4	Insert data into database within 1 seconds of request
NF5	Remove data from database within 1 seconds of request
NF6	Get list of relevant data from database within 1 second of request
NF7	Allow only admin users the ability to add and remove bars

Functional Requirements

ID	*Requirement and Description*	*Backlog ID*
F1	Handle user request for adding a bar to the database	
F2	Handle user request for removing a bar from the database	
F3	Handle user request to query bar database and return all relevant data	
F4	Return relevant confirmation/error for each user request	
F5	Implement authenticated users through a log-in system	

F6	Store sensitive data in a properly secured format	

Assumptions

Add Bar

- Traffic is not greater than capacity
- The primary keys for bars is names and address
- The primary keys cannot be updated once inserted
- Bars can only be added using this endpoint, they cannot be deleted

Delete Bar

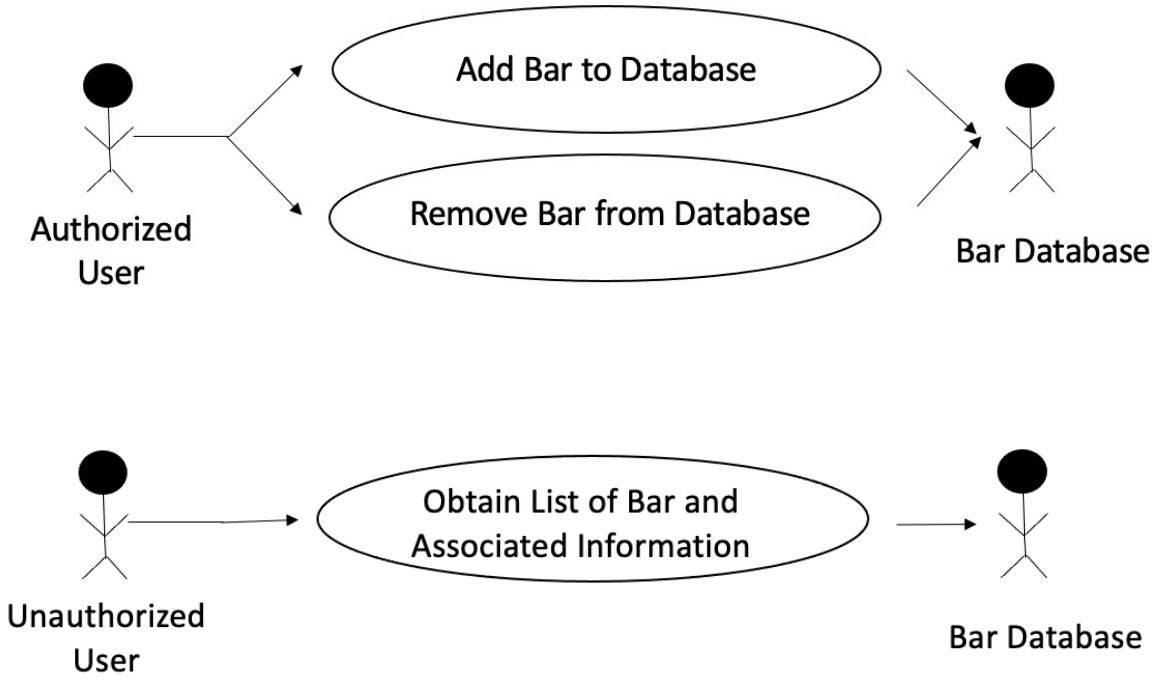
- The primary keys (name and address) must be provided to delete the bar

Get Bar

- Bars are returned as an array of JSON objects

System Modeling

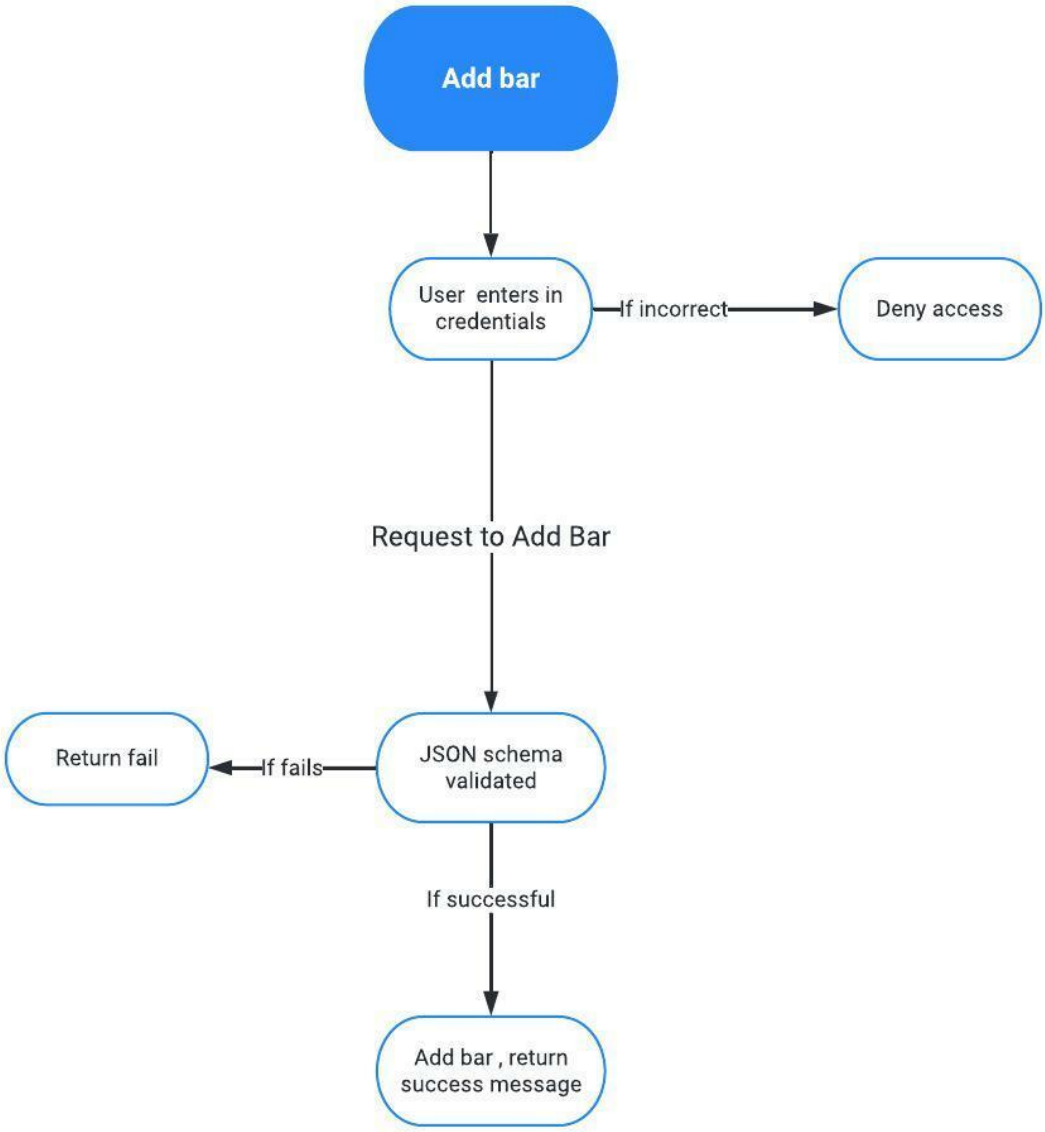
Use Cases



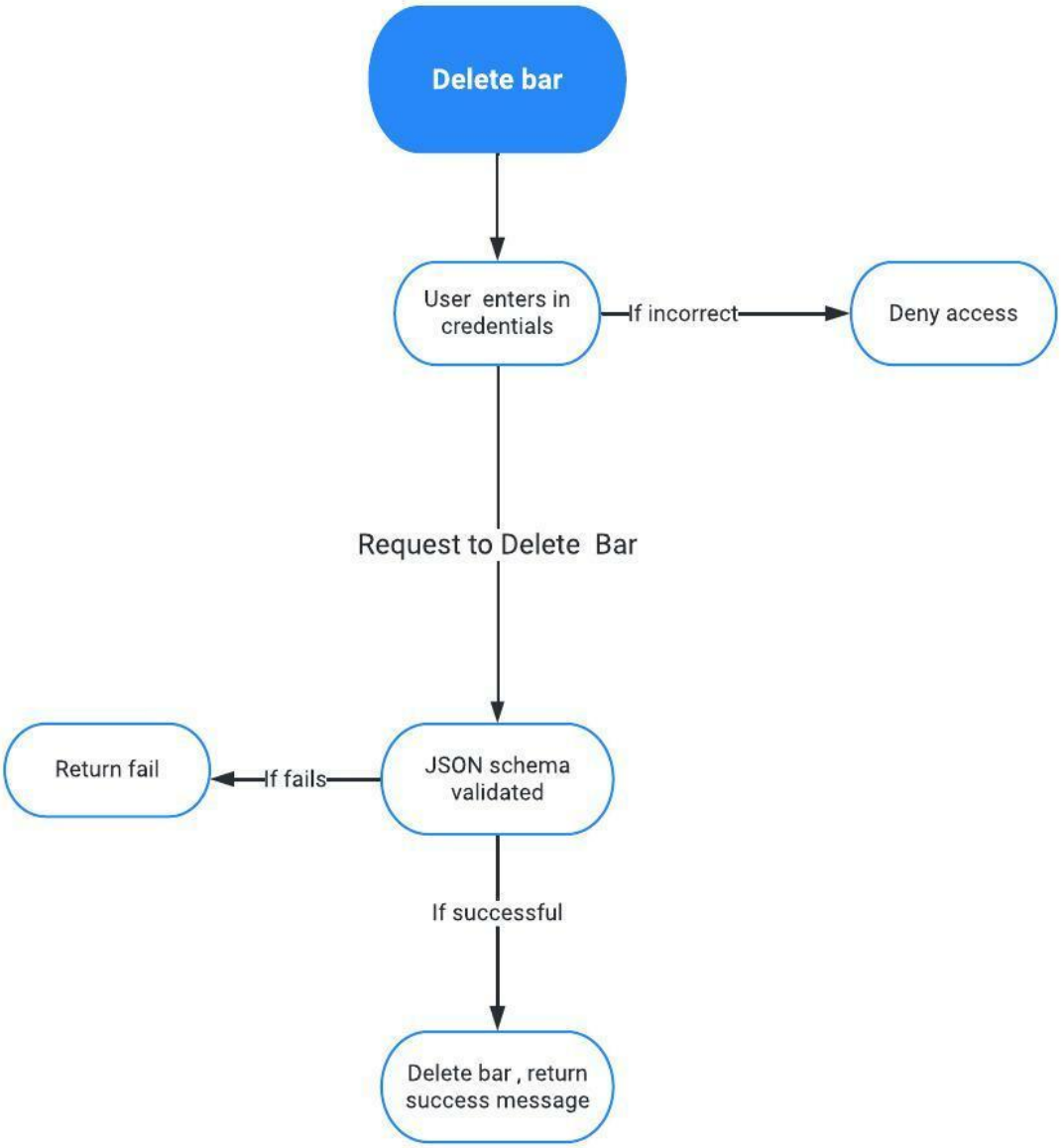
Our use case for our process of modifying and obtaining bar data would involve two kinds of users and the bar database. The authorized user would be able to add bars and delete bars from the database, while the unauthorized users could only access the list of all bars.

Conceptual Model

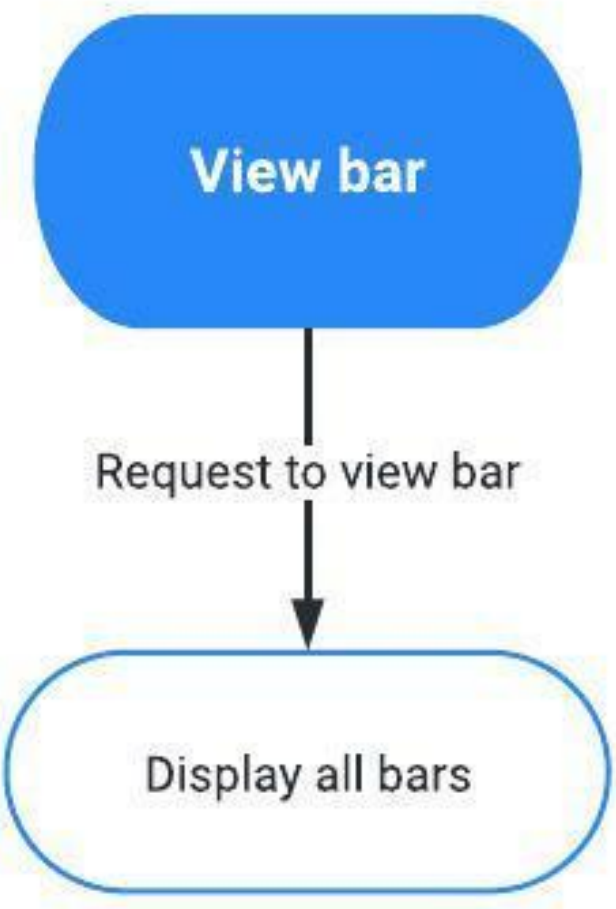
Add Bar



Delete Bar

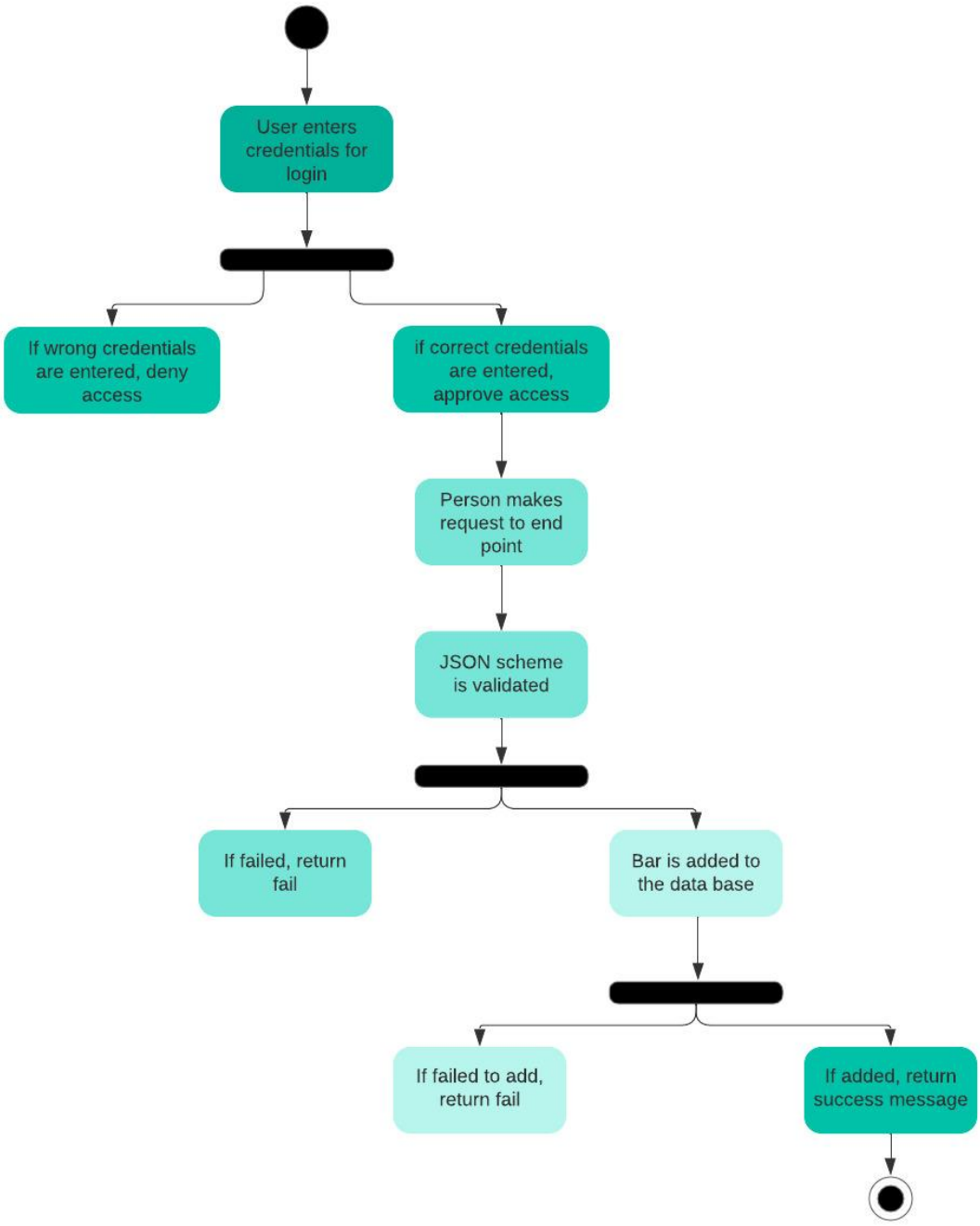

Delete bar
User enters in credentials
If incorrect
Deny access
Request to Delete Bar
Return fail
If fails
JSON schema validated
If successful
Delete bar , return success message

Get Bar

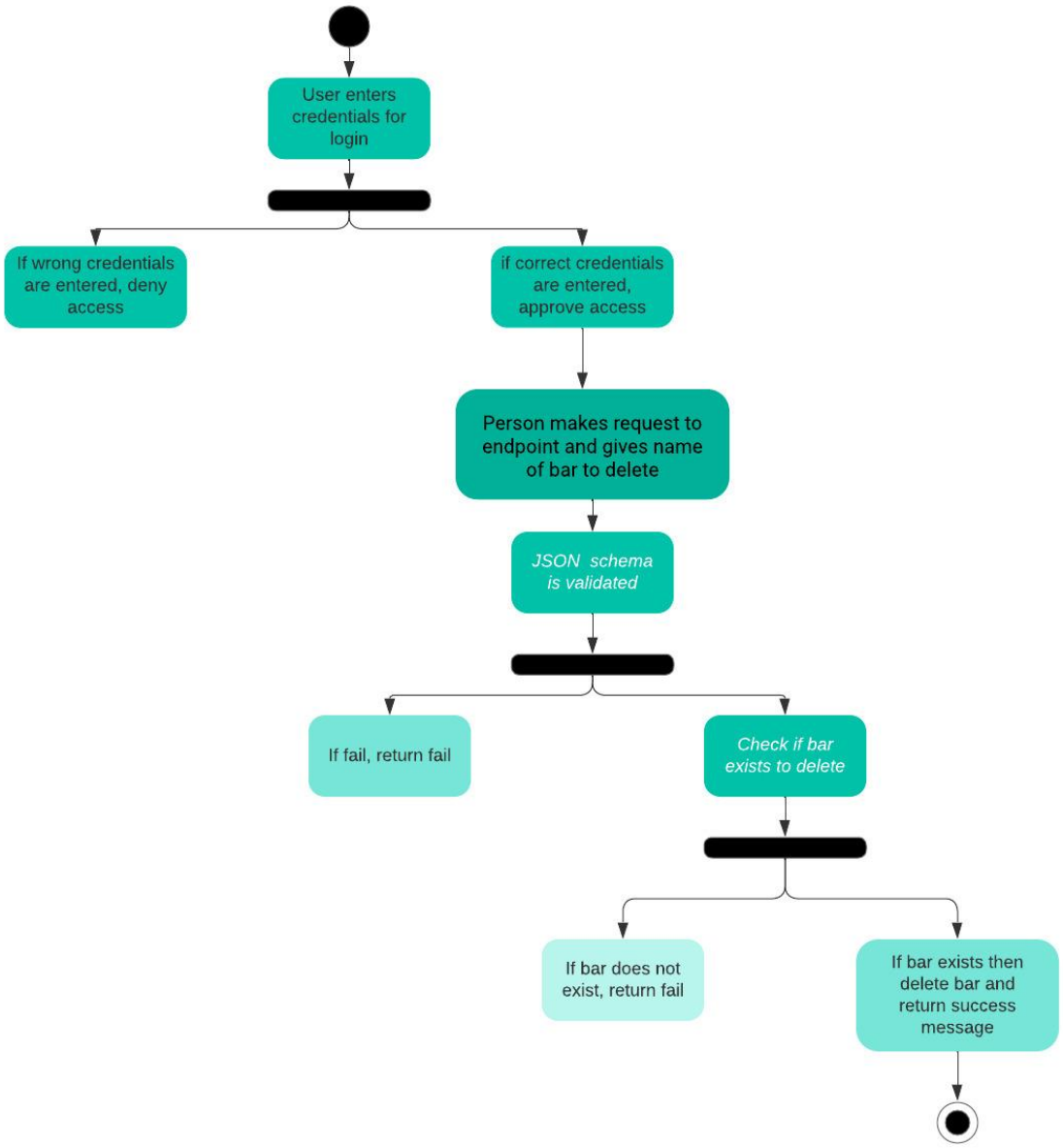


Activity Diagram (Logical View)

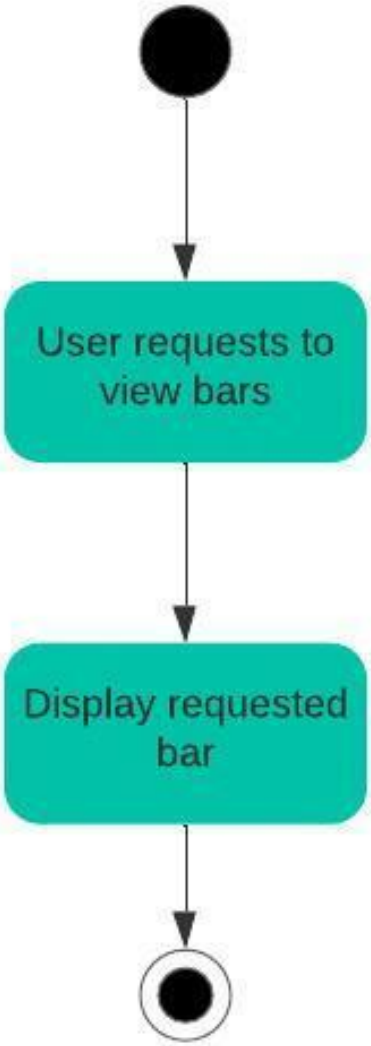
Add Bar


User enters credentials for login
If wrong credentials are entered, deny access
if correct credentials are entered, approve access
Person makes request to end point
JSON scheme is validated
If failed, return fail
Bar is added to the data base
If failed to add, return fail
If added, return success message

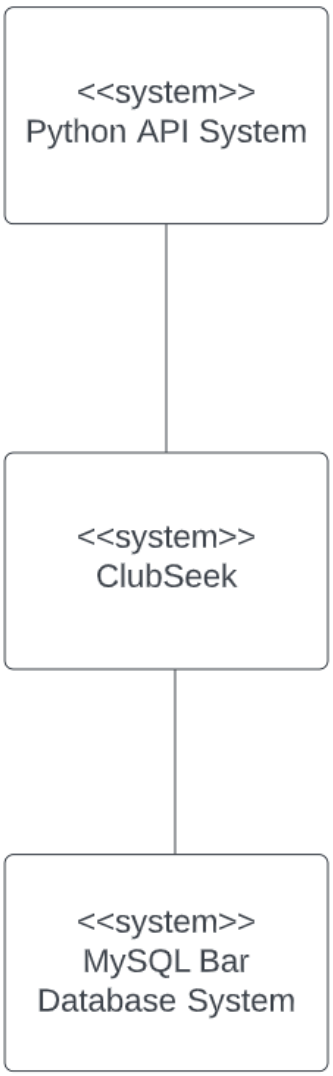
Delete Bar


User enters credentials for login
If wrong credentials are entered, deny access
if correct credentials are entered, approve access
Person makes request to endpoint and gives name of bar to delete
JSON schema is validated
If fail, return fail
Check if bar exists to delete
If bar does not exist, return fail
If bar exists then delete bar and return success message

Get Bar

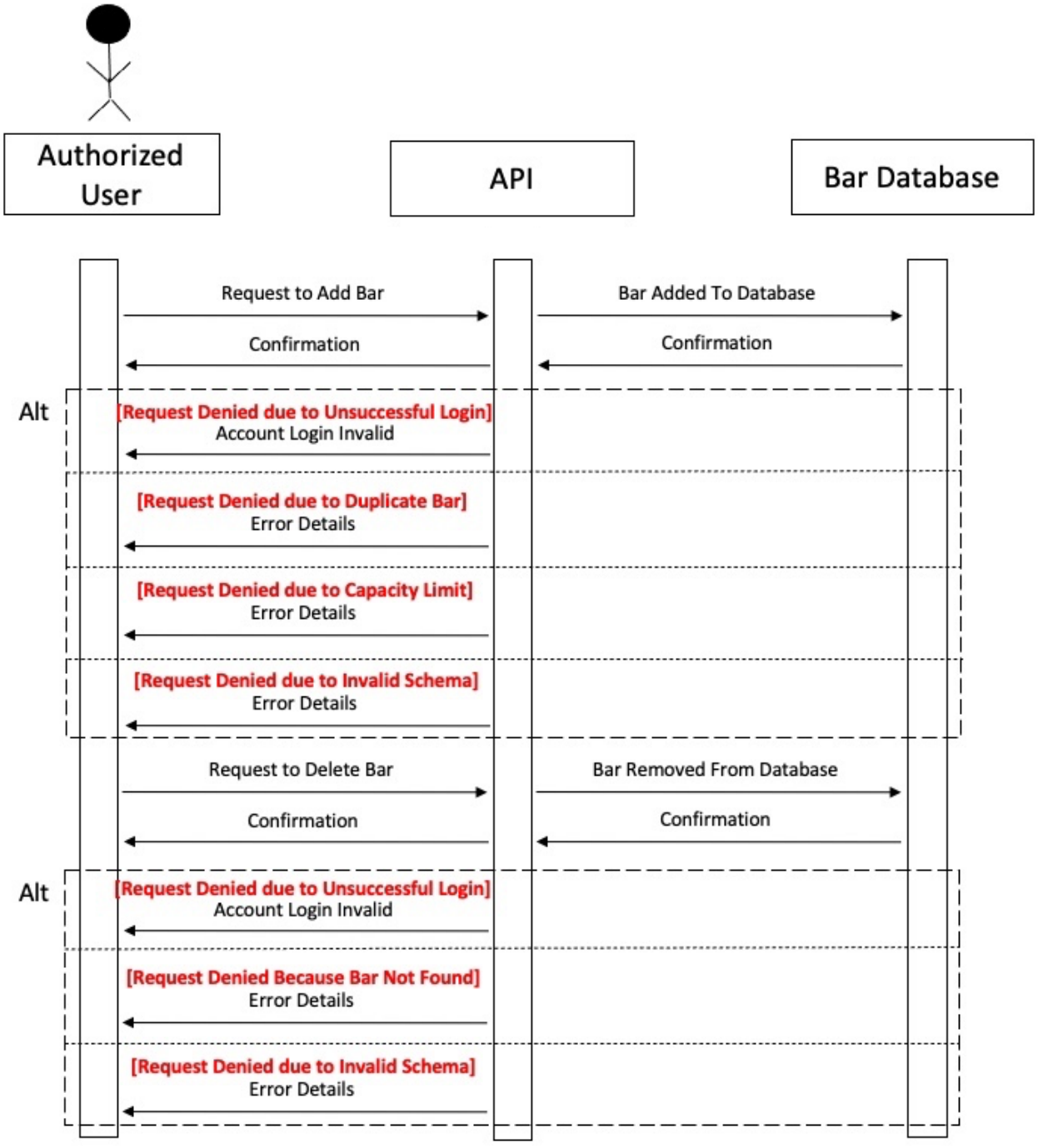


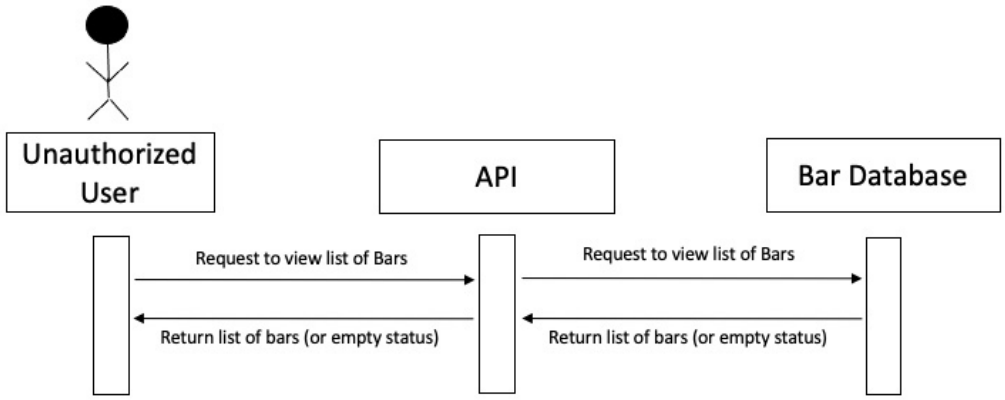
Context Model



This context model shows the *ClubSeek* as a whole when put in conjunction with surrounding systems in its software environment. Visually we can see that *ClubSeek* is connected to two other systems: Python API system and MySQL Bar Database system. These two systems show a general overview of how our software system as a whole interacts with every other system component. It is unlikely that this model will change from sprint to sprint since the systems utilized should not change.

Sequence Diagram (Process View)

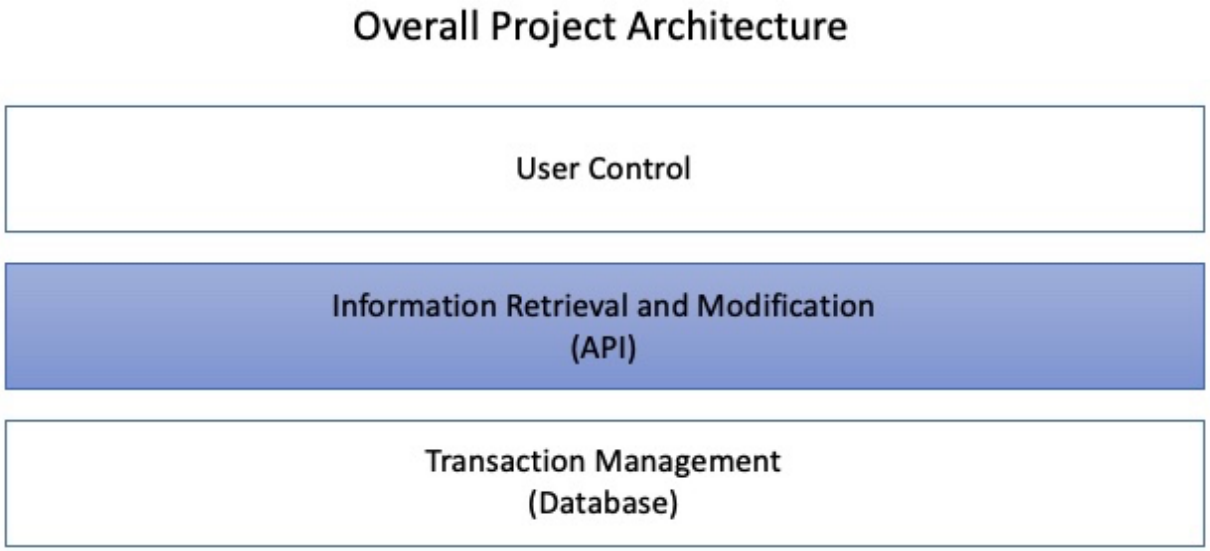


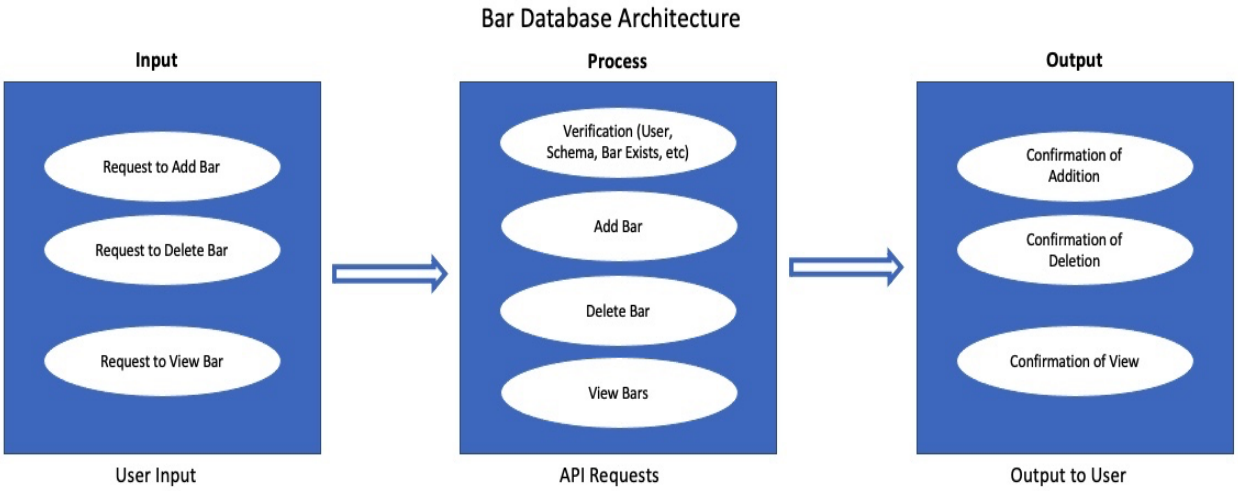


In order to account for the two types of users in Sprint 1, we have two sequence diagrams made for this sprint. For the authorized users, we have functionality for adding and deleting bars as well as alternate outcomes depending on the error. For the unauthorized user, the only functionality is viewing all bars and returning the list of all the bars.

Architectural Modeling:

Process Model





To properly model our system's architecture, we have two types of models that can be used to represent our software solution. The first model, which is the Overall Project Architecture, is a transaction-based information system model. This model is most appropriate for this project because we plan on having three main layers: the user layer which interacts with the software, the information retrieval and modification layer which in our case is an API, and lastly the transaction management layer which is a database.

We also chose to represent the architecture of our database using a transaction processing system, since the user is interacting with the database using API calls. The user inputs a request to the API endpoint for the input stage, the back-end then commences the queries to the database, which then returns confirmation statements (whether they were successful or not).

Component Diagram

This architectural part of the system represents the component diagram which implements the Developmental View of the architectural system. This provides an implementation view for the system where we see the API and System Database working in conjunction to update the system and user with the desired bar while keeping the components used concise and to the point. Thus, the software system is broken down into the following components for simplicity: API Endpoint, System Database, and API Client. The API's implemented for this sprint were the Add Bar API, Delete Bar API (with authentication), and View Bar API (without authentication).

Specification

Add Bar

Function	Adds bar to the database
Description	Adds a bar to the database. Includes the following fields: name, wow factor, capacity, current traffic, and address
Inputs	User credentials
Source	Existing database of the bar
Outputs	Operation Successful A bar has traffic that exceeds capacity A bar already exists in the database Authentication credits missing or invalid
Destination	API file
Action:	Takes user input and credentials and inputs name, wow factor, capacity, current traffic, and address of the bar
Requires	User input
Precondition	User credentials, bar should exist
Postcondition	Displays added bar
Side effects	None

Delete Bar

Function	Deletes bar from database
Description	Deletes bar name, wow factor, capacity, current traffic, and address from database
Inputs	User credentials
Source	Existing bars in database
Outputs	Operation Successful There is no bar with the following name or address Authentication credits missing or invalid
Destination	API file
Action:	Removes a bar and all its relevant data from the database
Requires	User input

Precondition	User credentials, bar should exist
Postcondition	Deletes bar
Side effects	None

Get Bar

Function	Gets requested bar
Description	Displays the bar that is requested by the user
Inputs	Bar name and address
Source	Existing bar database
Outputs	Operation Successful No Clients or Bars to Return
Destination	API file
Action:	Returns an array of all the bars stored in the database in a JSON object
Requires	Bar name and address
Precondition	User input
Postcondition	Array of JSON objects
Side effects	None

Implementation

Problem Analysis

When we first decided on how to create our application, we knew that the problem statement called for an API for its application tier. However, we needed a data tier for the application. We could've either stored information in-memory or created a separate database that the API made requests to. We decided on a database because it allowed us to create a scalable, containerized solution. We chose Python as our coding language and SQL as our query language since we were familiar with both languages.

Containerization

To enable the seamless setup and connection of our API and Database, we utilized dockerFile and docker-compose files. We leveraged Docker to build our application into an image and ran images as containers with networking and storage setup for both the database and application.

Unit Tests and System tests were also integrated with Docker so users could easily test our application.

Application

The following dependencies were used for the following purposes in this sprint:

- Flask: Python web application framework used for API Endpoints
- mysql-connector-python: Used to run SQL Commands to Database
- Flask-HTTPAuth: Flask extension that provides Basic HTTP authentication for Flask routes
- flask-expects-json: JSON Schema Validation for API Requests
- Werkzeug: Used for text hashing in AuthZ
- get-docker-secret: Used to Pull Docker Secrets in Container

The three API endpoints getBar, viewBar, and deleteBar were implemented first as a POC for the application.

Software Testing

Unit Testing

In this sprint, we had one helper function that didn't utilize both our Python API and MySQL database. Therefore, we had two unit tests: one for the correct authentication username and password combination and one for a combination failing authentication.

For our testing across all sprints, we utilized Pytest, a testing framework for Python that allows for efficient automated testing that reports which specific tests failed. An example of a failed test is shown below:

```
major-project-group-4-clubseek-1  | ================================= FAILURES =================================
major-project-group-4-clubseek-1  | _____________________________ test_verify_password_fail _____________________________
major-project-group-4-clubseek-1  |
major-project-group-4-clubseek-1  |     def test_verify_password_fail():
major-project-group-4-clubseek-1  | >       assert clubseek.helpers.verify_password("sidhu","fish") == "sidhu"
major-project-group-4-clubseek-1  | E       AssertionError: assert None == 'sidhu'
major-project-group-4-clubseek-1  | E        +  where None = <function verify_password at 0x7fbf5ed60b80>('sidhu', 'fish')
major-project-group-4-clubseek-1  | E        +    where <function verify_password at 0x7fbf5ed60b80> = <module 'clubseek.helpers' from '/app/clu
bseek/helpers.py'>.verify_password
major-project-group-4-clubseek-1  | E        +      where <module 'clubseek.helpers' from '/app/clubseek/helpers.py'> = clubseek.helpers
major-project-group-4-clubseek-1  |
major-project-group-4-clubseek-1  | tests/test_unit_clubseek.py:10: AssertionError
major-project-group-4-clubseek-1  | ========================= short test summary info ==========================
major-project-group-4-clubseek-1  | FAILED tests/test_unit_clubseek.py::test_verify_password_fail - AssertionErro...
major-project-group-4-clubseek-1  | ======================== 1 failed, 1 passed in 0.46s ========================
```

Our unit tests call the functions themselves (as shown in the photo below), so they have regression testing. If the functions themselves change where it no longer authenticates properly, these tests will fail.

Our tests are shown in an image below:

```python
import clubseek.main
import clubseek.helpers
import json

def test_verify_password():
    assert clubseek.helpers.verify_password("admin","password") == "admin"

def test_verify_password_fail():
    assert clubseek.helpers.verify_password("sidhu","fish") != "sidhu"
```

At the end of our sprint, these unit tests were successful, as shown below.

```
major-project-group-4-clubseek-1  | ============================= test session starts ==============================
major-project-group-4-clubseek-1  | platform linux -- Python 3.9.12, pytest-7.1.2, pluggy-1.0.0
major-project-group-4-clubseek-1  | rootdir: /app
major-project-group-4-clubseek-1  | collected 2 items
major-project-group-4-clubseek-1  |
major-project-group-4-clubseek-1  | tests/test_unit_clubseek.py ..                                          [100%]
major-project-group-4-clubseek-1  |
major-project-group-4-clubseek-1  | ============================== 2 passed in 0.49s ===============================
major-project-group-4-clubseek-1 exited with code 0
```

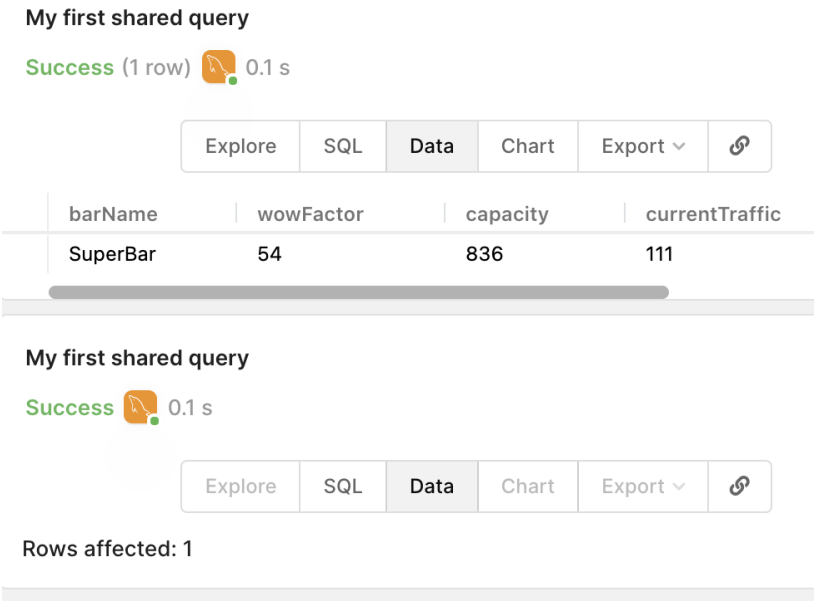
Component Testing

For component testing, we tested that the schema was correct for the add, delete, and view operations. Therefore, the component that was getting tested was just the database.

In the adding a bar entry to the database test (test_Add_In_Database), we used the following SQL commands:

```sql
# Add a Bar
INSERT INTO Bars (barName, wowFactor, capacity, currentTraffic, address)
    VALUES
        ('SuperBar', 54, 836, 111, '255 Sidhu Way Eoin, NJ 07726');

SELECT * FROM Bars;
```

The result of these commands is below:

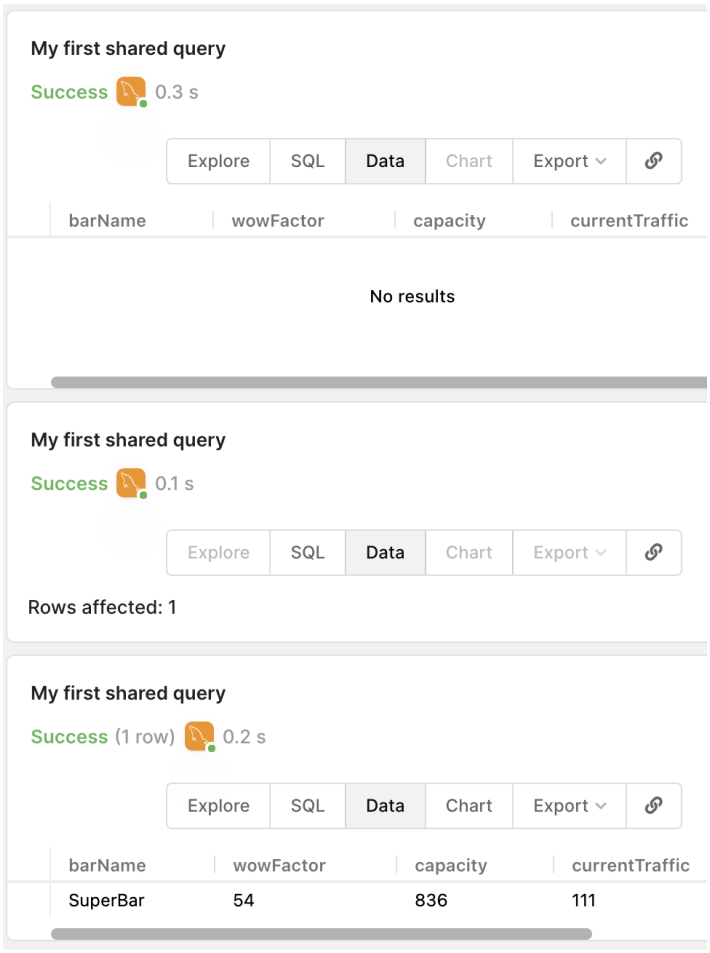


Therefore, the schema we were using across our project for the bars were correct, as the request successfully added a bar entry.

In the adding a bar entry to the database test (test_Del_In_Database), we used the following SQL commands:

```
# Delete a Bar
INSERT INTO Bars (barName, wowFactor, capacity, currentTraffic, address)
    VALUES
        ('SuperBar', 54, 836, 111, '255 Sidhu Way Eoin, NJ 07726');

SELECT * FROM Bars;

DELETE FROM Bars;

SELECT * FROM Bars;
```

The result of the SQL commands is below:



The SQL command was able to successfully add a bar, view it showing that the bar was successfully added, delete the bar, and then view it showing that the bar was successfully deleted. Therefore through these component tests, we know that the database is working as intended and is ready for the system tests where the Python API is integrated to perform the adds and deletes with authentication.

System Testing

In this sprint, we expanded upon the previous system tests to account for testing. We added the authorization logins for the delete and adding operations of bars to the database. There are now a total of 7 system tests: test_bar_adding_no_auth(), test_bar_adding(), test_bar_deleting_bar_without_auth(), test_bar_deleting_bar(), test_bar_fail_deleting_bar(), test_bar_300_bars(), and test_view_300_bars(). The session is opened with the database, and the tests perform the intended operation. For example, the test_bar_adding() test actually adds the bar to the database and the successful test_bar_deleting_bar() test actually deletes the bar. We used pytest to automate the testing where the operations are tested to see whether they produce the intended SQL status code. For example, a successful addition to the bar should produce a 200 code, an unsuccessful addition due to having no authentication would produce a 401 code, and so on. A picture of all tests running successfully is shown below:

```
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_adding_no_auth PASSED       [ 14%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_adding PASSED                [ 28%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_deleting_bar_without_auth PASSED [ 42%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_deleting_bar PASSED          [ 57%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_fail_deleting_bar PASSED     [ 71%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_add_300_bars PASSED          [ 85%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_view_300_bars PASSED             [100%]
major-project-group-4-integrationtest-1  |
```

The functional requirements of adding and deleting bars to the database was also fulfilled. The successful tests show that the functional requirement of implementing a login system was fulfilled as it allows for users with the authentication information to perform the secure commands of adding and removing bars. The non-functional security requirement was added successfully in this sprint, as now only admins, or users with the authentication account and password, can perform the sensitive operations of adding and deleting bars.

Although there is some overhead in starting up the environment, the add and delete requests are performed within a second, following the requirements as shown below:

```
major-project-group-4-integrationtest-1  | 0.13s call     tests/test_integration_clubseek.py::test_bar_adding
major-project-group-4-integrationtest-1  | 0.13s call     tests/test_integration_clubseek.py::test_bar_deleting_bar
```

With authentication, the operation to add a bar takes 0.13s and the operation to delete a bar takes 0.13s, as shown by the time the test takes to successfully add/delete a bar. We added a test to add 300 bars to the database, and then view all the contents of the database. The results of the test are shown below:

```
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_add_300_bars PASSED          [ 50%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_view_300_bars PASSED             [100%]
major-project-group-4-integrationtest-1  |
major-project-group-4-integrationtest-1  | ============================== slowest durations ===============================
major-project-group-4-integrationtest-1  | 38.27s call     tests/test_integration_clubseek.py::test_bar_add_300_bars
major-project-group-4-integrationtest-1  | 0.01s call     tests/test_integration_clubseek.py::test_view_300_bars
```

The tests were successful and the system was able to handle 300 requests at once. Also, the system returned all the bars (300) in under 0.01s, well exceeding the requirement of returning all the bars in under 1s. Also, since we were able to add 300 bars to the database, the system meets the requirement of being able to store 100+ bars.

The code of our system tests is shown below:

```python
from xml.etree.ElementTree import tostring
from clubseek import *
from requests.auth import HTTPBasicAuth
import requests
import time
import json

passwordAdmin = "password"

# Wait for API Endpoint to Connect to Database
while True:
    try:
        response = requests.get("http://clubseek:3000/readiness")
        if response.status_code == 200:
            break
    except:
        print("Waiting for API Endpoint")
        time.sleep(5)
        continue

def test_bar_adding_no_auth():
    request = [{
        "barName": "SuperAwais",
        "wowFactor": 54,
        "capacity": 836,
        "currentTraffic": 111,
        "address": "255 Sidhu Drive Eoin, NJ 08841"
    }]

    addBar = requests.post("http://clubseek:3000/bars", json = request)

    assert addBar.status_code == 401

def test_bar_adding():
    # Add Bar
    request = [{
        "barName": "SuperAwais",
        "wowFactor": 54,
        "capacity": 836,
        "currentTraffic": 111,
        "address": "255 Sidhu Drive Eoin, NJ 08841"
    }]

    addBar = requests.post("http://clubseek:3000/bars", json = request, auth = HTTPBasicAuth('admin', passwordAdmin))

    getBar = requests.get("http://clubseek:3000/bars")

    expected = [dict(
        address = "255 Sidhu Drive Eoin, NJ 08841",
        barName = "SuperAwais",
        capacity = 836,
        currentTraffic = 111,
        wowFactor = 54
        )]

    assert json.loads(getBar.content) == expected

def test_bar_deleting_bar_without_auth():
    request = {
        "barName": "SuperAwais",
        "address": "255 Sidhu Drive Eoin, NJ 08841"
        }

    # Delete Bar
    delBar = requests.delete("http://clubseek:3000/bars", json = request)

    getBar = requests.get("http://clubseek:3000/bars")

    assert(delBar.status_code == 401)
```

```python
def test_bar_deleting_bar():
    request = {
        "barName": "SuperAwais",
        "address": "255 Sidhu Drive Eoin, NJ 08841"
        }

    # Delete Bar
    delBar = requests.delete("http://clubseek:3000/bars", json = request, auth = HTTPBasicAuth('admin', passwordAdmin))

    getBar = requests.get("http://clubseek:3000/bars")

    assert(delBar.status_code == 200)


def test_bar_fail_deleting_bar():
    request = {
        "barName": "SuperAwais",
        "address": "255 Sidhu Drive Eoin, NJ 08841"
        }

    # Delete Bar
    delBar = requests.delete("http://clubseek:3000/bars", json = request, auth = HTTPBasicAuth('admin', passwordAdmin))

    getBar = requests.get("http://clubseek:3000/bars")

    assert(delBar.status_code == 400)


def test_bar_add_300_bars():
    i = 0
    while True:
        name = str(i)
        request = [{
            "barName": name,
            "wowFactor": 54,
            "capacity": 836,
            "currentTraffic": 111,
            "address": name
        }]
        addBar = requests.post("http://clubseek:3000/bars", json = request, auth = HTTPBasicAuth('admin', passwordAdmin))
        i += 1
        if i > 299:
            break
    assert(addBar.status_code == 200)

def test_view_300_bars():
    viewBar = requests.get("http://clubseek:3000/bars")
    assert(viewBar.status_code == 200)
```

Project Backlog

ID	*Requirement and Description*	*Implemented?*
B1	Add a secondary database that keeps track of all queries and requests made to the database	
B2	Create an algorithm to recommend a bar to a user based on given parameters	

Sprint 2

Overview

The focus of this sprint was the implementation of the algorithm to actually recommend a bar for a user. In order to do so, we asked the user to provide some basic information and then a suggestion was generated based on the input parameters.

Last Commit: https://github.com/radical-teach/major-project-group-4/pull/8

Requirements Engineering

Non-Functional Requirements

ID	*Requirement and Description*
NF8	Recommend a bar within 1 second of request

Functional Requirements

ID	*Requirement and Description*	*Backlog ID*
F7	Given user input recommend a bar for them to go to	B2

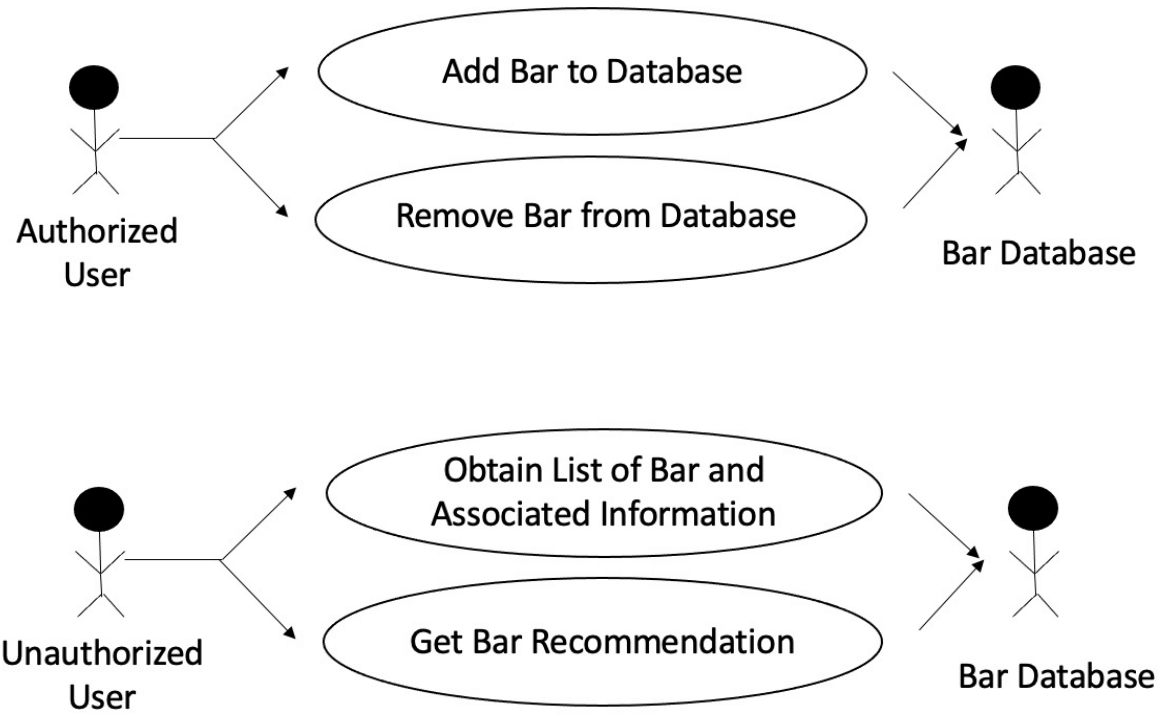
Assumptions

Bar Selection

- Once a bar has been recommended, the individual's information and bar information is stored
- The primary keys of the client are name and phone number
- The users table is cleared every night
- Only one request is allowed per user per night
- Users must provide the following input: name, phone number, minimum wow factor, maximum capacity, preference
- The user must specify a preference (either capacity or wow factor). In the case of a tie, whichever bar has a higher value of the preference attribute the person selects, that bar will be recommended

System Modeling

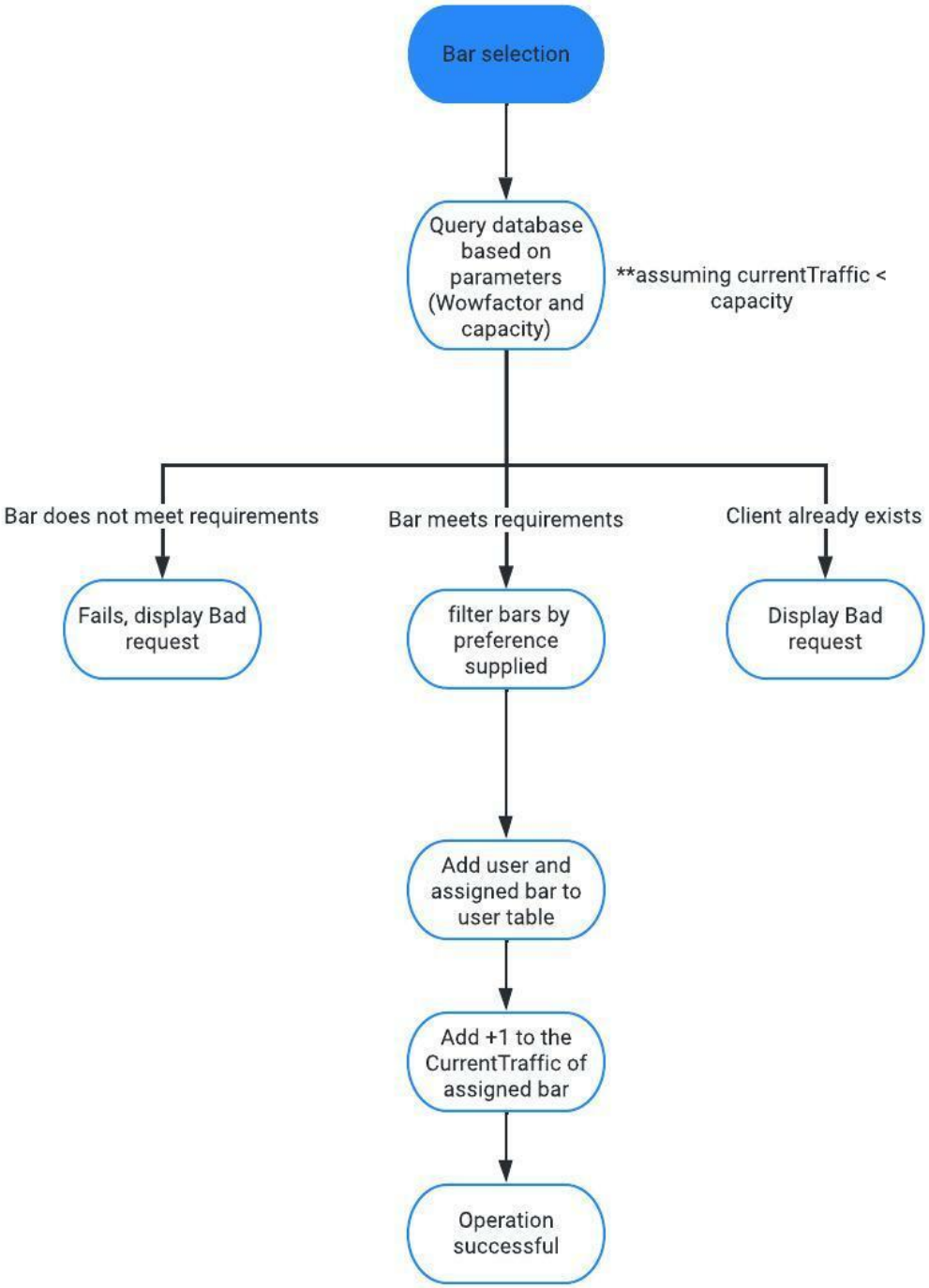
Use Cases



In this sprint, new functionality was added to the unauthorized user. Now an unauthorized user can get a bar recommendation based on parameters that they input.

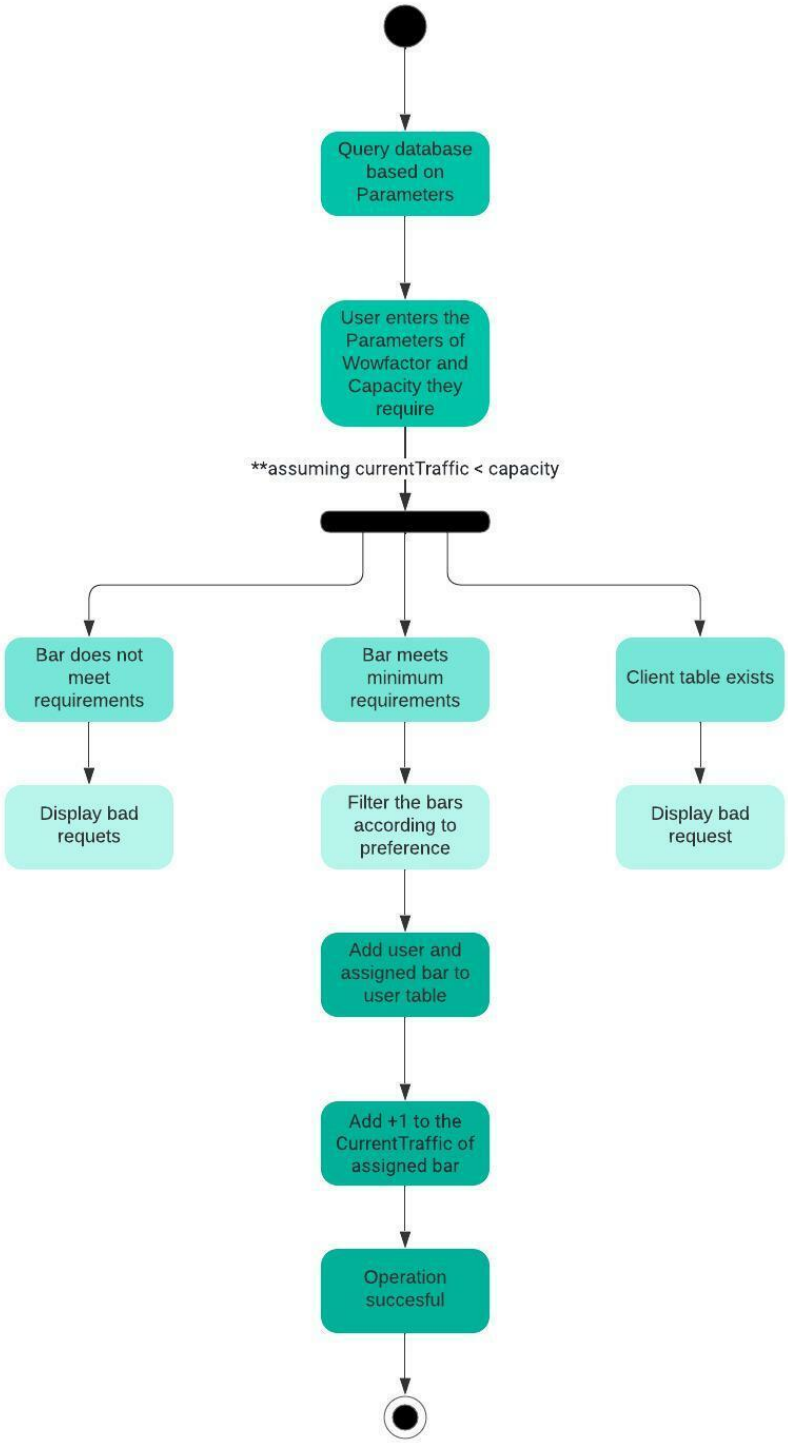
Conceptual Model

Bar Selection

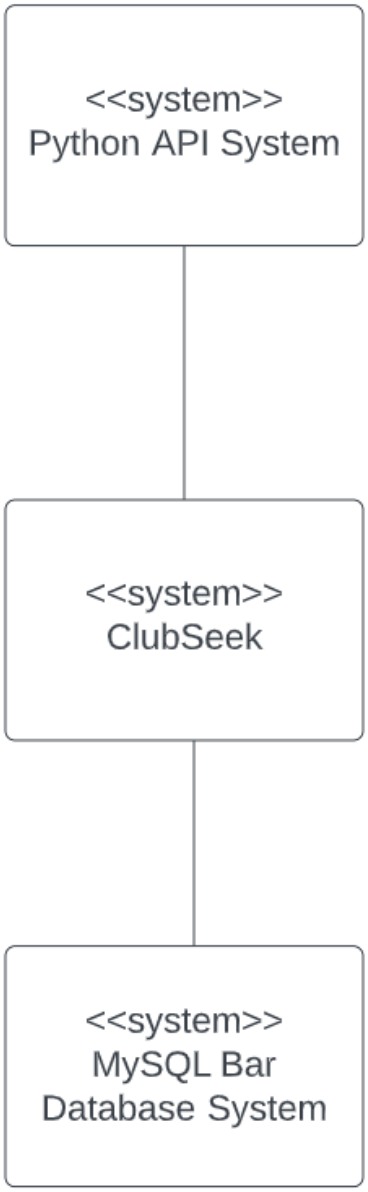


Activity Diagram (Logical View)

Bar Selection

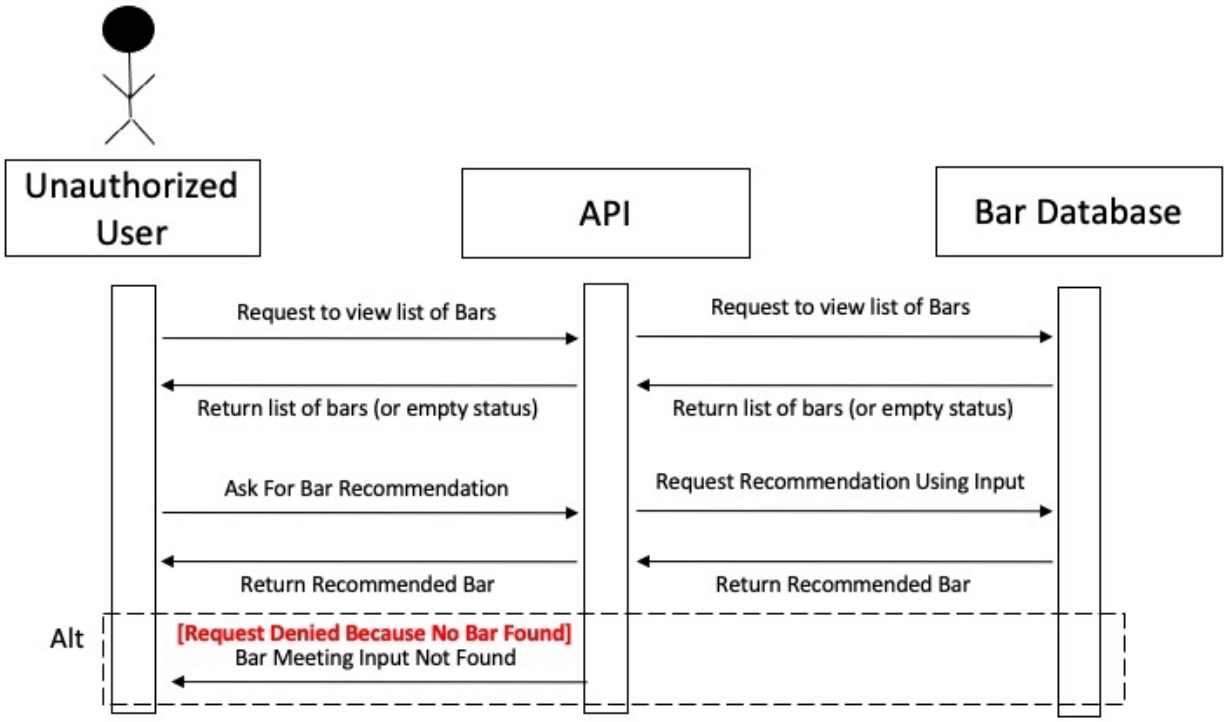


Context Model



This context model shows the *ClubSeek* as a whole when put in conjunction with surrounding systems in its software environment. Visually we can see that *ClubSeek* is connected to two other systems: Python API system and MySQL Bar Database system. These two systems show a general overview of how our software system as a whole interacts with every other system component. This was alluded to earlier, but nevertheless, it must be stated that the model did not change from last sprint to this sprint since the systems utilized did not change.

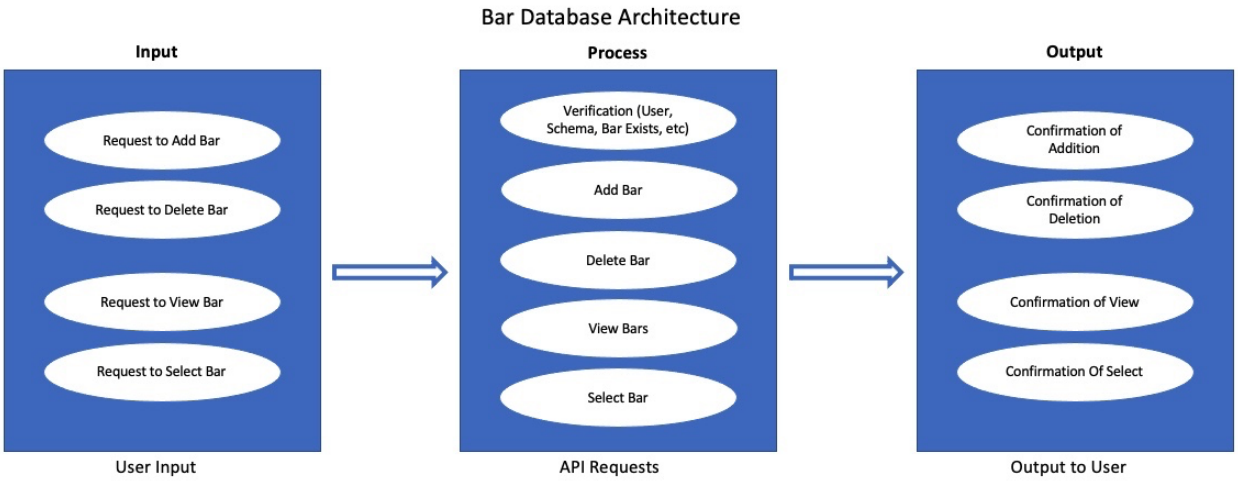
Sequence Diagram (Process View)



In this sprint, the unauthorized user sequence needed to be updated with the new functionality. As you can see, a new request for bar recommendation can be made and depending on the result of the request, either a bar will be returned or an error will be returned if there are no bars that meet the user's specifications.

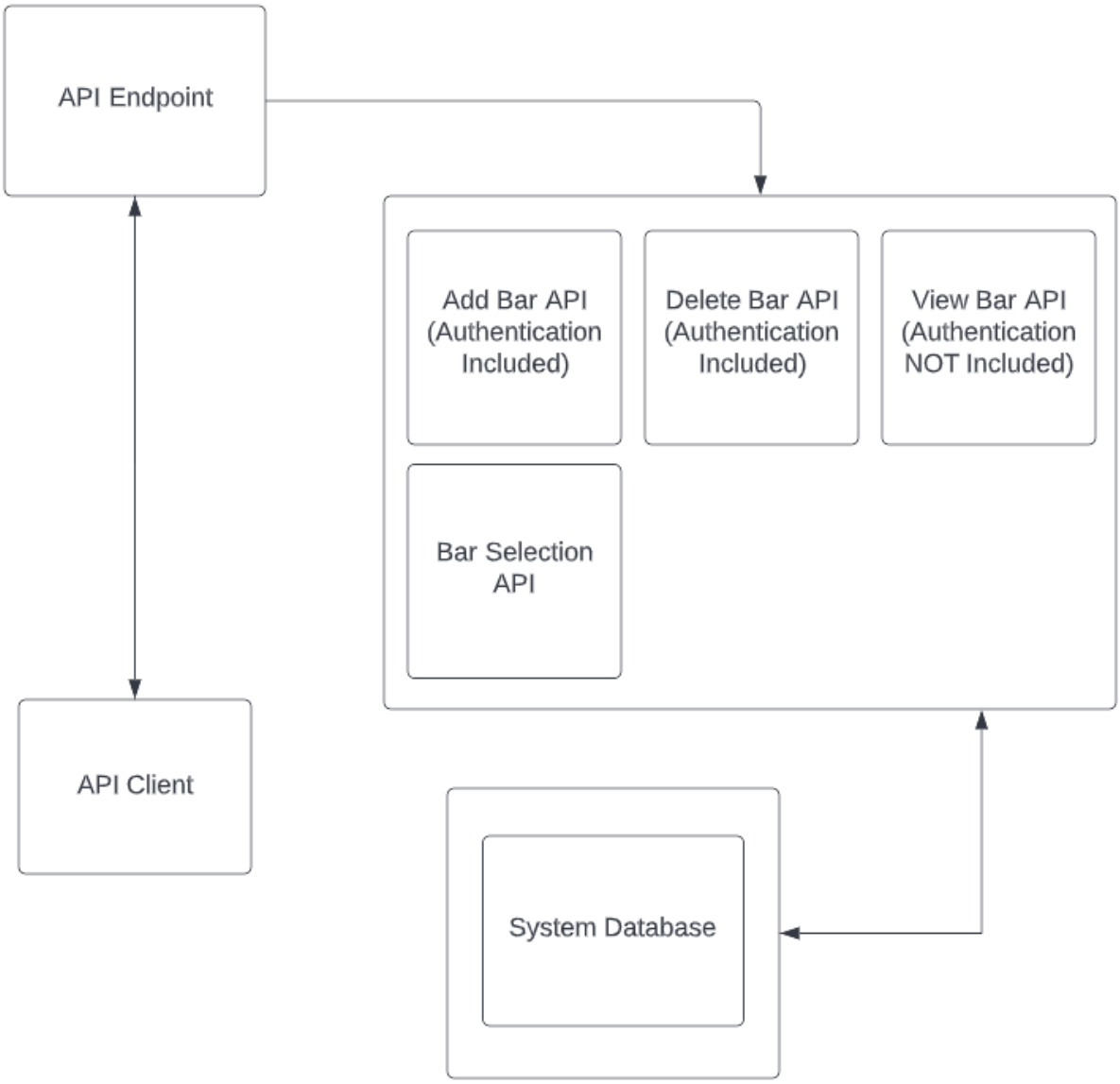
Architectural Modeling

Process Model



In this sprint, we had to update the transaction processing system, since we added functionality for the user to be recommended a bar, which is seen by the additional input, process, and output in the Bar Database Architecture diagram.

Component Diagram



This architectural section of the system represents the component diagram which implements the Developmental View of the architectural system. This provides an implementation view for the system where we see the API and System Database working in conjunction to update the system and user with the desired bar while keeping the components used concise and to the point. Thus, the software system is broken down into the following components for simplicity: API Endpoint, System Database, and API Client. The API's implemented for this sprint were the Add Bar API, Delete Bar API, View Bar API, and Bar Selection API. This version of the diagram

includes a Bar Selection API in addition to the API's mentioned in the previous sprint. The Bar Selection API does not include authentication.

Specification

Bar Selection

Function	Matches user to a bar based on parameters
Description	Uses parameters like wowfactor and capacity to match users with the bar they would like
Inputs	User name, phone number, wowfactor, capacity, preference
Source	Existing bar database
Outputs	Operation Successful No bars meet minimum requirements A client already exists at the users table
Destination	API file
Action:	This API finds the best bar for an individual based on capacity and wowfactor. In the case of a tie, whichever parameter was preferred, the bar with a higher value of the preferred parameter will be returned
Requires	User name, phone number, existing bar, capacity, wowfactor, preference
Precondition	currentTraffic < capacity (assumed)
Postcondition	Displaying the bar match
Side effects	None

Implementation

The barSelection API endpoint was implemented in this sprint to enable users to have Bars recommended to them based on their personal preferences.

Secrets Management

We utilized Docker Secrets to pass Database and API credentials to containers. Docker enabled us to containerize and run our application in any environment. Additionally, we only stored the hash of our API credential passwords in our repository. Therefore, passwords can only be validated and not read from the GitHub Repository. Lastly, we stored passwords for GitHub and AWS (so we can push images to private repositories) in the native secrets management tool from GitHub.

SQLAlchemy

Writing raw Database Queries proved to be too complicated and prone to error, so we switched to a tool called SQLAlchemy for database operations. SQLALchemy had built in sessions management, object relational mapping, and an abstraction layer that allowed us to write SQL using Python language. Once we put in the work to switch, we found it much easier and more efficient to handle database operations.

The following dependencies were used for the following purposes in this sprint:

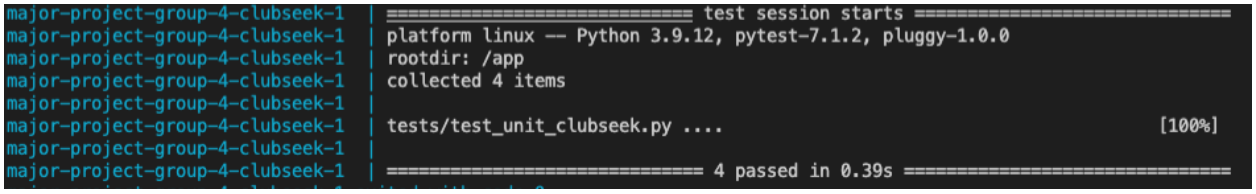
- SQLAlchemy: Used to connect and perform CRUD operations on MySQL Database
- Flask-SQLAlchemy: extension for Flask that adds support for SQLAlchemy to an application

Software Testing

Unit Testing

In this sprint, we had two helper functions: one for determining the bar entry with the lowest current traffic (the number of people in the bar), and one for determining the bar entry with the greatest wowFactor. Since these tests called the functions themselves, they are also regression tests as changes to the functions that cause different than expected results would fail the tests.

These unit tests, as well as the ones from the previous sprint, all passed.



The code of the two new unit tests are as follows:

```
def test_bar_greatest_wow():
    allBarObjects = []
    allBarObjects.append(
        clubseek.main.Bars(
            address = "cheese",
            barName = "pizza",
            capacity = 365,
            currentTraffic = 12,
            wowFactor = 88
        )
    )

    allBarObjects.append(
        clubseek.main.Bars(
            address = "pizza",
            barName = "cheese",
            capacity = 365,
            currentTraffic = 12,
            wowFactor = 77
        )
    )

    greatestBar = clubseek.helpers.getGreatestWow(allBarObjects)

    assert greatestBar.address == "cheese"

def test_bar_lowest_traffic():
    allBarObjects = []
    allBarObjects.append(
        clubseek.main.Bars(
            address = "cheese",
            barName = "pizza",
            capacity = 330,
            currentTraffic = 120,
            wowFactor = 88
        )
    )

    allBarObjects.append(
        clubseek.main.Bars(
            address = "pizza",
            barName = "cheese",
            capacity = 365,
            currentTraffic = 110,
            wowFactor = 77
        )
    )

    greatestBar = clubseek.helpers.getLowestCapacity(allBarObjects)

    assert greatestBar.address == "cheese"
```

Component Testing

System Testing

In this sprint, we added tests for returning a bar for the user to go to. We added a test for when the user successfully gets recommended a bar, for when no bar entry matches the user's specifications for wowFactor and the current traffic, and for when the user tries to make multiple bar requests in a single night. The results are our test are below:

```
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_adding_no_auth PASSED       [  9%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_adding PASSED               [ 18%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_selection PASSED            [ 27%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_selection_duplicate_Request PASSED [ 36%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_selection_no_avail_Bars PASSED [ 45%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_deleting_bar_without_auth PASSED [ 54%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_deleting_bar PASSED         [ 63%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_fail_deleting_bar PASSED    [ 72%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_add_300_bars PASSED         [ 81%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_view_300_bars PASSED            [ 90%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_selection_with_many_bars PASSED [100%]
```

The new tests test_bar_selection(), test_bar_selection_duplicate_Request(), test_bar_selection_no_avail_Bars(), and test_bar_selection_with_many_bars() all passed. In addition, all the previous system tests passed. This signals that none of the additions to the code in Sprint 2 caused any new defects, faults, or errors.

Again, we used the SQL codes of 200 (successful request), 211 (no bars available), and 400 (duplicate request) to establish that the code produced the right results with the edge cases.

In order to test whether the selection algorithm for returning the right bar for the user meets the timing constraint in the requirements, we added a test to select the bar after adding the 300 bars to the database.

The results of the timing test:

```
major-project-group-4-integrationtest-1  | ============================== slowest durations ==============================
major-project-group-4-integrationtest-1  | 37.43s call     tests/test_integration_clubseek.py::test_bar_add_300_bars
major-project-group-4-integrationtest-1  | 0.15s call     tests/test_integration_clubseek.py::test_bar_selection_with_many_bars
major-project-group-4-integrationtest-1  | 0.14s call     tests/test_integration_clubseek.py::test_bar_deleting_bar
major-project-group-4-integrationtest-1  | 0.14s call     tests/test_integration_clubseek.py::test_bar_adding
major-project-group-4-integrationtest-1  | 0.13s call     tests/test_integration_clubseek.py::test_bar_fail_deleting_bar
major-project-group-4-integrationtest-1  | 0.03s call     tests/test_integration_clubseek.py::test_bar_selection
major-project-group-4-integrationtest-1  | 0.02s call     tests/test_integration_clubseek.py::test_view_300_bars
major-project-group-4-integrationtest-1  | 0.01s call     tests/test_integration_clubseek.py::test_bar_selection_duplicate_Request
major-project-group-4-integrationtest-1  | 0.01s call     tests/test_integration_clubseek.py::test_bar_deleting_bar_without_auth
major-project-group-4-integrationtest-1  | 0.01s call     tests/test_integration_clubseek.py::test_bar_selection_no_avail_Bars
major-project-group-4-integrationtest-1  |
major-project-group-4-integrationtest-1  | (23 durations < 0.005s hidden.  Use -vv to show these durations.)
major-project-group-4-integrationtest-1  | ============================ 11 passed in 38.21s ============================
```

Selecting the correct bar with 300 bars to choose from only took 0.15s, well below the 1s timing constraint. Also, these tests show how the functional requirement of recommending the correct bar is achieved.

The added system tests are shown below:

```python
def test_bar_selection():
    request = {
        "name": "Eshaan Mathur",
        "phoneNumber": "732-732-7787",
        "minWowFactor": 50,
        "maxTraffic": 700,
        "preference": "capacity"
    }

    barSel = requests.get("http://clubseek:3000/barSelection", json = request)

    assert (barSel.status_code == 200)

def test_bar_selection_duplicate_Request():
    request = {
        "name": "Eshaan Mathur",
        "phoneNumber": "732-732-7787",
        "minWowFactor": 50,
        "maxTraffic": 700,
        "preference": "capacity"
    }

    barSel = requests.get("http://clubseek:3000/barSelection", json = request)

    assert (barSel.status_code == 400)

def test_bar_selection_no_avail_Bars():
    request = {
        "name": "Eshaan Mathur",
        "phoneNumber": "732-732-7787",
        "minWowFactor": 80,
        "maxTraffic": 700,
        "preference": "wowFactor"
    }

    barSel = requests.get("http://clubseek:3000/barSelection", json = request)

    assert (barSel.status_code == 211)
```

```
def test_view_300_bars():
    viewBar = requests.get("http://clubseek:3000/bars")
    assert(viewBar.status_code == 200)

def test_bar_selection_with_many_bars():
    request = [{
            "barName": "target",
            "wowFactor": 70,
            "capacity": 836,
            "currentTraffic": 111,
            "address": "target"
        }]
    addBar = requests.post("http://clubseek:3000/bars", json = request, auth = HTTPBasicAuth('admin', passwordAdmin))

    request = {
        "name": "Joe Smith",
        "phoneNumber": "732-555-5555",
        "minWowFactor": 65,
        "maxTraffic": 300,
        "preference": "wowFactor"
    }

    barSel = requests.get("http://clubseek:3000/barSelection", json = request)

    assert (barSel.status_code == 200)
```

The existing and new requirements were tested and we were able to show that our system was consistent with these requirements through our old and new tests.

Project Backlog

ID	*Requirement and Description*	*Implemented?*
B1	Add a secondary database that keeps track of all queries and requests made to the database	
B2	Create an algorithm to recommend a bar to a user based on given parameters	Y
B3	Keep track of users and what bars they have been recommended	
B4	Update the current traffic of the bar that is recommended to the user	

Sprint 3

Overview

The focus of sprint three was to button up the project and address the issues regarding updating the bar information as recommendations were provided. Since we want our tool to be usable in real time, every time a recommendation is provided we need to increase the traffic within the bar to keep the data up to date.

Requirements Engineering

Non-Functional Requirements

ID	*Requirement and Description*
NF9	Update current traffic of bar within 1 seconds of recommending a bar to a user

Functional Requirements

ID	*Requirement and Description*	*Backlog ID*
F8	Keep track of what bars each user has been recommended to	B3
F9	Update wowFactor or capacity of a bar entry	B4

Assumptions

View Users

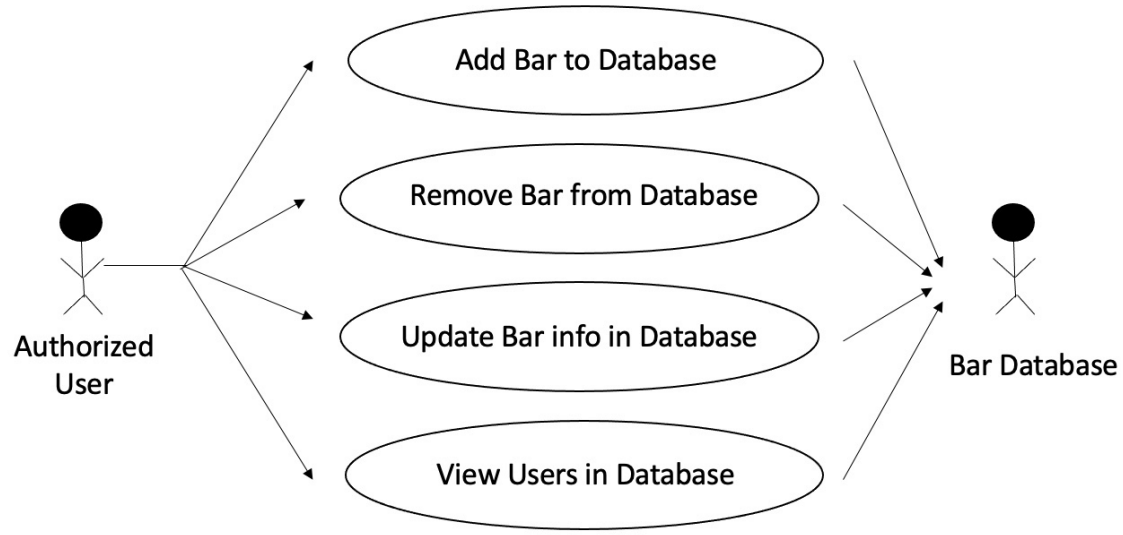
- Users are returned as an array of JSON objects

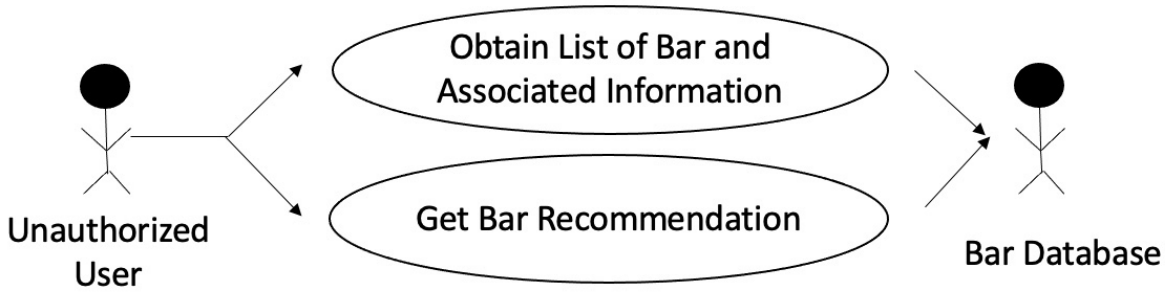
Update Bar Data

- The primary keys (bar name and address) cannot be updated once entered
- The current traffic will never be greater than the capacity

System Modeling

Use Cases

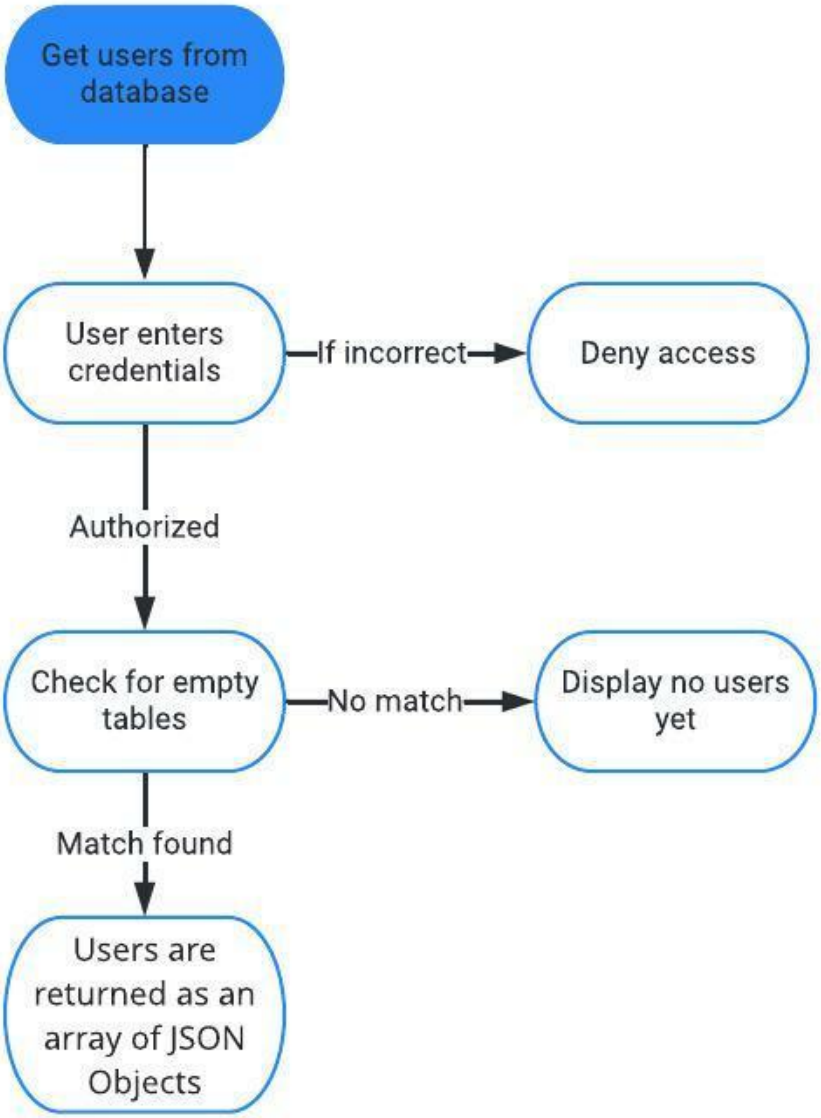




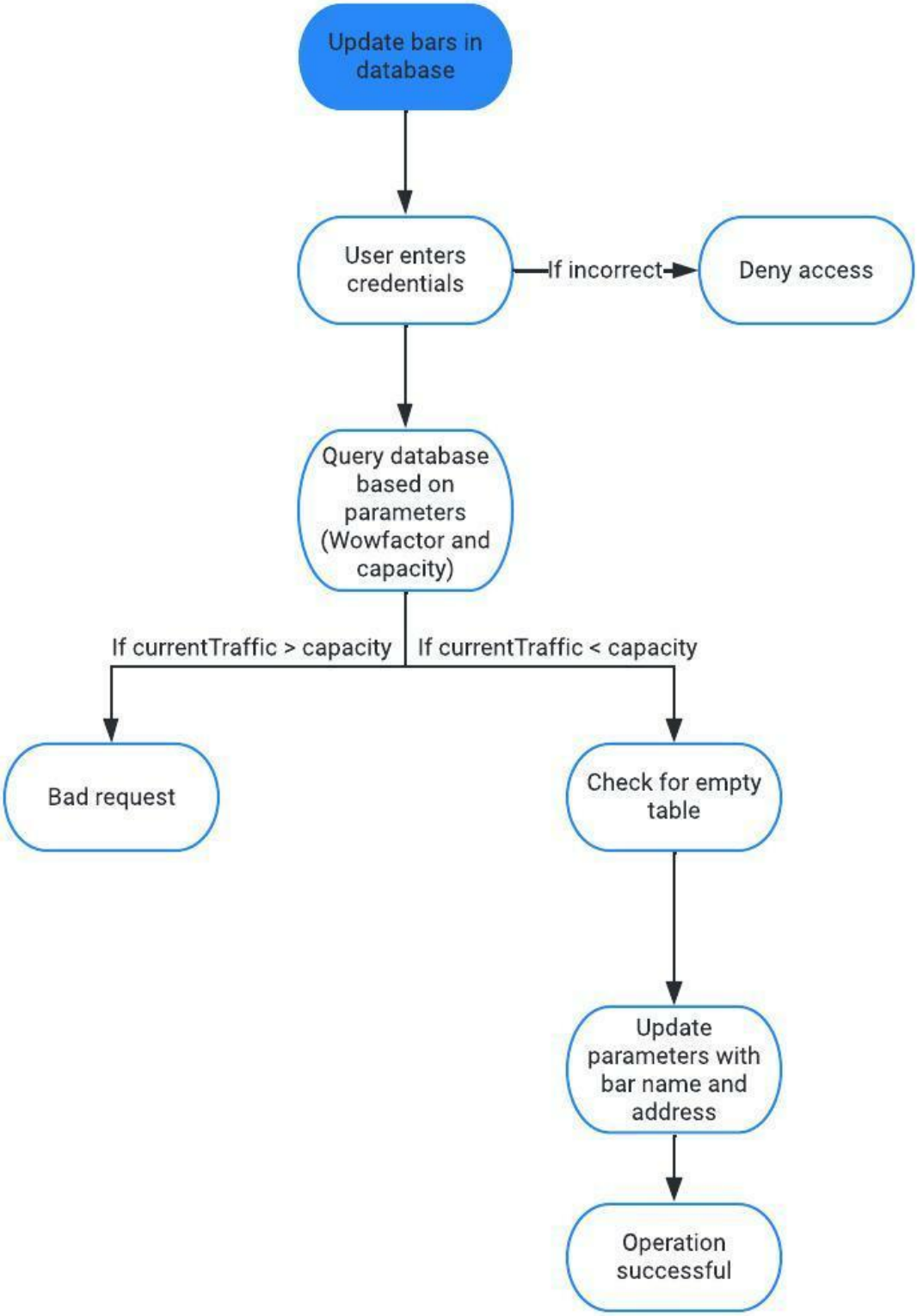
In this sprint, new functionality was added to the authorized user. Now, the actions that an authorized user can perform include updating bar info in the database as well as view the users and what bars they have been recommended to. In this way, we can keep track of exactly which bars we recommended to which users and update the capacity as people use our tool to go to particular bars.

Conceptual Model

View Users

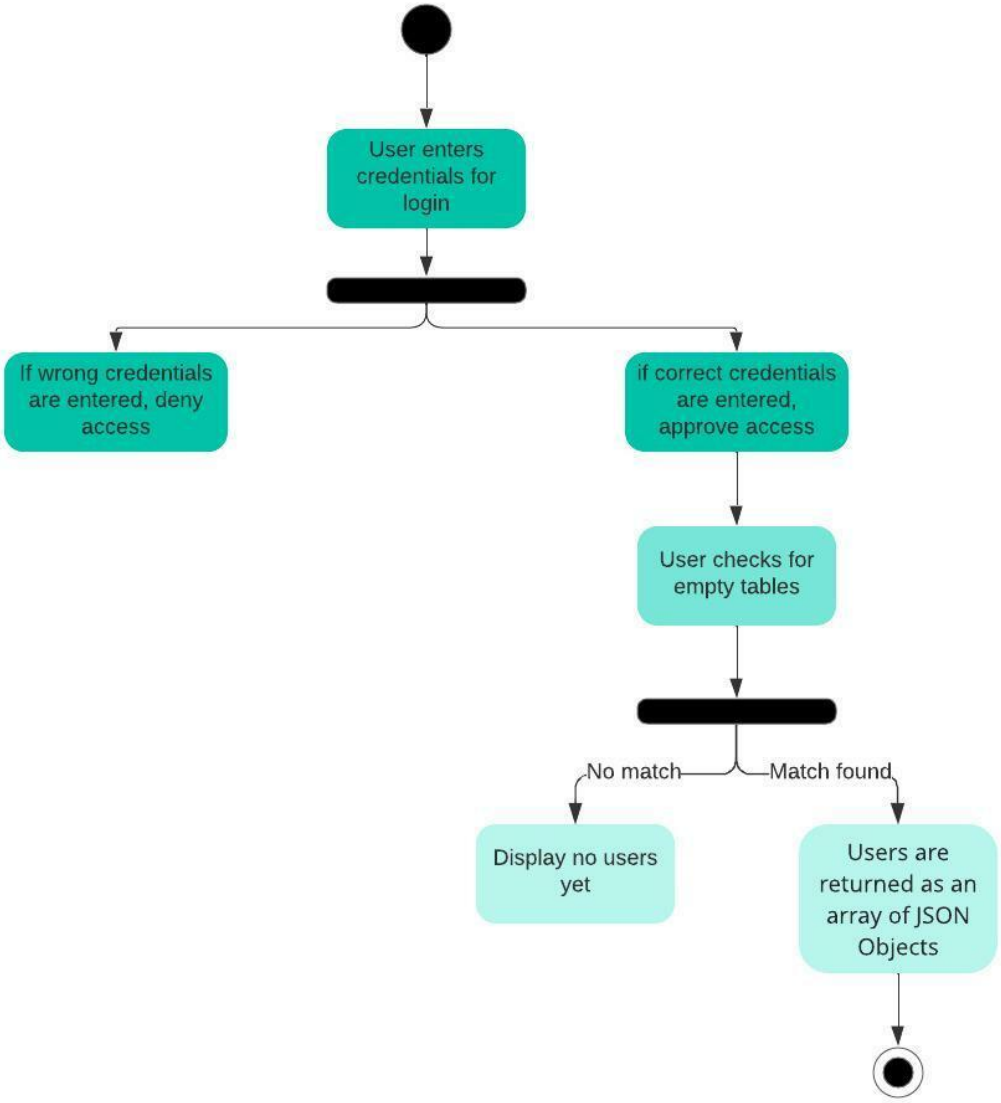


Update Bar Data

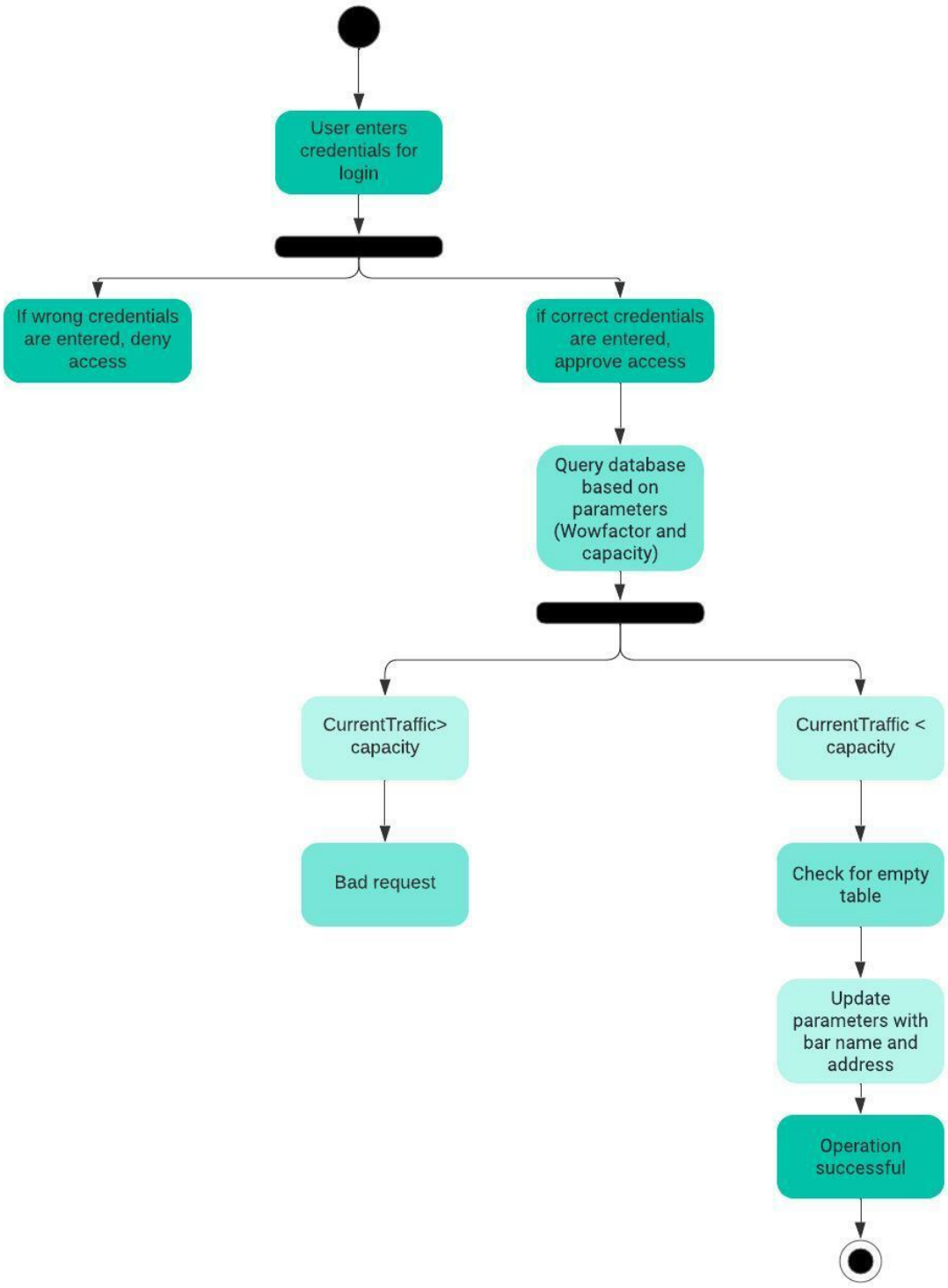


Activity Diagram (Logical View)

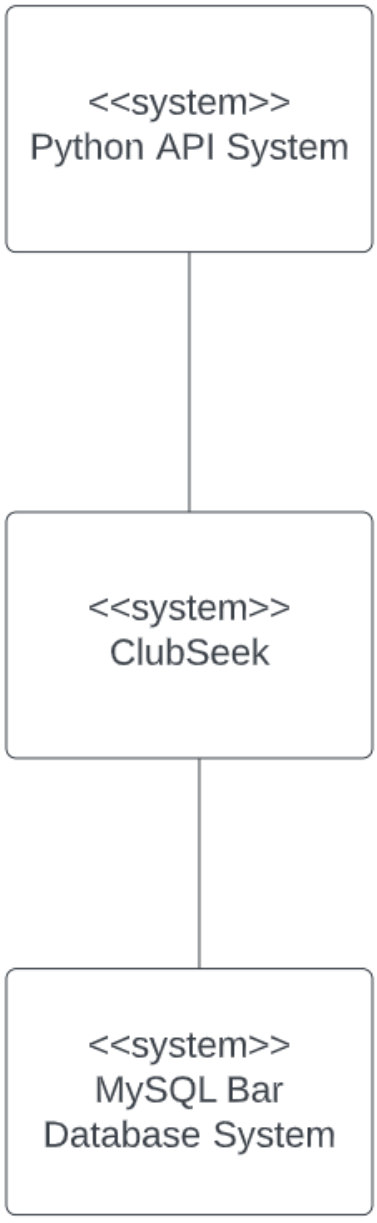
View Users



Update Bar Data

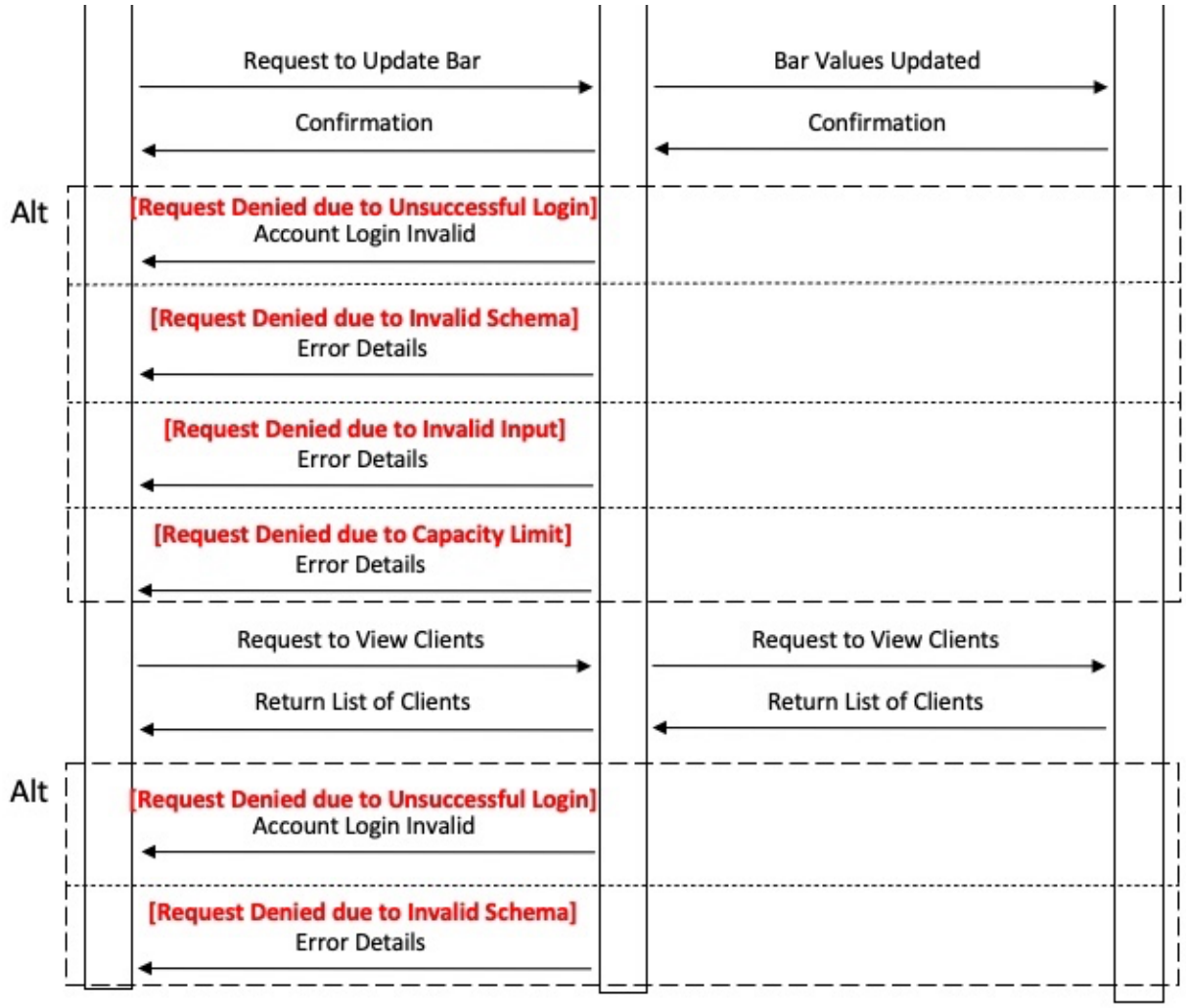


Context Model



This context model shows the *ClubSeek* as a whole when put in conjunction with surrounding systems in its software environment. Visually we can see that *ClubSeek* is connected to two other systems: Python API system and MySQL Bar Database system. These two systems show a general overview of how our software system as a whole interacts with every other system component. This was alluded to earlier, but nevertheless, it must be stated that the model did not change from last sprint to this sprint since the systems utilized did not change.

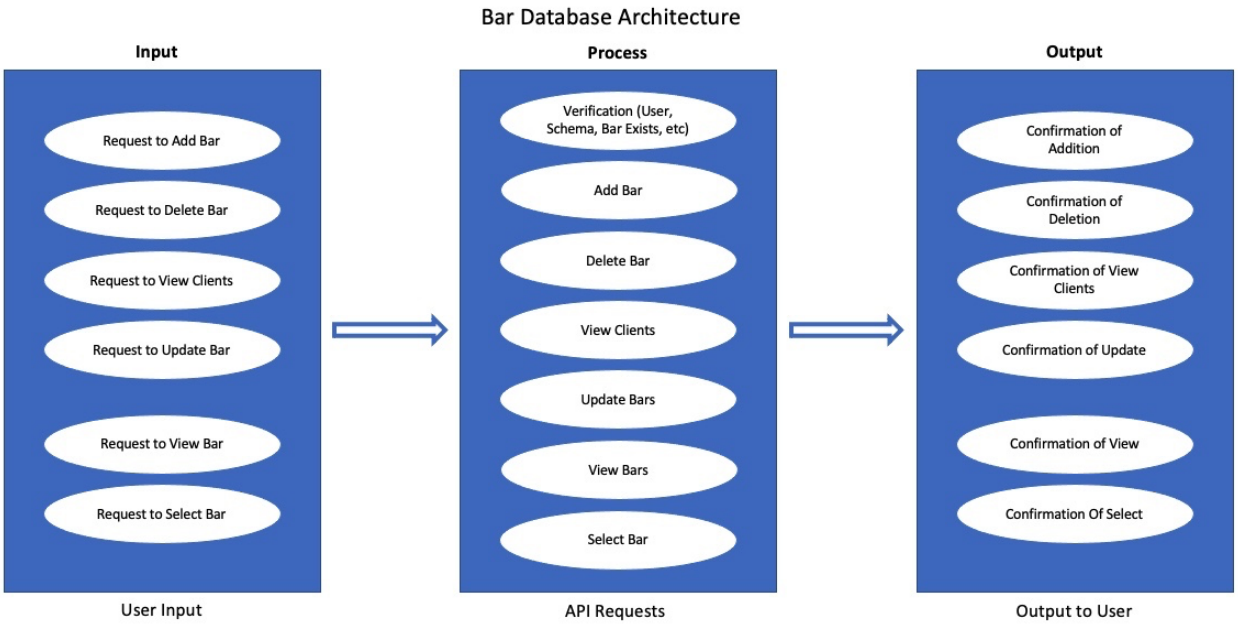
Sequence Diagram (Process View)



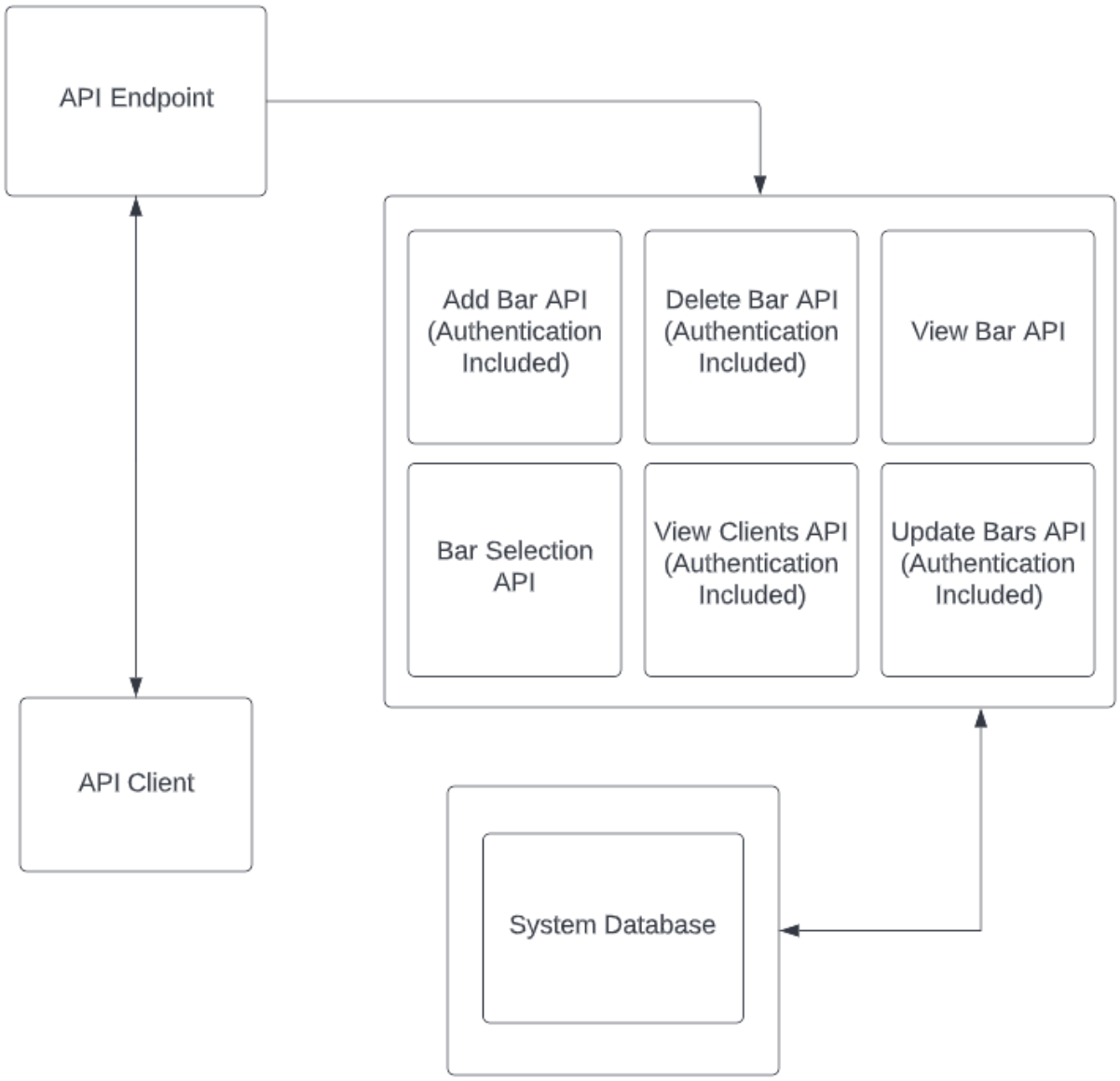
In this sprint, the authorized user sequence needed to be updated with the new functionality. As you can see, a new request to update bar values can be made by authorized users, and there are four possible alternatives for receiving an error. Similarly, an authorized user can also get a list of clients and there are two alternative outcomes that result in errors.

Architectural Modeling

Process Model



Similar to the last sprint, we had to update the transaction processing system, since we added functionality for users to view clients and their requests as well as update the values within the bar as recommendations are made. In this final version of the transaction processing system,



we see our complete project having six inputs, seven processes, and six outputs.

Component Diagram

This architectural section of the system represents the component diagram which implements the Developmental View of the architectural system. This provides an implementation view for the system where we see the API and System Database working in conjunction to update the system and user with the desired bar while keeping the components used concise and to the point. Thus, the software system is broken down into the following components for simplicity: API Endpoint, System Database, and API Client. The API's implemented for this sprint were the Add Bar, Delete Bar, View Bar, Bar Selection, View Clients, and Update Bars. This version of the

diagram includes a View Clients API and Update Bars API in addition to the API's mentioned in the previous sprints. View Clients includes authentication, as does Update Bars.

Specification

View Users

Function	Gets users from the database
Description	Displays existing users in the database at tables
Inputs	User credentials
Source	Existing bar database
Outputs	Operation Successful No clients or bars to return Authentication credentials missing or invalid
Destination	API file
Action:	Checks for empty tables
Requires	User credentials, existing bar and users at tables
Precondition	Authorized
Postcondition	Displaying the users at tables
Side effects	None

Update Bar Data

Function	Updates information of the bars
Description	Based on parameters, user is able to check for empty tables and update the parameters with bar name and address
Inputs	User credentials, wowfactor, capacity
Source	Existing bar database
Outputs	Operation successful, bar updated Bad request
Destination	API file
Action:	If currentTraffic < capacity, checks for empty tables and user is able to update information If currentTraffic > capacity, bad request
Requires	User authorization, parameters
Precondition	currentTraffic < capacity
Postcondition	Bar is updated
Side effects	None

Implementation

In the final sprint, we added the ability to update existing bars and view users' bar recommendations.

Documentation

In this sprint, we properly documented all the API Endpoints of our application. We documented their request schema, sample response body, and HTTP Response codes. Additional documentation on the application and API can be found on the README or the Postman API Documentation

GitOps

To enable GitOps, we utilized GitHub Actions. For one, we created GitHub Actions that tested if Unit and Integration Tests passed in Pull Request. By making the main branch protected and making the GitHub actions required for all Pull Requests, we ensured that only code that was

properly tested made it to the main branch. We also set up GitHub Actions to push images to the GitHub Container Registry and AWS Elastic Container Registry when a PR is merged to main. This meant that we always had an up to date container in AWS ECR and our GitHub repo for clients to pull and use. This could also be used to create a pipeline for automated deployment of application updates.

Software Testing

Unit Testing

In this sprint, all development activities involved functions from dependencies, no unit tests were added in this sprint. The functions created in this sprint fell under system testing, as our functions utilized both our Python API and MySQL database. Unit tests were completed in sprints 2 and 3, as they incorporated helper functions that did require interfacing between our two systems: the Python API and

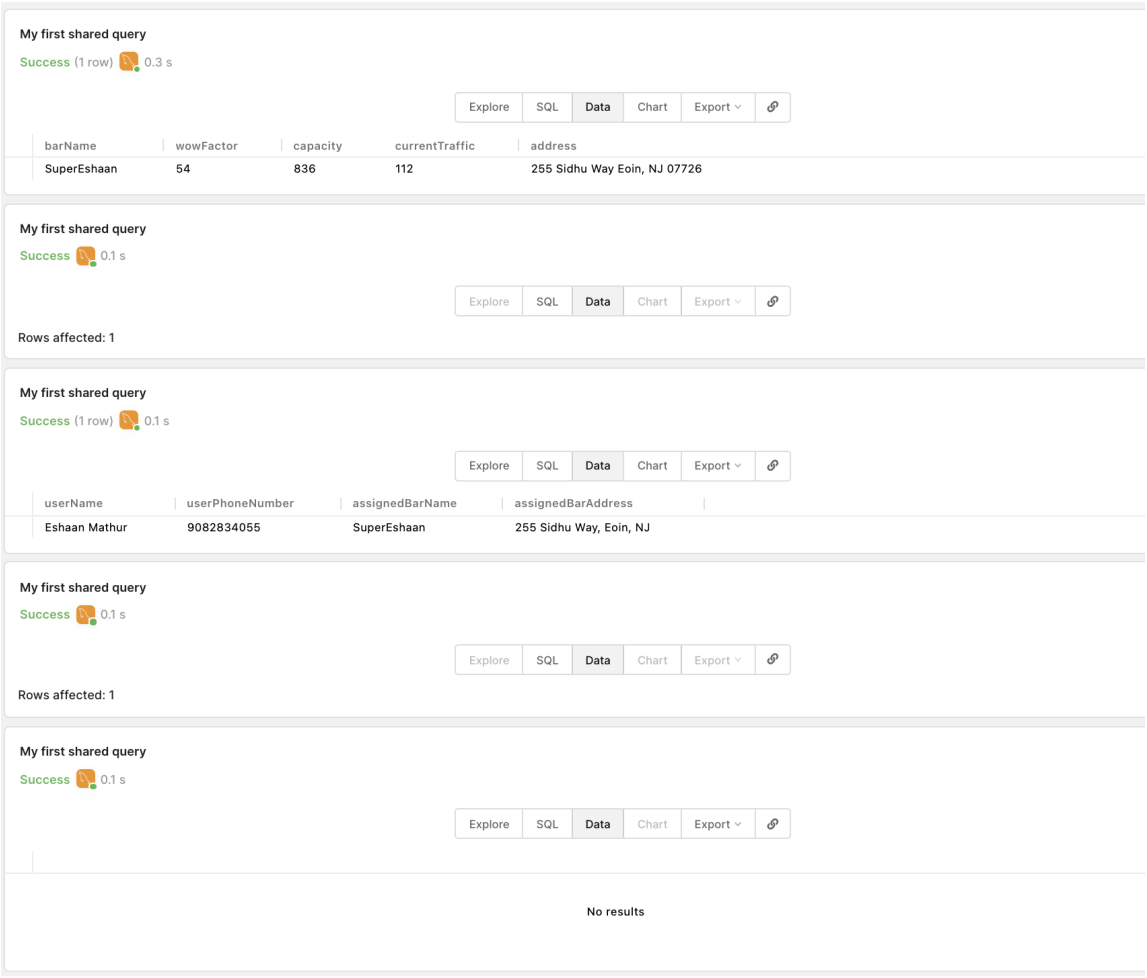
Component Testing

In this sprint, we tested the database component to see whether adding the user to the client list and updating the bar entry's currentTraffic field by 1 would occur within the timing constraint of our non-functional requirement introduced this sprint.

The SQL commands on our database for this test (test_update_bar_on_selection_speed) is below:

```
# Add one to the CurrentTraffic of a Bar
INSERT INTO Bars (barName, wowFactor, capacity, currentTraffic, address)
    VALUES
        ('SuperEshaan', 54, 836, 111, '255 Sidhu Way Eoin, NJ 07726');

INSERT INTO Users (userName, userPhoneNumber, assignedBarName, assignedBarAddress)
    VALUES
        ('Eshaan Mathur', '9082834055', 'SuperEshaan', '255 Sidhu Way, Eoin, NJ');

SELECT * FROM Users;

UPDATE Bars SET (currentTraffic = currentTraffic + 1)
    WHERE barName = 'SuperEshaan' AND address = '255 Sidhu Way Eoin, NJ 07726';

SELECT * FROM Bars;
```

The output of the series of SQL commands in our test is below:



As shown in the output, the bar was successfully added, a user was added to that bar, and the currentTraffic value of the bar entry was incremented by 1. Using the time values of the 2nd and 4th boxes (corresponding to the increment currentTraffic value and the add user to bar entry respectively), we can determine the time to be equal to <=0.2 seconds, each step took <=0.1 seconds each.

System Testing

In this sprint, we added system tests for the new functionality of storing the user information of what bars they went to, and the functionality of updating the wowFactor and/or capacity of the bar entries in the database. The following is the result of the new and old tests after this sprint:

```
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_adding_no_auth PASSED       [  6%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_adding PASSED                [ 13%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_updating_bar PASSED          [ 20%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_bad_update PASSED            [ 26%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_selection PASSED             [ 33%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_selection_duplicate_Request PASSED [ 40%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_selection_no_avail_Bars PASSED [ 46%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_view_clients PASSED              [ 53%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_view_clients_no_auth PASSED      [ 60%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_deleting_bar_without_auth PASSED [ 66%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_deleting_bar PASSED          [ 73%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_fail_deleting_bar PASSED     [ 80%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_add_300_bars PASSED          [ 86%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_view_300_bars PASSED             [ 93%]
major-project-group-4-integrationtest-1  | tests/test_integration_clubseek.py::test_bar_selection_with_many_bars PASSED [100%]
major-project-group-4-integrationtest-1  |
```

The new tests test_bar_updating_bar(), test_bar_bad_update(), test_view_clients(), and test_view_clients_no_auth() all passed. Therefore, the system still fulfills the previous requirements, while now meeting the functional requirements of being able to view the clients of the bars when authorized and meeting the requirement of being able to update the data of a bar within the database.

The timing constraint requirement of updating the bar's current traffic within 1 second was able to be tested through the test_bar_updating_bar() test. The test takes 0.13s to update the bar entry, meeting the timing constraint by far.

```
major-project-group-4-integrationtest-1  | 0.13s call     tests/test_integration_clubseek.py::test_bar_updating_bar
```

The code of the new tests are shown below:

```python
def test_bar_updating_bar():
    request = {
        "barName": "SuperAwais",
        "wowFactorChange": 60,
        "capacityChange": 850,
        "currentTrafficChange": 150,
        "address": "255 Sidhu Drive Eoin, NJ 08841"
    }

    upBar = requests.put("http://clubseek:3000/bars", json = request, auth = HTTPBasicAuth('admin', passwordAdmin))
    assert (upBar.status_code == 200)

def test_bar_bad_update():
    request = {
        "barName": "SuperAwais",
        "wowFactorChange": 60,
        "capacityChange": 850,
        "currentTrafficChange": 900,
        "address": "255 Sidhu Drive Eoin, NJ 08841"
    }

    upBar = requests.put("http://clubseek:3000/bars", json = request, auth = HTTPBasicAuth('admin', passwordAdmin))
    assert (upBar.status_code == 400)
```

```python
def test_view_clients():
    request = {
    }

    viewClients = requests.get("http://clubseek:3000/users", json = request, auth = HTTPBasicAuth('admin', passwordAdmin))

    assert (viewClients.status_code == 200)

def test_view_clients_no_auth():
    request = {
    }

    viewClients = requests.get("http://clubseek:3000/users", json = request)

    assert (viewClients.status_code == 401)
```

Project Backlog

ID	*Requirement and Description*	*Implemented?*
B1	Add a secondary database that keeps track of all queries and requests made to the database	
B2	Create an algorithm to recommend a bar to a user based on given parameters	Y
B3	Keep track of users and what bars they have been recommended	Y
B4	Update the current traffic of the bar that is recommended to the user	Y
B5	Deploy containers to Kubernetes for fault tolerance and high availability	
B6	Develop a front end to provide a UI before mass-market release	

Conclusion and Evaluation

Dependability Implementation

Security

Throughout our sprints we kept security in mind as a lot of the features (like adding, deleting, updating bars, viewing users, bar selection/match) cannot be accessed or be successful without user authorization. If the wrong credentials are entered, the user will be denied access to the database. This would fall under operational security as this concerns the operation and use of the organization's system.

We further protected our data and accounts by hashing and storing them in this form, thus it would be very difficult to access our system without being an authorized user.

Project Management Concept

Team Work

Most software projects are a group project, but we have to keep in mind that a group should be cohesive. The people in our team are highly motivated individuals and have the team spirit required for this project to be successful.

The Scrum master made sure of all team members' availability to hold meetings and was very flexible with all of it while making sure everyone was on the same page and interacted with each other in productive ways. The group was extremely cohesive as we all learned from each other and continually improved ourselves to match equal contributions towards this project. We also thoroughly enjoyed and motivated each other to work during the day. We did not hesitate to fix other team members' mistakes, irrespective of who did it. Everyone knew what was expected and what was going on.

To begin with, the group was formed and picked for it to be diverse, containing members from different backgrounds and strengths for it to all blend together very well in everyone's best interest. Even though roles were not defined or assigned, we are a very diverse group as we had people working on programming, testing and documentation separately, yet together. We all discussed our strengths and weaknesses and decided on the tasks that we could take on that were in our best interest so we could provide a high quality product in the end. This was also done to keep the individual motivated all throughout the process and improve on their personal goals. It is safe to say that this group consisted of the right balance of task oriented, self oriented and interaction oriented people. We had top quality communication and response time

even out of meetings when individuals were working on their own aspects of the project. Since we were more of an informal group, we got a chance to grow as friends, causing the communication to be easier and creating a better working environment.

Advanced SWE Topic Implementation

Software Reuse

The advanced SWE topic we chose to implement was software reuse. While we have different aspects of software reuse across the components of our system, we went into the original planning of the program with a specific software reuse approach in mind. In the textbook in the Software Reuse Chapter (Chapter 15), we decided to utilize the Application System Integration approach.

Approach	Description
Application frameworks	Collections of abstract and concrete classes are adapted and extended to create application systems.
Application system integration	Two or more application systems are integrated to provide extended functionality.
Architectural patterns	Standard software architectures that support common types of application system are used as the basis of applications. Described in Chapters 6, 11, and 17.

Instead of only using a Python application system, we decided to integrate a Python application system with a MySQL database to add extended functionality. With the MySQL database, it allowed users to receive data about the bars and make a selection with a lot more efficiency. It also enabled us to implement the authorized user functionality with each request, allowing extra security for preventing unauthorized users from adding, deleting, changing bars, or obtaining information about what bars our clients are at.

Database integration was one of the 5 web application frameworks listed in the Applications Frameworks section of Software Reuse. Database integration provided the features to our system that were listed in the textbook, as we utilized SQLAlchemy to extend our objects to multiple tables in the main database.

Validation

Are we building the right product?

The main prompt of this product was to produce a software solution to help users determine which bar they should attend based on their coolness requirement as well as security. Looking back at the integration and unit testing, we can conclude that our product is indeed the right product for the job. Our testing shows that our solution meets all the requirements of the problem statement, and delivers a solution in a timely and efficient manner.

Verification

Are we building the product right?

Throughout the entire process of scrum calls and working on the program we used static verification.

This was a way to verify the correctness of the program. Using the sequence diagram and the use cases, we were able to verify when we would get errors in the program and in what use cases the program functioned as we wanted it to. In every instance, we were able to verify that we received the outputs that we expected; we returned errors when errors should be returned and confirmed that all API calls functioned properly. In this way, since our testing was rigorous and thorough, and we had no surprise outputs, we can confidently say that our product meets all requirements and functions properly.