ClassCheck: Attendance Tracking System

Software Engineering Team Project Report #3

Spring 2025

05/05/2025

Submitted by:

Array of Hope

Team Members:

- Absa Fall - af1072@scarletmail.rutgers.edu (Absa1072)
- Rashmee Gade - rrg91@scarletmail.rutgers.edu (rashmee-g)
- Prachi Patel - psp129@scarletmail.rutgers.edu (psp129)
- Nethra Sakthivel - ns1410@scarletmail.rutgers.edu (ns1410)
- Navya Terapalli - nt434@scarletmail.rutgers.edu (navyaterap)

Course Name: Software Engineering (14:332:452)

Github: https://github.com/Absa1072/ClassCheck

Table of Contents

Report 2: Part 2

Report 1 Part 1:

DESCRIPTION:

In many people's lives as a student at Rutgers University, the issue of attendance in classes is a prevalent one. It is often time consuming and confusing for professors to figure out an efficient way to take attendance without having to roll call the entire class, while maintaining integrity of going to class. The solution to this is to create a location-based system that automatically marks attendance when a student enters the classroom area. The student will have to input their schedule and their class locations and the system will automatically mark attendance when the student enters the classroom area. ClassCheck is a web-based platform that makes it easier for both students and teachers to track attendance. It aims to replace the manual attendance tracking by offering a user-friendly platform that records and stores a student's attendance data. There are three key features on this app. Unregistered users can access a quick demo showing the registration process, which includes entering their NETID and selecting their classes and teachers. They will also have access to a map of Rutgers University. Registered students will be able to login using their NETID and input their class schedules with assigned teachers. They can view their schedules, see class locations pinned on the map, and track their attendance for each class. Finally, registered teachers can sign in with their email, input their class schedule, and gain access to rosters and individual student attendance records. If a student is at the class location at class time, there will be a P for present marked in their attendance sheet. Similarly, if they are not at the location at class time an A for absent will be marked. If a class is cancelled by the teacher that day's attendance will not be counted.

The project focuses on backend development using HTML, CSS and JavaScript to ensure data management and API integration.

Authentication services will be integrated using third party services like institutional login (NetId) or Google. Geolocation services will be integrated using third party services such as Google Maps API.

DATA SOURCE:

For the purposes of development and testing, ClassCheck will use synthetic data. This involves sample user accounts (students and instructors), class schedules, and attendance records made manually or using scripts. Synthetic data ensures privacy while also allowing for the simulation of a variety of use cases. However, we will integrate real-world data in specific areas through the use of third-party APIs like for the geolocalisation and authentication. The real world data will be the class coordinates which can be compared to the student's device coordinates when the attendance is being marked.

DATA COLLECTIONS AND ACCESSIBILITY:

ClassCheck organizes its data into structured collections in order to ensure seamless data retrieval and security measures for different user roles:

- ➔ **User Profiles:** Stores personal information, roles, and authentication details of users.
- ➔ **Attendance Records:** Logs timestamps, geolocation data, class sessions, and attendance status for each student.
- ➔ **Class Schedules:** Manages course timings, assignments, and attendance requirements.
- ➔ **Attendance Exceptions:** Stores delay reports and instructor-approved attendance overrides.
- ➔ **University map:** show the university map and class locations.

These collections are shared among registered users and some are available for public access.

For registered users: Professors, students, and administrators who have authenticated access through third-party authentication services. They will have access to these data collections: **user profiles, attendance records, class schedules, attendance exceptions** and **university map.**

For unregistered users: Limited to informational content like quick demo for registration process and an insight of the platform. They will have access to a **university map** and **user registration option.**

BUSINESS OPERATIONS ON DATA COLLECTIONS:

Every data collection in ClassCheck facilitates a number of procedures designed to improve functionality:

- **User Profiles:**
    - Register new users (students, professors, administrators).
    - Authenticate users based on credentials.
    - Modify or update user roles (e.g: upgrade a user to professor status)
- **Attendance Records:**
    - Mark attendance manually or via automated check-in with geolocation.
    - Retrieve and display attendance history per student.
    - Generate attendance reports for administrative review.
    - Prevent attendance marking if students are outside the classroom location.
- **Class Schedules:**
    - Assign students to specific courses.
    - Update or modify schedules as needed.
    - Enforce attendance policies based on institutional requirements.
- **Attendance Exceptions:**
    - Allow students to submit delay reports due to transportation or other issues.
    - Enable instructors to review and approve/reject attendance modification requests.
- **University Map**
    - Allow students and visitors to retrieve map data.
    - Add or update building locations and classroom details.
    - Modify locations if classroom assignments change.

BUSINESS RULES/POLICIES:

- **Time Constraints:** Students must check in within the first 15 minutes of class to be marked present.
- **Location Validation**: Students must be physically inside the classroom to be considered present; students waiting outside will remain unverified until confirmed by the instructor.
- **Handling location Failures:** If attendance is not marked due to location issues, students can manually verify their attendance and professors can enter it manually.
- **Traffic/Public Transportation Delays:** Students can submit delay reports, which instructors can review and override attendance records when necessary.
- **Role-Based Permissions:** Professors and administrators have distinct privileges for modifying attendance data.
- **Security Protocols:** Unauthorized access attempts trigger security alerts and there can be some temporary access restrictions.
- **Custom Attendance Policies:** Institutions can define their own rules, such as excused absences and late arrival penalties.
- **Absence Notifications:** The system will notify students via email if they are marked absent to keep them accountable and also allow them to dispute inaccuracies.
- **Data Accuracy**: Attendance records cannot be change except by authorized users to ensure reliability

Search: The user can search for the description of business policies by typing them in the search bar.

REQUIREMENTS:

1. **REQ-1: Attendance Not Marked Due to Disabled or Inaccurate Geolocation**

Description: The system must detect when a student arrives at their scheduled class location but fails to be marked present due to disabled or inaccurate geolocation services.

Business Concepts/Operations:

- Relies on geolocation tracking to verify attendance
- Allow students to manually verify if tracking fails
- Instructors can review and override attendance records when necessary

User Story: As a student, I want to be notified if my geolocation is inaccurate or disabled so that I can take appropriate action to verify my attendance.

2. **REQ-2: Attendance Affected Due to Traffic/Public Transportation Delays**

Description: The system must allow students to report unexpected delays due to traffic or public transportation issues and request attendance exceptions.

Business Concepts/Operations:

- Students experiencing delays due to traffic or public transport can submit delay reports
- Instructors review these requests and approve or reject them based on validity

User Story: As a student, I want to report transportation delays so that my attendance can be considered despite circumstances beyond my control.

3. **REQ-3: Student Marked Present While Waiting Outside the Classroom (This was implemented because the maxDistanceAllowed between user and classLocation is**

miniscule, meaning user has to be in same room as class)

Description: The system must prevent students from being marked present if they are outside the classroom but still within the geographic attendance radius.

Business Concepts/Operations:

- The system detects when a student is near but not inside the classroom.
- If flagged, instructors review and confirm actual attendance.
- Students must enter the classroom to be considered present, and those flagged outside remain unverified

User Story: As an instructor, I want to ensure that students are physically inside the classroom when marked present so that no false attendance statuses are recorded.

4. **REQ-4: Multi-Factor Authentication for Secure Login (This was implemented using Auth0)**

Description: The system must implement multi-factor authentication (MFA) for registered users to enhance security

Business Concepts/Operations:

- Ensures secure login for students and teachers
- Prevents unauthorized access to attendance data
- Supports authentication via email or app-based verification

User Story: As a registered user, I want an additional layer of authentication when logging in so that my account and attendance records remain secure.

5. **REQ-5: Attendance Summary and Reports for Teachers**

Description: The system must generate weekly and monthly attendance reports for teachers to monitor student participation

Business Concepts/Operations:

- Provides data visualizations of attendance trends
- Flags students with excessive absences
- Allows teachers to export reports in CSV/PDF formats

User Story: As a teacher, I want to see attendance reports so that I can track student engagement and identify at-risk students

6. **REQ-6: Class Location Verification & Updates**

Description: The system must allow teachers to verify and update class locations in case of room changes

Business Concepts/Operations:

- Ensures accuracy in location-based attendance tracking
- Teachers can update class locations if the assigned room changes
- Notifications sent to students as location updates

User Story: As a teacher, I want to update my class location when necessary so that students are correctly tracked for attendance.

7. **REQ-7: Automated Attendance Notifications**

Description: The system must send push-notifications when it's time to head to class as well as after attendance has been successfully marked.

Business Concepts/Operations:

- Ensures students are aware of when attendance tracking will begin
- Students receive a notification if their attendance has been successfully tracked
- If error occurs, send notification to student so that they can speak to their instructor for override

User Story: As a student, I want to receive automated reminders about attendance tracking so I can make decisions to ensure my location is properly tracked

Due to time constraints, we weren't able to fully offer these highlighted requirements from Report 1, but we did implement an attendance exception portal where students can describe the reason for their absence, and on the professor's side of their respective dashboard, the professor can manually review the appeal requests and accept/reject the attendance exception.

GLOSSARY OF BUSINESS CONCEPTS:

1. **Attendance Tracking System**: a digital system designed to record and manage the attendance of students
2. **User Authentication**: a security process that verifies the identity of a user before granting access to the ClassCheck system.
3. **User Profiles**: Digital representations of individuals using the ClassCheck system, including students, professors, and administrators. Contains personal details, user roles, and authentication credentials.
4. **Push Notifications**: Alerts sent to students reminding them about attendance statuses, or issues related to attendance tracking.
5. **Data Accuracy & Integrity**: Ensures that attendance data is reliable and only modifiable by authorized personnel.
6. **Security Protocols**: Procedures and systems to detect and respond to unauthorized access, ensuring that sensitive data remains protected.
7. **Custom Attendance Policies**: ClassCheck's specific rules that dictate how attendance is recorded, such as rules for lateness, excused absences, and dispute resolution.
8. **Manual Override**: The ability for authorized users (e.g., instructors) to modify or input attendance data in case of errors or special circumstances.
9. **Role-Based Access Control**: A system that provides specific permissions and access rights based on the user's role (e.g., student, professor, administrator).
10. **University Map**: A geolocation-enabled map displaying classroom buildings and locations within Rutgers University, helping in verifying student presence.

11. **Attendance Reports**: Automatically generated summaries provided to instructors to review attendance trends and help students retrieve their attendance records.
12. **Geolocation Validation:** Comparing a student's real-time location to a predefined classroom location to verify presence.
13. **Attendance Exception:** A record submitted by students when they're unable to attend class due to valid reasons (e.g., transportation delay), pending the professor's approval or rejection.
14. **Notification Gateway:** The internal system module responsible for sending absence or attendance alerts to students and/or professors.

Absence Notification : An alert sent to a student informing them of the students absence for their scheduled class.

Report 1 Part 2:

USE CASES:

1. **UC1: Mark Attendance**

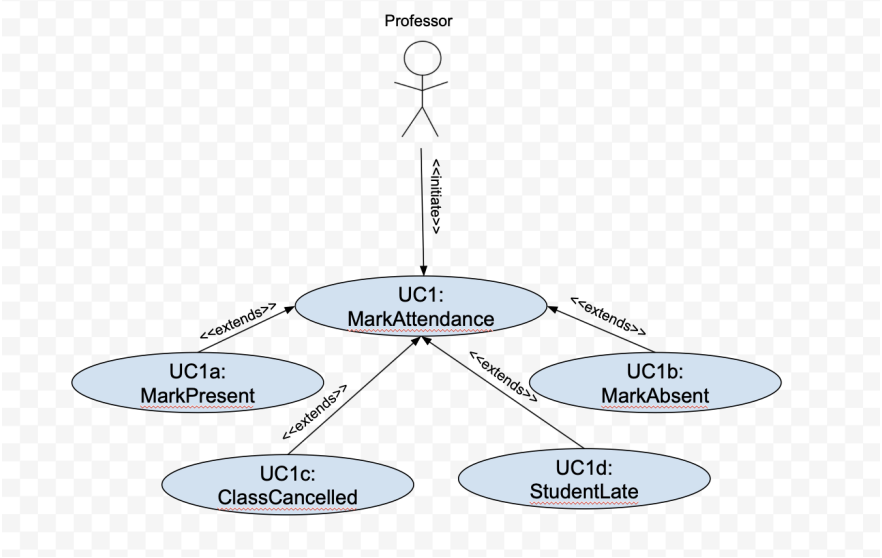
Owner: Nethra Sakthivel

-> UC1a: Mark Student Present

-> UC1b: Mark Student Absent

-> UC1c: Class Cancellation

-> UC1d: Student Arrives Late to Class



UC1	**MarkAttendance**

Related Requirements	REQ1, REQ2, REQ3, REQ6
Initiating Actors	Professor
Actor's Goals	To mark the attendance of the students in their class at that time
Participating Actors	Students and the Attendance System
Preconditions	• Class must be held that day • Students must be on roster • Class must be in the intended location
Minimal Guarantees	Students will be marked for attendance regardless of lateness
Success Guarantees	Students are accurately marked for their attendance for that class.
Flow of Events	1. Student picks the class of choice to be marked for attendance 2. The students geolocations will be updated (UC4) 3. Based on the geolocation of students, they are either marked as absent or present.

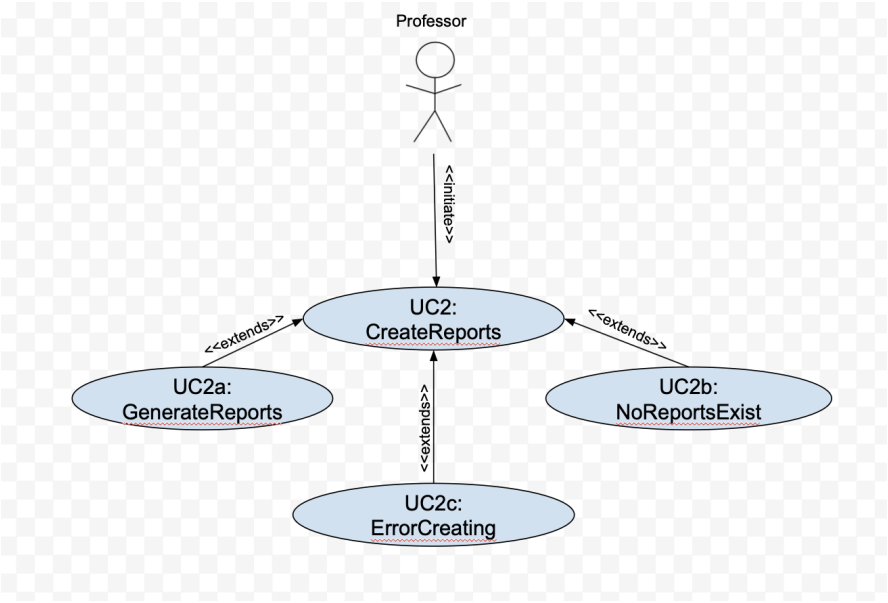
2. **UC2: Create Attendance Reports**

Owner: Rashmee Gade

-> UC2a: Generate Report

-> UC2b: No Report Exists for Intended Date

-> UC2c: Error in Generating Report



UC2	**CreateReports**
Related Requirements	REQ4, REQ5

Initiating Actors	Professor
Actor's Goals	To get attendance reports for a certain day
Participating Actors	Attendance System
Preconditions	• Class should be held that day • There should be data inputted for that day
Minimal Guarantees	The system does not generate an incomplete or inaccurate report.
Success Guarantees	An accurate attendance report is generated for the date desired.
Flow of Events	1. Students attendance is marked in the system 2. Professor initiates CreateReports 3. The system finds the data for the desired date 4. The system generates a report indicating the students who were present, absent

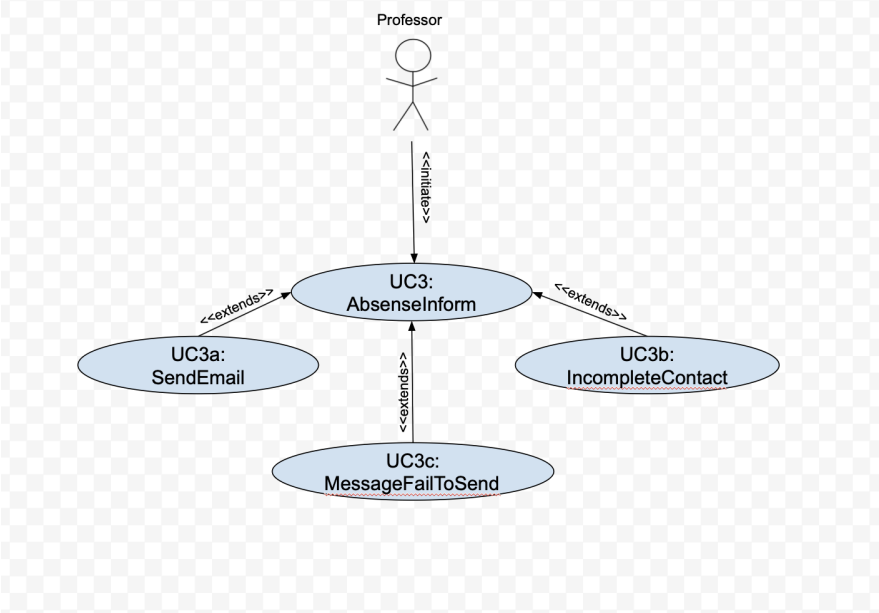
3. UC3: Inform Students of Absences

Owner: Navya Terapalli

-> UC3a: Send Email to Student

-> UC3b: Contact Information is Incomplete

-> UC3c: Message Fails to Send



UC3	**AbsenceInform**
Related Requirements	REQ1, REQ2, REQ3, REQ7
Initiating Actors	Professor
Actor's Goals	Have the system send the student a message

	about their absence from class
Participating Actors	Student and Attendance System
Preconditions	● The students geolocation must show that they are not at the intended class location ● Class must be held that day
Minimal Guarantees	Students do not receive incorrect notifications about absences.
Success Guarantees	Students receive an accurate notification about their absence.
Flow of Events	1. Teacher initiates MarkAttendance 2. The students geo-location is tracked 3. Students whose geolocation is not at class will be sent a notification regarding their absence from class

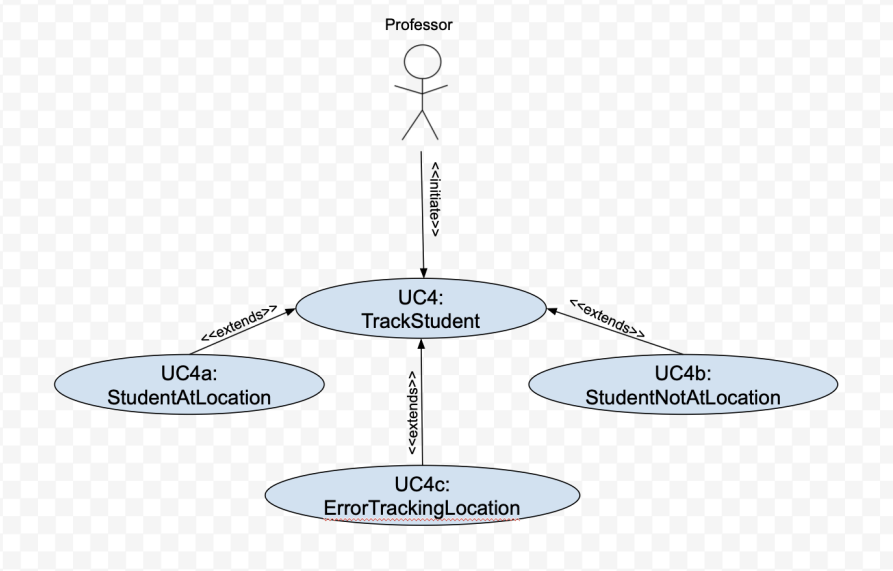
4. **UC4: Track Student Location**

Owner: Prachi Patel

-> UC4a: Student is at Class Location

-> UC4b: Student is not at Class Location

-> UC4c: Error Tracking Student Location



UC4	**TrackStudent**
Related Requirements	REQ1, REQ2, REQ3, REQ6
Initiating Actors	Professor

Actor’s Goals	Verify and log the students location during class time
Participating Actors	Students, Attendance System, GeoLocation API
Preconditions	● Students location services must be enabled ● The system must be working
Minimal Guarantees	Student tracking does not track the student if it was denied access.
Success Guarantees	The system logs the students location at the time of class.
Flow of Events	1. Professor starts to take attendance 2. The students are asked if they want their location tracked 3. Students' geo-locations are tracked to determine whether they are at the coordinates of class

5. **UC5: Authenticate The User**

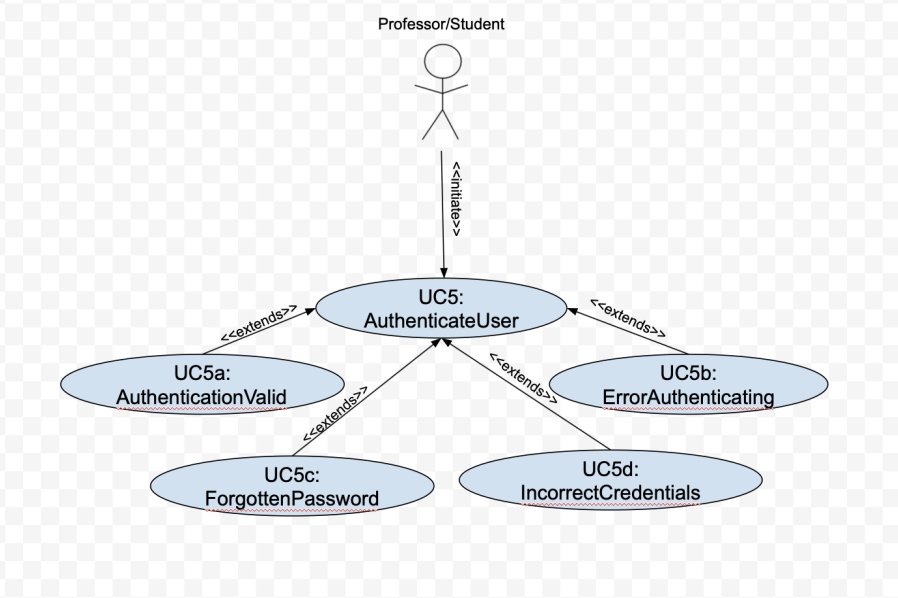
Owner: Absa Fall

-> UC5a: AuthenticationValid

-> UC5b: ErrorAuthenticating

-> UC5c: ForgottenPassword

-> UC5d: IncorrectCredentials



UC5	**AuthenticateUser**
Related Requirements	REQ4
Initiating Actors	Professor or Student
Actor's Goals	Successfully log into the attendance system

Participating Actors	Attendance System and User-Authentication API
Preconditions	● The user must input valid credentials ● The system must be working
Minimal Guarantees	Unauthorized users cannot gain access
Success Guarantees	Authorized users can gain access
Flow of Events	1. Professor/Students open the website 2. Professor/Students attempt to login to the system 3. The system verifies the credentials with the database 4. Professor/Student gains access and logs in

REPORT 1 CONTRIBUTIONS:

- **Prachi Patel** - Use Cases
- **Nethra Sakthivel** - Description
- **Rashmee Gade** - Requirements
- **Navya Terapalli** - Requirements
- **Absa Fall** - Business Operation/Data Collections & Rules/Policies

SUBTEAM STRUCTURE:

Frontend & Authentication System Team: works on attendance dashboard, schedule display, login UI and implement authentication forms

- **Members:** Absa Fall, Rashmee Gade
- **Tech Used:** HTML, JavaScript

Backend & Database Team: handle API requests for login and process attendance data, set up Firebase database

- **Members:** Prachi Patel, Navya Terapalli, Nethra Sakthivel
- **Tech Used:** FirebaseDB, Node.js, Postman

Report 2: Part 1

DOMAIN MODELS:

Entities:

- Users (netID, name, email, role, classes): Represents system users such as students and professors. In the database, students are stored under Users and professors are stored under Professors.
- Attendance Record (netID, classSession, status, date, time, location): Gets a student's attendance status for each class. In the database, this is stored under AttendanceRecords.
- Absence Notification (netID, studentStatus, message): Notifies students and professors when a student is marked absent. In the database, we will extract this data from AttendanceRecords.
- Class Schedule: This includes courseName, and represents a scheduled class for that semester and is linked to the latitude and longitude for that specific class. In the database, this information is stored under Classes.
- Location(Latitude , Longitude): The coordinates of where both the class is and where the student is at the time of their scheduled class. These coordinates for each class are stored under Classes and the coordinates for each student's attendance is in AttendanceReports.

Services:

LocationChecker(read) → Using geolocation services to track the students location, if the student is an allowable distance from the class location, this leads to the attendance marker

button. If the student is not an allowable distance from class location, this again leads to the attendance marker button.

AttendanceMarker(write) → If a student is at an allowable distance, AttendanceMarker writes present. If not, AttendanceMarker writes absent.

Collections:

User Collection: This data will be collected as the user logins into the system through the login page. The users will then be stored in our database. It will store the users email, netID, and role (professor or student).

Class Collection: This data will be collected manually and populated into the database. It will contain all the classes that are taking place, their location (longitude and latitude), the professor of that course, start time, and end time.

Attendance Record Collection: This data will be collected every time the mark attendance button is clicked initiating the markAttendance function. The data will be auto-populated in the database with the user's netID, course name, attendance status, and location (longitude and latitude).

API Validation: The input data for user collection will be validated by the role-based access control API and the authentication API. These APIs will help separate users by their role and give certain access to the respective roles based on their login.The input data for class collection will not be validated by an API because the data for those will be manually put into the database, however, the location data from each class will be used to validate the students attendance at class through the geolocation API. Attendance Record Collection will be validated using the

geolocation API as students' attendance will be marked accordingly based on their location in comparison to the class they are attending when the markAttendance use case is in play.

1. **UC1: MarkAttendance Pseudocode & Domain Model**

```
Function MarkAttendance(netID, courseName, latitude, longitude)
    classSession = getClassSession(courseName)

    classLocation = classSession.location
    distanceAllowed = dist // undetermined distance from classroom that is still valid

    studentDistance = classLocation - (latitude,longitude)

    if studentDistance > distanceAllowed
        status = absent
    if studentDistance < distanceAllowed
        status = present

    // create a new attendance record to put in the database
    attendanceRecord = {
        netID
        courseName
```

status

date

studentLocation // longitude and latitude of student

}

Workflow:

1. Fetch class information : User inputs class name ex : CS 201

   Expected response :

   coursename : “ —-----”

   Location: {

   Latitude: —------

   Longitude —---

   }

   Session date : —-----

   Session time : —-------

   }

   Error handling : if courseName is invalid or the session doesn't exist: raise

   classSessionNotFoundError

2. Calculate distance: uses geolocation services to find distance

   studentDistance = classLocation - (latitude,longitude)

   Error handling: Invalid coordinates

   { “ error: “Invalid latitude/longitude formatting”}

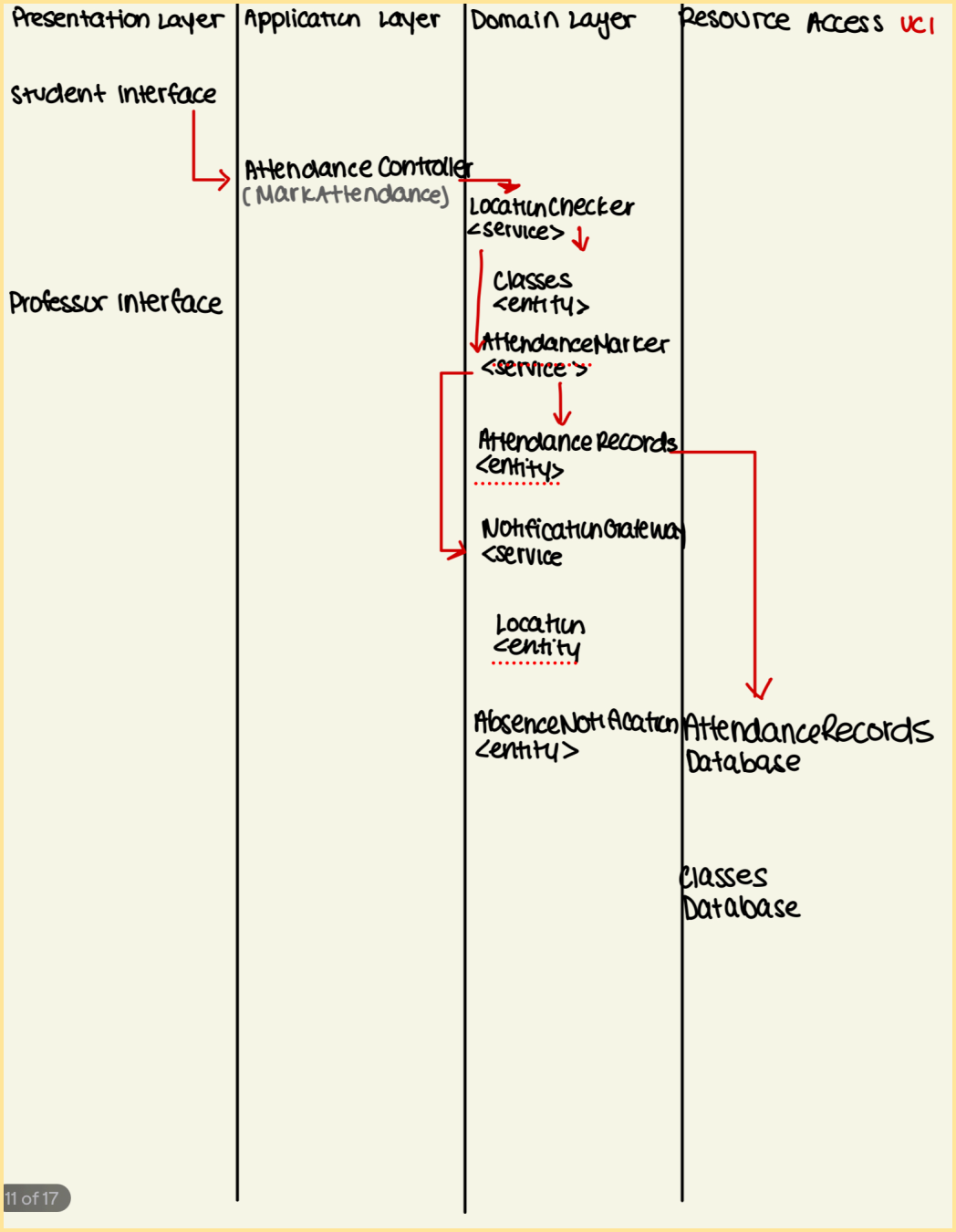
3. Validate distance

distance allowed will be defined later on

Error handling : if distanceAllowed not initialized, configurationError will rise

4. Create and save attendance Record : send to Attendance Database / student database

   Error handling : If attendance fails to save into database DatabaseInsertError {error: “ Could not save attendance record” }



Classes < entity> → stores class information like name, dates of the class, times, and location (longitude and latitude)

AttendanceRecordsDatabase< entity> → takes students' attendance and has information such as NETID, classes, physical location longitude and latitude.

Location<entity> → shows the longitude and latitude of class locations and can be used to compare the students location with the class location to see if the user is in class.

AbsenceNotification < entity> → The message that is sent to student and professor when a student is marked absent.

2. **UC2: CreateReports Pseudocode & Domain Model**

```
Function CreateReports(courseName, date)
    classSession = getClassSession(courseName)

    records = fetchAttendanceReports(date)

    If records(empty)
        Return error

    Reports = {}

    For Each record in records
        student = record.netID
        date = date
        status = record.status

    Return record
```

Workflow:

1. Fetch class information : User inputs class name and date

   Expected response :

   coursename : “ —-----”

   Location: {

   Latitude: —-------

   Longitude —---

   }

   Session date : —-----

   Session time : —-------

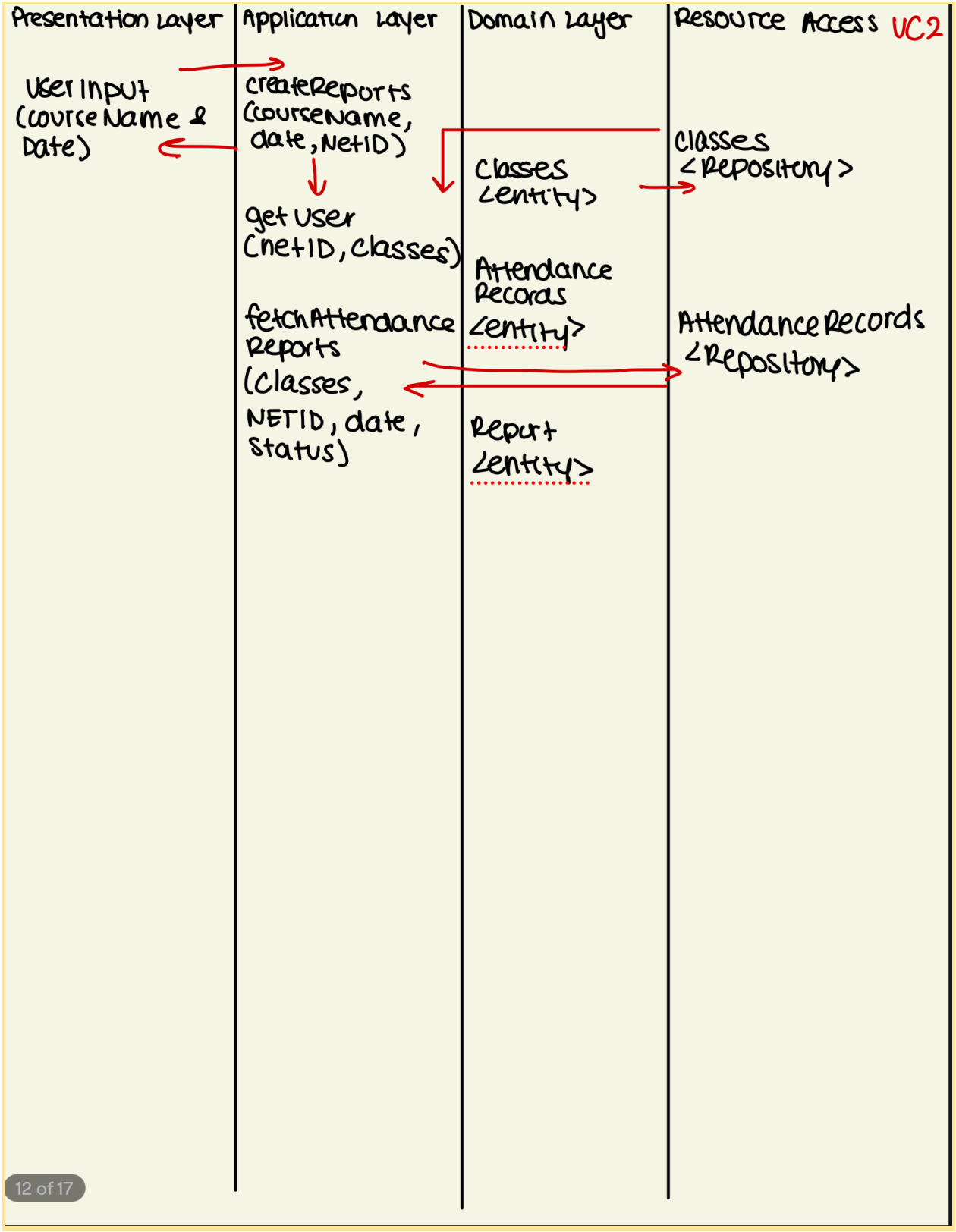
   }

   Error handling : if courseName is invalid or the session doesn't exist: raise

   classSessionNotFoundError

2. Get attendance records : request sent to attendance records database (repository)

   Error handling : If no records exist { "error": "No attendance records found }


Presentation Layer
Application Layer
Domain Layer
Resource Access UC2
User Input (courseName & Date)
createReports (courseName, date, NetID)
Classes <Repository>
Classes <Entity>
get User (netID, classes)
Attendance Records <Entity>
fetchAttendance Reports (classes, NETID, date, status)
Attendance Records <Repository>
Report <Entity>
12 of 17

Classes<entity> → Shows the session for a course and has information like course name, time and location coordinates. It validates whether a person is absent because of the class location coordinates.

AttendanceRecords <entity> → This is a record of attendance for all students in the class that a professor can view containing netID of students and their attendance for a particular day.

Report<entity> → the full list of attendance for a particular student on a specific class

classSession < repository >--> a repository that stores the classSession data and can be used to check if the class is in the repository before adding to the students list of classes.

AttendanceRecords<repository> → repository that stores data from AttendanceRecords entity, and fetches full attendance records for a date.

3. **UC3: AbsenceInform Pseudocode & Domain Model**

```
Function AbsenceInform(netID, date, courseName)
        classSession = getClassSession(courseName)
        studentName = getName(netID)

        records = fetchAttendanceReports(netID, date)

        if netID(empty) or date(empty)
                print “unable to send message”
                Return error
```

if record(empty)

print “unable to generate message”

Return error

message = studentName + netID + “ was absent on “ + date + “ in ” + classSession

SendNotification(studentName + message)

Return Success

Workflow: User inputs netID, courseName and date

1. Fetch class information : User inputs class name and date

   Expected response :

   coursename : “ —-----”

   Location: {

   Latitude: —------

   Longitude —---

   }

   Session date : —-----

   Session time : —------

   }

   Error handling : if courseName is invalid or the session doesn't exist: raise

   classSessionNotFoundError

2. Get studentsName : sent to user information sub system, student database

Error handling : invalid netId or user not found UserNotFoundError { "error": "Student with netID abc123 not found" }

3. Get attendance records : request sent to attendance records database (repository)

   Error handling : If no records exist { "error": "No attendance records found }

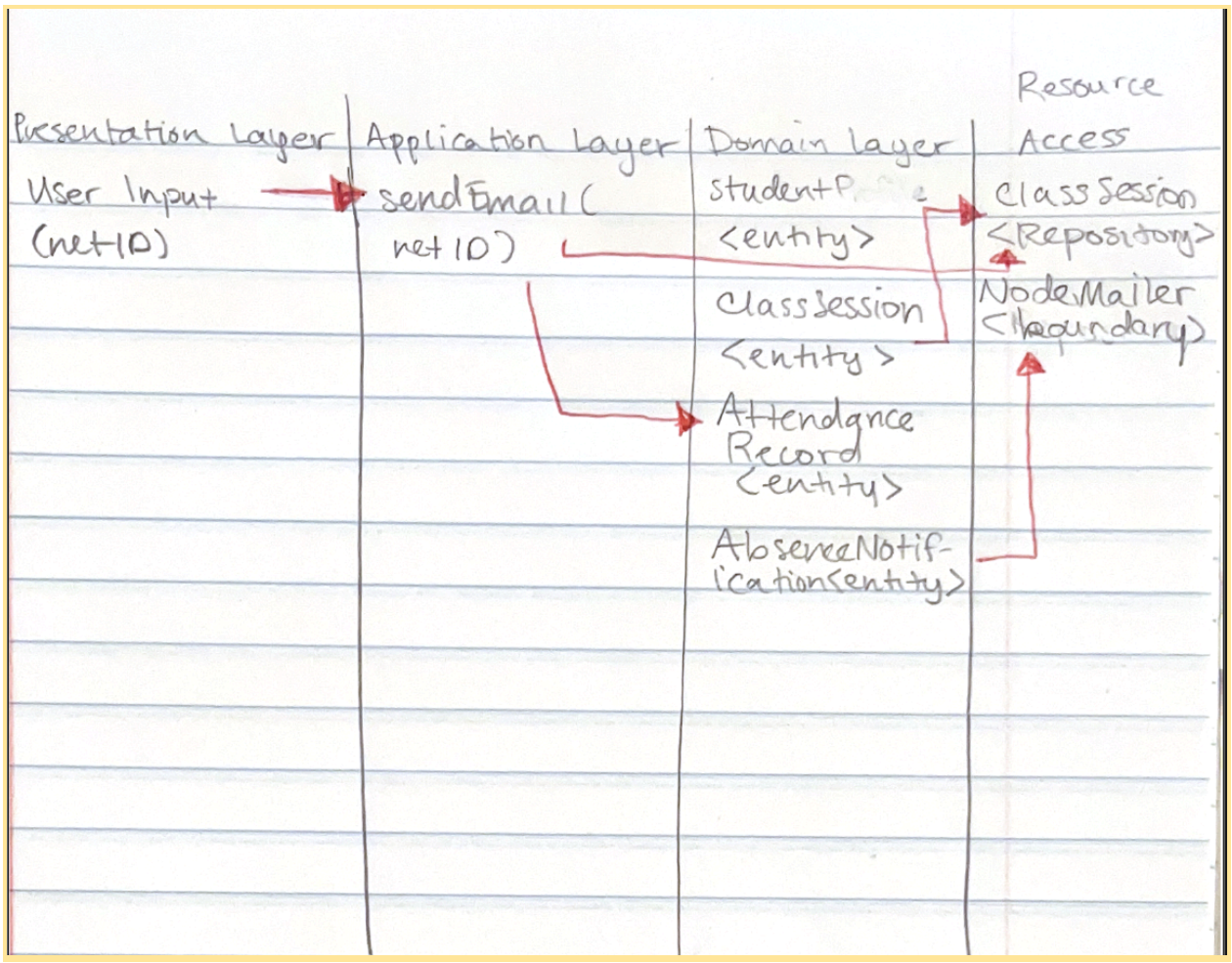
4. Validate inputs : make sure all inputs are valid : handled locally before sending to any databases

   Error handling : { "error": "Missing netID or date" }

5. Send notification to professor : notificationGateway

   Send to professor that _____ was present/absent on _____

   Error Handling : NotificationSendFailure { "error": "Failed to send absence notification" }



Classes<repository> → stores and retrieves class session data, verifying if the class session exists and matches the attendance record.

Attendance <repository> →

stores and retrieves AttendanceRecord entities, letting the system check if the student was marked absent for a given date and class.

User <repository> →

stores and retrieves Student entities. Fetches the student's name and email address using their netID.

NodeMailer<boundary> →

External dependency that sends the email notification and handles delivery to the student's email address.

4. **UC4: TrackStudent Pseudocode & Domain Model**

```
Function TrackStudent(netID)
    studentLocation = getGeolocation(netID)
    currentTime = getTime()

    if location(empty)
        print "error tracking students location"

    LocationRecord = {
        netID
        currentTime
        studentLocation
    }
```

Workflow: User inputs netID

1. Get students location : request sent to Geolocation service system

   Error Handling : GPS off or device is unreachable

Raise GeolocationUnavailableError :{ "error": "Unable to retrieve location for abc123" }

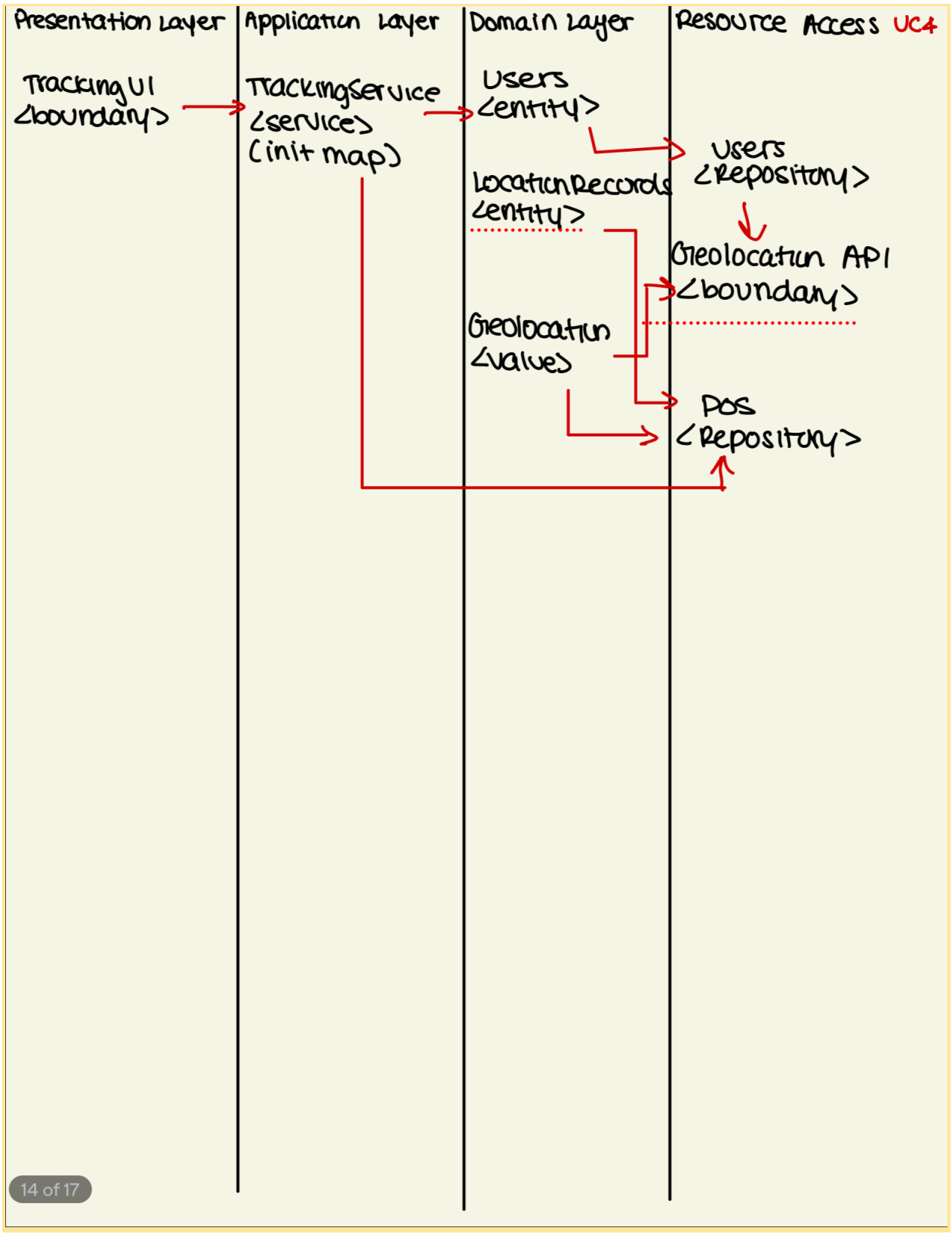
2. Validate location : using geolocation API services

   Error handling : student location unavailable : { "error": "Student location not available" }

3. Create location Records : save in location Repository

   Error handling : if location services produce error for student location

   LocationRecordInsertError:


Presentation Layer
Application Layer
Domain Layer
Resource Access UC4
Tracking UI
<boundary>
TrackingService
<service>
(init map)
Users
<entity>
Users
<Repository>
LocationRecords
<entity>
Geolocation API
<boundary>
Geolocation
<values>
POS
<Repository>
14 of 17

InitMap <service> →

Handles the process of checking the student's current location against the class location to determine if they are present.

LocationRecord <entity> →

Represents the student's latitude and longitude coordinates to record their location.

Geolocation <value> →

The actual latitude and longitude pair used to compare the student's location to the class location.

GeolocationAPI <boundary> →

External service that retrieves the geolocation data from the user..

Pos <repository> →

Stores and retrieves saved class position (pos) data or logged student location data

5. **UC5: AuthenticateUser Pseudocode & Domain Model**

```
Function AuthenticateUser(username, password)
    currentUser = getUserFromDatabase(username)

    if username(empty)
        print “Username is invalid”
        return error

    if currentUser.getPassword() !== password
        print “Incorrect Password”
        return error

    if currentUser.getPassword() == password
        print “user authentication is successful”
        return homePage.currentUser()
```

Workflow: User inputs username and password

1. Retrieve user from database : User credentials (username password pair) and is requested from userRepository

   Error Handling : if username is not found : UserNotFoundError :{ "error": "User not found" }

Password incorrect for the username password pair : PasswordError: {“ error”: “ Incorrect password “}

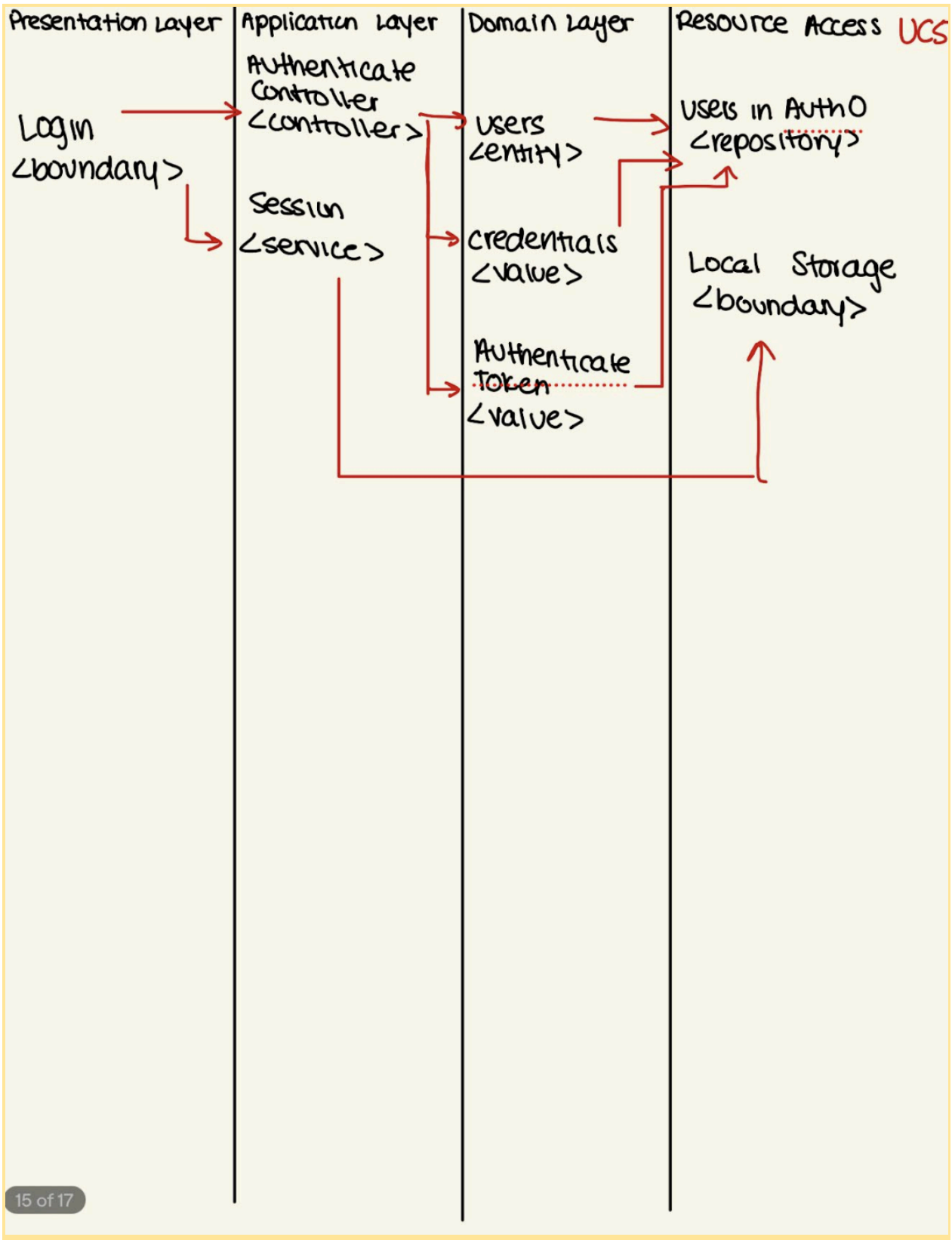
2. Input validation : local validation before calling userRepository

   Error handling : if username section is empty : { "error": "Username cannot be empty" }

3. Password verification : If username not matched from the credentials domain layer , raise AuthenticationFailedError: { "error": "Incorrect password" }

4. Successful authentication : go to home page

   Error handling : error : “ Try again “


Presentation Layer
Application Layer
Domain Layer
Resource Access
UCS
Authenticate Controller <controller>
Login <boundary>
Session <service>
Users <entity>
Credentials <value>
Authenticate Token <value>
Users in Auth0 <repository>
Local Storage <boundary>
15 of 17

Login <boundary> →

The user interface where the user enters their username, email, and password to log in.

AuthenticationController <controller> →

Handles the login request, checks the provided credentials, and handles authentication.

Session <service> →

Manages the creation and expiration of user sessions after login.

User <entity> →

Represents the user in the system and their user information like netID, name, email, password, and whether they are a student or professor

Credentials <value> →

Login credentials of the user for authentication.

AuthenticationToken <value> →

Creates a token after login for the user's authenticated session.

Users in Autho0 <repository> →

Stores and retrieves user entities with their netID's, emails, and passwords.

LocalS torage <boundary> →

Boundary that keeps track of active sessions and tokens.

Data Validation for API Operation:

- **GeoLocationValidation**
    - Used in: /attendance/mark
    - Validates: Student's current location vs. classroom coordinates
- **AttendancePolicyService**
    - Used in: /attendance/mark
    - Validates: Check-in timing, late arrival policy (15 min), prevents marking if too late
- **ExceptionHandlingService**
    - Used in: /exceptions/submit
    - Validates: reasons for delay; forward to instructor for approval or rejection
- **UserAuthService**
    - Used in: /user/login
    - Validates: User credentials via Auth0; assigns user role

DATABASE SCHEMA:

For this project, we will be utilizing **Firebase Realtime Database** to manage the structured data required by the ClassCheck attendance tracking system, since it is well-suited for handling complex relationships between entities such as students, professors, class schedules, and attendance records. Firebase provides a scalable, cloud-based solution that supports real-time data synchronization, making it well-suited for our use case of geolocation-based attendance tracking. The data structure is designed to model core functionalities such as student and professor profiles, class schedules, attendance records, and real-time updates. The following section outlines the Firebase data structure used to support ClassCheck's real-time functionality and role-based access.

DATABASE SCHEMA : NoSQL Database JSON Format:

- **Attendance:**

```
"Attendance": {
  "courseName": {
    "date": {
      "netid1": {
        "attendanceStatus": "text",
        "checkinTime": time,
        "latitude": num (coordinates),
        "longitude": num (coordinates),
        "netID": "text",
        "studentName": "text"
      },
      "netid2": {
        "attendanceStatus": "text",
        "checkinTime": time,
        "latitude": num (coordinates),
        "longitude": num (coordinates),
        "netID": "text",
        "studentName": "text"
      },
      "netid3": {
        "attendanceStatus": "text",
        "checkinTime": time,
        "latitude": num (coordinates),
        "longitude": num (coordinates),
        "netID": “text”,
        "studentName": “text”
      }
    }
  }
}
```

- **AttendanceExceptions**

```
"AttendanceExceptions": {
  "exc1": {
    "additionalNotes": “text”,
    "classID": “text”,
    "date": “text”,
    "professorNetID": “text”,
    "reason": “text”,
    "status": "text",
    "studentName": "text",
    "studentNetID": "text"
  },
  "exc2": {
    "additionalNotes": "text",
    "classID": "text",
    "date": "text",
    "professorNetID": "text",
    "reason": "text",
    "status": “text”,
    "studentName": “text”,
    "studentNetID": “text”
  },
  "exc3": {
    "additionalNotes": “text”,
    "classID": “text”,
    "date": “text”,
    "professorNetID": “text”,
    "reason": “text”,
    "status": “text”,
    "studentName": “text”,
    "studentNetID": “text”
  },
},
```

- **AttendanceRecords**

```
"AttendanceRecords": {
  "netID": {
    "recordID": {
      "courseName": “text”,
      "date": “text”,
      "netID": “text”,
      "status": “text”,
      "studentLocation": {
        "lat": num (coordinates),
        "lon": num (coordinates)
      }
    },
    "recordID": {
      "courseName": “text”,
      "date": “text”,
      "netID": “text”,
      "status": “text”,
      "studentLocation": {
        "lat": num (coordinates),
        "lon": num (coordinates)
      }
    },
    "recordID": {
      "courseName": “text”,
      "date": “text”,
      "netID": “text”,
      "status": “text”,
      "studentLocation": {
        "lat": num (coordinates),
        "lon": num (coordinates)
      }
    },
    "recordID": {
      "courseName": “text”,
      "date": “text”,
      "netID": "text",
      "status": "tex",
      "studentLocation": {
        "lat": num (coordinates),
        "lon": num (coordinates)
```

```
      }
    },
    "recordID": {
      "courseName": "text",
      "date": "text",
      "netID": "text",
      "status": "text",
      "studentLocation": {
        "lat": num (coordinates),
        "lon": num (coordinates)
      }
    },
  },
},
```

- **Classes:**

```
"Classes": {
   "courseName": {
     "lat": num (coordinates),
     "lon": num (coordinates)
   },
   "courseName": {
     "lat": num (coordinates),
     "lon": num (coordinates)
   },
   "courseName": {
     "lat": num (coordinates),
     "lon": num (coordinates)
   },
   "courseName": {
     "lat": num (coordinates),
     "lon": num (coordinates)
   },
   "courseName": {
     "lat": num (coordinates),
     "lon": num (coordinates)
   },
   "courseName": {
     "lat": num (coordinates),
     "lon": num (coordinates)
   },
   "courseName": {
     "lat": num (coordinates),
     "lon": num (coordinates)
   }
 },
```

- **Professors:**

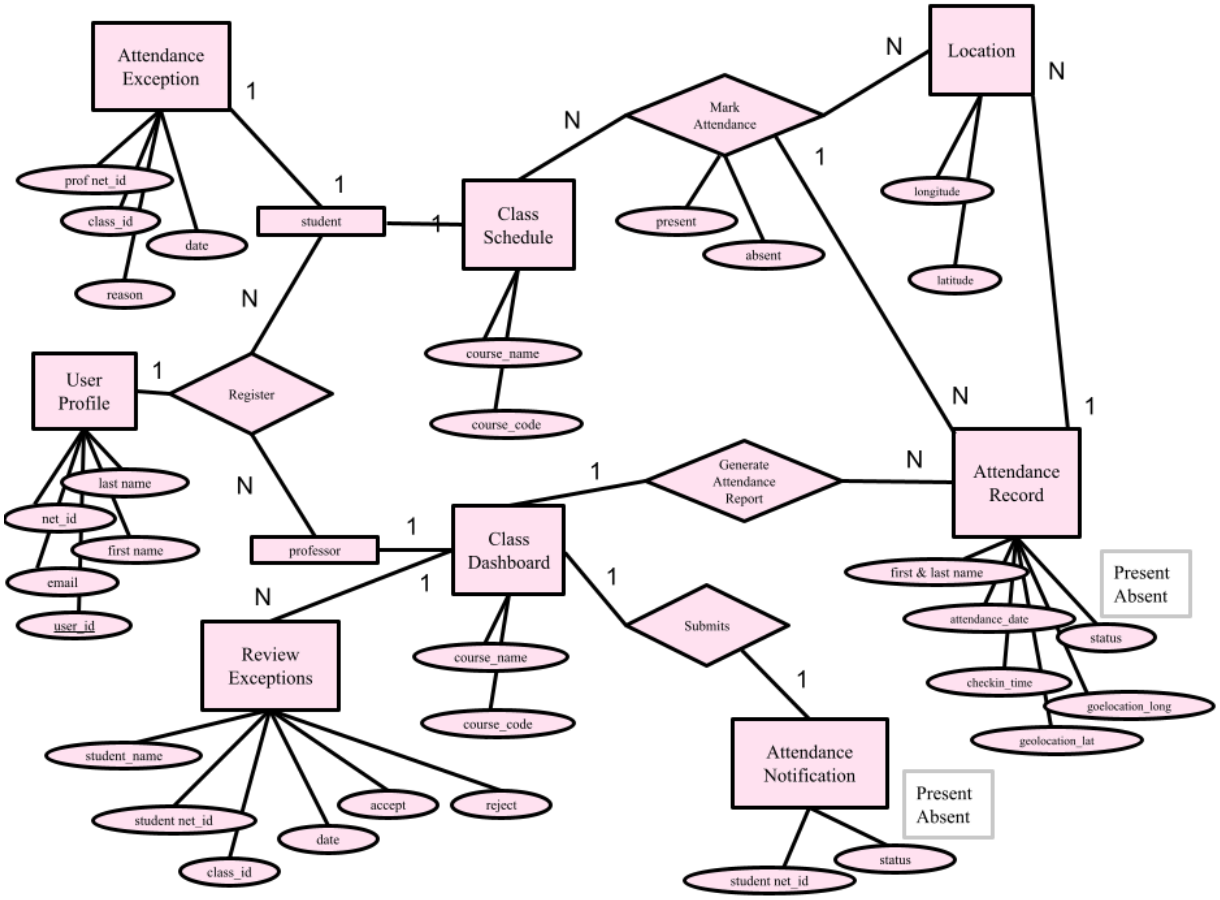
```
"Professors": {
   netIDP": {
     "classes": [
       "text",
       “text”
     ],
     "firstName": "text",
     "lastName": "text",
     "netID": "text",
     "password": "text",
     "role": "professor"
   },
   "netID": {
     "classes": [
       “text"
     ],
     "firstName": "text",
     "lastName": "text",
     "netID": "text",
     "password": "text",
     "role": "professor"
   },
},
```

- Users:

```
"Users": {
   "netID": {
     "classes": [
       "text",
       "text"
     ],
     "firstName": "text",
     "lastName": "text",
     "netID": "text",
     "password": "text",
     "role": "student"
   },
   "netID": {
     "classes": [
```

```
        "text"
      ],
      "firstName": "text",
      "lastName": "text",
      "netID": "text",
      "password": "text",
      "role": "student"
    },
},
```

ER Diagram - ClassCheck:



MULTIPLICITIES :

- UserProfile → 1 user : Multiple Attendance Records
- ClassSchedule → 1 class: Multiple Attendance Records
- Location → 1 student location : Multiple Attendance Records
- Attendance Record : 1 class → Multiple Attendance Records
- UserProfile → 1 professor : Multiple Classes on Dashboard
- User Profile → 1 student : Multiple Classes on Schedule
- Location → 1 Class Location : Multiple Class Schedules
- UserProfile → 1 student : Multiple exception requests
- AttendanceRecord → 1 record : 1 class
- AttendanceNotification → 1 notification : Submitted for 1 Attendance Record
- AttendanceException → 1 exception request : Submitted by 1 student

To ensure data privacy, security, and proper authorization within the ClassCheck system, each data collection (or database table) has clearly defined access permissions based on user roles. These roles—**students**, **professors**, and **administrators**—determine the level of interaction a user may have with each dataset, such as whether they can read, write, or modify records. For instance, students may only view their own attendance or submit exception requests, while professors manage class schedules and attendance records. Administrators have full access across all collections to maintain the system and enforce institutional policies. The table below outlines the access rights granted to each actor type for each data collection.

Access Controls for Each Data Collection:

Data Collection	Actor Type	Permission Type	Description
UserProfile	Students, Professors	Write-only (self)	Can create their own profile.
ClassSchedule	Students	Read-only	View their enrolled class schedule.
ClassDashboard	Professors	Read-only	Professors can view their classes.
AttendanceRecord	Professors	Read-only	View attendance reports (daily/student report), print/download reports.
AttendanceException	Students	Write-only (create)	Submit delay/absence requests with reasons.
	Professors	Both Read & Write	Review and approve/reject

			exceptions.
Location	Professors	Read-only	Used to validate location during attendance.
	Students	Read-only	View location details for class navigation.
MapData	Users (Students)	Read-only	Track user location with coordinates.

Report 2: Part 2

REST API SPECIFICATION:

Here we define the REST-based APIs for the operations listed in the first project proposal:

1. **User Profiles - Navya Terapalli**

Operation	**Resource URI**	**Method**	**Inputs**	**Returns**	**HTTP Response Code**	**Input data validation**
Register New User	/users/register	POST	name, netID, password	New user registered	201 Created, 409 Conflict	Email must be valid .edu format, password >8 characters
Authenticate user	/users/login	POST	netID, password	Validated user in database	200 OK, 401 Unauthorized	Email format valid, password not empty
Modify	/users/{id}	PATCH	role	netID, new	200 OK,	Role must

user role	/role			role	400 Bad Request, 404 Not Found	be 'student', 'professor', or 'admin'

2. **Attendance Records - Nethra Sakthivel**

Operation	**Resource URI**	**Method**	**Inputs**	**Returns**	**HTTP Response Code**	**Input data validation**
Mark attendance (manual/automatic)	/attendance/check-in	POST	netId, classId, location (lat, lon)	Status and attendanceID	200 OK, 403 Forbidden, 400 Bad Request	Location must be within range of classroom
Get student attendance history	/attendance/history/{userId}	GET	netID	A list of attendance records	200 OK, 404 Not Found	netID must exist
Generate admin report	/attendance/reports	GET	courseName, date	A report of attendance stats	200 OK, 400 Bad Request,	classID, valid date format

3. **Class Schedules - Absa Fall**

Operation	**Resource URI**	**Method**	**Inputs**	**Returns**	**HTTP Response Code**	**Input data validation**
Assign student to course	/classes/{classId}/enroll	POST	netID	A message confirming student has been assigned to a course	200 OK, 404 Not Found, 409 Conflict	nettId must be valid and not already enrolled
Update schedule	/classes/{classId}/schedule	PUT	newSchedule (JSON	Updated schedule info	200 OK, 400 Bad Request	time format valid, no conflicts
We did not implement this API operation because it did not align with						

our current goals.						

4. **Attendance Exceptions - Rashmee Gade**

Operation	**Resource URI**	**Method**	**Inputs**	**Returns**	**HTTP Response Code**	**Input data validation**
Submit exception request	/exceptions /submit	POST	Professor netID, classID, date, reason	requestID, action (accept or reject)	201 Created	reason not empty, valid datetime
Review Request	/exceptions /{id}/revie w	POST	decision (approve/re ject), professor netID	Status of decision	200 OK, 404 Not Found	Only instructors allowed to view requests

5. **University Map - Prachi Patel**

Operation	**Resource URI**	**Method**	**Inputs**	**Returns**	**HTTP Response**	**Input data validation**

					Code	
Retrieve map data	/map	GET	N/A	mapData (JSON)	200 OK	N/A
Add/update building	/map/buildings	POST	coordinates	userLocation	201 Created, 400 Bad Request	Coordinates format valid

THIRD PARTY APIS:

- We will be using a **geolocation maps service** which will serve as a locator for the browser and return the browser/devices location. This API will be used to determine the exact location of the browser that the web server is using. The use cases that use this API are UC1: MarkAttendance and UC4: TrackStudent.
    - Link: https://console.cloud.google.com/google/maps-hosted/discover/geolocation?inv=1&invt=Abtdlg&authuser=1&project=braided-verve-450919-c0
- We will also be using an **authentication service** which will be implemented when the user logs into the system. This API will be used to authenticate the user that is creating an account through the login page for the first time or trying to login to the website again. The use cases that use this API are UC5: AuthenticateUser.
    - Link: https://auth0.com/features/single-sign-on
- We will also be using a third-party database through Firebase Realtime Database. This will be implemented in the server to get user and professor profiles, class location information, student attendance information, and also post students attendance status. The use cases that use this API are UC1: MarkAttendance, UC2: CreateReports, UC3: NotifyStudentAbsence, and UC4: TrackStudent.
    - Link: https://causal-cacao-457203-s1-default-rtdb.firebaseio.com/

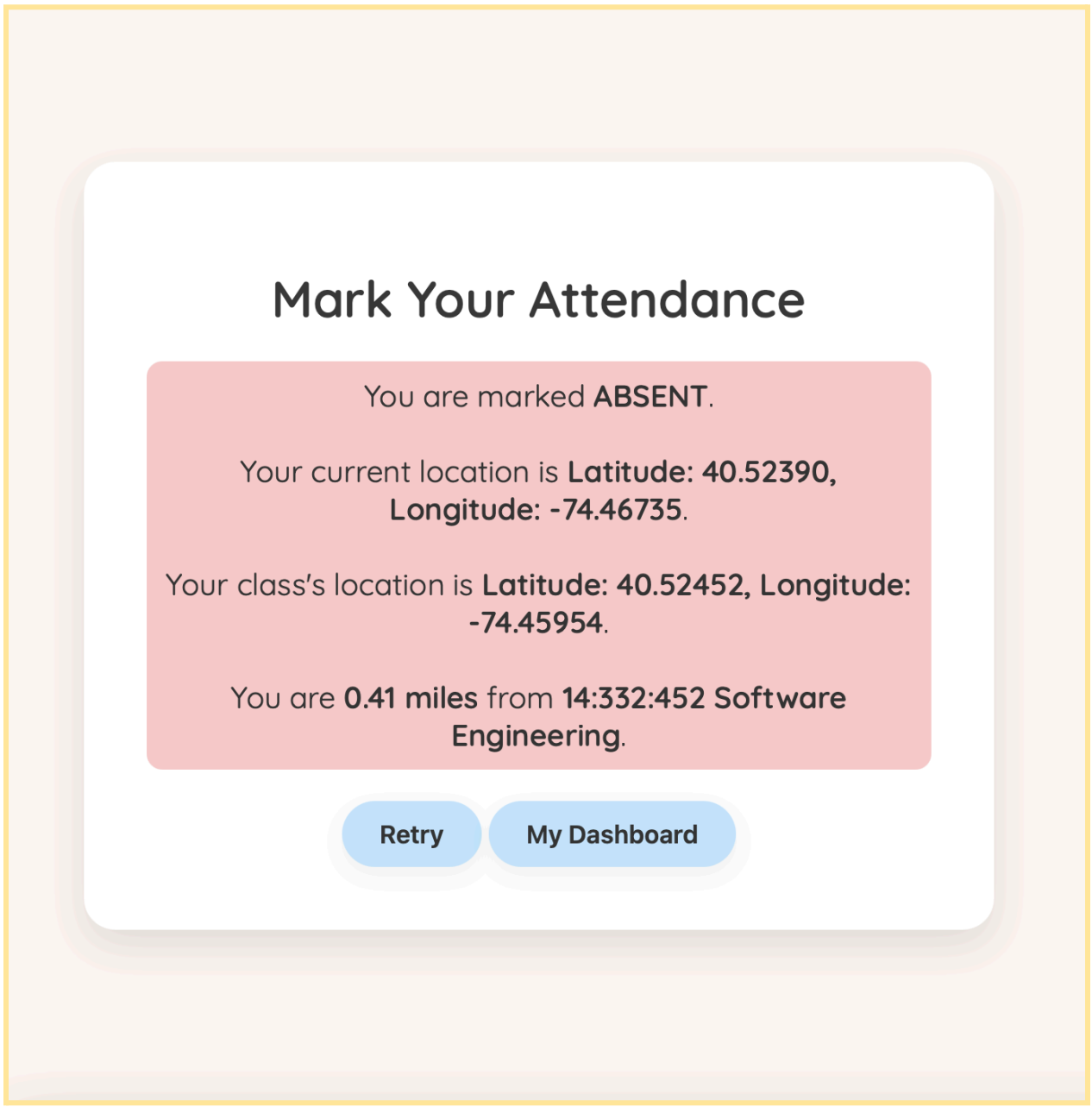
We didn't use the third party API: https://warrant.dev/use-cases/role-based-access-control/ we handled role based access with Auth0.

USE CASE OWNERSHIP:

- **UC1 – Mark Attendance**

  Owned by: *Nethra Sakthivel*

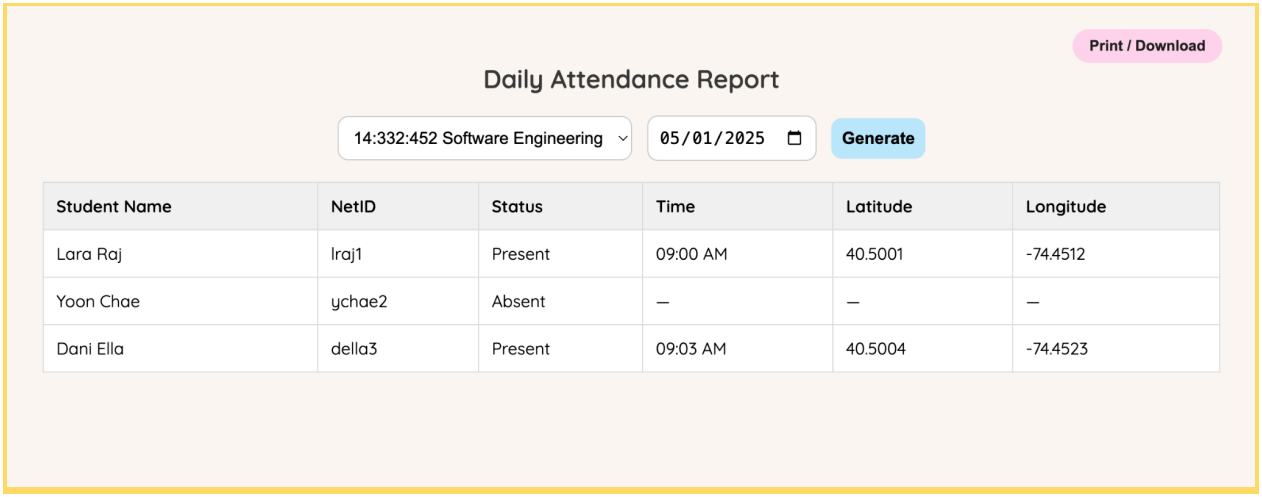
  Description: Enables students to mark their attendance as present or absent using geolocation.

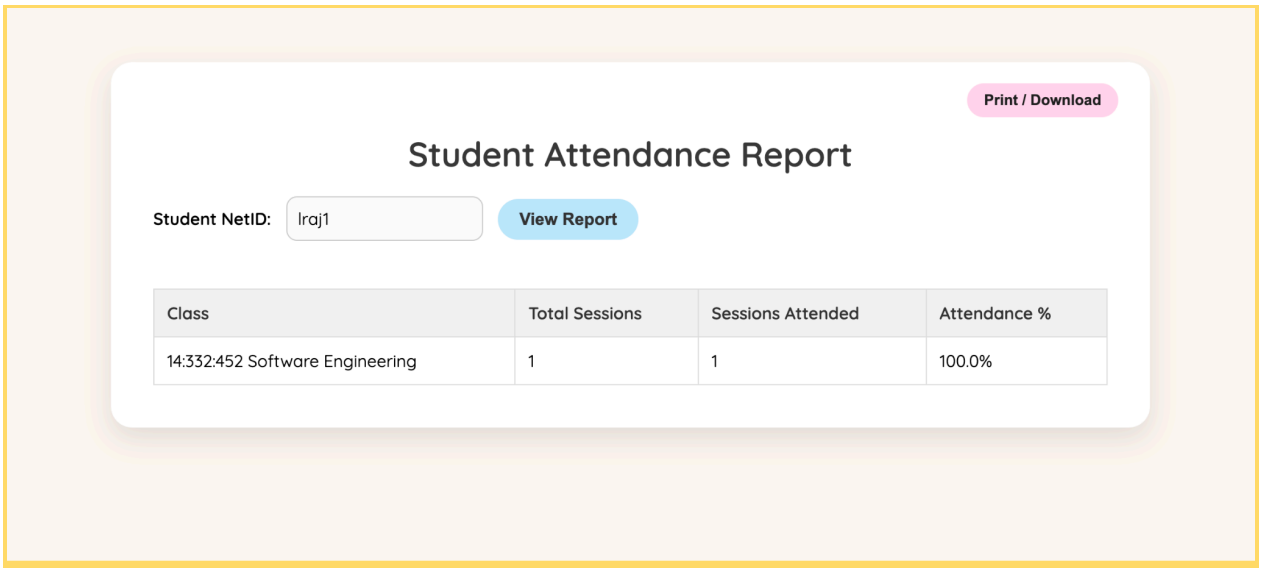


- **UC2A – Create Attendance Reports**

  Owned by: *Rashmee Gade*

  Description: Allows professors to generate reports showing attendance records in real time, with the options of viewing attendance through a daily report or a student attendance report.

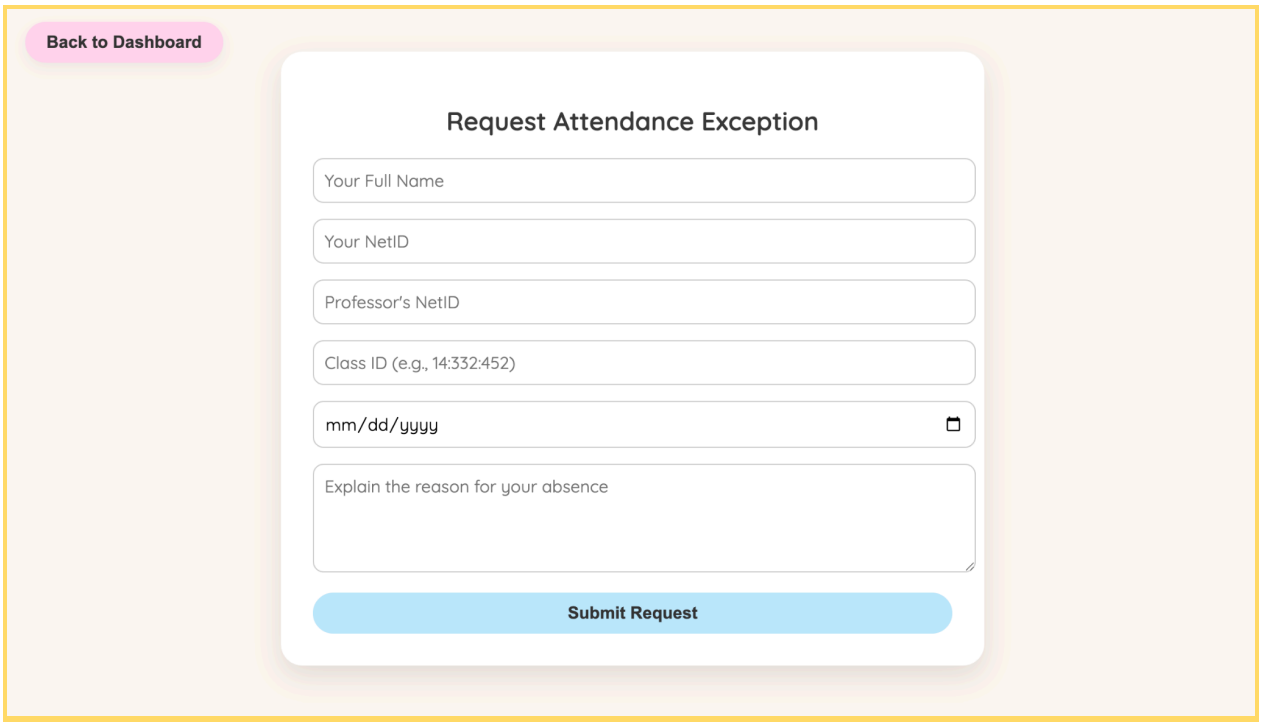


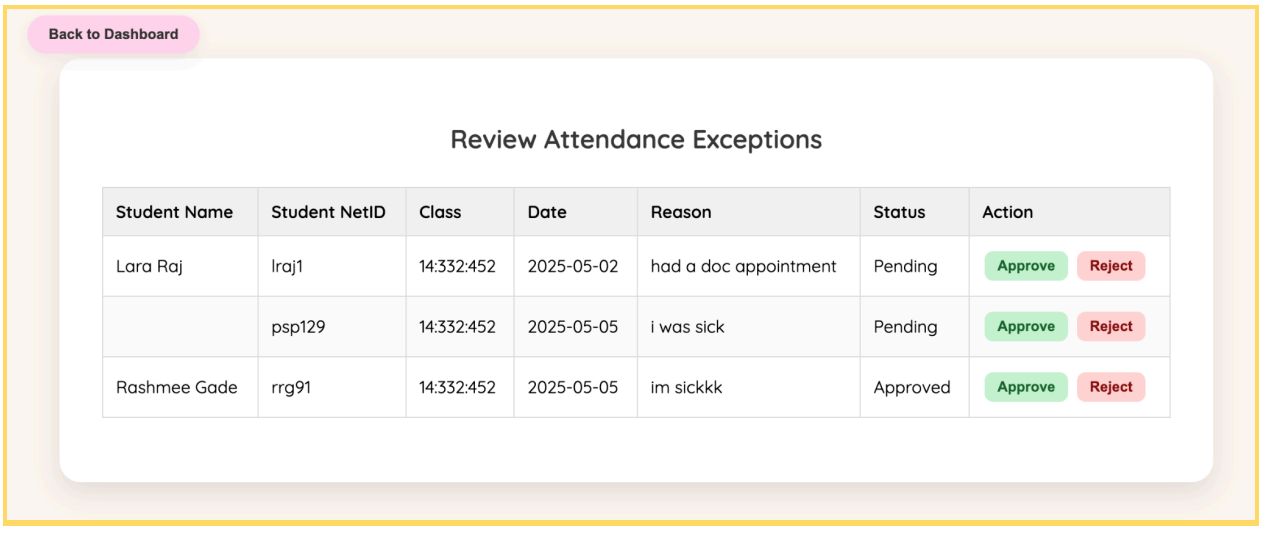


- **UC2B- Create Absence Exception**

Owned by: Rashmee Gade

Description: Allows students to send an absence exception appeal with information about their netID, date, and absence reason. Then the professor can review it from their dashboard and approve or reject the exception form.

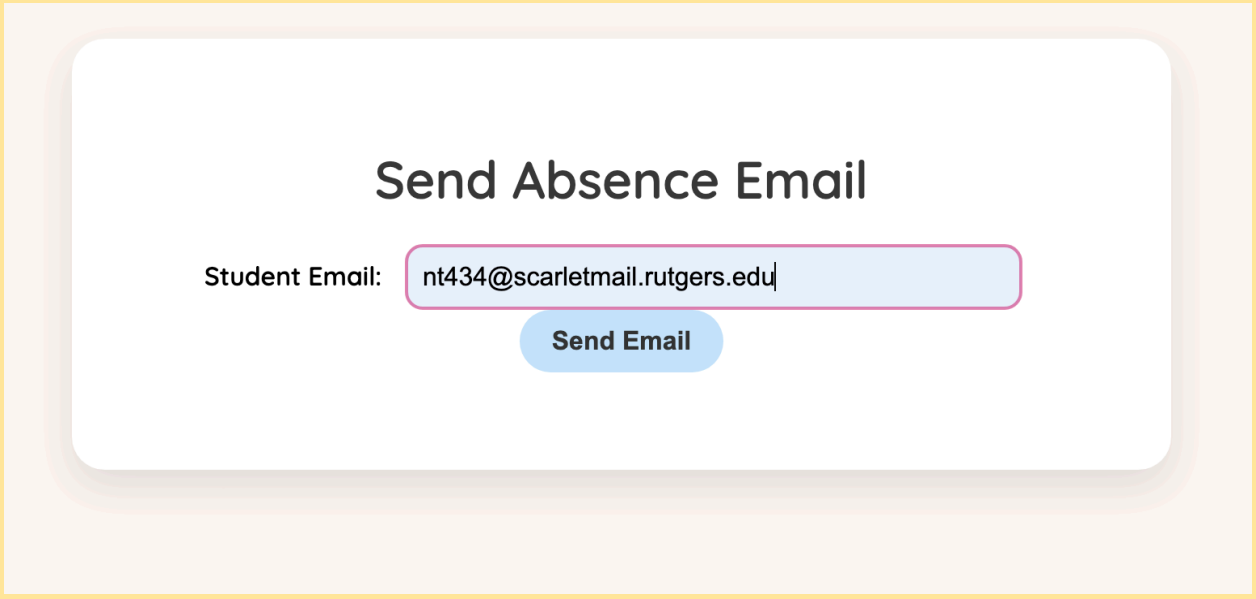


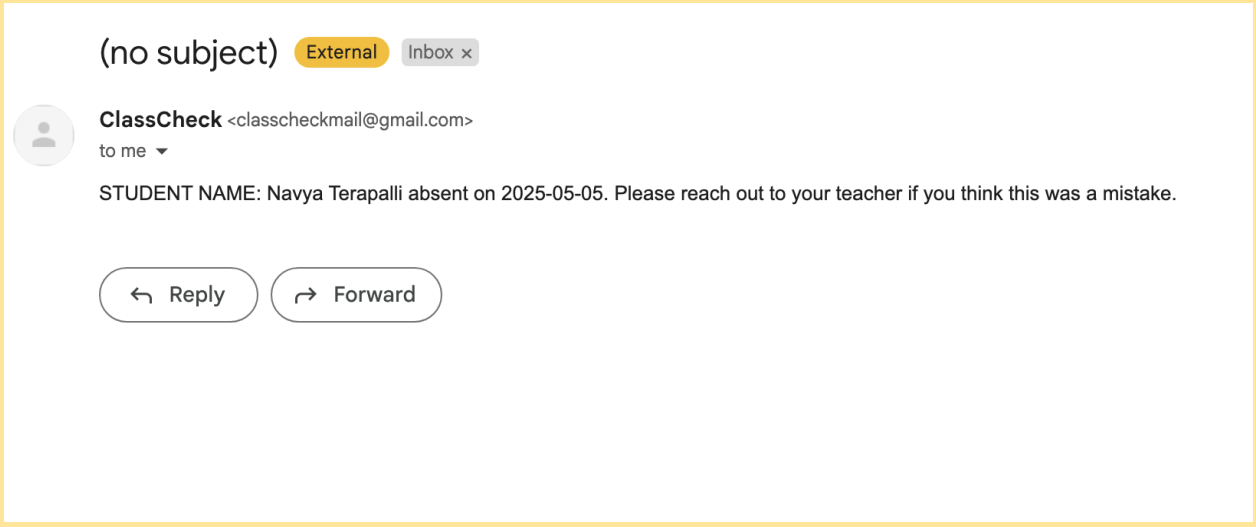


- **<u>UC3 – Inform Students of Absences</u>**

  Owned by: *Navya Terapalli*

  Description: Sends notifications from the professor's side to students who were absent from class. Tells them what class was missed and the date of absence.

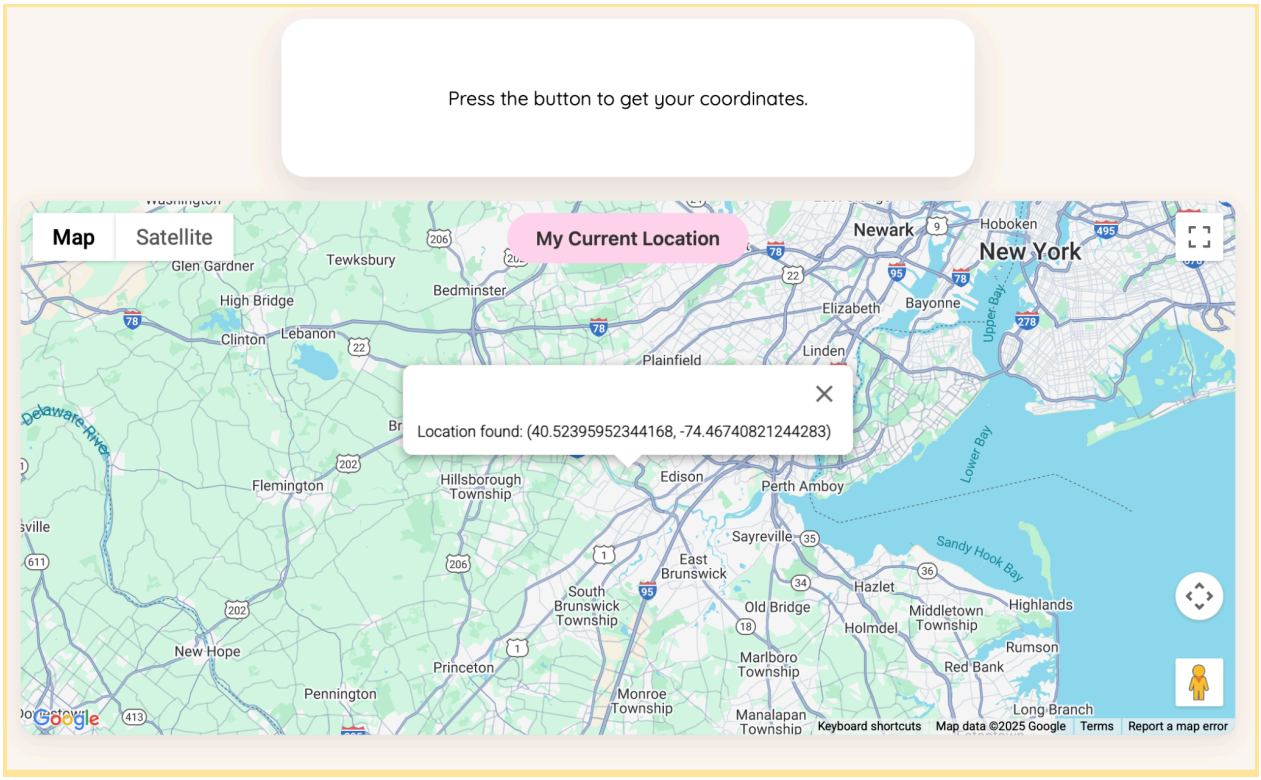




- **<u>UC4 – Track Student Location</u>**

  Owned by: *Prachi Patel*

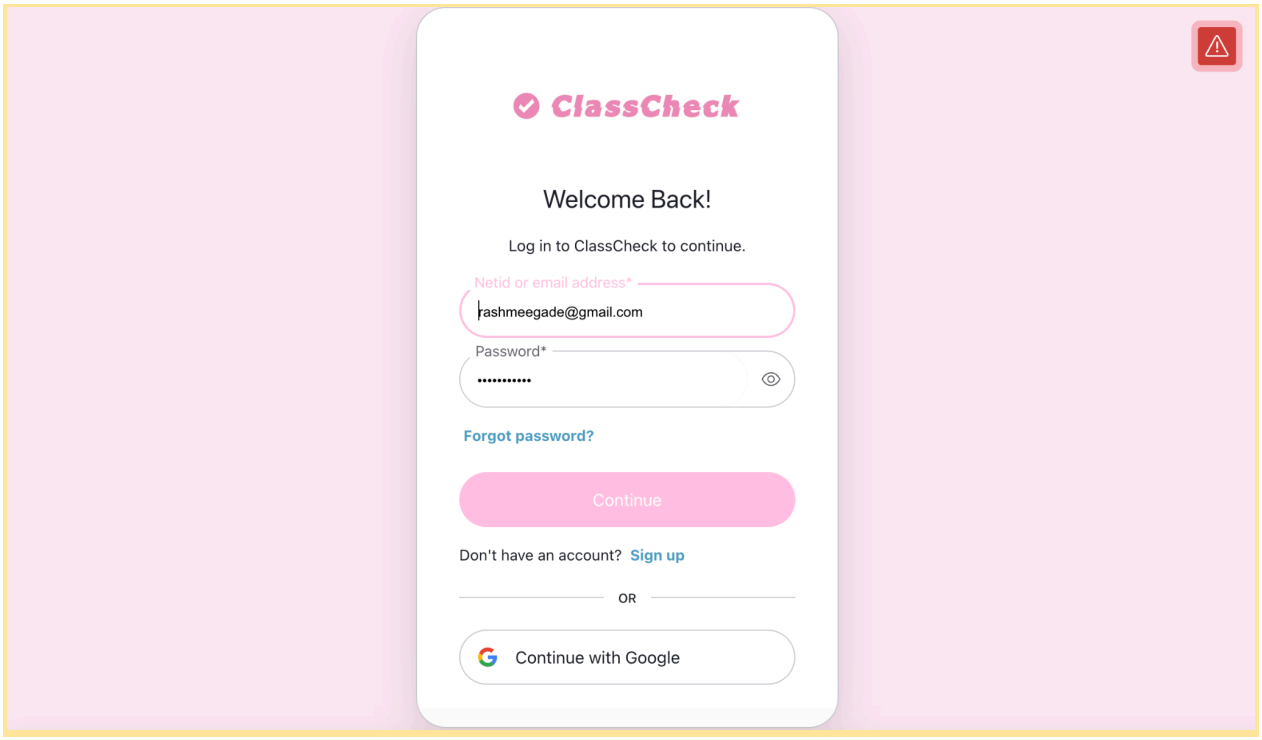
  Description: Get students location coordinates through a third-party API to use to mark their attendance in the system.



- **<u>UC5 – Authenticate the User</u>**

  Owned by: *Absa Fall*

  Description: Manages login process using third-party services such as Auth0 for secure access.



REPORT 2 CONTRIBUTIONS:

- **Prachi Patel** - Domain Model (User Collections, API Validation, Pseudocode) & Third-Party APIs
- **Navya Terapalli** - REST API Specification
- **Nethra Sakthivel** - All domain models visuals and pseudocode error handling
- **Rashmee Gade** - Data Schema Design (ER Diagram, SQL Table Definitions) & Access Control Permissions Mapping
- **Absa Fall** - Glossary of Business Concepts, Use Case Ownership, Data Source & Data Validation of API operation

Report 3

API IMPLEMENTATION:

- Firebase Realtime Database: **POST (create-profile)**

In this API, we are creating a new item in the Users collection of the Firebase Realtime Database if done through the student profile. This takes the firstName, lastName, netID, password, and classes, and posts this in the Users collection.

```
// Create Profile for Student API
app.post('/create-profile', async (req, res) => {
  const { netID, firstName, lastName, password, classes, role } = req.body;

  if (!netID || !firstName || !lastName || !password || !classes || classes.length === 0) {
    return res.status(400).json({ error: 'Missing a required field. Please fill out all parts of the profile.' });
  }

  try {
    const userRef = db.ref(`Users/${netID}`);
    await userRef.set({
      firstName,
      lastName,
      password,
      classes,
      role,
    });

    res.status(200).json({ message: 'Profile created successfully!' });
  } catch (err) {
    console.error('Error saving profile:', err);
    res.status(500).json({ error: 'Failed to save profile' });
  }
});
```

- Firebase Realtime Database: **POST (create-teacher-profile)**

In this API, we are creating a new item in the Professor collection of the Firebase Realtime Database if through the professor profile. This takes the firstName, lastName, netID, password, and classes, and posts this in the Professors collection.

```
// Create Profile for Professor API
app.post('/create-teacher-profile', async (req, res) => {
  const { netID, firstName, lastName, password, classes, role } = req.body;

  if (!netID || !firstName || !lastName || !password || !classes || classes.length === 0) {
    return res.status(400).json({ error: 'Missing a required field. Please fill out all parts of the profile.' });
  }

  try {
    const profRef = db.ref(`Professors/${netID}`);
    await profRef.set({
      firstName,
      lastName,
      password,
      classes,
      role,
    });

    res.status(200).json({ message: 'Profile created successfully!' });
  } catch (err) {
    console.error('Error saving profile:', err);
    res.status(500).json({ error: 'Failed to save profile' });
  }
});
```

- Firebase Realtime Database: **GET (get-user)**

In this API, we are retrieving/querying data from the database, based on the netID attribute. Through this we are able to get the users classes that they have selected and their personal information like name and netID through the User collection.

```
//Get student user from database API
app.get('/get-user', async (req, res) => {
  const netID = req.query.netID;

  if(!netID){
    return res.status(400).json({error:' Missing netID, cannot get user'});
  }

  try {
    const userRef = db.ref(`Users/${netID}`);
    const snapshot = await userRef.once('value');
    const userData = snapshot.val();

    if(!userData){
      return res.status(400).json({error: 'Missing user data/user not found'});
    }

    return res.status(200).json(userData);
    } catch (error) {
      console.error('Error saving profile', error);
      res.status(500).json({error: 'Failed to get profile'});
  }
});
```

- Firebase Realtime Database: **GET (get-prof)**

In this API, we are retrieving/querying data from the database, based on the netID attribute. Through this we are able to get the professors' classes that they have selected and their personal information like name and netID through the Professors collection.

```
// Get professor from database API
app.get('/get-prof', async (req, res) => {
  const netID = req.query.netID;

  if(!netID){
    return res.status(400).json({error:' Missing netID, cannot get user'});
  }

  try {
    const profRef = db.ref(`Professors/${netID}`);
    const snapshot = await profRef.once('value');
    const userData = snapshot.val();

    if(!userData){
      return res.status(400).json({error: 'Missing user data/user not found'});
    }

    return res.status(200).json(userData);
    } catch (error) {
      console.error('Error saving profile', error);
      res.status(500).json({error: 'Failed to get profile'});
  }
});
```

- Firebase Realtime Database: **GET (get-classes)**

In this API, we are retrieving the data from the database based on the netID and studentClasses attributes. Through this we are able to get the classes that the user has registered for whether it is a student or professor from either the User collection and Professor collection. The **Users and Classes collections** are **linked** as the Users collections takes the classes data from the Classes collection when they check off their specified classes in the profile.html file.

```
// get users classes from firebase
app.get('/get-classes', async (req, res) => {
  try {
    const classesRef = db.ref('Classes');
    const snapshot = await classesRef.once('value');
    const data = snapshot.val();
    res.status(200).json(data);
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch classes' });
  }
});
```

- Google Maps Geolocation: **POST (save-location)**

In this API, we are taking the user's coordinates which are retrieved on the front-end in the geolocation.html file by an API call where the browser asks the user whether they allow for the browser's location to be tracked. This data is then used on the backend in attendance.html to mark the students attendance status. We are then creating a new item in the AttendanceRecords collection based on the netID of the user. This takes the location data of the user and saves that along with the attendance status of the class from the Classes collection to create a new item to save in. At each instance, a new item is created with this save-location POST.

```
// save users location to firebase
app.post('/save-location', async(req, res) => {
  const {netID, courseName, status, date, time, lat, lon} = req.body;

  if (!netID || !courseName || !status || !lat || !lon) {
    return res.status(400).json({ error: 'Missing required fields.' });
  }

  if(!status){
    return res.status(400).json({error: "Missing students attendance status."});
  }

  try {
    const userRef = db.ref(`AttendanceRecords/${netID}`);
    await userRef.push({
      netID,
      courseName,
      status,
      date: date || new Date().toISOString().split('T')[0],
      time,
      lat,
      lon
    });

    res.status(200).json({message: 'Attendance saved!'});
  } catch (err) {
    console.error('Error saving attendance', err);
    res.status(500).json({error: 'Failed to save attendance'});
  }
});
```

- Firebase Realtime Database: **GET (get-student-attendance)**

In this API, we are retrieving/querying data from the database, based on the netID attribute. Through this we are able to get the attendance records for their classes that they have marked their attendance for through the AttendanceRecords collection. This is called in the studentReports.html file when trying to gather records based on the netID.

- Auth0: **GET (/login)**

In this API, we are using the server function res.oidc.login() which redirects the user to the auth0 login page for authentication. The login process begins when the user clicks on the "Login" from the index page which triggers a GET request from the auth0 endpoint. The parameters such as scope, client_id, redirect_uri are passed to define what type of user information is being requested, which application in Auth0 is requesting access and where the user should be redirected to after a successful login.

Parameter	**Value**	**Purpose**
response_type	code	Initiates Oauth code flow
client_id	VJwDlz26K2Duzl8GpWxOTHPpAnxs5nPE	Identifies ClassCheck application
redirect_uri	https://absa1072.github.io/ClassCheck/callback.html	Where auth0 redirects after login
Scope	openid profile email	Requests access to basic user info

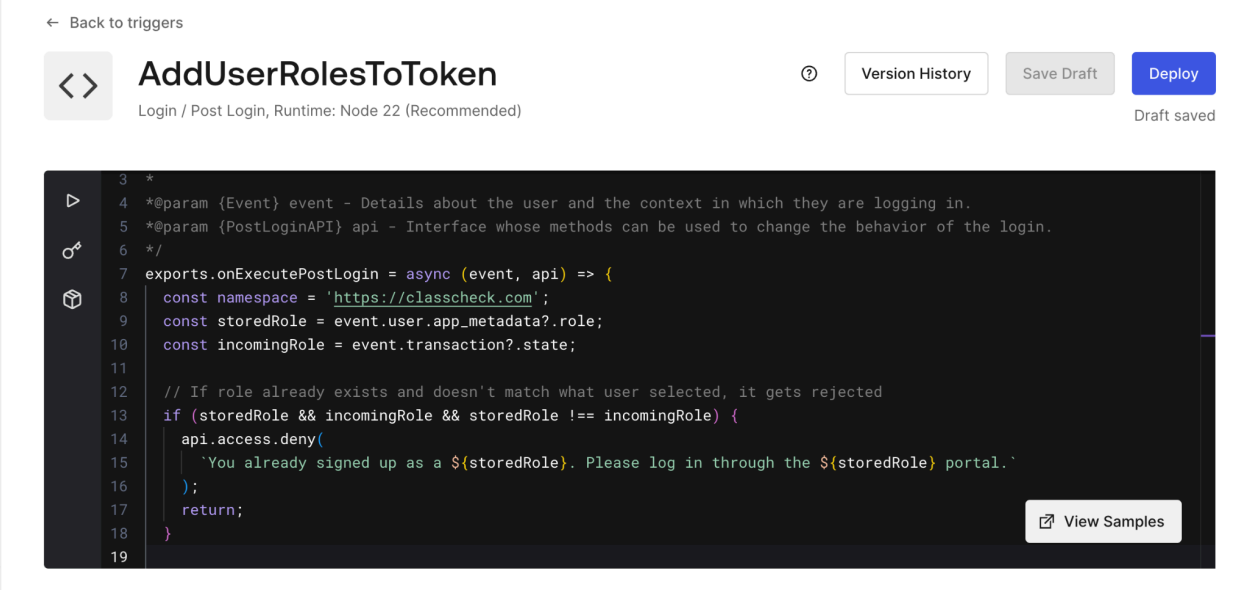
- Auth0: **GET(/callback)**

In this API, we extract the authorization code and the state (role). After the user is successfully authenticated, Auth0 redirects the user to a pre-defined callback URL which is callback.html. This will handle the response from Auth0 which includes an authorization code. So the callback script uses the @auth0/auth0-spa-js library to exchange the code for an access token and ID token. It then decodes the user's information and role from the ID token and stores it in local storage. Based on the assigned role, the user will be redirected to either a student or professor page.

```
var express = require("express");
var router = express.Router();

router.get("/", (req, res) => {
    console.log(req.oidc.isAuthenticated());
    res.render("index", {
        title: "ClassCheck",
        isAuthenticated: req.oidc.isAuthenticated(),
    });
});

router.get('/callback', (req, res) => {
    const returnTo = req.appSession?.returnTo || '/';
    res.redirect(returnTo);
});

module.exports = router;
```

- Auth0 action: **onExecutePostLogin**

This API is implemented using Auth0 action and runs after the user logs in. It retrieves the user's role from their app_metadata and sets the role as a custom claim on both the ID token and access token. This is to make sure that the frontend can consistently ready the role and control access based on it. The custom claim uses a namespaced format like https://classcheck.com/role.



- Auth0: **GET(/studentOrProfessor.html)**

In this API, users are asked to select whether they are a student or professor before signing up. That selection is passed along via state during the Auth0 signup process. The auth0 post login action **(onExecutePostLogin)** then uses that information to save the user's role to app_metadata. The role of the user is then reused in all future sessions.

- Nodemailer: **POST (/API/absence form)**

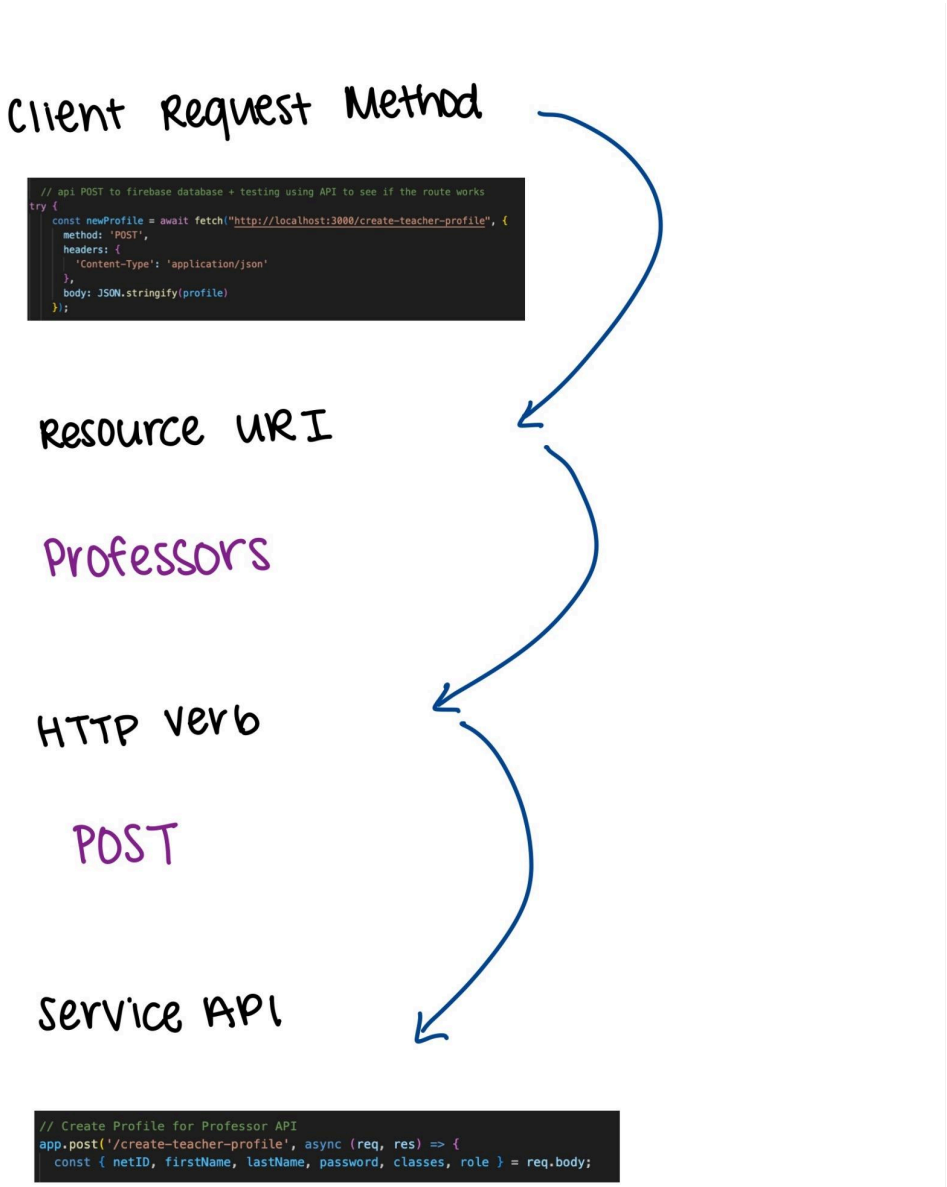
In this API, we handle the forwarding of emails using an email transport. Our specification uses gmail as it services scarletmail addresses. On the professor dashboard, you can push a notification to the absent student by entering their netID. After fetching the email information from the user entity, the server function .sendMail() forwards an absence message that contains their name, date of absence, and information about appeal instructions to the student's scarletmail address.

Server Side Routing of HTTP Requests Diagrams for Important Functions:

- **POST (create-profile)**

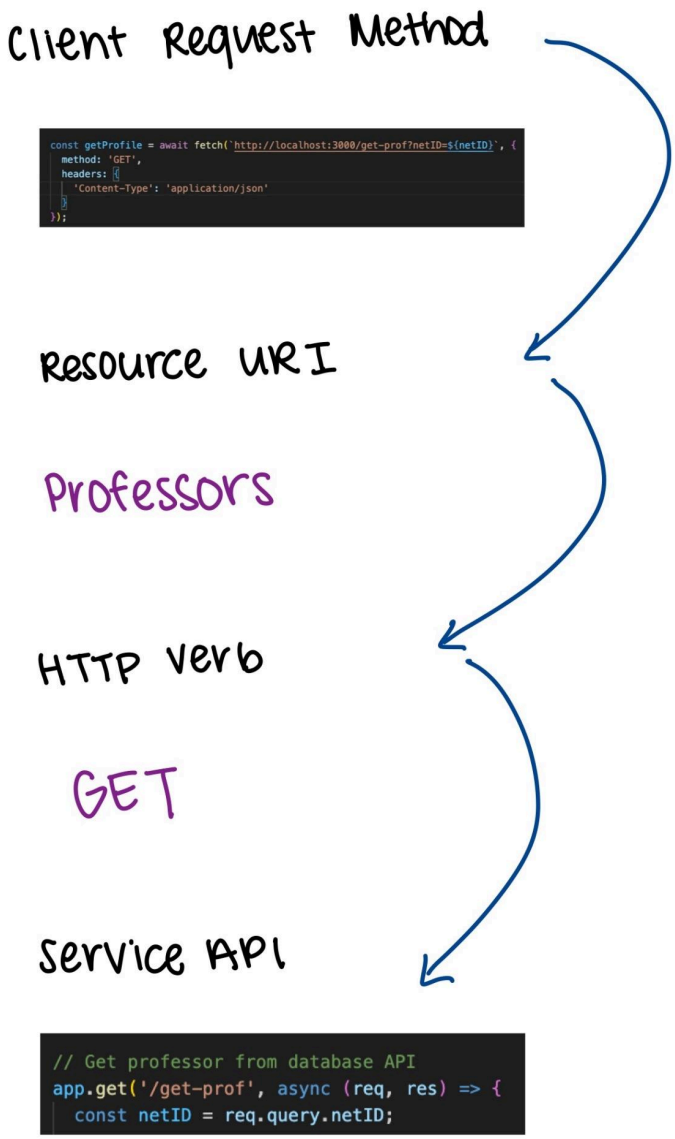
In this request diagram, we are requesting the create-profile method in profile.html where the API is taking data and posting it in the Users collection resource. It is saving the data through a POST http request. In the backend in server.js, we are creating the service that is called by the server when the request is implemented.

- **POST (create-teacher-profile)**



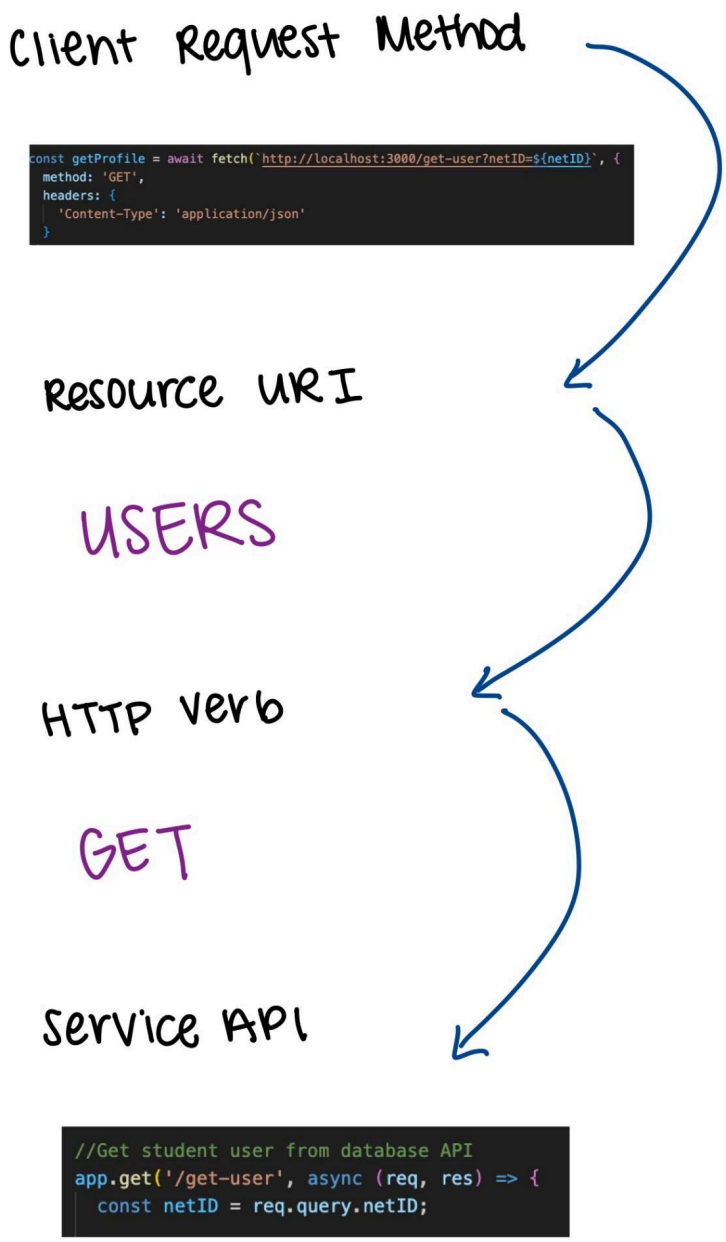
In this request diagram, we are requesting the create-teacher-profile method in profileProfessor.html where the API is taking data and posting it in the Professors collection resource. It is saving the data through a POST http request. In the backend in server.js, we are creating the service API that is called by the server when the request is implemented.

- **GET (get-prof)**



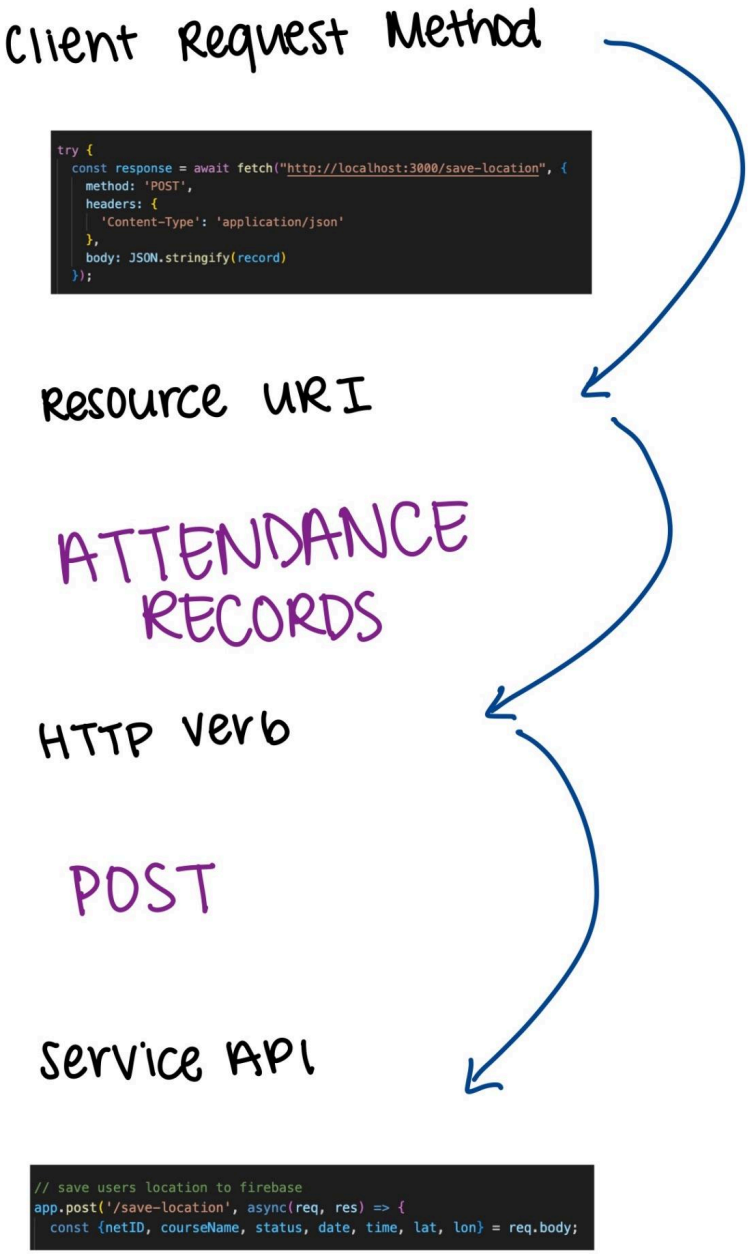
In this request diagram, we are requesting the get-prof method in professorDashboard.html where the API is taking data and posting it in the Professors collection resource. It is getting the data through a GET http request. In the backend in server.js, we are creating the service API that is called by the server through a query.

- **GET (get-user)**



In this request diagram, we are requesting the get-user method in myClasses.html where the API is taking data and posting it in the Users collection resource. It is getting the data through a GET http request. In the backend in server.js, we are creating the service API that is called by the server through a query.

- **POST (save-location)**



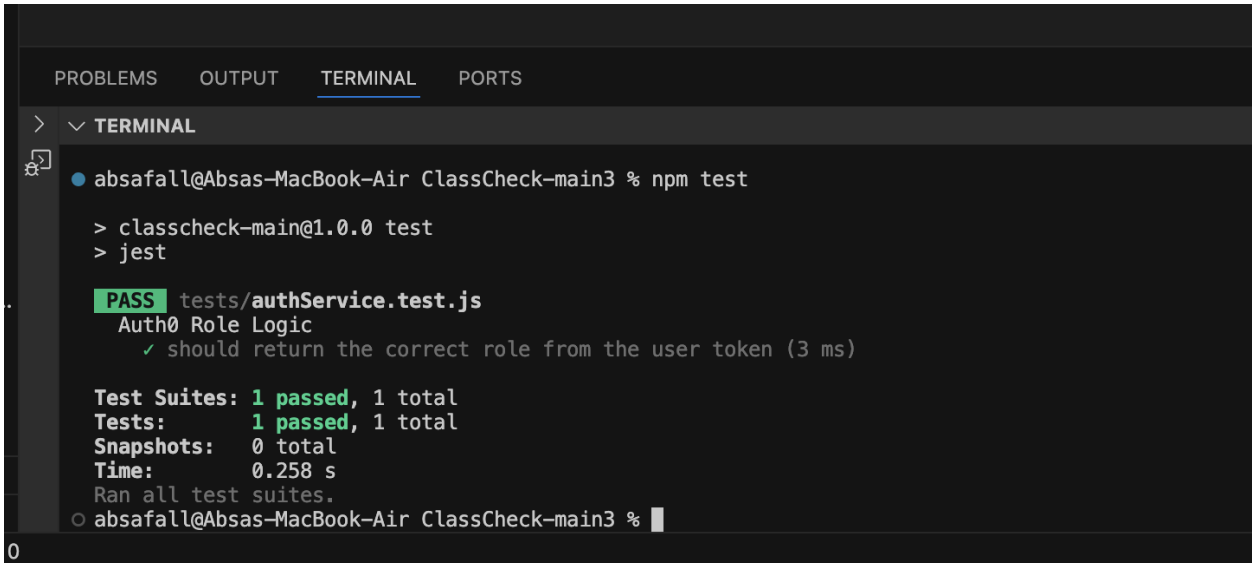
In this request diagram, we are requesting the save-location method in attendance.html where the API is taking data and posting it in the AttendanceRecords collection resource. It is

saving the data through a POST http request. In the backend in server.js, we are creating the service API that is called by the server when the request is implemented.

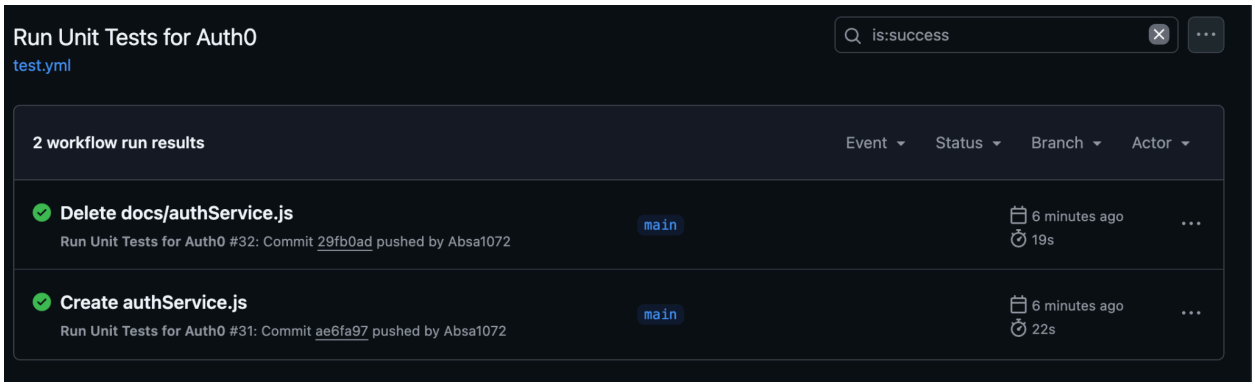
UNIT TEST WORKFLOW:

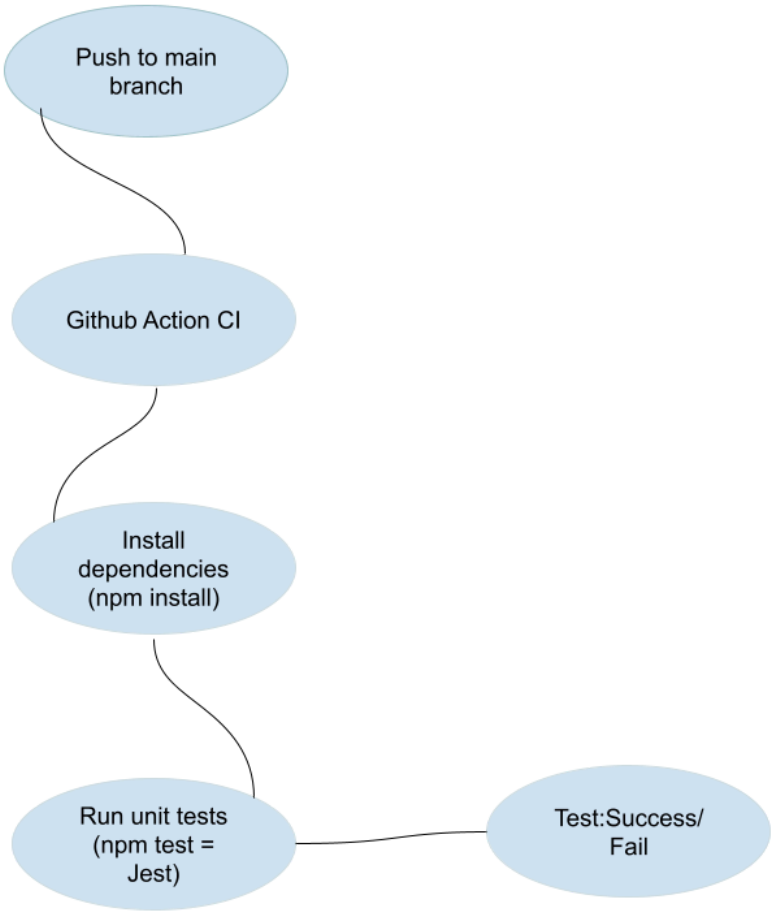
Unit tests were set up using Jest to make sure the Auth0 login process and role handling are working properly. GitHub Actions workflow was also added so that tests run automatically every time we push changes to the repository.

- Tested locally



- Through Github Actions





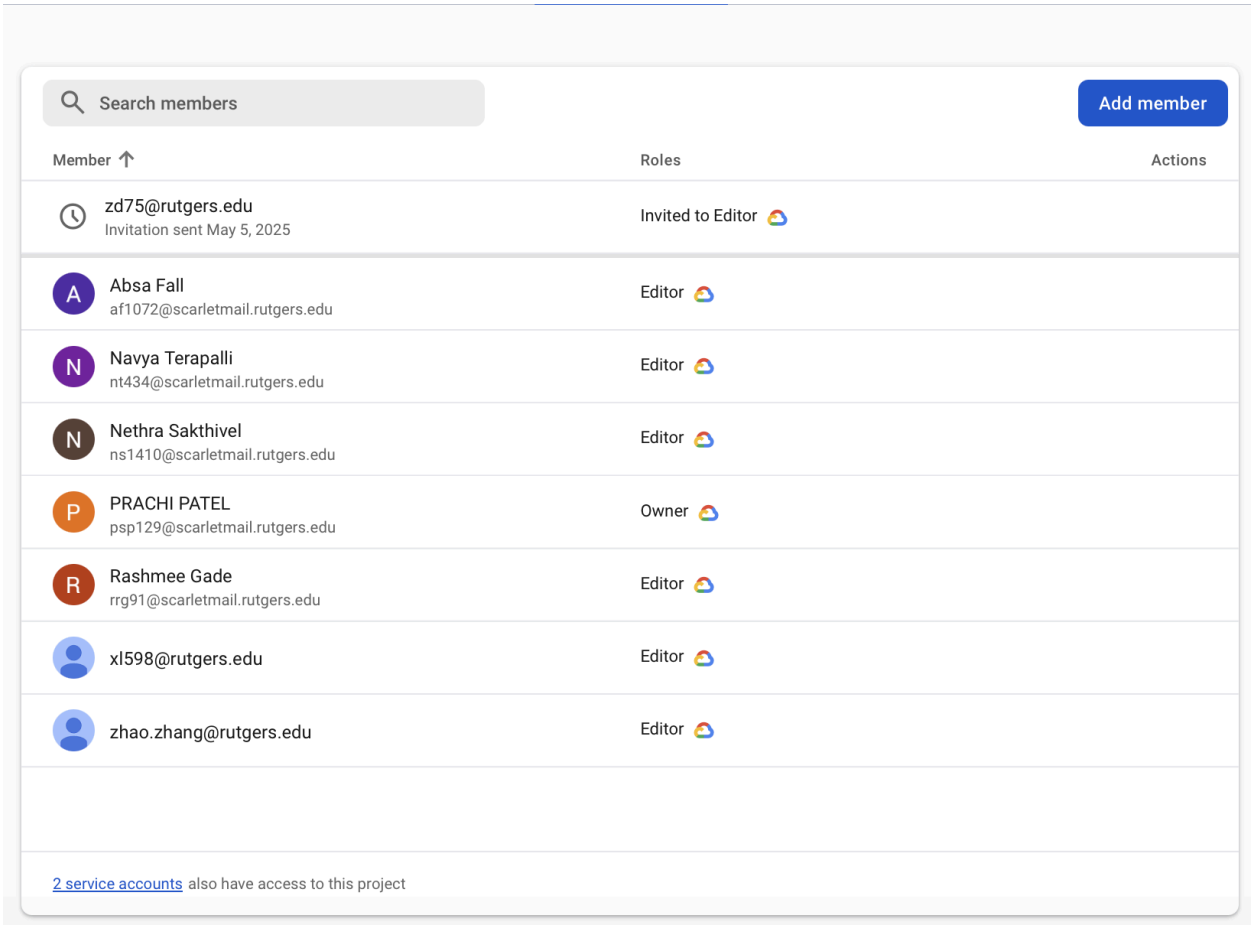
DAG Diagram of the Workflow

API CONTRIBUTION TABLE:

API	IMPLEMENTED BY:	TESTED BY:
Create-Profile	Nethra Sakthivel	All
Create-TeacherProfile	Navya Terapalli	All
Get-User	Prachi Patel	All
Get-Prof	Absa Fall	All
Get-Classes	Nethra Sakthivel	All
Save-Location	Prachi Patel	All
Get-StudentAttendance	Rashmee Gade	All
NodeMailer	Navya Terapalli	All
GET/logn	Absa Fall	All
GET/callback	Absa Fall	All

DATABASE ACCESS INSTRUCTIONS :

1. Login with your credentials to firebase through Rutgers emails using this link: (TA's and Professor have collaboration access). We added the professor and TA as editors on the firebase.



Troubleshooting Guide:

1. If the student profile displays the error “Failed to save via API”, then go into vscode and run profile.html through the live server. This occurs because of a callback issue in callback.html after using auth0 authentication.
2. If the professor profile displays the error “Failed to save via API”, then go into vscode and run profileProfessor.html through the live server. This occurs because of a callback issue in callback.html after using auth0 authentication.
3. When the professor clicks the button to send the absence notification to email, there may be an error message that says “Failed to send email.” This is because the nodemailer transport (Gmail) requires an encryption key to be able to send the notification, but does not because of a problem with the Google Cloud Platform. The Cloud Service, while once functional, no longer sends an email because the Cloud Platform does not register the encryption key anymore. This error currently has not been solved, but given more time, we would resolve the issue with the transporter.

CONTRIBUTIONS TABLE:

NAME	CONTRIBUTIONS	PERCENTAGE
Nethra Sakthivel	• Edited domain models from Report 2 for use cases 1 and 2 to bring them up to date. • Added multiplicities of records. • Added missing glossary definitions • updated entities from Report 2. • Edited requirements page to fit time constraints and edited how we implemented the requirements we had time to implement.	20%
Prachi Patel	• Create table of contents • Edited entities,	20%

	services, collections, and API validation in Report 2. ● Edit third-party APIs to include Firebase Realtime Database in Report 2. ● Report 3: API Implementation and Server-Side HTTP Requests Diagrams. Also wrote the descriptions and how all the parts connect together. ● Create Firebase Realtime Database with the implemented the service key to connect the firebase with admin access and create all the collections.	

	• Create the API routes in server.js	
Navya Terapalli	• Edited domain models from report 2 for use case 3,4,5 and descriptions for respective entities • Modified API Specifications from Report 2 • Added to the API Implementation section on Report 3 for NodeMailer and Auth0 APIs • Edited the Requirements from Report 1 to adequately reflect what was finally implemented • Modified the Use Cases to reflect the	20%

	current implementations of ClassCheck in Report 2. ● Added pictures of front-end of implemented Use Cases in Report 1. ● Created the Troubleshooting Guide.	
Rashmee Gade	● Moved the Requirements section to before the Use Cases in Report 1 for better logical flow. ● For Database Schema, changed the database we were utilizing from MySQL to Firebase. ● Edited the ER	20%

	diagram to better visualize our current project. • Added multiplicity relationships below the ER diagram. • Edited the Access Control Table. • Added images for the FrontEnd Use Cases in Report 1. • Added Use Case 2B (Attendance Exceptions).	
Absa Fall	• Report 3 - API Implementation. • Remove third party API from report 2 (https://warrant.dev/use-cases/role-based-access-control/) and implement it through	20%

	Auth0. ● Implement unit test workflow both in GitHub and locally for Auth0. ● DAG diagram workflow.	