SmartTutor

Project Team

Munam Mustafa 21I-0460
Alian Anwar 21I-0730
Muhammad Raza 21I-0511

Session 2021-2025

Supervised by

Ms. Nirmal Tariq

Co-Supervised by

Ms. Marium Hida

Department of Computer Science

National University of Computer and Emerging Sciences
Islamabad, Pakistan

December, 2024

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The need for personalized and interactive learning methods has grown significantly in the world today. In order to provide an engaging and efficient learning environment, the SmartTutor combines modern technology including real-time emotional analytics, and video communication.

The purpose of this project is to provide a web-based platform with functionality like feedback systems, social learning communities, video calling, and tutor-student matching based on preferences that enable smooth tutor-student interactions. Via this platform, the facial emotions of the students will be captured in real time in a video call session between tutor and student, which will allow tutor to adjust their teaching strategies.

Background:

The education sector has witnessed rapid evolution over the past decade, driven by technological advancements aimed at improving learning experiences. However, despite the availability of online learning platforms and digital tools, personalized and engaging learning remains a significant challenge. Traditional education systems often fail to cater to individual student needs, leading to disengagement and suboptimal learning outcomes. Additionally, the lack of real-time feedback mechanisms prevents tutors from dynamically adjusting their teaching strategies based on student engagement or emotional response during sessions.

One key issue is the difficulty students face in finding tutors whose teaching styles align with their learning preferences. Generic tutor-student matching systems often rely on superficial criteria, such as subject expertise or availability, neglecting factors like teaching methodology, student personality, and emotional compatibility. This mismatch frequently results in less effective learning sessions.

Moreover, existing systems have limited parental involvement in a child's academic journey. Many platforms lack mechanisms to provide detailed progress tracking, emotional engagement analytics, or collaborative features for tutors, students, and parents to work

together in fostering academic improvement.

Several platforms have attempted to address these issues. For instance, systems like smarttutor.co. offered online tutor-student pairing but lacked real-time emotional analytics, which are crucial for improving session interactivity and effectiveness. Similarly, research studies emphasized the importance of adaptive learning but failed to integrate technological solutions for emotional engagement tracking into live sessions.

Recognizing these gaps, SmartTutor was conceived as an innovative solution to create a comprehensive, user-friendly platform that addresses these challenges. By integrating real-time emotional analytics, personalized tutor matching, and collaborative tools for all stakeholders, SmartTutor seeks to redefine the online learning landscape and foster a more engaging, effective educational experience.

[1].

1.1 Existing Solutions

Over the years, several platforms have emerged to address the challenges in education, particularly in connecting students with tutors. While these systems provide valuable services, they often fall short of delivering comprehensive and personalized learning experiences. Below is an overview of some of the existing solutions, their features, and their limitations.

Table 1.1: Comparison of Existing Solutions

System Name	System Overview	System Limitations
Beacon Tutoring Pakistan	Beacon Tutoring Pakistan connects students with tutors based on location, subject preference, and availability. The platform focuses on creating an optimal tutor-student match to ensure academic progress.	Lacks advanced features like video calling or emotional analytics to monitor student engagement. Primarily focuses on basic tutor-student matching, limiting its scope in fostering interactive and adaptive learning.
Preply	Preply is a global platform that facilitates tutor-student connections for online learning. It offers search filters such as subject expertise, hourly rates, and tutor language.	Does not incorporate emotional analytics to gauge student engagement or provide feedback. Also lacks collaborative tools for parents and tutors to monitor progress.
SmartTutor (Proposed)	SmartTutor combines real-time emotional analytics, personalized tutor matching, video calling, and detailed progress tracking. The system enables dynamic tutor-student interaction while involving parents in the learning process.	The system doesn't have such limitations. This system provides a better environment for students to find a best tutor for themselves and align with them.

This comparison highlights the gaps in existing solutions, such as the absence of emotional analytics and real-time feedback mechanisms, and establishes the need for a more advanced platform like *SmartTutor*.

1.2 Problem Statement

In today's educational environment, finding a right tutor for student according to their preferences is still very difficult. This process also lacks personalization which leads to a mismatch between the learning style of student and the teaching style of tutor. This difficulty in finding a right tutor results in less effective tutoring sessions which results in an outcome which is not improved or is less effective.

Moreover, traditional education system lacks in **student engagement** which also fails to provide an interactive learning environment. Without knowing that how the students are reacting to the lecture delivered to them, tutors are unable to adjust their teaching strategies.

Inadequate involvement of parents is also one of the major problem which leads to an unimproved progress of a student. This disconnect between the tutor and the parent result in an uninformed progress of a child in front of his/her parents which also leads to a bad student progress.

The SmartTutor: An Enhanced Learning System seeks to overcome these challenges by giving a more individualized, engaging, and inclusive tutoring experience.

1.3 Scope

SmartTutor is a platform that provides effective matching of tutor based on the student's preferences. Tutors can list their qualifications, skills, availability, and areas of expertise by creating their profiles, whereas the students can search and select the best tutor according to their requirements. It also features Video Calling integrated with real time emotional analytics that allow the teachers to monitor the student's engagement levels during live sessions. Additionally, it creates social learning communities where students can collaborate and share knowledge. It also generates detailed reports to monitor both teacher and students performance, which helps in improving outcomes. This collaborative learning will keep parents connected with their children.

1.4 Modules

Following are the modules and features that SmartTutor consist of:

1.4.1 User Interface (UI)

It involves designing a user friendly and responsive interface for both tutor and students for their ease.

1. User **friendly design** using different frameworks i.e react.

2. A **responsive design** that works for a different devices i.e mobile, laptop.

1.4.2 User Management

This module will help admin to manage users i.e student, tutor, parent profiles.

1. **Admin dashboard** where admin can **manage accounts** of different users.
2. This will also help admin to **track and review** the activity of users.

1.4.3 Authentication and Authorization

This module will allow user to make a successful register or login to their accout that will also ensure that authorizd users have access to the website.

1. **Secure authentication** for end users.
2. Role based access control to ensure specific user functionalities.

1.4.4 Security and Privacy

This module ensures that all the data of users is safe.

1. **End to end encryption** in video calling.
2. **Data encryption** to ensure security of data.

1.4.5 Version Control and Data Backup

This module will ensure that the data generated is protected and is not lossed.

1. **Data backups**.
2. **Data redundancy**.

1.4.6 Real time Synchronization

This module will ensure real time communication between users.

1. **Real time synchronization** for video calling and emotional analytics.

1.4.7 Tutor Matching Based on Student Preferences

This module allows students to find the best tutor based on personalized preferences, ensuring an optimized tutor-student pairing.

1. **Preference based finding** on bases of subject, availability etc.
2. **Tutor profiles** including their qualifications, experience etc.

1.4.8 Video calling

This module allows a secure video calling environment where students and tutors can interact with each other.

1. **Video and audio** streaming for smooth communication.
2. **Screen sharing** option for both student and tutors.

1.4.9 Real time emotional analytics

This module will catch the emotional state of students during video calling which will help tutors to adjust their teaching methods.

1. **Emotion detection** to identify emotions like satisfaction, confusion etc.
2. **Real-time emotional** feedback for tutors to adjust their teaching strategies.

1.4.10 Social Learning Communities

This module will create social learning communities in which students can collaborate and interact

1. **Group Creation** for a learning environment which is collaborative.
2. **Shared Resources Repository** where user can send helping material.

1.4.11 Report Generation

This module will give a report to parents about their child's learning progress while tutoring sessions.

1. **Progress Reports** where reports of student progress is displayed to parent.
2. **Exportable Reports** in PDF or png format, which can be downloaded by parents .

1.4.12 Feedback

This module will facilitate parents and students to provide a feedback to a particular tutor.

1. **Feedback Form** to provide an anonymous feedback for tutor.

1.5 Work Division

For each module and respective feature, assign responsibility to a team member.

Table 1.2: Table 1

Name	**Registration**	**Responsibility / Module / Feature**
Alian Anwar	21i-0730	Module 1: User Interface (UI) - Iterations 1-4 Module 3: Authentication and Authorization - Iteration 2 Module 6: Real-Time Synchronization - Iteration 3 Module 7: Tutor Matching Based on Student Preferences - Iteration 2 Module 12: Feedback and Rating System - Iterations 3-4 Testing - Iterations 1-4
Munam	21i-0460	Module 2: User Management - Iterations 1-2 Module 4: Security and Privacy - Iteration 2 Module 8: Video Calling - Iterations 3-4 Module 10: Social Learning Communities - Iterations 3-4 Module 12: Feedback and Rating System - Iterations 3-4 Testing - Iterations 1-4
Raza	21i-0511	Module 5: Version Control and Data Backup - Iterations 1-2 Module 9: Real-Time Emotional Analytics - Iterations 2-3 Module 11: Report Generation - Iterations 2-3 Module 12: Feedback and Rating System - Iterations 3-4 Testing - Iterations 1-4

Chapter 2

Project Requirements

This section outlines the necessary requirements for the successful completion of the project. Project requirements can be divided into two main categories: functional requirements, which describe the system's core operations, and non-functional requirements, which specify performance, security, and usability standards.

2.1 Use-case

This section outlines in detail the major use cases offered on the SmartTutor platform. A use case is a description of how users and the system interact to achieve some goals. It specifies who the actors are, the flow of events, and the possible terminations that may be brought about. These use cases describe our intended scenario, which in turn ensures that the system behaves correspondingly for various scenarios, thus meeting the functional requirements defined for each user class.

The following are some use cases that describe the core functionalities of the SmartTutor platform, like user sign-up, interaction between tutor and student, video calling, report generation about progress, and real-time emotional analytics etc.

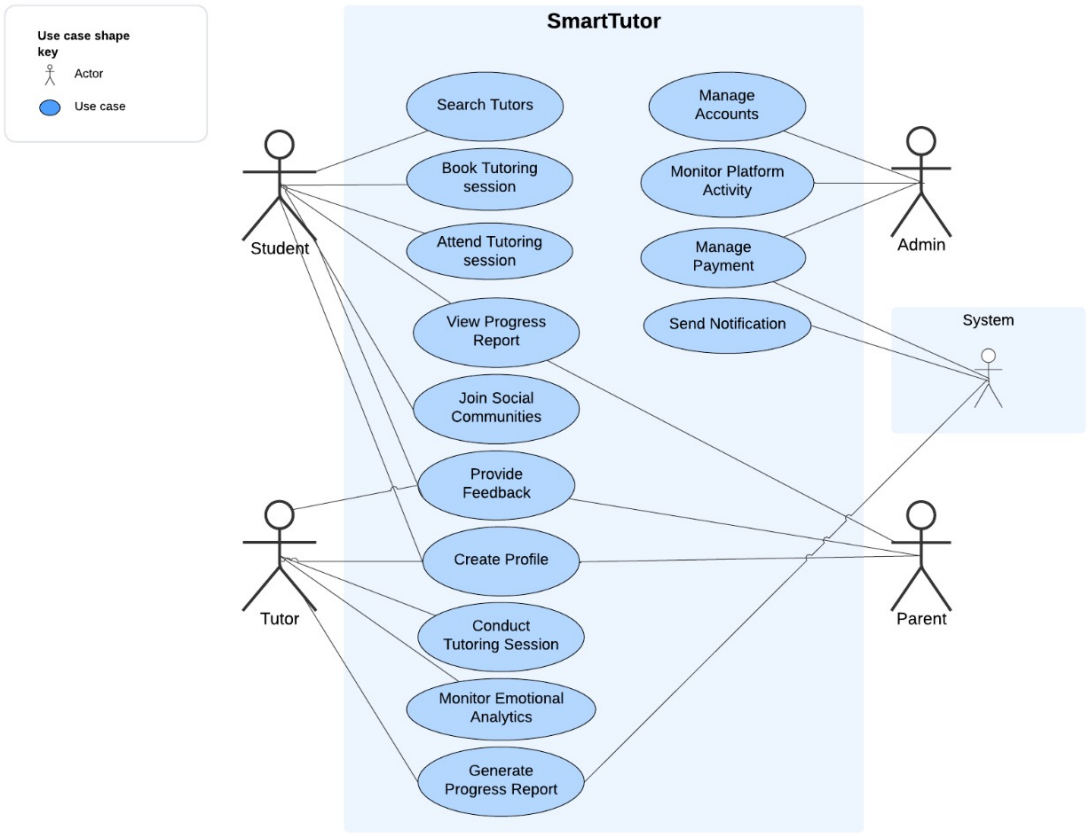

Figure 2.1: Use Case Diagram

2.1.1 Use Case 1: Register Student

- **Actor:** Student
- **Scope:** SmartTutor Web Application
- **Level:** User Goal
- **Precondition:**
  1. Students should have a valid email address or phone number.
  2. Students have opened the sign-up page.
- **Success Guarantee:**
  1. The student's account is successfully created and the student can now log in.
- **Main Success Scenario:**

1. The student goes to the registration page.
2. The student enters their personal details, email, and password.
3. The system validates the email and password.
4. The system checks for duplicate emails.
5. The system confirms the verification and completes the registration process.

- **Extension:**

1. The student enters an invalid email address or password format.
2. The system displays that the email is already registered.
3. The system fails to verify the email.

2.1.2 Use Case 2: Create Profile for Tutor

- **Actor:** Tutor
- **Scope:** SmartTutor Web Application
- **Level:** User Goal
- **Preconditions:**

1. The tutor is already registered and successfully logged into the platform.
2. The tutor has selected the edit or create profile option.

- **Success Guarantee:**

1. The tutor's profile is successfully created and it is available for students to search and select.

- **Main Success Scenario:**

1. The tutor logs into the system using valid credentials.
2. The tutor goes to the "Create Profile" option.
3. The tutor enters personal details, contact information, and edits the profile picture.
4. The tutor provides qualifications, degree, certifications, and experience in specific subjects.
5. The tutor sets availability time and price for tutoring.
6. The system verifies the details entered by the tutor.

7. After verification, the system publishes the tutor's profile.

- **Extension:**

  1. The tutor does not select any subject, and the system warns the tutor to select at least one subject.
  2. The tutor does not upload any qualifications or degree.
  3. The system fails to verify the tutor's qualifications.

2.1.3 Use Case 3 : Search for Tutor

- **Actor:** Student
- **Scope:** SmartTutor Web Application
- **Level:** User Goal
- **Preconditions:**

  1. The student must be logged in.

- **Success Guarantee:**

  1. The student gets a list of tutors matching their preferences.

- **Main Success Scenario:**

  1. The student logs into the system using valid credentials.
  2. The student goes to the "Find Tutor" section.
  3. The system displays search filters like subject, availability, rating, and price.
  4. The student selects their preferences and searches.
  5. The system displays a list of tutors matching the search.
  6. The student gets the list and checks tutor profiles.

- **Extension:**

  1. The system displays that no tutor matches the search criteria and suggests alternative profiles.

2.1.4 Use Case 4: Book a Tutoring Session

- **Actor:** Student
- **Scope:** SmartTutor Web Application
- **Level:** User Goal
- **Preconditions:**
  1. The student has contacted and selected a tutor.
  2. The tutor has approved the request and is available for the session.
- **Success Guarantee:**
  1. The session is successfully scheduled.
- **Main Success Scenario:**
  1. The student logs into the system using valid credentials.
  2. The student goes to the selected tutor's profile.
  3. The student selects a suitable time slot and submits the session request.
  4. The system sends a notification to the tutor for the session request.
  5. If the tutor accepts the request, the system confirms the booking and notifies the student.
- **Extension:**
  1. The tutor declines the session request.
  2. No time slots are available.

2.1.5 Use Case 5: Attend Tutoring Session

- **Actor:** Student, Tutor
- **Scope:** SmartTutor Web Application
- **Level:** User Goal
- **Preconditions:**
  1. The student has selected a tutor.
  2. The tutor is available for the session.

3. The session is scheduled and confirmed.

- **Success Guarantee:**
  1. Both the student and tutor have successfully joined the video call.
- **Main Success Scenario:**
  1. The student clicks the "Join Session" button in their dashboard.
  2. The tutor also clicks "Join Session" from their dashboard.
  3. The system starts a video call between the student and tutor.
  4. The video session begins, and both can communicate.
- **Extension:**
  1. Poor audio and video quality due to network issues.
  2. No one joined the session.

2.1.6 Use Case 6: Monitor Real-Time Emotional Analytics

- **Actor:** Tutor, Student
- **Scope:** SmartTutor Web Application
- **Level:** User Goal
- **Preconditions:**
  1. The student must have their video camera on.
  2. The emotional analytics feature must be enabled.
- **Success Guarantee:**
  1. The tutor receives real-time emotional feedback during the session.
  2. The tutor can change their teaching strategies according to the emotional feedback.
- **Main Success Scenario:**
  1. When the student and the tutor start the video call, the system begins tracking the student's facial expressions.
  2. The system analyzes the emotions such as confusion, focus, frustration, etc.
  3. The tutor adjusts their teaching strategy according to the feedback.

4. The system provides the emotional feedback for the complete session.
5. At the end, the tutor can review an emotional analytics report.

- **Extension:**

1. The student disables the camera.
2. The system gives incorrect emotions.

2.1.7 Use Case 7: Generate Progress Report

- **Actor:** Tutor, Student, Parent
- **Scope:** SmartTutor Web Application
- **Level:** User Goal
- **Preconditions:**

1. The tutor and student must have attended tutoring sessions.
2. The system must store student performance and emotional analytics data.

- **Success Guarantee:**

1. The tutor receives a detailed report on the student's progress.

- **Main Success Scenario:**

1. The tutor goes to the student's profile and selects the "Generate Report" option.
2. The system analyzes the performance and emotional analytics data.
3. The tutor adds other assessment data and comments.
4. The system provides the data in the form of a report.
5. The tutor reviews the report and sends it to the student and parents.

- **Extension:**

1. The system fails to get the student's data.
2. The system fails to generate the report.

2.1.8 Use Case 8: Provide Feedback

- **Actor:** Student, Parent
- **Scope:** SmartTutor Web Application
- **Level:** User Goal
- **Preconditions:**
  1. The tutor and student must have attended at least one tutoring session.
- **Success Guarantee:**
  1. The feedback is successfully provided and added to the tutor's profile.
- **Main Success Scenario:**
  1. After attending the tutoring session, the student can access the feedback option.
  2. The student rates the tutor and writes a review.
  3. The system checks the review for anything inappropriate.
  4. After the check, the student submits the feedback.
  5. The system updates the tutor's profile by adding the review.
- **Extension:**
  1. The system detects inappropriate content in the review, and the student fails to submit the review.

2.1.9 Use Case 9: Join Social Learning Communities

- **Actor:** Student
- **Scope:** SmartTutor Web Application
- **Level:** User Goal
- **Preconditions:**
  1. The student must be logged in to the platform.
  2. The social learning communities must be available on their account.
- **Success Guarantee:**

1. The user can join, post, and interact in the learning communities.

- **Main Success Scenario:**

1. The student logs into the system using valid credentials.
2. The student goes to the "Join Learning Community" section.
3. The system displays a list of available communities.
4. The student can participate in the community by posting questions and sharing resources.
5. The system checks and notifies the student of successful participation.

- **Extension:**

1. The student is unable to join the community due to age or role restrictions.
2. The student is banned or removed from the community.

2.1.10 Use Case 10: Manage Accounts

- **Actor:** Admin
- **Scope:** SmartTutor Web Application
- **Level:** Admin Function
- **Preconditions:**

1. Admin must be logged in to the platform.

- **Success Guarantee:**

1. Admin can successfully manage the selected accounts, including creating, updating, or deleting.

- **Main Success Scenario:**

1. Admin logs into the system using valid credentials.
2. The admin navigates to the "Manage Accounts" section.
3. The system displays a list of user accounts.
4. Admin selects an account for action.
5. After performing the desired actions, the admin submits the changes.
6. The system updates the changes accordingly.

- **Extension:**

1. The admin tries to delete an account that has pending payments, and the system prevents this action.

2.1.11 Use Case 11: Monitor Platform Activity

- **Actor:** Admin
- **Scope:** SmartTutor Web Application
- **Level:** Admin Function
- **Preconditions:**
  1. Admin must be logged in to the platform.
- **Success Guarantee:**
  1. Admin can successfully monitor the platform activity.
- **Main Success Scenario:**
  1. Admin logs into the system using valid credentials.
  2. The admin goes to the "Monitor Platform Activity" section.
  3. The system displays a list of activities to monitor, such as logins and errors.
  4. Admin selects a specific activity to monitor.

2.1.12 Use Case 12: Manage Payments

- **Actor:** Admin, System
- **Scope:** SmartTutor Web Application
- **Level:** Admin Function
- **Preconditions:**
  1. Admin must be logged in to the platform.
- **Success Guarantee:**
  1. Payments are successfully processed and recorded.
- **Main Success Scenario:**
  1. Admin logs into the system using valid credentials.

2. Admin goes to the “Manage Payments” section.
3. Admin views the pending payments for tutors and billing information for students.
4. The system records the payment actions.
5. Admin reviews the payment history.

- **Extension:**

1. The system experiences any issue during payment processing.

2.1.13 Use Case 13: Send Notification

- **Actor:** System
- **Scope:** SmartTutor Web Application
- **Level:** System Function
- **Preconditions:**

1. Any notification trigger event such as account updates, session reminders, etc., occurs.

- **Success Guarantee:**

1. Notifications are sent by the system to the users based on the event.

- **Main Success Scenario:**

1. Any predefined event occurs.
2. The system generates a notification.
3. The system identifies the users based on the notification type.
4. The system sends the notification to the users.
5. Users receive the notification.

- **Extension:**

1. The system fails to deliver the notification.

2.2 Functional Requirements

In this section, we establish the primary functional requirements for the SmartTutor system. These requirements are focused on the very basics that the system should represent to assist with students, tutors, and parents. Every single one outlines a separate feature that the platform needs in order to provide personalized tutoring experiences, real-time emotional analytics, secure communication or comprehensive progress tracking.

2.2.1 User Interface (UI)

Following are the requirements for module 1:

1. **FR1:** An **easy to use API** system which the tutor and student can quickly be achieved through, React for optimized UI dynamic rendering.
2. **FR2:** The system will be capable of **automatic layout adjustment** based on the screen size (for mobile phones, tablets, laptops) and shall work in a responsive way to provide an optimal user experience across platforms.

2.2.2 User Management

Following are the requirements for module 2:

1. **FR1:** The System will provide an admin **Dashboard for managing User** account and User Profile like Add, Update, Delete and View students/Parent/Tutor profile.
2. **FR2:** The system should **provide the admin with the visibility** to follow what users are doing like log in history, session data, how they are using platform and that they comply to proper usage of defined rules of the platform.

2.2.3 Authentication and Authorization

Following are the requirements for module 3:

1. **FR1: Authentication shall be secured** to create and authenticate the users with encrypted credentials, in order to avoid unauthorized access of user data.
2. **FR2:** The system will have fine-grained **role-based access control** to grant a minimum functionality for different kinds of users, who are students, tutors, parents and admin.

2.2.4 Security and Privacy

Following are the requirements for module 4:

1. **FR1: End-to-end encryption** for video calls between tutors and students to secure communication.
2. **FR2:** All user **data will be encrypted**, both in transit and at rest, so a third party cannot access sensitive information like personal details or login credentials.

2.2.5 Version Control and Data Backup

Following are the requirements for module 5:

1. **FR1:** The system shall have **automatic regular data backup** so that user data (account info, session logs) is maintained and can be recovered should it ever be lost.
2. **FR2: Data redundancy** to store another copy of important data in more than one place, ensuring that critical systems can be restored in the event of a disk or system failure.

2.2.6 Real-time Synchronization

Following are the requirements for module 6:

1. **FR1:** The system has to ensure that there is **no delay in the transmission** of audio and video streams between tutors and students during video calling sessions, so it provides real-time synchronization.
2. **FR2:** The system will provide **real-time capture and transmission** of Emotional Data when video-calling so as to provide immediate feedback to the tutors.

2.2.7 Tutor Matching Based on Student Preferences

Following are the requirements for module 7:

1. **FR1:** Students will have the ability to **search for tutors** that fit with their learning style based on subjects, availability and language allowing the student a more personalised tutor matching their learning requirements.

2. **FR2:** The system will show **extensive tutor profiles** including qualifications, experience and subject matter as to allow the students so as to choose or select a proper tutor.

2.2.8 Video Calling

Following are the requirements for module 8:

1. **FR1:** There will be a provision for **secure video and audio streaming**, to ensure lag-free communication of students with the tutors during tutoring sessions.

2.2.9 Real-time Emotional Analytics

Following are the requirements for module 9:

1. **FR1:** The system shall use **emotion detection** to recognise the emotional state (satisfaction, confusion, frustration) of a student during video calls.
2. **FR2:** The system will give **real-time emotional feedback** to tutors and this data can be used by the tutor to modify their teaching strategies and engage students better in class sessions.

2.2.10 Social Learning Communities

Following are the requirements for module 10:

1. **FR1:** The system shall **enable students to interact** with each other and tutors within the community spaces through joining of groups for collaborative learning.

2.2.11 Report Generation

Following are the requirements for module 11:

1. **FR1: Progress reports** that summarize student learning outcomes, session attendance, and emotional engagement during tutoring sessions will be generated for parents by the system.
2. **FR2:** The system will additionally **enable parents to download the reports** generated with a PDF/PNG exportable format to have an easy access and share of their child's progression details.

2.2.12 Feedback

Following are the requirements for module 12:

1. **FR1:** The system would give the user a feedback form to put in the **suggestions and feedback** of the parent and student on tutors so that everybody could share their thoughts without giving own identity.

2.3 Non-Functional Requirements

This is part of our format for an outline that specifies non-functional requirements for the SmartTutor platform. These quality properties are essential to the functioning of a system and its usefulness. With a view to driving designate development and evaluation of the proposed SmartTutor platform, herein we specify the core non-functional requirements.

2.3.1 Reliability

Reliability indicates the level of consistency that you expect from SmartTutor to be able to perform the service. These are the requirements that specify reliability aspects of the system:

1. **Mean Time Between Failures (MTBF):** The system will have a Mean Time Between Failure (MTBF) of at least 1,000 hours, having low failures times is important to providing an efficient learning experience for the students and tutors.

2. **Definition of failure:** Any event that causes the platform to fail to perform a centrally critical ability (for example, video calling operation, user authentication operation or database content retrieval) is called as software failure.

3. **Consequences of Failure:** Software failed and tutoring sessions were broken, critical data is lost, end users lose trust. The system will include strong error handling and notification mechanisms to let the user know if there are potential issues as soon as possible.

4. **Protection from Failure:** It will also implement strategies like input validation, regular code reviews and automated testing to help prevent failures. All ingredients in this list must pass stress tests and be capable of handling maximum load.

5. **Error Detection Strategy:** The system will employ logging and monitoring mechanisms to enable real-time detection of errors. Alerts will be generated to admins

in case of catastrophic failure or performance degradation, hence implying quick identification and response.

6. **Error Correction Strategy:** The system shall be equipped with roll-back capability in case it fails so as to restore the previous stable state. The system shall also have a dedicated support team whose job will primarily entail addressing the issues and determining remedies to ensure swift corrective actions to get things back on the normal track.

2.3.2 Usability

Usability requirements focus on ensuring that the SmartTutor platform provides an intuitive and efficient user experience. These requirements encompass ease of learning, ease of use, error avoidance and recovery, the efficiency of interactions, and accessibility. The specified usability requirements will assist the user interface designer in creating an optimum user experience.

1. **USE-1:** The system should allow users to register and login within three interactions at most, in addition to keeping the sign-up and authentication process as straightforward as possible.

2. **USE-2:** Help on the system should be available from every page so immediate assistance and resources to use the system properly will be given.

3. **USE-3:** The system shall have a clear navigation, meaning it shall be easy for a user to find access to one of the functions, such as tutor matching and video calling, within two clicks from the dashboard.

4. **USE-4:** The platform should use error messages that are friendly to the user and also indicate how such errors may be rectified in case of input error at the time of registration, login, or tutor selection processes.

5. **USE-5:** The system shall embrace accessibility standards, which should ensure that users with facing difficulties can access and employ the platform effectively.

6. **USE-6:** The platform should enable users to provide information relating to their experience with the session undertaken with each tutor to facilitate continuous improvement in usability as informed by user feedback.

7. **USE-7:** The system should enable users to update their profiles with personal information and preferences in at least three interactions to ensure the process can be as smooth and efficient as possible.

8. **USE-8:** The platform will be providing visible feedback in the case of successful actions like form submission, attempted joining a video call, and saved settings-for example, through loading icons or confirmation messages-to remind users that their actions are successfully being processed.

2.3.3 Performance

This section describes the performance requirements for most operations of the system. The requirements ensure that the SmartTutor application operates efficiently and effectively under expected usage conditions.

1. **PER-1:** The system will ensure that 95 percent of video calls set between students and tutors are connected within 3 seconds with an Internet connection of at least 20 Mbps or higher.
2. **PER-2:** The platform will process and return the results of tutor searches based on student preferences within 2 seconds for 90 percent of queries.
3. **PER-3:** The system will be able to support as many as 500 simultaneous users without degrading performance. As such, all users can still enjoy smooth interactions even during peak usage times.
4. **PER-4:** The site shall generate and display a progress report to parents within 5 seconds of the request being placed. Access to critical information will thus be prompt.
5. **PER-5:** The system shall provide the user with the facility of upload and sharing of resources with maximum upload time being 5 seconds for file sizes of up to 10 MB over a standard broadband connection.
6. **PER-6:** The system shall be able to provide feedback on real-time emotional analytics regarding changes in the emotional state of the student to the tutors within 5 seconds during video calls to adjust teaching strategies well in time.
7. **PER-7:** The platform shall have the load time of the user interface for users within 2 seconds of getting to the homepage or the dashboard at 90 percent.

2.3.4 Security

In this section, security requirements that protect data and the SmartTutor platform were specified. These security requirements aim to prevent unauthorized access, support data breach protection, and enforce data confidentiality, integrity, and availability for the user.

1. **SEC-1:** The system should be capable of withstanding, without any breach, at least 99.9 percent attempted accesses to it in an unauthorized manner.

2. **SEC-2:** The platform shall ensure that sensitive information related to its users remains protected- both personal and academic information- such that during any possible breach, no more than 0.01 percent of the information is leaked.

3. **SEC-3:** The platform shall detect and log all failed login attempts and suspicious activity; such can be reviewed and analyzed within 1 hour of the event so as not to expose it for any longer period.

4. **SEC-4**: The system will protect 100 percent of all user data, including communication between students, tutors, and parents through encryption protocols while in transit so that it is impossible to intercept or tamper with such data.

Chapter 3

System Overview

3.1 Architectural Design

The architecture provided represents a three-tier architecture. Here's a breakdown of each layer:

1. **Presentation Layer**
    - **Description:** This is the layer where the user's interface and his interaction are given. Here, all the functionalities involved with the display of information along with taking user input are handled.
    - **Components:**
        - **Web Interface:** This allows users to interact with the system via a GUI.
2. **Application Layer**
    - **Description:** It is also referred to as the business logic layer. This is the layer where the core application functionalities reside. It processes user inputs, executes operations based on predefined business rules, and communicates with the data layer.
    - **Components:**
        - **User Management Module:** Handles user registration, login, and profile management.
        - **Session Management Module:** Manages tutoring sessions, including scheduling and attendance.
        - **Analytics Module:** Processes data related to user performance and emotional analytics.
        - **Progress Module:** Tracks student progress and generates reports.

- **Feedback Module:** Collects and manages feedback from users.
- **Community Module:** Facilitates social interactions and community features.

3. **Data Layer**

- **Description:** This layer is responsible for data storage and management. It interacts with the database to perform operations like create, read, update, and delete (CRUD).
- **Components:**
  - **Database:** The relational database (like MySQL or PostgreSQL) where all data entities (users, sessions, feedback, etc.) are stored.

Conclusion: The architecture follows a three-tier model, where:

- The Presentation Layer is concerned with user interface and experience.
- The Application Layer focuses on business logic and processes.
- The Data Layer is responsible for data management and storage.

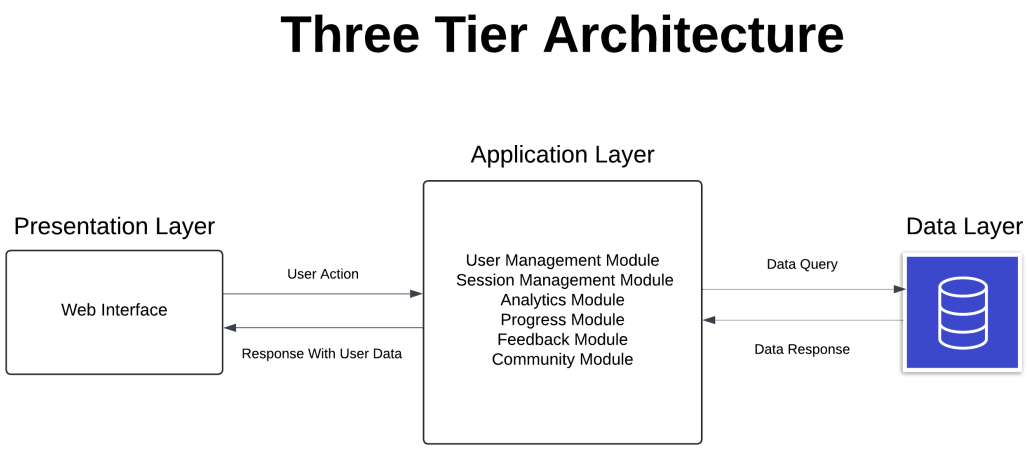

Figure 3.1: Architecture Diagram

3.2 Data Design

In the SmartTutor system, the information domain is transformed into various data structures for the purpose of efficient storage, processing, and further organization of large-sized data entities: users (students, tutors, parents, admins), sessions, feedback, emotional analytics, preferences, and payments.

3.2.1 Major Data Entities

Here are the primary data entities involved in the SmartTutor system, detailing their attributes:

3.2.1.1 Users

- **Table Name:** Users - **Attributes:** - `UserID` (Primary Key): Unique identifier for each user. - `Username`: Unique username for user login. - `Password`: Hashed password for security. - `Email`: Email address of the user. - `Role`: Specifies the user's role (Student, Tutor, Parent, Admin). - `RegistrationDate`: Date when the user registered. - `PhoneNumber`: Contact number of the user.

3.2.1.2 TutorProfiles

- **Table Name:** `TutorProfiles` - **Attributes:** - `TutorID` (Primary Key, Foreign Key referencing UserID): Unique identifier for the tutor. - `Qualifications`: Educational qualifications of the tutor. - `SubjectsSpecialization`: Subjects the tutor specializes in. - `Rating`: Average rating based on feedback. - `AvailabilitySchedule`: Tutor's available hours for sessions. - `Experience`: Number of years of tutoring experience.

3.2.1.3 StudentProfiles

- **Table Name:** `StudentProfiles` - **Attributes:** - `StudentID` (Primary Key, Foreign Key referencing UserID): Unique identifier for the student. - `Grade`: Current grade level of the student. - `Interests`: Subjects or activities the student is interested in. - `CurrentTutor`: ID of the current tutor assigned to the student. - `GuardianName`: Name of the guardian or parent. - `School`: School the student attends. - `Preferences`: Preferences related to learning styles or subjects.

3.2.1.4 ParentProfiles

- **Table Name:** `ParentProfiles` - **Attributes:** - `ParentID` (Primary Key, Foreign Key referencing UserID): Unique identifier for the parent. - `Children`: List of children associated with this parent. - `ContactInformation`: Contact details for the parent.

3.2.1.5 AdminProfiles

- **Table Name:** `AdminProfiles` - **Attributes:** - `AdminID` (Primary Key, Foreign Key referencing UserID): Unique identifier for the admin. - `Permissions`: Specific permissions granted to the admin. - `Role`: Defines the level of admin access.

3.2.1.6 Sessions

- **Table Name:** `Sessions` - **Attributes:** - `SessionID` (Primary Key): Unique identifier for each tutoring session. - `StartTime`: Start time of the session. - `EndTime`: End time of the session. - `Duration`: Duration of the tutoring session (in minutes). - `Status`: Current status of the session (Scheduled, Completed, Cancelled). - `VideoLink`: Link to the video meeting (if applicable). - `EmotionalAnalytics`: Any emotional analytics associated with the session.

3.2.1.7 Feedback

- **Table Name:** `Feedback` - **Attributes:** - `FeedbackID` (Primary Key): Unique identifier for each feedback entry. - `FeedbackText`: Text comments provided by the student regarding the session. - `Rating`: Rating given by the student (e.g., on a scale of 1-5). - `IsAnonymous`: Indicates whether the feedback is anonymous. - `DateSubmitted`: Date when the feedback was submitted.

3.2.1.8 Reports

- **Table Name:** Reports - **Attributes:** - `ReportID` (Primary Key): Unique identifier for each report. - `StudentID` (Foreign Key referencing UserID): The ID of the student for whom the report is generated. - `TutorID` (Foreign Key referencing UserID): The ID of the tutor involved in the sessions. - `EmotionalAnalyticsSummary`: Summary of emotional analytics for the student. - `PerformanceScore`: Score or data representing the student's performance. - `GeneratedDate`: Date when the report was generated.

3.2.1.9 EmotionalAnalytics

- **Table Name:** `EmotionalAnalytics` - **Attributes:** - `EmotionID` (Primary Key): Unique identifier for emotional analytics entry. - `SessionID` (Foreign Key referencing SessionID): The ID of the session being analyzed. - `StudentID` (Foreign Key referencing UserID): The ID of the student whose emotions are being analyzed. - `EmotionalType`: Type of emotion (e.g., Happy, Frustrated). - `Timestamp`: Date and time of the emotional state recording.

3.2.1.10 Payments

- **Table Name:** `Payments` - **Attributes:** - `PaymentID` (Primary Key): Unique identifier for each payment. - `StudentID` (Foreign Key referencing UserID): The ID of the student making the payment. - `TutorID` (Foreign Key referencing UserID): The ID of the tutor receiving the payment. - `Amount`: Total amount paid. - `PaymentDate`: Date of the payment transaction. - `PaymentMethod`: Method used for payment (e.g., Credit Card, PayPal). - `PaymentStatus`: Status of the payment (e.g., Completed, Pending).

3.2.1.11 Preferences

- **Table Name:** `Preferences` - **Attributes:** - `PreferenceID` (Primary Key): Unique identifier for each preference entry. - `UserID` (Foreign Key referencing UserID): The ID of the user associated with the preferences. - `PreferredTutor`: ID of the preferred tutor for the user. - `LearningStyle`: Preferred learning style (e.g., Visual, Auditory). - `SubjectPreferences`: Subjects the user prefers for learning.

3.2.2 Data Storage and Organization

The data for the SmartTutor system is organized into structured tables within a relational database management system (RDBMS) such as MySQL. The database schema is designed to support the following tables and relationships:

1. **Users Table:** Stores basic user information that is common to all users (students, tutors, parents, and admins).
2. **TutorProfiles Table:** Stores additional profile details specific to tutors, with a foreign key relationship to the Users table.
3. **StudentProfiles Table:** Stores additional profile details specific to students, with a foreign key relationship to the Users table.
4. **ParentProfiles Table:** Stores information specific to parents, with a foreign key relationship to the Users table.
5. **AdminProfiles Table:** Contains information specific to admins, with a foreign key relationship to the Users table.
6. **Sessions Table:** Logs all scheduled tutoring sessions, linking students and tutors through foreign keys.
7. **Feedback Table:** Captures feedback from students after sessions, linking back to the Sessions table through a foreign key.// 8. **Reports Table:** Maintains performance reports for users, linking back to the Users table through a foreign key.
9. **EmotionalAnalytics Table:** Stores emotional analytics data linked to each session and user.

10. **Payments Table:** Logs payment transactions associated with sessions, linking students and tutors.
11. **Preferences Table:** Stores user preferences related to learning styles and subjects.

3.2.3 Data Processing

The SmartTutor system performs several operations in data processing to ensure efficient management and manipulation of data. Key functionalities through processing are as follows:

- **CRUD Operations:** Each entity is permitted to support all or some CRUD operations, extending it to manage user accounts, session bookings, feedback, emotional analytics, preferences, and payment transactions.
- **Data Validation:** Input validation ensures that inputted data is consistent and integrity-preserving, such as ensuring valid email formats, enforcing user roles, and validating payment amounts.
- **Querying:** SQL queries are used to fetch data from the database, including available tutors, retrieving session histories, generating performance reports, and tracking payments.

3.2.4 Data Security

To ensure data security, the system employs several measures:

- **Encryption:** User passwords are stored in a hashed format using secure hashing algorithms, and payment information is also encrypted.
- **Access Control:** Role-based access control is implemented to restrict access to sensitive data based on user roles.

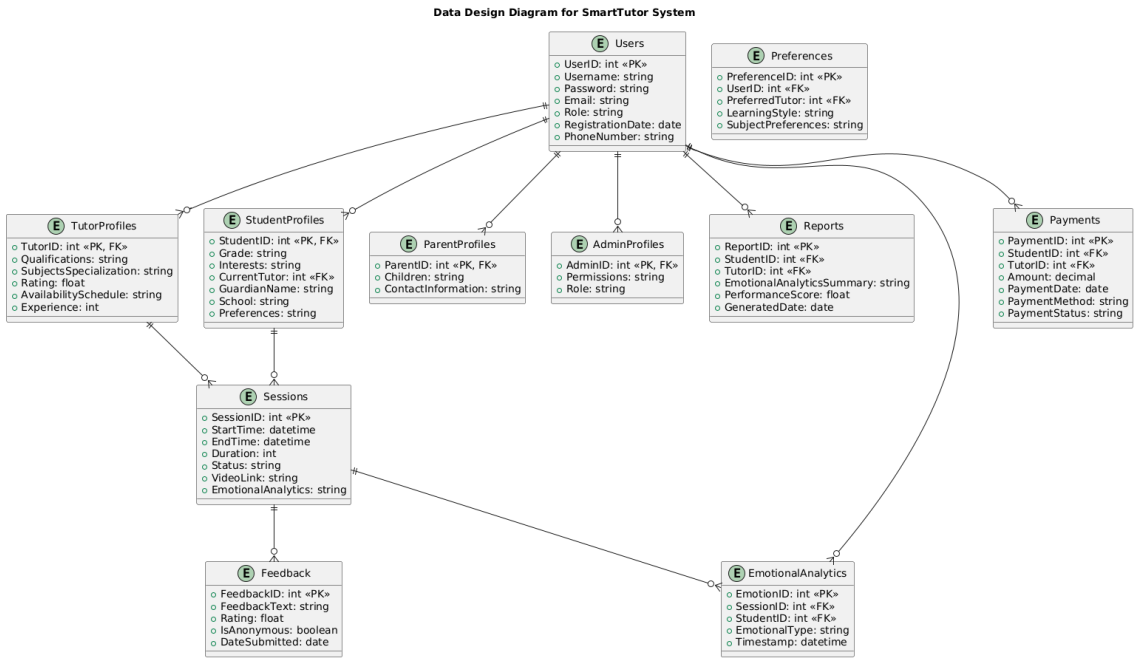

Figure 3.2: Data Design Diagram

3.3 Domain Model

Following is the domain model which provides a visual representation of the key entities within the SmartTutor platform and their relationships.

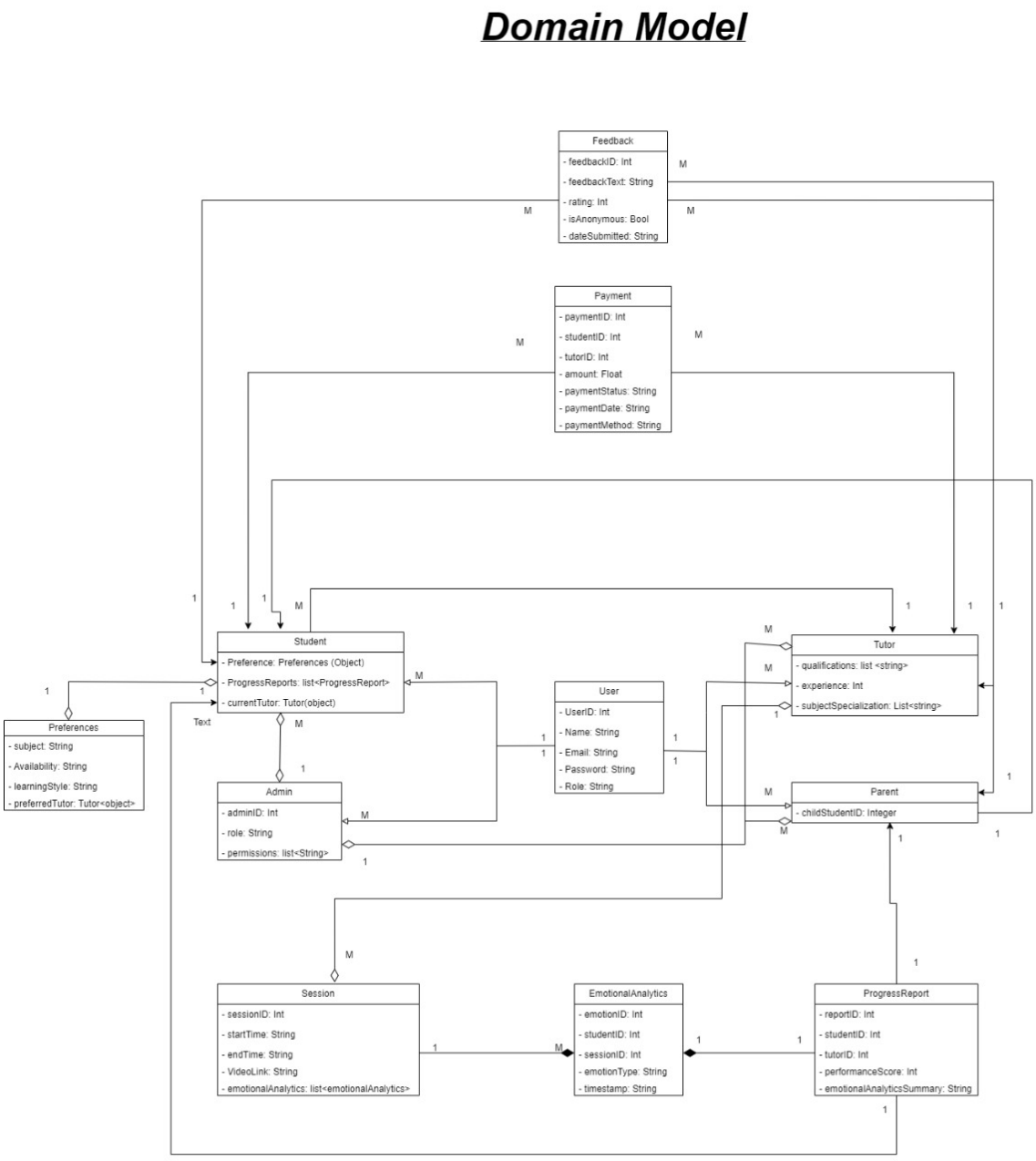

Figure 3.3: Domain Model

3.4 Design Models

This part describes the design models that can be applied to the SmartTutor system, working with an object-oriented development. Every model contains deep descriptions together with visual representations to better support the architecture and interaction of the system. The models used for describing this system are:

• Activity Diagram

• Class Diagram
• Class-level Sequence Diagram
• State Transition Diagram

These diagrams illustrate both dynamic behavior and structural organization of the system thus achieving clearness and transparency in the whole phase of design.

Design Models for Object Oriented Development Approach

3.4.1 Activity Diagram

The activity diagram is used to represent the flow of activities and actions in the Smart-Tutor system. It depicts the processes involved and their sequential order; hence, it demonstrates interaction between users and the system. In this respect, it represents how different activities are interconnected so that the workflow of the application can be understood—user actions, decision points, and parallel processes. This model is basic to the assessment of operation efficiency and the finding of areas in user interaction and system functionality that need to be optimized.

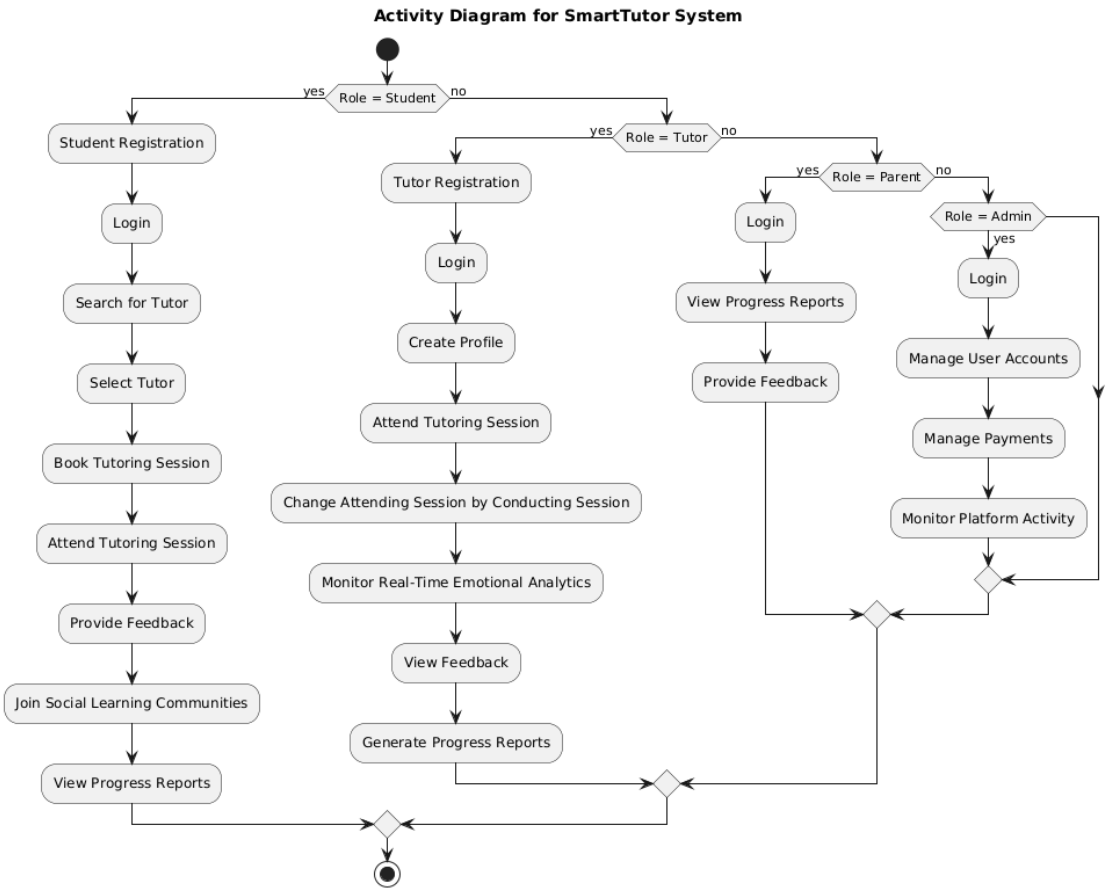

Figure 3.4: Activity Diagram

3.4.2 Class Diagram

The class diagram is a basic view of SmartTutor, mapping the structure of the system together with associated relationships between its fundamental entities. All classes in the system are indicated along with their attributes, methods, as well as visibility. Being able to depict different classes interact with the use of associations, inheritance, and dependencies, makes the class diagram very suitable to give an ideal view of object-oriented design. This is the model that would allow people to understand how an application was structured and thus talk about the application coherently for the development process with stakeholders.

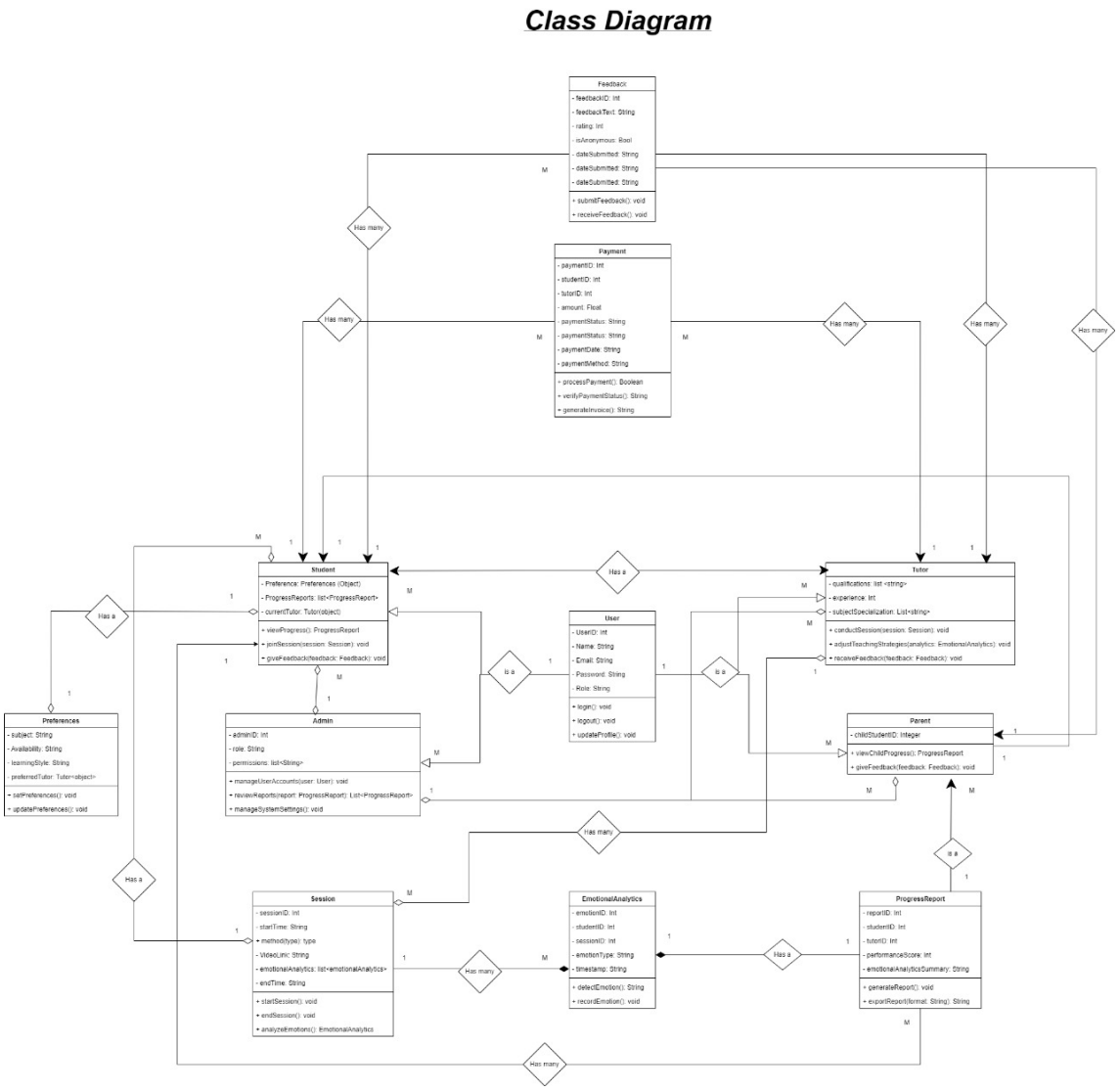

Figure 3.5: Class Diagram

3.4.3 Sequence Diagram

The sequence diagram is a very essential tool used to demonstrate the time-evolving behavior of the objects within the SmartTutor. It represents the communication or message sequence among the classes and different components in a specific use case or scenario. It reveals the control flow and timing aspect involved in achieving some particular task through the way in which objects collaborate. The model has been very helpful in understanding exactly how the application works, identifying those potential bottlenecks, and making sure all interactions are aligned with overall system requirements.

3.4.3.1 SSD 1: Student Register

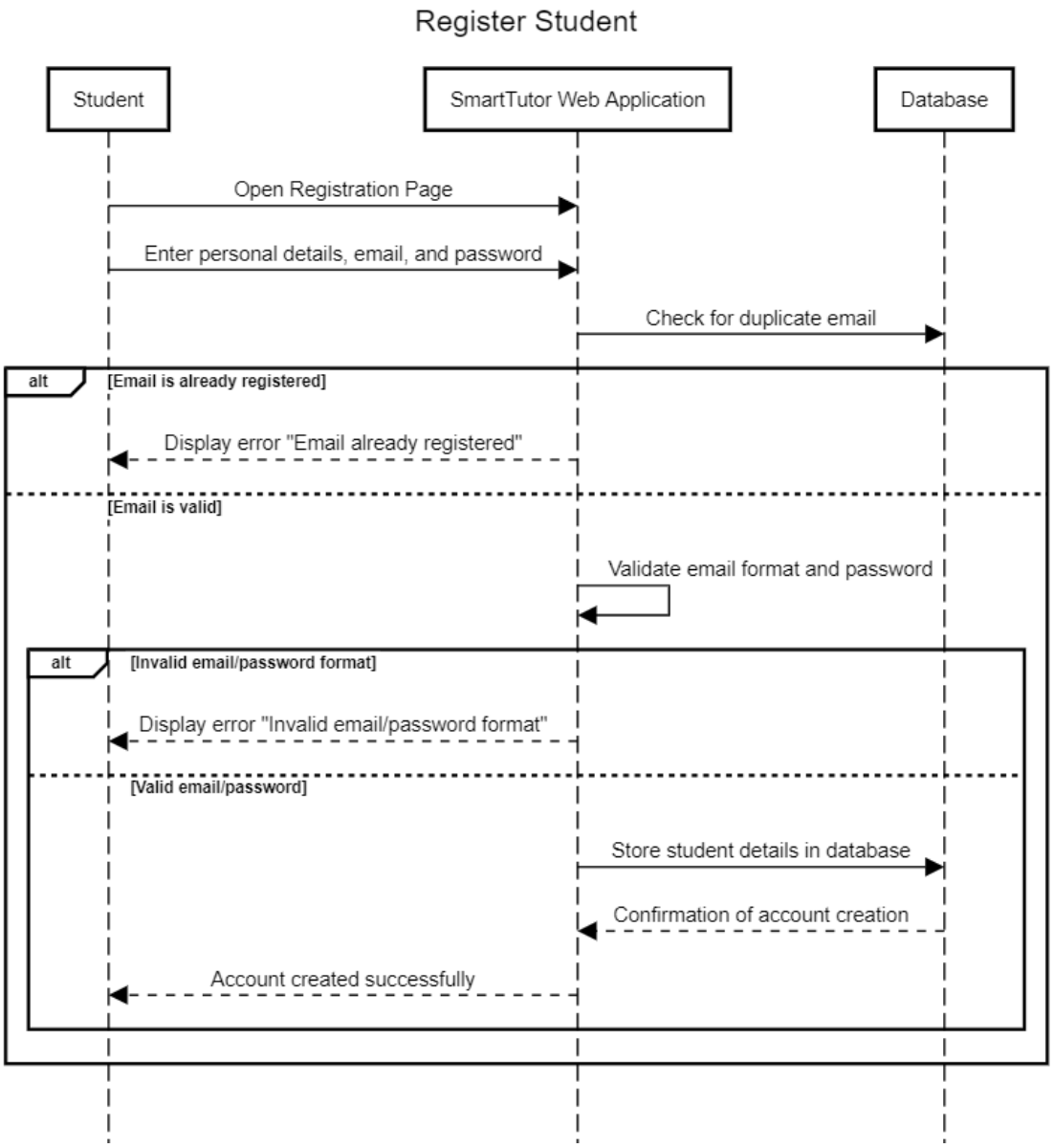

Figure 3.6: SSD (Student Register)

3.4.3.2 SSD 2: Tutor Profile Creation

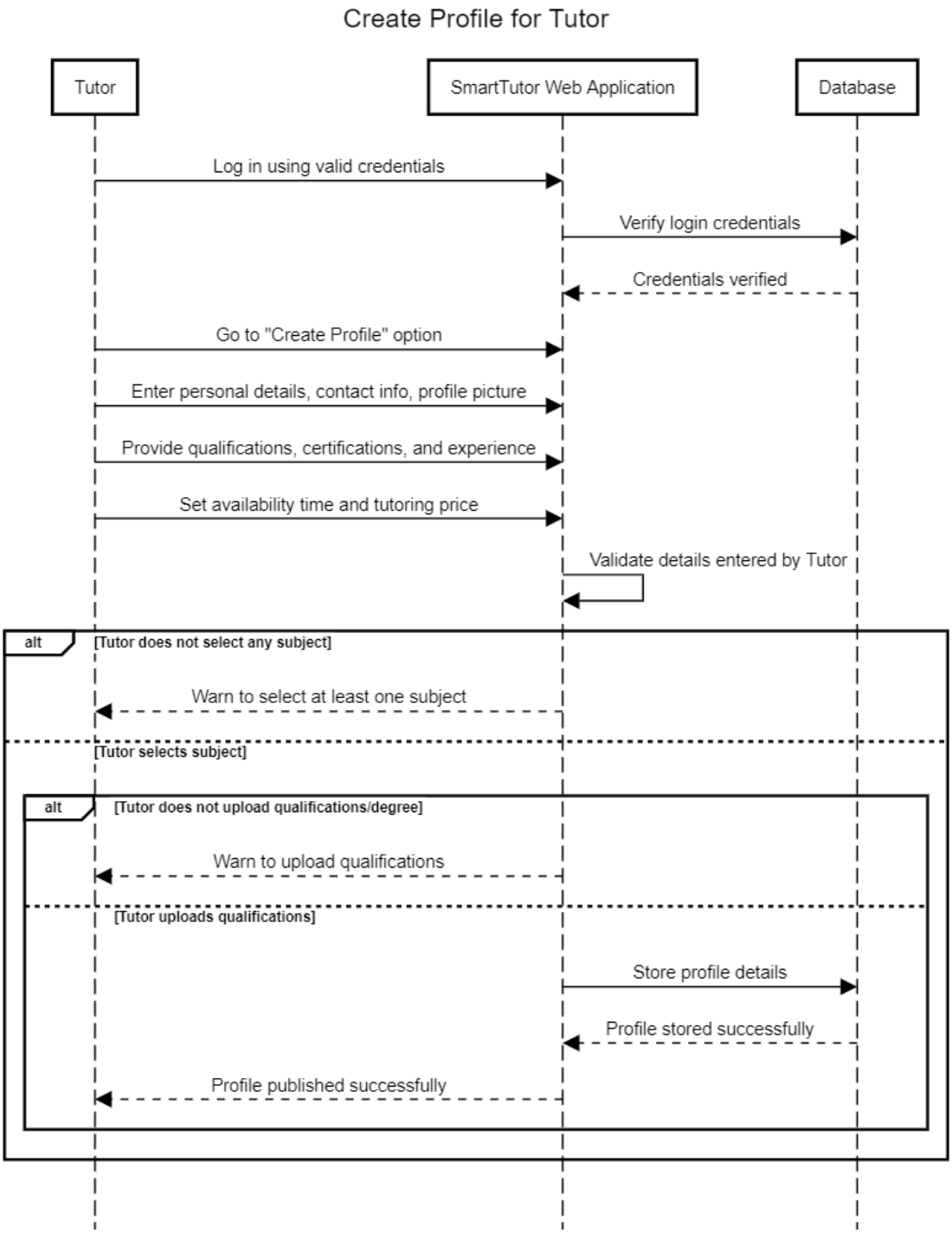

Figure 3.7: SSD (Tutor Profile Creation)

3.4.3.3 SSD 3: Search For Tutor

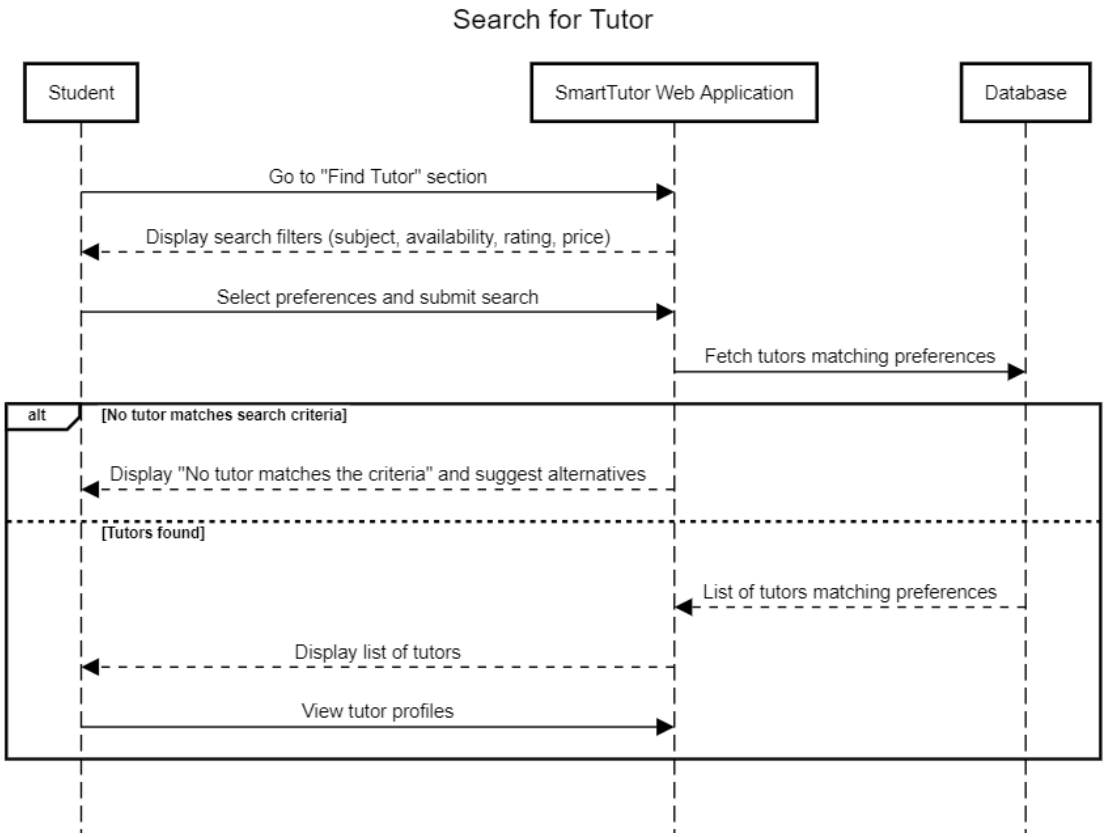

Figure 3.8: SSD (Search For Tutor)

3.4.3.4 SSD 4: Book Tutoring session

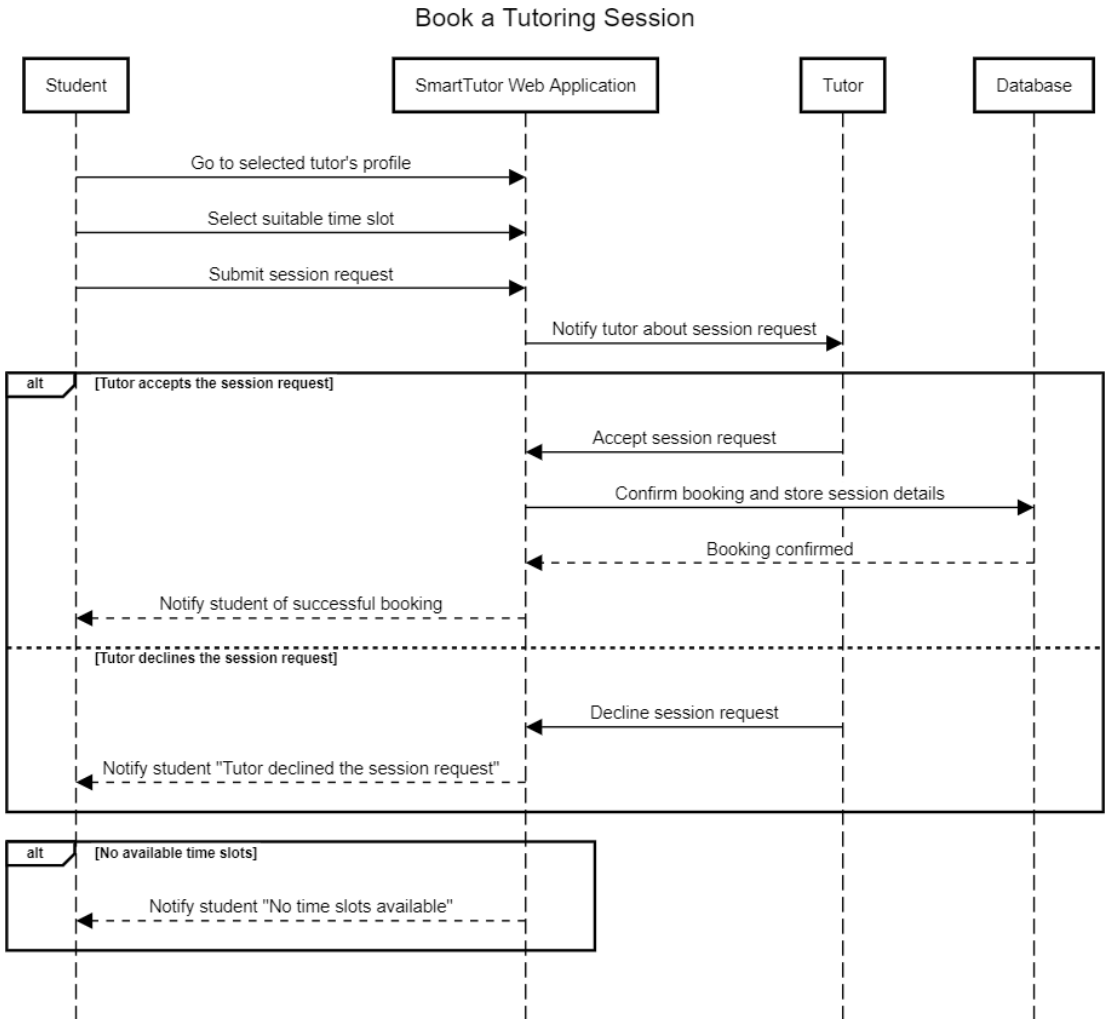

Figure 3.9: SSD (Book Tutoring session)

3.4.3.5 SSD 5: Attend Session

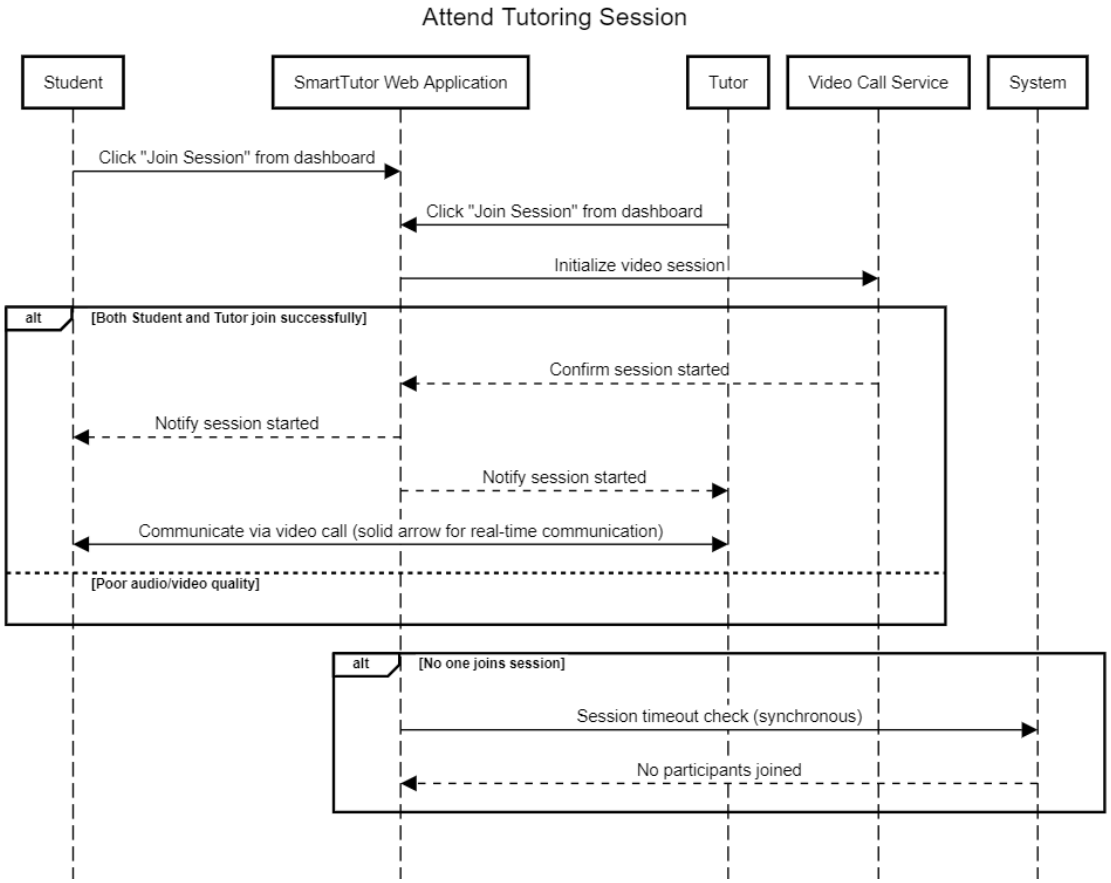

Figure 3.10: SSD (Attend Session)

3.4.3.6 SSD 6: Emotional Analytics

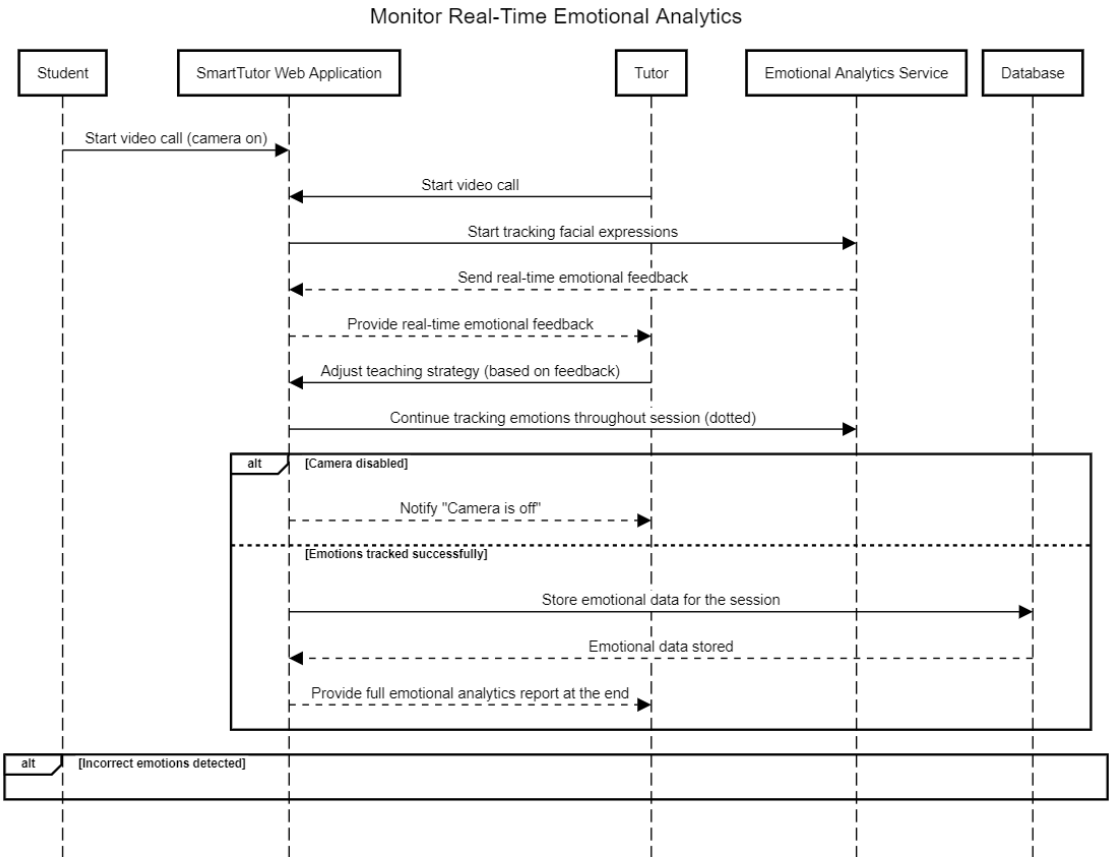

Figure 3.11: SSD (Emotional Analytics)

3.4.3.7 SSD 7: Generate Progress Report

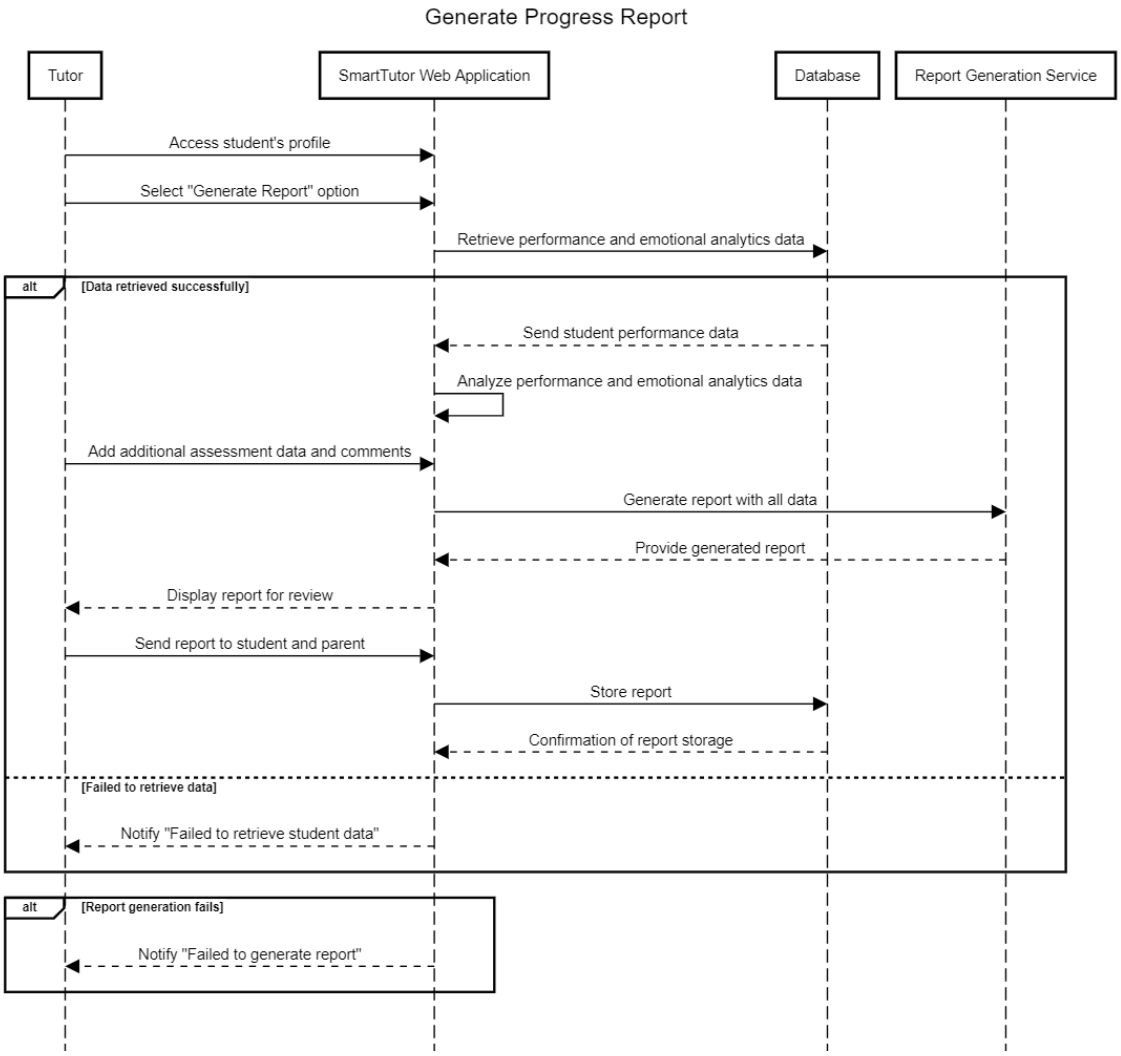

Figure 3.12: SSD (Generate Progress Report)

3.4.3.8 SSD 8: Feedback

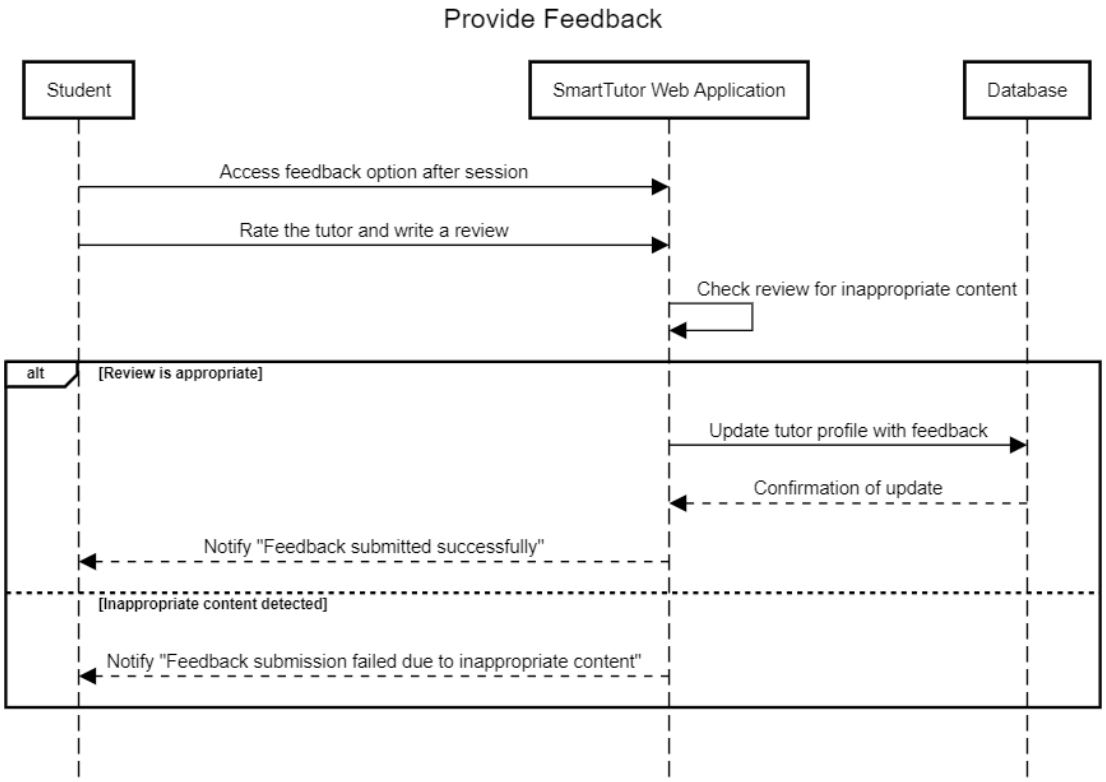

Figure 3.13: SSD (Feedback)

3.4.3.9 SSD 9: Join Communities

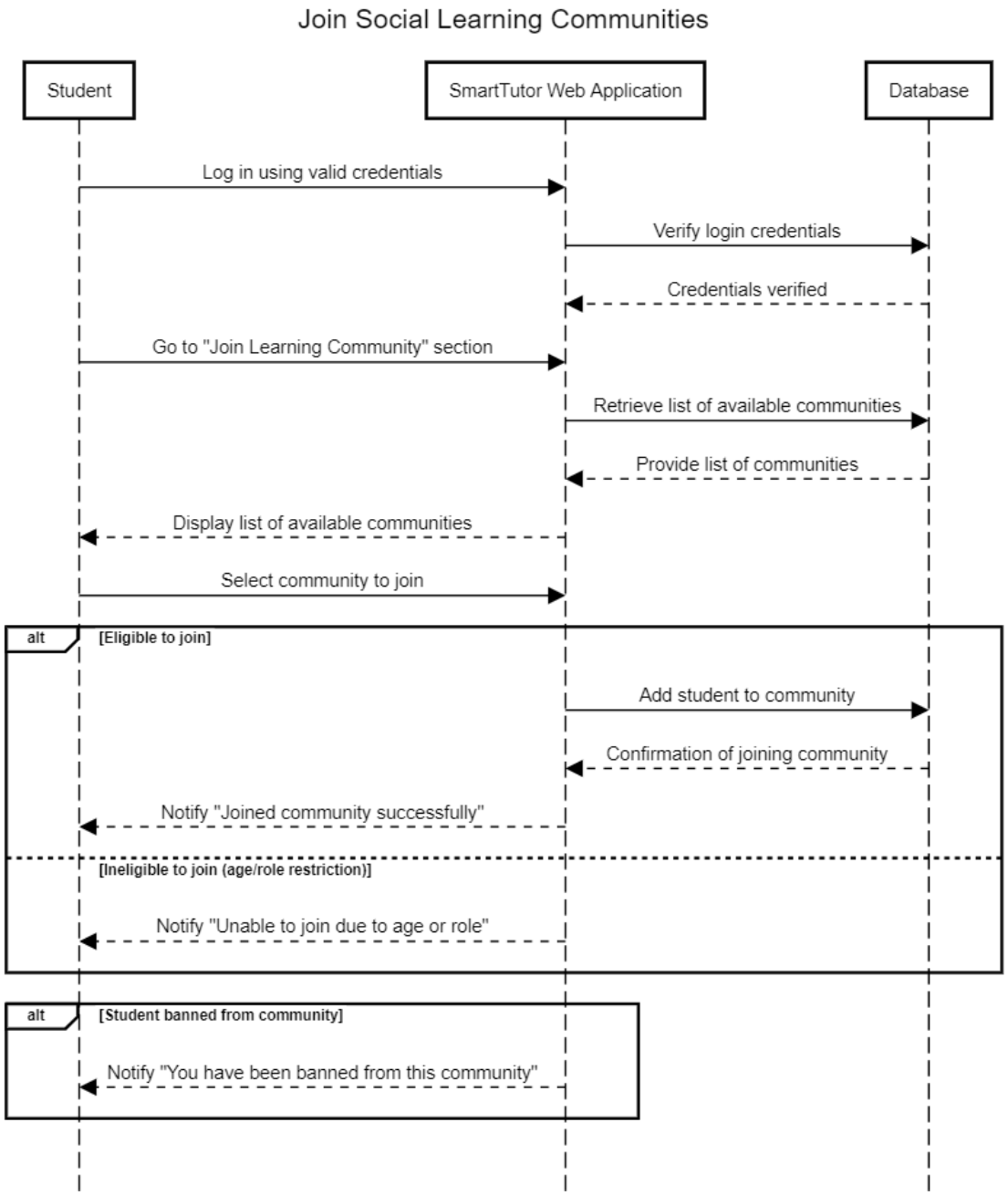

Figure 3.14: SSD (Join Communities)

3.4.3.10 SSD 10: Manage Account

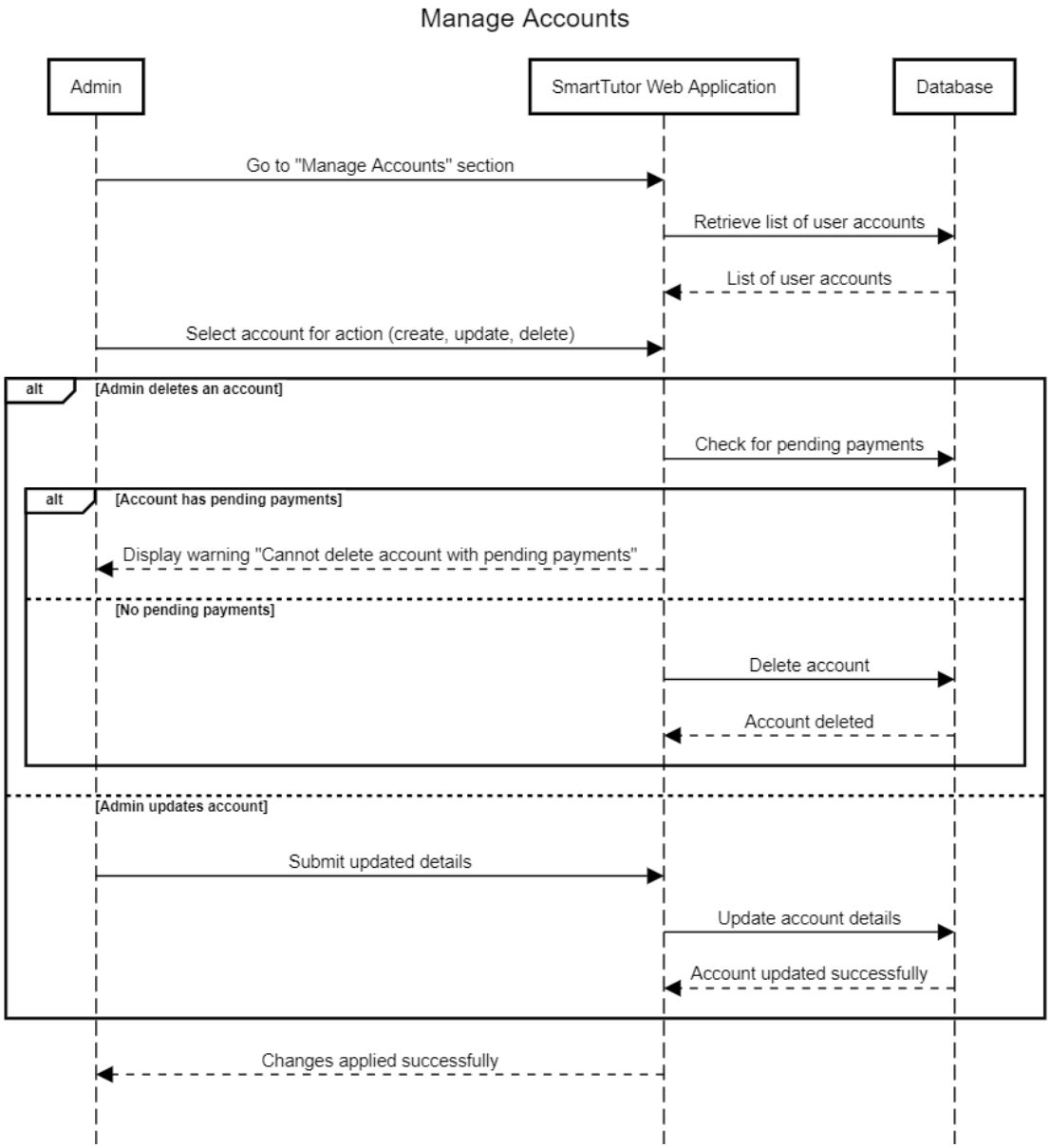

Figure 3.15: SSD (Manage Account)

3.4.3.11 SSD 11: Monitor Platform Activity

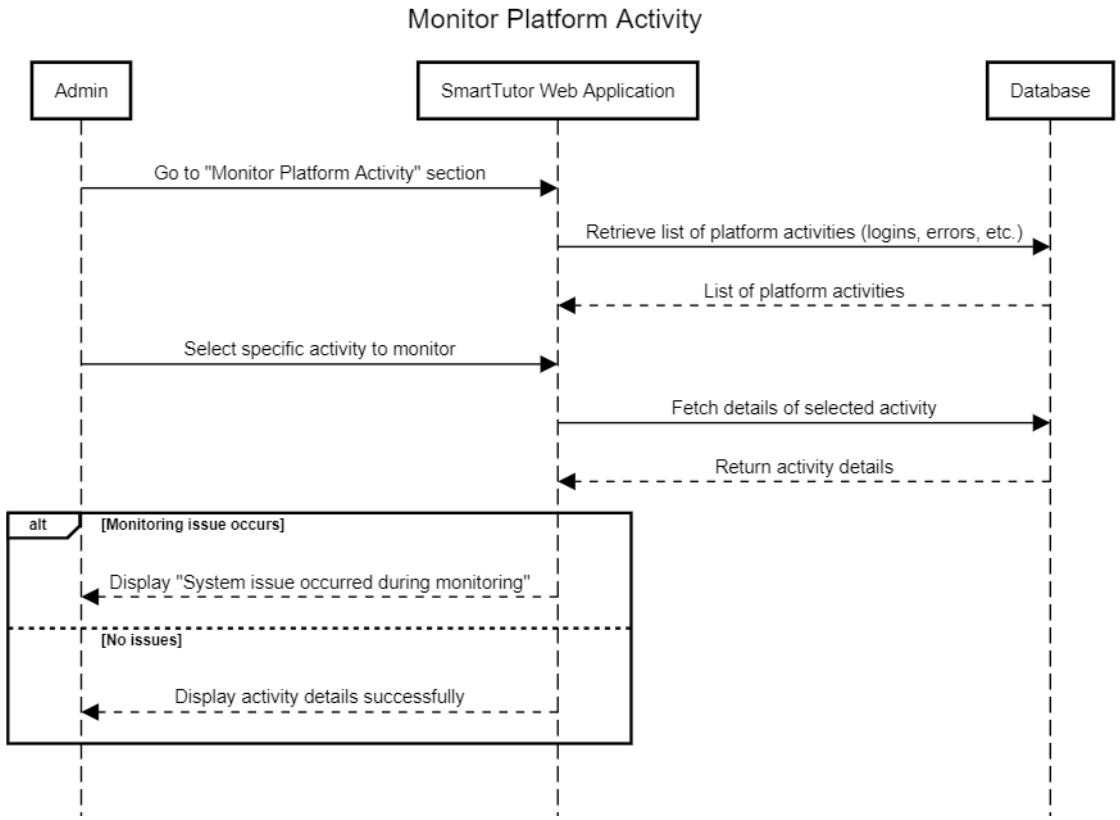

Figure 3.16: SSD (Monitor Platform Activity)

3.4.3.12 SSD 12: Manage Payment

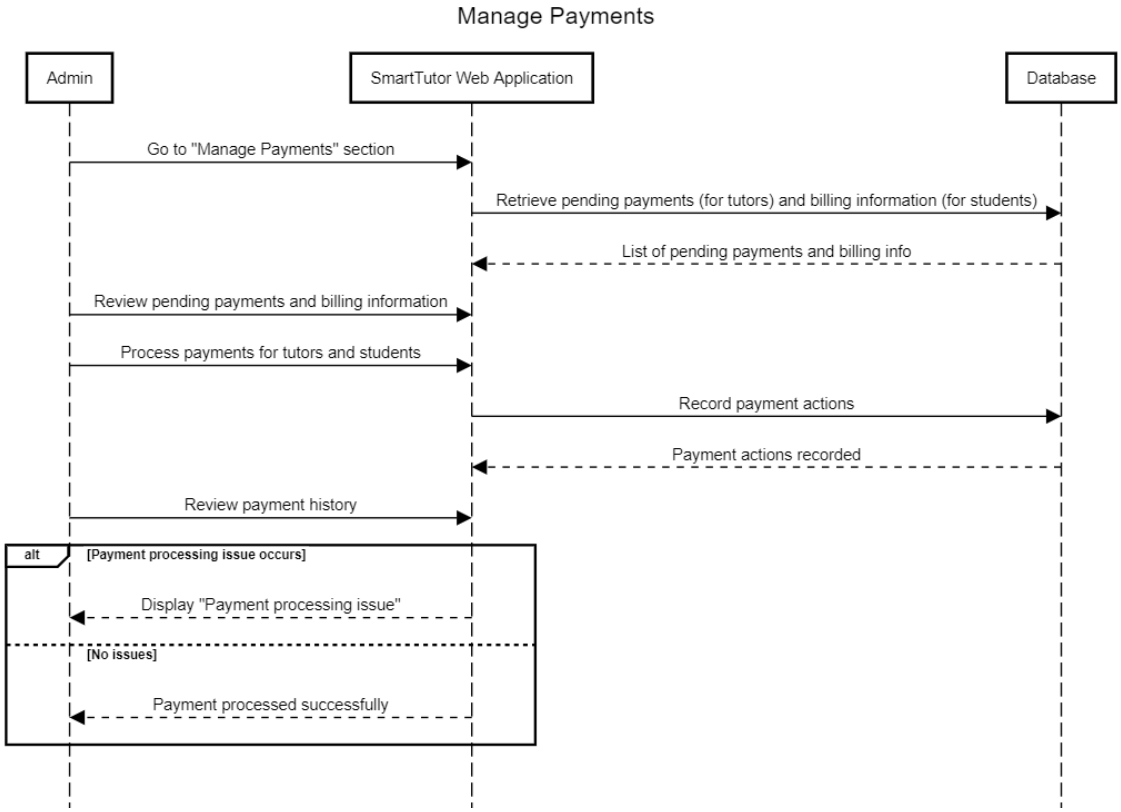

Figure 3.17: SSD (Manage Payment)

3.4.3.13 SSD 13: Send Notification

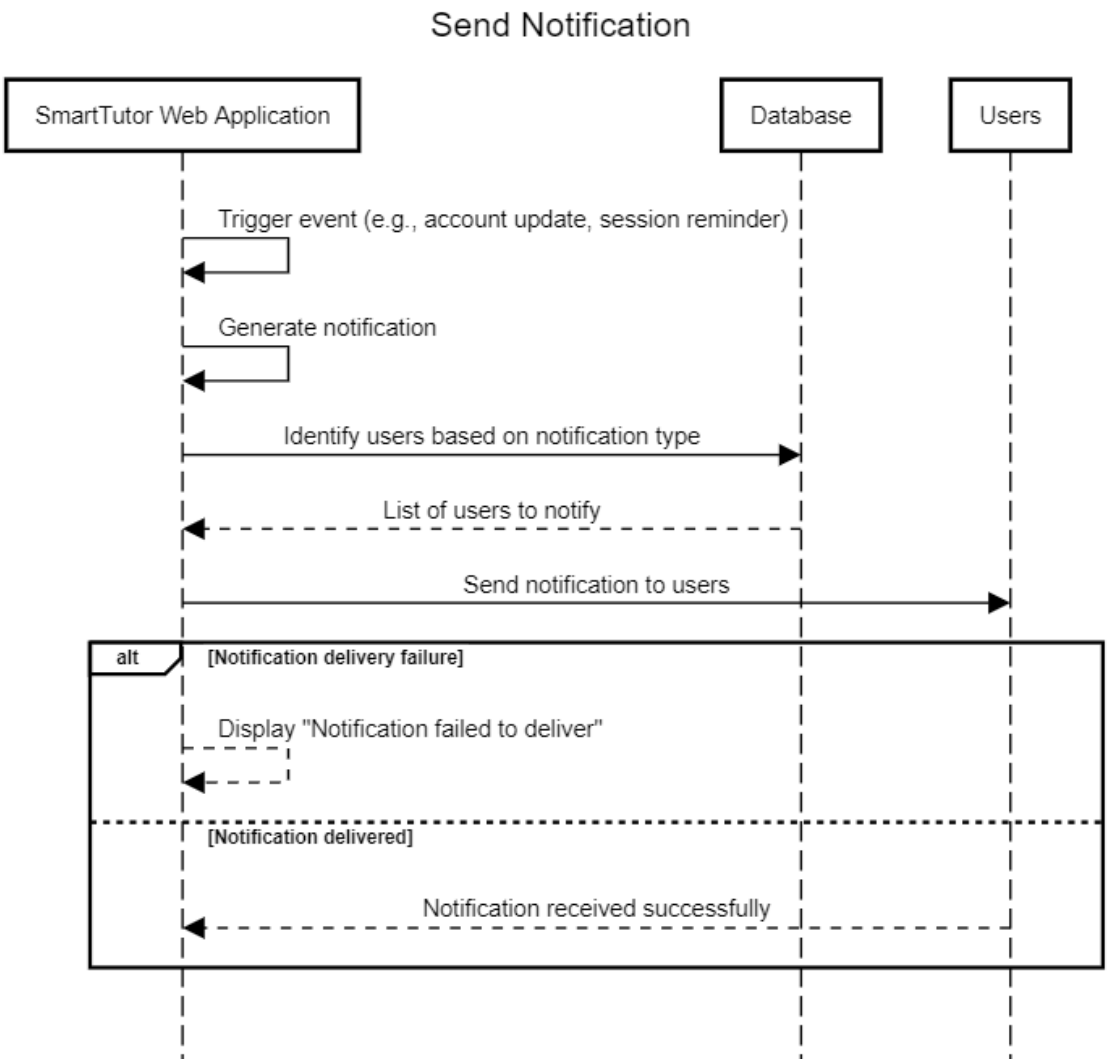

Figure 3.18: SSD (Send Notification)

3.4.3.14 SSD 14: User Login

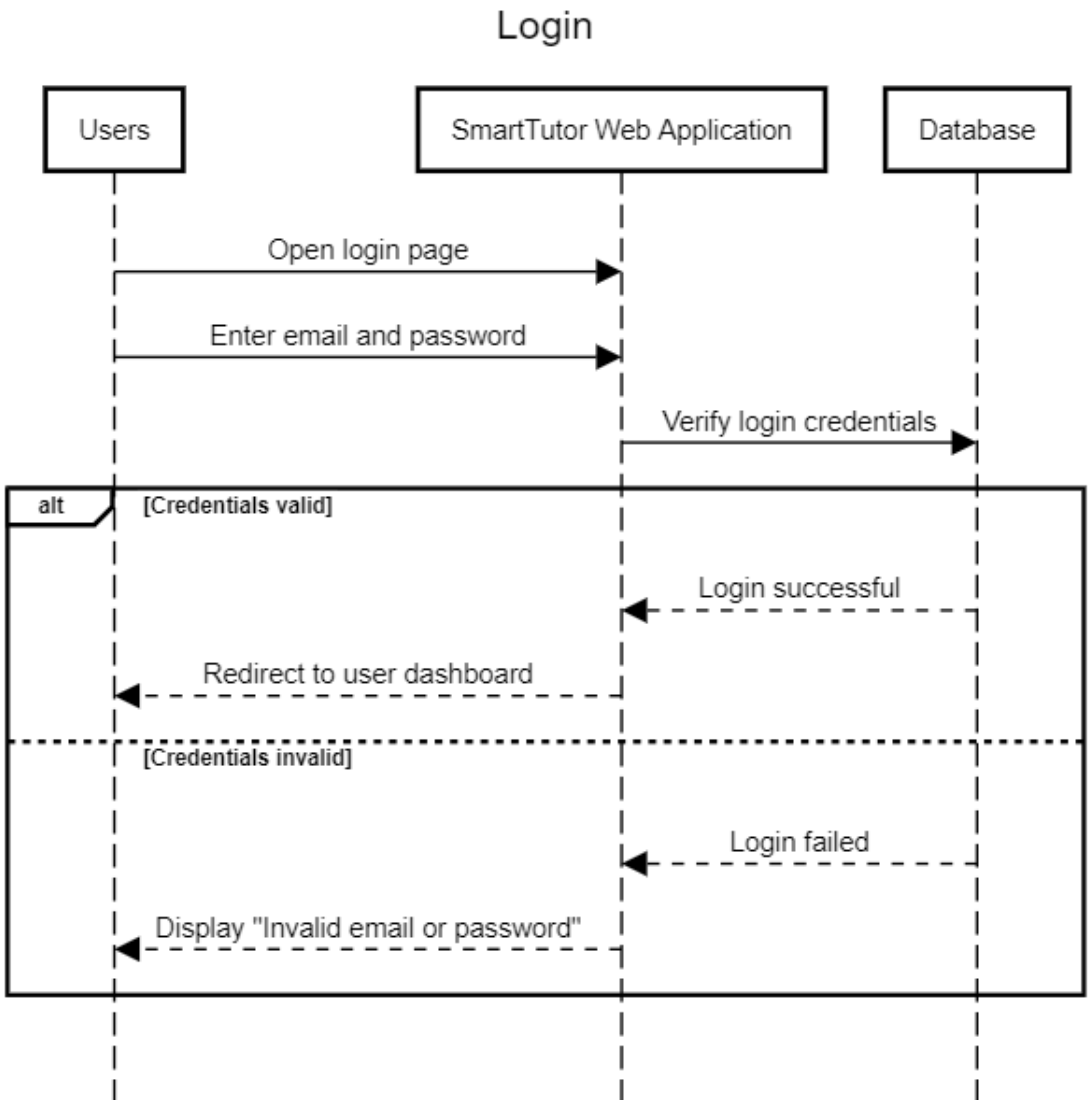

Figure 3.19: SSD (User Login)

3.4.4 State Transition Diagram

This state transition diagram represents the dynamic behavior of the SmartTutor system because, although it describes all those states to which different elements of the system may change, it also indicates transitions from one to another. It actually shows graphically how the system might respond to various events or conditions and how the states change due to user interactions and backend processes. This diagram can especially be helpful in tracking complex functionalities, such as user sessions, account management, and notification handling. In detailing states and transitions, the state transition diagram serves to assist in identifying problem areas and optimizing processes to ensure smooth flow within the application.

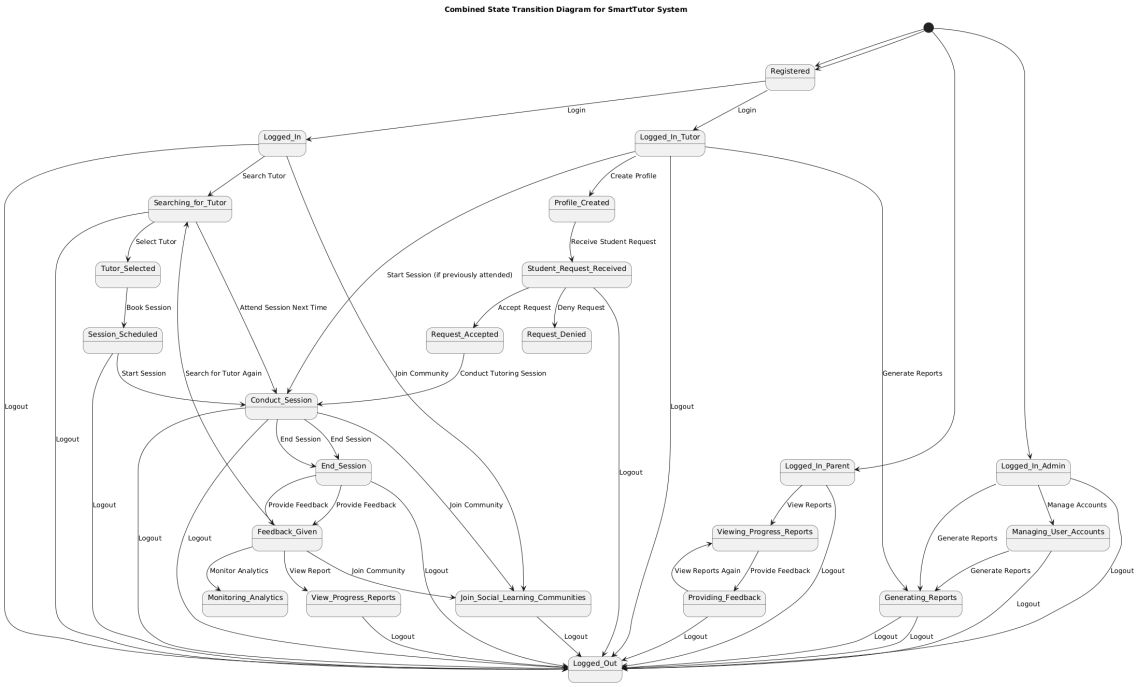

Figure 3.20: State Diagram

Chapter 4

Implementation and Testing [UPTO THE CURRENT ITERATION ONLY]

Give a general description of the functionality, context, and design of your project. Provide any background information if necessary.

4.1 Algorithm Design

Following are the algorithms used in our project alog with their psudocode.

4.1.1 Signup module

The `signup` module in the project utilizes various algorithms to manage the user registration process effectively and securely. These algorithms are implemented to handle key functionalities such as user data validation, password security, and account creation. Below is the detailed description of the algorithms used:

- **Password Hashing Algorithm:** This algorithm uses the `bcrypt` library to hash user passwords securely. By hashing passwords with a unique salt, it ensures that sensitive information is protected, even if the database is compromised.
- **Roll Number Generation Algorithm:** The project generates unique roll numbers for students, tutors, and parents. This is done by appending a specific identifier (such as `'S'`, `'T'`, or `'P'`) to the current year and a random sequence of digits. For parents, the roll number is derived from the associated student's roll number.

- **Email OTP Verification Algorithm:** To ensure only verified users can register, the system checks the status of a One-Time Password (OTP) in the database. If the OTP is invalid or unauthorized, the user is redirected to the OTP verification process.

- **User Existence Check Algorithm:** Before creating a new account, the system verifies the uniqueness of the username, email, or roll number by querying the database. If duplicates are found, the registration process is halted to maintain data integrity.

- **Email Sending Algorithm:** Using the `nodemailer` library, the system sends account details, such as credentials, to users via email. This ensures that users are informed and can access their accounts conveniently.

- **Random Password Generation Algorithm:** The system generates a random password for parent accounts during the registration process. This ensures secure initial credentials, which can be changed later by the user.

Signup algorithm psudocode:

```
IMPORT nodemailer
IMPORT bcrypt

// Module Export
EXPORT function(app, db) {

    // Helper Functions
    FUNCTION generateStudentRollNo():
        RETURN "Year + S + RandomDigits"

    FUNCTION generateTutorRollNo():
        RETURN "Year + T + RandomDigits"

    FUNCTION generateParentRollNo(studentRollNo):
        RETURN studentRollNo.replace('S', 'P')

    FUNCTION generateRandomPassword():
        RETURN RandomStringOfLength(8)

    FUNCTION checkOtpEmailExistence(email, callback):
        QUERY db FOR otp_table WHERE email = email
        RETURN callback(err, results > 0)
```

Figure 4.1: Signup Psudocode

```
    FUNCTION checkOtpStatus(email, callback):
        QUERY db FOR otp_table WHERE email = email AND status = 'authorized'
        RETURN callback(err, results > 0)

    FUNCTION checkStudentExistence(rollNo, username, email, callback):
        QUERY db FOR student_table WHERE rollNo OR username OR email EXISTS
        RETURN callback(err, results > 0)

    FUNCTION checkUserExistence(username, email, role, callback):
        TABLE = role === "tutor" ? "tutor_table" : "student_table"
        QUERY db FOR TABLE WHERE username OR email EXISTS
        RETURN callback(err, results > 0)

    FUNCTION checkTutorRollNoExistence(rollNo, callback):
        QUERY db FOR tutor_table WHERE rollNo EXISTS
        RETURN callback(err, results > 0)

    // POST Endpoint
    app.post('/api/signup', FUNCTION(req, res) {
        EXTRACT role, fullname, username, email, contact, password, confirm_password, parent_email, parent_contact, student_roll_no FROM req.body

        // Validate input
        IF anyRequiredFieldsMissing():
```

Figure 4.2: Signup Psudocode

```
            RETURN res.status(400).send("All fields are required.")

        IF password !== confirm_password:
            RETURN res.status(400).send("Passwords do not match.")

        // OTP Check
        checkOtpEmailExistence(email, FUNCTION(err, exists) {
            IF !exists:
                RETURN res.redirect("/otp_verification.html?email=" + email)

            checkOtpStatus(email, FUNCTION(err, isAuthorized) {
                IF !isAuthorized:
                    RETURN res.redirect("/otp_verification.html?email=" + email)

                // Configure Email Transport
                CONFIGURE nodemailer TRANSPORTER

                IF role === "student":
                    // Check if student exists
                    checkStudentExistence(student_roll_no, username, email, FUNCTION(err, exists) {
                        IF exists:
                            RETURN res.status(400).json({ message: "Student already exists." })
```

Figure 4.3: Signup Psudocode

```
70
71                      // Hash Password
72                      bcrypt.hash(password, 10, FUNCTION(err, hashedPassword) {
73                          INSERT student INTO student_table
74                          GENERATE parentPassword
75                          bcrypt.hash(parentPassword, 10, FUNCTION(err, hashedParentPassword) {
76                              INSERT parent INTO parent_table
77
78                              // Send Emails
79                              SEND student email
80                              SEND parent email
81                              RETURN res.json({ message: "Student and Parent registered successfully!" })
82                          })
83                      })
84                  })
85              ELSE IF role === "tutor":
86                  // Check if tutor exists
87                  checkUserExistence(username, email, role, FUNCTION(err, exists) {
88                      IF exists:
89                          RETURN res.status(400).json({ message: "Tutor already exists." })
90
91                      GENERATE tutorRollNumber
92                      checkTutorRollNoExistence(tutorRollNumber, FUNCTION(err, exists) {
```

Figure 4.4: Signup Psudocode

```
90
91                      GENERATE tutorRollNumber
92                      checkTutorRollNoExistence(tutorRollNumber, FUNCTION(err, exists) {
93                          WHILE exists:
94                              GENERATE new tutorRollNumber
95
96                          bcrypt.hash(password, 10, FUNCTION(err, hashedPassword) {
97                              INSERT tutor INTO tutor_table
98                              INSERT profile status INTO tutor_profile_status_table
99
100                             // Send Confirmation Email
101                             SEND tutor email
102                             RETURN res.json({ message: "Tutor registered successfully!" })
103                         })
104                     })
105                 })
106             ELSE:
107                 RETURN res.status(400).send("Invalid role.")
108         })
109     })
110 })
111 }
112
```

Figure 4.5: Signup Psudocode

4.1.2 OTP Verification Module

The `otp_verification.js` module is responsible for securely handling the verification of One-Time Passwords (OTPs) in the system. It validates user-submitted OTPs and ensures that only authorized users can proceed with the registration or login process. The primary functionalities of this module include:

- **Input Validation:** Ensures the required fields, such as email and OTP, are provided in the request.

- **Database Querying:** Verifies if the provided OTP and email match an entry in the `otp_table` where the OTP's status is marked as 'unauthorized'.

- **OTP Authorization:** If a valid OTP is found, the module updates its status in the database to 'authorized', signaling successful verification.

- **Record Cleanup:** Schedules the deletion of the OTP record from the database after one minute to enhance security and prevent re-use of the OTP.

- **Error Handling:** Provides appropriate error messages and logging for invalid inputs, database errors, or unauthorized OTP attempts.

This module employs algorithms for secure OTP verification, status updates, and scheduled record cleanup. The pseudocode representation of its workflow is provided below.

```
ALGORITHM: OTP Verification and Cleanup

INPUT: email, otp (from user request)

BEGIN
    1. Validate Input:
        IF email OR otp is missing THEN
            RETURN error response ("Email and OTP are required").

    2. Query Database:
        Execute SQL query to find matching OTP record:
        SELECT * FROM otp_table
        WHERE email = ? AND otp = ? AND status = 'unauthorized'.

        IF no matching record is found THEN
            RETURN error response ("Invalid OTP or email").

        ELSE
            3. Update OTP Status:
                Execute SQL query to update OTP status:
                UPDATE otp_table
                SET status = 'authorized'
                WHERE email = ? AND otp = ?.
```

Figure 4.6: OTP Verification Psudocode

```
3. Update OTP Status:
    Execute SQL query to update OTP status:
    UPDATE otp_table
    SET status = 'authorized'
    WHERE email = ? AND otp = ?.

    IF update fails THEN
        RETURN error response ("Error verifying OTP").
    ELSE
        RETURN success response ("Verification successful").

        4. Schedule OTP Record Deletion:
            AFTER 1 minute:
                Execute SQL query to delete OTP record:
                DELETE FROM otp_table
                WHERE email = ? AND otp = ?.

                IF deletion fails THEN
                    LOG error ("Error deleting OTP record").
                ELSE
                    LOG success ("OTP record deleted").
END
```

Figure 4.7: OTP Verification Psudocode

4.1.3 Tutor Profile Creation Module

The `tutor_profile_creation.js` module enables tutors to create and submit their profiles, including personal details, educational background, availability, and file uploads such as degree certificates and profile pictures. This module is essential for collecting and organizing tutor-specific data in the database. The main features of this module include:

- **File Uploads:** Handles uploading and storing degree certificates and profile pictures using the Multer middleware. Uploaded files are renamed based on the `Tutor_ID` for easier identification.
- **Input Validation:** Validates the tutor's email and roll number against existing records in the `tutor_table` to ensure the authenticity of the data.
- **Data Insertion and Overwriting:** Deletes any existing profile data associated with the `Tutor_ID` and inserts the new profile data into the `tutor_profile_data_table`.
- **Profile Status Update:** Updates the tutor's profile status in the `tutor_profile_status_table` to *pending*, signaling that the profile is awaiting review.
- **Error Handling and Logging:** Provides comprehensive error messages and logs for database operations, file uploads, and renaming errors.

The following pseudocode illustrates the logical flow of the profile creation process.

```
ALGORITHM: Tutor Profile Creation
INPUT: Profile data (personal details, educational background, availability, etc.)
       Files (degree certificate and profile picture)
BEGIN
    1. Initialize Multer middleware for handling file uploads:
        a. Set destination folder based on the file type (degree or profile picture).
        b. Use a temporary filename format (timestamp-originalname).

    2. Handle POST request to '/api/tutor_profile_creation':
        a. Extract form data from the request body.
        b. Extract uploaded files (degree and profile picture) from the request.

    3. Validate Tutor's Email and Roll Number:
        a. Query database (\texttt{tutor\_table}) to find matching \texttt{Tutor\_ID}.
        b. IF no match is found THEN
            RETURN error response ("Please enter a valid email and roll number").
        c. ELSE retrieve \texttt{Tutor\_ID}.

    4. Handle File Renaming:
        a. Rename the degree certificate to \texttt{Tutor\_ID.extension}.
        b. Rename the profile picture to \texttt{Tutor\_ID.extension}.
        c. IF renaming fails THEN
            RETURN error response (e.g., "Error saving file").
        d. Store renamed filenames as \texttt{degreeLink} and \texttt{profilePicLink}.

    5. Insert Profile Data:
        a. Delete existing profile data in \texttt{tutor\_profile\_data\_table} for the \texttt{Tutor\_ID}.
        b. Insert new profile data into \texttt{tutor\_profile\_data\_table}.
        c. Include file links (\texttt{degreeLink} and \texttt{profilePicLink}) in the data.

    6. Update Profile Status:
        a. Update the tutor's status in \texttt{tutor\_profile\_status\_table} to "pending".

    7. Respond to the Client:
        a. RETURN success response ("Profile created successfully, status set to pending").
        b. Redirect to the login page.

    8. Error Handling:
        a. Log and RETURN appropriate error responses for any failures in database operations or file handling.
END
```

Figure 4.8: Tutor Profile Creation Psudocode

Figure 4.9: Tutor Profile Creation Psudocode

4.1.4 Login Module

The `login.js` module handles the login process for various user roles (Admin, Student, Tutor, Parent) in the SmartTutor application. It validates user input, verifies credentials against the database, and handles redirection based on the user's role and profile status. Key features of this module include:

- **Roll Number Validation:** Ensures that the provided roll number follows the correct format using a regular expression. The valid format is `XXA-XXXX`, `XXS-XXXX`, `XXT-XXXX`, `XXP-XXXX` where `X` represents a number.

- **Dynamic Table Selection:** Based on the roll number, the correct table (Admin, Student, Tutor, or Parent) is selected for credential verification.

- **Password Validation:** The system compares the provided password with the stored password in the database. For most roles, `bcrypt` is used for secure password comparison.

- **Profile Status Check (for Tutors):** For tutor users, the system checks whether the profile is created. If not, the user is redirected to the profile creation page.

- **Redirection Based on Role:** After successful login, users are redirected to their respective dashboards based on their role (Admin, Tutor, Student, Parent).

- **Error Handling:** Comprehensive error messages are provided for invalid roll numbers, passwords, and issues with database queries.

The following pseudocode illustrates the logical flow of the login process.

```
BEGIN login process
    Receive user input (roll number, password)
    Validate roll number using regex
        IF roll number does not match format
            RETURN error "Invalid roll number format"

    SELECT table based on roll number:
        IF roll number contains 'A' ⇒ admin_table
        IF roll number contains 'S' ⇒ student_table
        IF roll number contains 'T' ⇒ tutor_table
        IF roll number contains 'P' ⇒ parent_table

    Execute SQL query to fetch user data from corresponding table

    IF query fails
        RETURN error "Database query error"

    IF user found
        Compare stored password with input password
        IF password valid
```

Figure 4.10: Login Psudocode

```
        IF password valid
            IF user is tutor
                Check tutor profile status from tutor_profile_status_table
                IF profile status is 'Not created'
                    Redirect to tutor profile creation page
                ELSE
                    Store user details in session
                    Redirect to tutor dashboard
            ELSE
                Store user details in session
                Redirect to respective dashboard based on role:
                    - Admin dashboard for 'admin_table'
                    - Student dashboard for 'student_table'
                    - Parent dashboard for 'parent_table'
        ELSE
            RETURN error "Invalid roll number or password"
    ELSE
        RETURN error "Invalid roll number or password"

END login process
```

Figure 4.11: Login Psudocode

4.1.5 Admin Handle Tutor Profile Status Module

The `admin_handle_tutor_profile_status.js` module allows administrators to view tutors with pending profiles and update their profile status (accepted or rejected). This module also handles the process of sending an email notification to the tutor after the profile status update. The main features of this module include:

- **Fetching Tutor Data with Pending Status:** The module retrieves tutors with a pending profile status from the `tutor_profile_data_table`, joining it with the `tutor_profile_status_table` and `tutor_table` to get the complete tutor details.
- **File Path Assignment:** For each tutor, the module attempts to find the tutor's profile picture and degree certificate based on the tutor's ID. It checks for different file extensions and constructs the corresponding file paths.
- **Updating Profile Status:** Administrators can update the profile status of a tutor, marking it as either `accepted` or `rejected`. The updated status is saved in the

`tutor_profile_status_table`.

- **Email Notification:** After updating the profile status, an email is sent to the tutor, informing them of the outcome. The email content is dynamic, reflecting whether the profile was accepted or rejected.

- **Error Handling:** The module includes error handling for database queries and email sending operations, ensuring the application provides meaningful feedback in case of failures.

The following pseudocode illustrates the logical flow of the tutor profile status handling process.

```
BEGIN Fetching tutor data with pending status

    Execute SQL query to fetch tutor data with status 'pending' from the following tables:
        tutor_profile_data_table (tpd)
        tutor_profile_status_table (tps)
        tutor_table (tt)

    FOR each tutor in the results
        Define potential file extensions for profile picture (jpeg, jpg, png) and degree files (pdf, doc, docx)

        Search for profile picture:
            FOR each image extension
                IF file exists at tutor_pic/{Tutor_ID}.{ext}
                    Set profilePicPath to /tutor_pic/{Tutor_ID}.{ext}
                    BREAK

        Search for degree certificate:
            FOR each degree extension
                IF file exists at tutor_degree/{Tutor_ID}.{ext}
                    Set degreeFilePath to /tutor_degree/{Tutor_ID}.{ext}
```

Figure 4.12: Admin Handle Tutor Profile Status Psudocode

```
    IF error occurs while updating profile status
        Return error "Failed to update profile status"

    Fetch tutor's email address from tutor_table based on tutorId

    IF error occurs while fetching email
        Return error "Failed to fetch tutor email"

    Construct email subject and message based on status (accepted or rejected)

    Use Nodemailer to send email:
        Set up transporter using Gmail service
        Send email with the subject, message, and tutor's email address

    IF error occurs while sending email
        Return error "Failed to send email"

    Return success message: "Profile status updated to {status} and email sent to tutor"

END Updating tutor profile status
```

Figure 4.13: Admin Handle Tutor Profile Status Psudocode

4.1.6 Student Search for Tutor Module

The `student_search_for_tutor.js` module allows students to search for tutors based on various criteria such as subject, availability, price, country, name, language, and grade. The module also retrieves the tutor's profile picture if available. The main features of this module include:

- **Filtering Tutors Based on Criteria:** Students can search for tutors by providing specific search criteria such as the tutor's name, subject taught, availability, price range, country, language, and grade. The query dynamically adjusts based on the provided search parameters.

- **SQL Query with Dynamic Filters:** The module constructs a dynamic SQL query based on the student's input. The query filters tutors who are marked as "accepted" in the `tutor_profile_status_table`.

- **Fetching Tutor Details:** The search results include tutor details like name, roll number, email, phone number, subjects taught, teaching experience, availability, country, language, teaching fee, and introduction.

- **Retrieving Profile Image:** The module checks for the presence of a profile image for each tutor by looking for specific image formats (`.jpg`, `.jpeg`, `.png`). If an image is found, the file path is returned with the tutor's details.

- **Error Handling:** The module handles errors such as database query failures or no tutors found based on the search criteria. Appropriate error messages are sent to the client.

The following pseudocode illustrates the logical flow of the student searching for a tutor.

```
BEGIN Student search for tutor
    Extract search criteria from the request body (subject, availability, price, country, name, language, grade)

    Initialize SQL query to fetch tutor details from tutor_table, tutor_profile_data_table, and tutor_profile_status_table
    WHERE profile status is 'accepted'

    IF name is provided
        Add condition to search for name in tutor_name (LIKE %name%)

    IF subject is provided
        Add condition to find subject in tutor_teaches_subject

    IF availability is provided and not 'Anytime'
        Add condition to find availability in tutor_availability_time

    IF price is provided
        Add condition to filter tutors based on fee range (+/- 1000 of the price)

    IF country is provided
        Add condition to filter tutors based on tutor_country
```

Figure 4.14: Student Search for Tutor Psudocode

```
        Execute the SQL query with the query parameters

        IF error occurs while querying the database
            Return error "Internal server error"

        IF no tutors found based on the criteria
            Return error "No tutors found"

        FOR each tutor in the results
            Initialize image path as null

            FOR each image format (jpg, jpeg, png)
                IF profile image exists for the tutor
                    Set imagePath to the tutor's image path

            Return tutor data with imagePath

        Return the tutors with their details and imagePath

    END Student search for tutor
```

Figure 4.15: Student Search for Tutor Psudocode

4.1.7 Parent Approve Tutor Appointed by Student Module

The `parent_approve_tutor_appointed_by_student.js` module allows parents to approve tutors appointed by students. The module retrieves details of tutors whose appointment is pending approval by the parent. The main functionalities of this module are as follows:

- **Fetching Tutor Approval Requests:** The parent can view details of tutors who have been appointed by students but are awaiting parental approval. The module fetches these tutors based on the parent's ID and the pending approval status.
- **SQL Query to Retrieve Tutor Information:** The SQL query retrieves relevant tutor details including name, email, contact number, city, country, language, specialization, teaching fee, and profile image. The query filters based on the parent's ID and pending approval status from the `student_approve_tutor_status_table`.
- **Retrieving Tutor Profile Image and Degree Document:** The module checks for the existence of the tutor's profile picture and degree document. If found, it includes

their URLs in the response for rendering in the client interface.

- **Error Handling:** The module handles cases such as query errors or when no pending tutors are found. Appropriate error messages are sent back to the client.

The following pseudocode outlines the logical flow of the `parent_approve_tutor_appointed_by_stude` process.

```
BEGIN Parent approve tutor appointed by student

    Extract parent ID from the request body

    Initialize SQL query to fetch tutors from the student_approve_tutor_status_table, tutor_table, and tutor_profile_data_table
    WHERE parent_approving_status is 'pending' AND Parent_ID matches the given parent ID

    Execute the SQL query with the parent ID as a parameter

    IF error occurs while querying the database
        Return error "Internal server error"

    IF no tutors found for approval
        Return error "No tutors found for approval"

    FOR each tutor in the results
        Initialize image path and degree link as null

        FOR each image format (jpg, jpeg, png)
            IF profile image exists for the tutor
```

Figure 4.16: Parent Approve Tutor Psudocode

```
    FOR each tutor in the results
        Initialize image path and degree link as null

        FOR each image format (jpg, jpeg, png)
            IF profile image exists for the tutor
                Set imagePath to the tutor's image path

        FOR each degree file format (pdf)
            IF degree file exists for the tutor
                Set degreeLink to the tutor's degree document link

        Return tutor data with imagePath and degreeLink

    Return the list of tutors with their details, imagePath, and degreeLink

END Parent approve tutor appointed by student
```

Figure 4.17: Parent Approve Tutor Psudocode

4.1.8 Tutor Handle Student Request Module

The `tutor_handle_student_request.js` module manages the response of a tutor to student requests. Tutors can either accept or reject a student's request for tutoring, and the module sends appropriate email notifications to both the student and their parent. The functionality is explained in detail below:

- **Handling Request Action:** The module processes two actions—`accept` or `reject`—based on the tutor's decision regarding a student's request. If the action is `accept`, the tutor is assigned to the student. If `reject`, the student's request is deleted from the database.

- **SQL Queries to Fetch Information:**

  - `getEmailsQuery`: Fetches the student and parent emails, along with the parent's ID, from the `student_approve_tutor_status_table`.

  - `getTutorDetailsQuery`: Retrieves detailed tutor information such as name, contact, teaching fee, availability, degree information, etc., from the `tutor_table` and `tutor_profile_data_table`.

- **Sending Emails:** If the tutor accepts or rejects the request, the module sends an email to the student and their parent. The emails are sent via Gmail using `nodemailer`. The email contains tutor details, including the tutor's contact information, teaching fee, availability, etc.

- **Error Handling:** The module checks for missing parameters in the request and ensures that the necessary information is available before proceeding with the database operations and email notifications. In case of any errors, appropriate error messages are returned.

The following pseudocode summarizes the logical flow of the `tutor_handle_student_request` process:

```
BEGIN Tutor handle student request

    Extract action, Student_ID, Tutor_ID from the request body

    IF any required parameter is missing
        Return error "Missing required parameters"

    Initialize SQL query to fetch emails of student and parent from the student_approve_tutor_status_table

    Execute the SQL query with Student_ID and Tutor_ID as parameters

    IF error occurs while fetching emails or no records are found
        Return error "Failed to fetch email details"

    Extract Student_Email, Parent_Email, and Parent_ID from the results

    Initialize SQL query to fetch tutor details from the tutor_table and tutor_profile_data_table

    Execute the SQL query with Tutor_ID as a parameter
```

Figure 4.18: Tutor Handle Student Request Psudocode

```
    Extract Student_Email, Parent_Email, and Parent_ID from the results

    Initialize SQL query to fetch tutor details from the tutor_table and tutor_profile_data_table

    Execute the SQL query with Tutor_ID as a parameter

    IF error occurs while fetching tutor details or no records are found
        Return error "Failed to fetch tutor details"

    IF action is 'accept'
        Insert tutor assignment record into the assigned_tutors_record_table
        Send acceptance email with tutor details to student and parent
        Return success message "Request accepted and emails sent"

    ELSE IF action is 'reject'
        Delete student request from the student_approve_tutor_status_table
        Send rejection email with tutor details to student and parent
        Return success message "Request rejected and emails sent"

END Tutor handle student request
```

Figure 4.19: Tutor Handle Student Request Psudocode

4.1.9 Community Management Module

The community management module in this project allows tutors to create communities, admins to approve or reject them, and students to join them. The module handles various tasks such as email notifications, database querying, and error handling.

4.1.9.1 Algorithms

Below is the detailed description of the algorithms used:

- **Email Sending Algorithm:** Initialize the email transporter with SMTP settings. Create mail options with recipient email, subject, and message body. Send the email and handle any errors. The following pseudocode summarizes the logical flow of the process:

```
procedure sendEmail(recipientEmail, subject, messageBody)
    transporter = createTransporter(SMTP settings)
    mailOptions = {
        from: "smarttutor253@gmail.com",
        to: recipientEmail,
        subject: subject,
        text: messageBody
    }
    result = transporter.sendMail(mailOptions)
    if result == "error" then
        output "Failed to send email"
        return "failure"
    else
        output "Email sent successfully"
        return "success"
    end if
end procedure
```

Figure 4.20: Email Sending Algorithm Pseudo code

- **Create Community Algorithm:** Prepare an SQL query to insert a new community into the database. Execute the query with provided community details. Output success or failure message based on the query result.

  The following pseudocode summarizes the logical flow of the process:

```
procedure createCommunity(name, description, subject, tutorId)
    sql = "INSERT INTO communities (name, description, subject, created_by, status) VALUES (?, ?, ?, ?, 'pending')"
    result = db.query(sql, [name, description, subject, tutorId])
    if result == "error" then
        output "Failed to create community"
        return "failure"
    else
        output "Community created successfully and is pending approval"
        return "success"
    end if
end procedure
```

Figure 4.21: Community Creation Algorithm Pseudocode

- **Get Pending Communities Algorithm:** Prepare an SQL query to fetch all communities with status "pending".Execute the query and output the results.

  The following pseudocode summarizes the logical flow of the process:

```
procedure getPendingCommunities()
    sql = "SELECT * FROM communities WHERE status = 'pending'"
    results = db.query(sql)
    if results == "error" then
        output "Failed to fetch communities"
        return "failure"
    else
        output results
        return "success"
    end if
end procedure
```

Figure 4.22: Pending Communities Retrieval Algorithm Pseudocode

- **Approve Community Algorithm:** Check if the provided status is either "approved" or "rejected". Fetch community details using the community ID. Fetch tutor email using the tutor ID from the community details. Update the community status in the database. Send an email to the tutor notifying them of the status update.

  The following pseudocode summarizes the logical flow of the process:

```
procedure approveCommunity(communityId, status)
    if status != "approved" AND status != "rejected" then
        output "Invalid status"
        return "failure"
    end if

    sql = "SELECT * FROM communities WHERE id = ?"
    community = db.query(sql, [communityId])
    if community == "error" OR community.length == 0 then
        output "Failed to fetch community details"
        return "failure"
    end if

    tutorId = community[0].created_by
    tutorSql = "SELECT Tutor_Email FROM tutor_table WHERE Tutor_ID = ?"S
    tutor = db.query(tutorSql, [tutorId])
    if tutor == "error" OR tutor.length == 0 then
        output "Failed to fetch tutor details"
        return "failure"
    end if

    tutorEmail = tutor[0].Tutor_Email
    updateSql = "UPDATE communities SET status = ? WHERE id = ?"
    result = db.query(updateSql, [status, communityId])
    if result == "error" then
        output "Failed to update community status"
        return "failure"
    end if

    messageBody = "Hello,\n\nYour community '" + community[0].name + "' has been " + status + ".\n\nThank you for using our platform!"
    sendEmail(tutorEmail, "Your Community Status Update", messageBody)
    output "Community has been " + status + ". Email sent to tutor."
    return "success"
end procedure
```

Figure 4.23: Community Approval Algorithm Pseudocode

- **Search Communities Algorithm:** Prepare an SQL query to fetch approved communities matching the subject. Execute the query with the provided subject and output the results.

  The following pseudocode summarizes the logical flow of the process:

```
procedure searchCommunities(subject)
    sql = "SELECT * FROM communities WHERE subject LIKE ? AND status = 'approved'"
    results = db.query(sql, ["%" + subject + "%"])
    if results == "error" then
        output "Failed to fetch communities"
        return "failure"
    else
        output results
        return "success"
    end if
end procedure
```

Figure 4.24: Community Search Algorithm Pseudocode

- **Join Community Algorithm:** Fetch community details using the community ID. Check if the community is approved. Check if the student has already joined the community. Insert the student into the community members table. Fetch student details using the student ID. Send an email to the student confirming their membership.

The following pseudocode summarizes the logical flow of the process:

```
procedure joinCommunity(studentId, communityId)
    communitySql = "SELECT * FROM communities WHERE id = ? AND status = 'approved'"
    community = db.query(communitySql, [communityId])
    if community == "error" OR community.length == 0 then
        output "Community not found or not approved"
        return "failure"
    end if

    checkSql = "SELECT * FROM community_members WHERE community_id = ? AND student_id = ?"
    membership = db.query(checkSql, [communityId, studentId])
    if membership == "error" OR membership.length > 0 then
        output "You have already joined this community"
        return "failure"
    end if

    joinSql = "INSERT INTO community_members (community_id, student_id, joined_at) VALUES (?, ?, NOW())"
    result = db.query(joinSql, [communityId, studentId])
    if result == "error" then
        output "Failed to join community"
        return "failure"
    end if

    studentSql = "SELECT * FROM student_table WHERE Student_ID = ?"
    student = db.query(studentSql, [studentId])
    if student == "error" OR student.length == 0 then
        output "Student not found"
        return "failure"
    end if

    studentEmail = student[0].Student_Email
    studentName = student[0].Student_Name
    messageBody = "Hello " + studentName + ",\n\nYou have successfully joined the community: '" + community[0].name + "'.\n\nThank you for using our platform!"
    sendEmail(studentEmail, "You have joined a community!", messageBody)
    output "You have successfully joined the community!"
    return "success"
end procedure
```

Figure 4.25: Community Joining Algorithm Pseudocode

- **Get Community Members Algorithm:** Prepare an SQL query to fetch members of a community. Execute the query with the provided community ID and output the results.

  The following pseudocode summarizes the logical flow of the process:

```
procedure getCommunityMembers(communityId)
    sql = "SELECT students.id AS student_id, students.name AS student_name, cm.joined_at FROM community_members cm JOIN students ON cm.student_id = students.id WHERE cm.community_id = ?"
    results = db.query(sql, [communityId])
    if results == "error" then
        output "Failed to fetch community members"
        return "failure"
    else
        output results
        return "success"
    end if
end procedure
```

Figure 4.26: Community Members Retrieval Algorithm Pseudocode

- **Get Joined Communities Algorithm:** Prepare an SQL query to fetch communities joined by the student. Execute the query with the provided student ID and output the results.

  The following pseudocode summarizes the logical flow of the process:

```
procedure getJoinedCommunities(studentId)
    sql = "SELECT c.id, c.name, c.subject, c.description FROM communities AS c INNER JOIN community_members AS cm ON c.id = cm.community_id WHERE cm.student_id = ?"
    results = db.query(sql, [studentId])
    if results == "error" then
        output "Failed to fetch communities"
        return "failure"
    else
        output results
        return "success"
    end if
end procedure
```

Figure 4.27: Joined Communities Retrieval Algorithm Pseudocode

- **Get Communities by Tutor Algorithm:** Prepare an SQL query to fetch communities created by the tutor. Execute the query with the provided tutor ID and output the results.

  The following pseudocode summarizes the logical flow of the process:

```
procedure getCommunitiesByTutor(tutorId)
    sql = "SELECT id, name, description, subject, status, created_at FROM communities WHERE created_by = ? AND status = 'approved'"
    results = db.query(sql, [tutorId])
    if results == "error" then
        output "Failed to fetch communities"
        return "failure"
    else
        output results
        return "success"
    end if
end procedure
```

Figure 4.28: Communities Created by Tutor Retrieval Algorithm Pseudocode

4.1.10 Community Chat and File Upload Module

The community chat and file upload module in this project allows users to communicate within communities by sending messages and uploading files. The module manages file uploads, message submissions, and the retrieval of both messages and files from the database.

4.1.10.1 Algorithms

Below is the detailed description of the algorithms used:

- **File Upload Algorithm:** Configure the file storage settings using `multer` to specify the destination folder and filename. Store the uploaded file in the designated folder

(`uploads/`) with a unique timestamp-based filename. Extract the file's MIME type (e.g., image, document, video) to determine its type. Insert the file's metadata, including the file URL, type, and sender's details, into the `community_files` table in the database. If an error occurs during the file upload process, return a failure message. Otherwise, confirm successful file upload and return the file URL.

This algorithm handles the file upload process, saving the file to a specified directory and inserting relevant data into the database.

```
procedure uploadFile(userId, communityId, file, role)
    destination = "uploads/"  // Set the directory to save the file
    fileName = currentTimestamp() + "-" + file.originalName  // Create a unique filename using the current timestamp and original file name
    saveFile(file, destination + fileName)  // Save the file to the designated folder

    fileType = extractFileType(file.mimetype)  // Extract the MIME type of the file (e.g., image, document, video)
    sql = "INSERT INTO community_files (community_id, sender_id, file_url, file_type, sender_role)"
    executeSQL(sql, [communityId, userId, fileUrl, fileType, role])  // Insert file metadata into the database
    if error then
        output "Failed to upload file"  // Handle error if the file upload fails
    else
        output "File uploaded successfully"  // Confirm successful file upload
        return fileUrl  // Return the file URL for future reference
end procedure
```

Figure 4.29: File Upload Algorithm Pseudocode

- **Message Sending Algorithm:** Extract the message details, including the sender's ID, community ID, message content, and sender's role. Insert the message into the `community_chat` table in the database. If an error occurs during the insertion, return a failure message. Otherwise, confirm successful message submission.

  This algorithm inserts a message into the community chat database and sends a response to confirm success.

```
procedure sendMessage(userId, communityId, message, role)
    sql = "INSERT INTO community_chat (community_id, sender_id, message, sender_role)"
    executeSQL(sql, [communityId, userId, message, role])  // Insert message details into the database
    if error then
        output "Failed to send message"  // Handle error if the message fails to send
    else
        output "Message sent successfully"  // Confirm message has been successfully sent
end procedure
```

Figure 4.30: Message Sending Algorithm Pseudocode

- **Message and File Retrieval Algorithm:** Retrieve messages and files from the `community_chat` and `community_files` tables using a `UNION` SQL query. For

each message or file, determine its type (either 'message' or 'file'), sender's name (based on role), and timestamp. Sort the results by the timestamp to ensure messages and files are displayed in chronological order. Return the sorted results to the user. If an error occurs during the retrieval process, return a failure message.

This algorithm retrieves both messages and files for a specific community, ordered by timestamp. It uses a UNION SQL query to combine results from both the community chat and community files tables.

```
procedure fetchMessagesAndFiles(communityId)
    sql = """
    SELECT 'message' AS type,
           CASE WHEN community_chat.sender_role = 'student' THEN student_table.Student_Name
                ELSE tutor_table.Tutor_Name END AS sender_name,
           community_chat.message,
           community_chat.sent_at AS timestamp,
           null AS file_url,
           null AS file_type,
           community_chat.sender_role
    FROM community_chat
    LEFT JOIN student_table ON community_chat.sender_id = student_table.Student_ID
    LEFT JOIN tutor_table ON community_chat.sender_id = tutor_table.Tutor_ID
    WHERE community_chat.community_id = ?
    UNION
    SELECT 'file' AS type,
           CASE WHEN community_files.sender_role = 'student' THEN student_table.Student_Name
                ELSE tutor_table.Tutor_Name END AS sender_name,
           null AS message,
           community_files.uploaded_at AS timestamp,
           community_files.file_url AS file_url,
           community_files.file_type AS file_type,
           community_files.sender_role
    FROM community_files
    LEFT JOIN student_table ON community_files.sender_id = student_table.Student_ID
    LEFT JOIN tutor_table ON community_files.sender_id = tutor_table.Tutor_ID
    WHERE community_files.community_id = ?
    ORDER BY timestamp ASC
    """
    executeSQL(sql, [communityId, communityId])  // Execute the SQL query to fetch messages and files
    if error then
        output "Failed to fetch messages and files"  // Handle error if fetching fails
    else
        output results  // Return results (messages and files ordered by timestamp)
end procedure
```

Figure 4.31: Message And File Retrieval Algorithm Pseudocode

4.2 External APIs/SDKs

Following table describes the SDKs which we have used in our project so far.

API and version	Description	Purpose of usage	API endpoint/function/class used
Nodemailer (version 6.7.2)	Email sending service	Used for sending email notifications for actions like acceptance and rejection of requests	nodemailer.createTransport, transporter.sendMail
Visual Studio Code (version 1.80)	Code editor and compiler	Used for writing and compiling the project code	N/A
MySQL (version 8.0)	Relational database management system	Used for database interactions, queries, and storing data for the application	db.query

4.3 Testing Details

Once the system has been successfully developed, testing has to be performed to ensure that the system working as intended.

4.3.1 Unit Testing

Each unit test is designed to test a specific function or method independently from other components, helping to identify issues directly related to the functionality being tested.

Following is the example of Unit testing:

TEST CASE ID	TEST OBJECTIVE	PRECONDITION	STEPS	TEST DATA	EXPEXTED RESULT	ACTUAL RESULT	PASS/FAIL
TC001	Admin login with username and password	Admin should be registered with valid username and password	Go to login. Enter the admin Roll and password	Roll number and Password	System displays admin dashboard	As expected	Pass
TC002	Verify user registration with OTP	User should have a valid email address	Go to signup and enter valid username, email and password	Entering email and password and OTP	System should send OTP and displays dashboard	As expected	Pass
TC003	Tutor Profile creation	Tutor should be logged in with valid credentials	Go to sign up and do the verification	Creating profile by entering details including image and degree document	Details should be stored and sent to admin for verification	As expected	Pass
TC004	Student search Tutor	Student should be logged in with valid credentials	Go to the dashboard and click Search for tutor	Entering each search like subject, language, price etc	System shows the tutor profiles matching the criteria	As expected	Pass
TC005	Tutor Accepts or rejects student request	Tutor should be logged in with valid credentials	Enter the dashboard and go to Student Requests	Student requesting the tutor for teaching	System displays the list of students with pending requests	As expected	Pass
TC006	Tutor creating social community	Tutor should be logged in and his/her profile should be approved	Go to the Create Community on dashboard and enter the community name and create	Creating the community by entering community name and subject	Approval request should be sent to the admin and created on approval	As expected	Pass
TC007	Student joining the community	Student should be logged in with valid credentials	Go to the Join Community and search for community and click join	Searching the community and joining, Sending and uploading messages	Student should join the community and upload and send message	As expected	Pass

Figure 4.32: Unit Testing

Bibliography

[1] A Kolyshkin and S Nazarovs. Stability of slowly diverging flows in shallow water. *Mathematical Modeling and Analysis*, 2007.