Software Engineering Project

Tutoring Software Platform *StudyBuddy*

Tsitsi Nyamutswa, Magdalena Leymańczyk, Aleksandra Klos

University Name: Warsaw University of Technology
Department Name: Faculty of Mathematics and Information Science
Date: November 26, 2023

1. Project Description

The fundamental objective of the project is to create a user-friendly platform that connects students with qualified tutors, making it easy for students to find and schedule a lesson. The tutors will have effective tools to manage their sessions, track students' progress and generate income while maintaining the quality of their services.

2. User Stories

Our User Stories consist of a front and reverse side, each for each actor - Student, Tutor, Parent/ Legal Guardian, Administrator. These are as follows:

- Student

As a… student

I want to…	so that…
register on the platform	I can access tutoring services and track my progress
login to my account	I can access my personalized dashboard, view my information, and use platform services
search for tutors based on specific criteria	I can find the right tutor to help me with my educational needs
request a tutoring session with a specific tutor	I can schedule personalized learning sessions when it suits me
have live video tutoring session with a chosen tutor	I can engage in interactive and real-time learning
track and view my progress and performance feedback from a tutor	I can monitor my educational growth
cancel a scheduled lesson	I can adjust my schedule in case plans change
rate a tutor after a tutoring session	I can share my experience and help other students make informed choices
access a list of my past lessons and view my upcoming lessons	I can keep track of my educational journey and be prepared for future sessions
be able to update my account	I can adjust my personal information and search preferences

Table 1: List of User Stories for a Student (Front Side)

Acceptance criteria checklist
(Functionality)

- Can I register on the tutoring platform, providing essential information for creating my student account?
- Can I securely log in to my account on the tutoring platform to access my personalized dashboard?
- Can I search for tutors based on specific subjects, location, and availability criteria?
- Can I request a tutoring session with a specific tutor, specifying the date and time that suits me?
- Can I cancel a scheduled lesson, providing a reason and receiving confirmation?
- Can I rate my tutor after a tutoring session, providing a rating and feedback comments?
- Can I update my account information, preferences, and settings, including password changes or account deactivation?

Non Functional Requirements
(Usability Reliability Performance Security)

- Is the platform available and accessible to students 24/7, with minimal downtime for maintenance or updates?
- Is the platform's user interface intuitive and easy to navigate for students?
- Is the platform accessible and easy to navigate to students with disabilities?
- Is the platform responsive on both desktop and smartphones?
- Does the platform work seamlessly across various web browsers and operating systems?
- Is the system dependable, ensuring that registrations, logins, searches, and session management occur accurately and consistently?
- Does the platform operate efficiently, handling a large number of students, lessons, and data without significant delays?
- Can a student be able to reset the password in case it is forgotten?
- Is user data, communication, and interactions secure, maintaining the privacy and confidentiality of students' educational information and account details?

Figure 1: User Story Card for a Student (Back Side)

- Tutor

As a tutor…

I want to…	so that…
create and manage my profile	I can provide information about my qualifications and teaching style
login to my account	I can access my personalized dashboard, manage my tutoring sessions, and interact with students
manage my availability and schedule tutoring sessions	I can efficiently connect with students

monitor and record student progress and performance	I can offer effective guidance and feedback
provide feedback to my students after a tutoring session	I can help them understand their strengths and areas for improvement
upload educational materials, such as videos, documents, and quizzes	students can access valuable resources for learning
access a list of my past lessons and view my upcoming lessons	I can keep track of my tutoring history and be prepared for upcoming sessions with students
be able to update my account	I can modify my personal information

Table 2: List of User Stories for a Tutor (Front Side)

Acceptance criteria checklist
(Functionality)

- Can I create and manage my tutor profile, providing information about my qualifications and teaching style?
- Can I securely log in to my tutor account on the tutoring platform to access my personalized dashboard?
- Can I manage my availability and schedule tutoring sessions efficiently, specifying subjects, date, and time?
- Can I monitor and record student progress and performance, providing effective guidance and feedback?
- Can I provide feedback to my students after tutoring sessions, helping them understand their strengths and areas for improvement?
- Can I upload educational materials, such as videos, documents, and quizzes, making them accessible to students?
- Can I access a list of my past lessons and view my upcoming scheduled lessons for tracking and preparation?
- Can I update my account information, including personal details, qualifications, and preferences when needed?

Non Functional Requirements
(Usability Reliability Performance Security)

- Is the platform available and accessible to tutors 24/7, with minimal downtime for maintenance or updates?
- Is the platform's user interface intuitive and easy to navigate for tutors?
- Is the platform responsive on both desktop and smartphones?
- Does the platform work seamlessly across various web browsers and operating systems?
- Is the system dependable, ensuring that registrations, logins, searches, and session management occur accurately and consistently?
- Does the platform operate efficiently, even when tutors have numerous lessons, resources, and data to manage?
- Can a tutor be able to reset the password in case it is forgotten?
- Is the tutor profile data, communication, and interactions secure, maintaining the privacy and confidentiality of tutor information?

Figure 2: User Story Card for a Tutor (Back Side)

- Parent/ Legal Guardian

As a parent/ legal guardian…

I want to…	so that…
register on the platform	I can access tutoring services and track my child's progress
login to my account	I can access my personalized dashboard, manage payments and communicate with tutors
have control over my child's tutoring account	I can monitor their progress, receive updates, and ensure a safe and productive environment
have the ability to communicate with my child's tutor	I can stay informed about my child's educational progress
have secure payment options and access to my billing history	I can conveniently pay for tutoring sessions of my child and track my financial transactions
set controls and preferences for my child's tutoring experience	I can ensure a safe learning environment for my child
access a list of my child's past lessons and view the upcoming lessons	I can see a past contribution and be prepared for their future sessions
be able to update my account	I can modify my personal information

Table 3: List of User Stories for a Parent/ Legal Guardian (Front Side)

Acceptance criteria checklist
(Functionality)

- Can I access a dedicated section in my parent dashboard to view my child's past tutoring lessons?
- Can I see a chronological list of past lessons with details such as date, time, tutor name, and subject?
- Can I click on a past lesson to access additional details, including any notes or feedback provided by the tutor?
- Can I view a separate list of my child's upcoming scheduled lessons, including date, time, and tutor information?
- Can I filter and sort my child's past and upcoming lessons by date, subject, or tutor name for ease of access?
- Can I receive reminders and notifications for my child's upcoming lessons based on my preferences?

Non Functional Requirements
(Usability Reliability Performance Security)

- Is the platform available and accessible to parents 24/7, with minimal downtime for maintenance or updates?
- Is the platform's user interface intuitive and easy to navigate for parents?
- Is the platform accessible and easy to navigate to parents with disabilities?
- Is the platform responsive on both desktop and smartphones?
- Does the platform work seamlessly across various web browsers and operating systems?
- Is the lesson history and schedule system dependable, ensuring accurate and up-to-date information for parents and their children?
- Does the lesson view process occur efficiently, even when parents have numerous children and lessons to track?
- Can a parent be able to reset the password in case it is forgotten?
- Is lesson data and communication secure, maintaining the privacy and confidentiality of parent and child information?

Figure 3: User Story Card for a Parent/ Legal Guardian (Back Side)

- Administrator

As an… administrator

I want to…	so that…
have the ability to manage user accounts, roles, and permissions	I can ensure the integrity and security of the platform
review tutor profiles and session recordings for quality and compliance	the platform maintains high-quality educational content
assist users with technical issues, answer questions, and offer guidance	users have a smooth and trouble-free experience on the platform
access to financial reports and data	I can analyze the platform's financial performance and make informed decisions
configure system settings	I can adjust and optimize platform functionality

Table 4: List of User Stories for an Administrator (Front Side)

Acceptance criteria checklist
(Functionality)

- Can I manage user accounts, roles, and permissions, including creating, modifying, and disabling user accounts as needed?
- Can I ensure the integrity and security of the platform by enforcing access controls and security measures?

- Can I review tutor profiles and session recordings to ensure quality and compliance
- with platform standards?
- Can I take actions based on the quality of tutor profiles and session recordings to maintain high-quality educational content?
- Can I assist users with technical issues, answer their questions, and offer guidance on using the platform effectively?
- Can I provide support and resolution for users' technical problems, ensuring a smooth and trouble-free experience on the platform?
- Can I access financial reports and data to analyze the platform's financial performance and make informed decisions?
- Can I configure system settings to adjust and optimize platform functionality to meet fast-changing needs and requirements?

Non Functional Requirements
(Usability Reliability Performance Security)

- Is the administrator interface user-friendly, making it easy to manage user accounts, review content, and configure system settings?
- Is the system dependable, ensuring that user accounts, content review, support, and financial analysis occur accurately and consistently?
- Does the administrative system operate efficiently, even when handling a large number of user accounts, content review tasks, and system configurations?
- Is administrator access and data handling secure, maintaining the privacy and confidentiality of user information, financial data, and system settings?

Figure 4: User Story Card for an Administrator (Back Side)

3. UML Use Case Diagrams

UML Use Case Diagrams are used for defining the requirements that a system must fulfill. They present actors, expected functionalities but also some dependencies between them. The relationships appear the following:

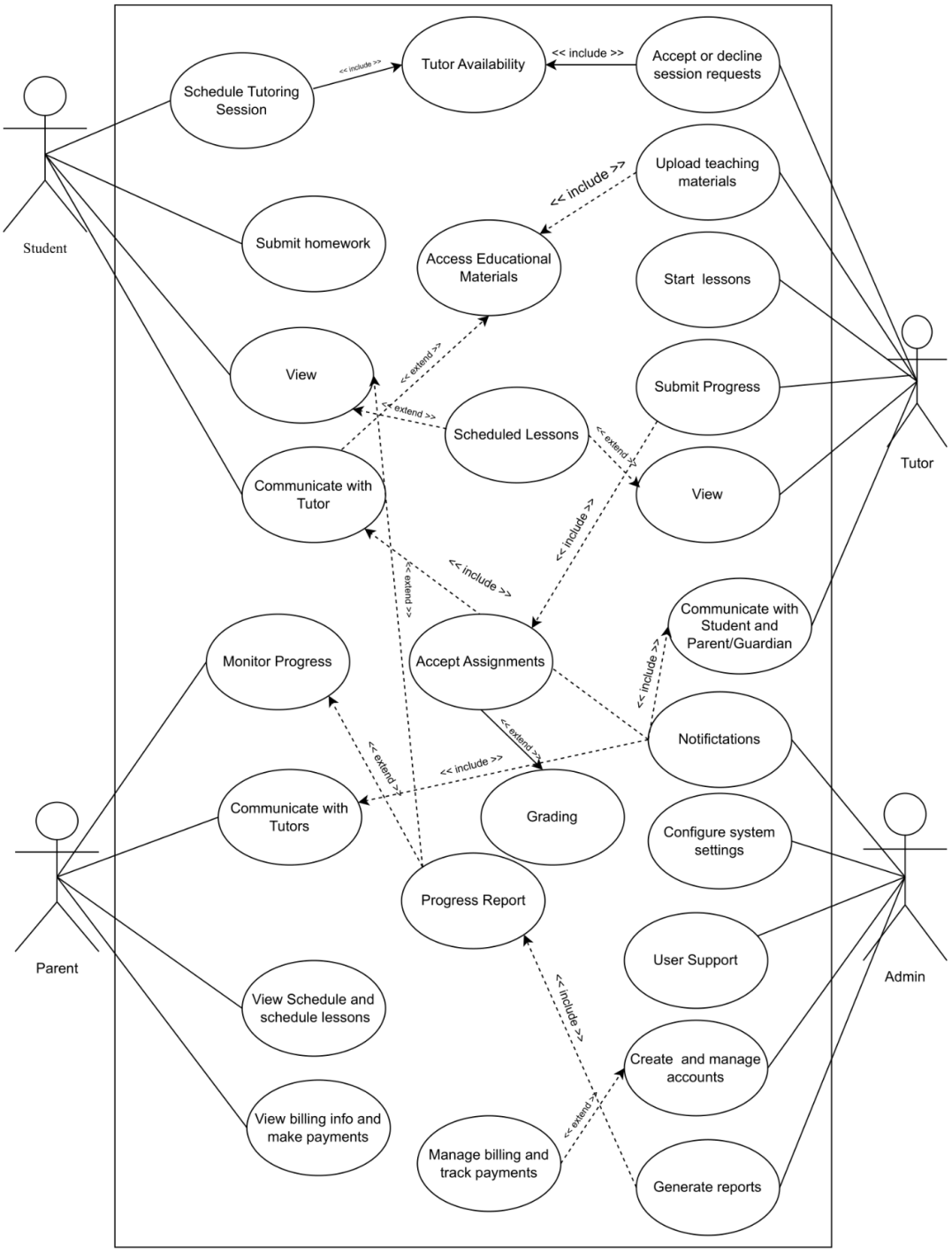

Figure 5: All relationships between actors captured in Use Case Diagrams

As seen above, each actor interacts with the system through various use cases. In other words, the diagram is designed to show the system from the perspective of different users, showing what each user can do within the system and how those actions relate to each other. The use cases are concentrated on a given type of user to provide a clear separation of functionalities.

- Student
    - *Schedule Tutoring Session*: the student can arrange a time for a tutoring session, which is dependent on tutor's availability.
    - *Submit Homework*: allows the student to upload homework, possibly for review or grading.
    - *View*: this use case allows the student to view scheduled lessons or see progress in the report.
    - *Scheduled Lessons*: use case that organizes and displays the student's lesson schedule.
    - *Communicate with Tutor*: provides a channel for direct communication between student and tutor. A notification appears when the tutor answers the request.
    - *Access Educational Materials*: the student can access learning materials, such as pdf exercises, videos or interactive modules as an extension for the lesson with a chosen tutor.

- Tutor
    - *Tutor Availability*: tutor updates their available times for conducting tutoring sessions.
    - *Accept or Decline Session Requests*: based on tutor's availability, tutor has the ability to accept or reject proposed sessions from students.
    - *Upload Teaching Materials*: allows tutor to provide educational materials, including giving access to their student.
    - *Start Lessons*: tutor begins the scheduled tutoring lesson.
    - *Submit Progress*: tutor can record progress of their students.
    - *Accept Assignments*: when submitting progress, the tutor receives completed assignments from students for review or grading.
    - *Grading*: tutor accepts assignments and after that, assesses them and provides grades.
    - *View*: similarly to the student, by this use case tutor can view scheduled lessons with his pupils.
    - *Communicate with Student and Parent/ Guardian:* enables the tutor to communicate with both the student and their parents or guardians regarding progress, lesson scheduling or payment. A notification can be seen when the response from the student or their parent appears.

- Parent
    - *Monitor Progress*: allows the parent to keep track of their child's progress through an online dashboard or progress report.
    - *Communicate with Tutors*: the parent can discuss their child's progress, concerns, scheduling or payments with a tutor as well as get notifications when the response appears.
    - *View Schedule and Schedule Lessons*: the parent can view the tutoring schedule and may have the ability to arrange/ book the session on behalf of their child.
    - *View Billing Info and Make Payments*: the parent can access billing information related to the tutoring services and make payment accordingly.

- Administrator
    - *Notifications:* the administrator can manage and distribute notifications to students, parents or tutors.
    - *Configure System Settings*: involves setting up and adjusting the system's operational parameters.
    - *User Support*: the administrator provides help and support to users encountering issues.
    - *Create and Manage Accounts*: the administrator has the authority to create new user accounts and manage existing ones as well as manage payment issues on the user's account level.
    - *Manage Billing and Track Payments***:** the administrator can monitor financial transactions within the system, such as tracking incoming payments and managing invoices.
    - *Generate Reports*: the administrator can generate various reports, such as connected to system usage, financials or other user's activity like progress.

Additionally, the Use Case diagram includes various relationships between use cases:

- <<include>> → this relationship indicates that the included use case ("derived" use case) is an inseparable part of the base use case. For example, *Schedule Tutoring Session* (base use case) includes *Tutor Availability* ("derived" use case)
- <<extend>> → this relationship suggests that an extending use case adds extra steps or information to the base use case under certain conditions.

4. UML Class Diagrams

The UML Class Diagrams are structural diagrams whose purpose is to design the overall architecture of the system and identify potential complexities in the existing code. They have been drawn in the following manner:

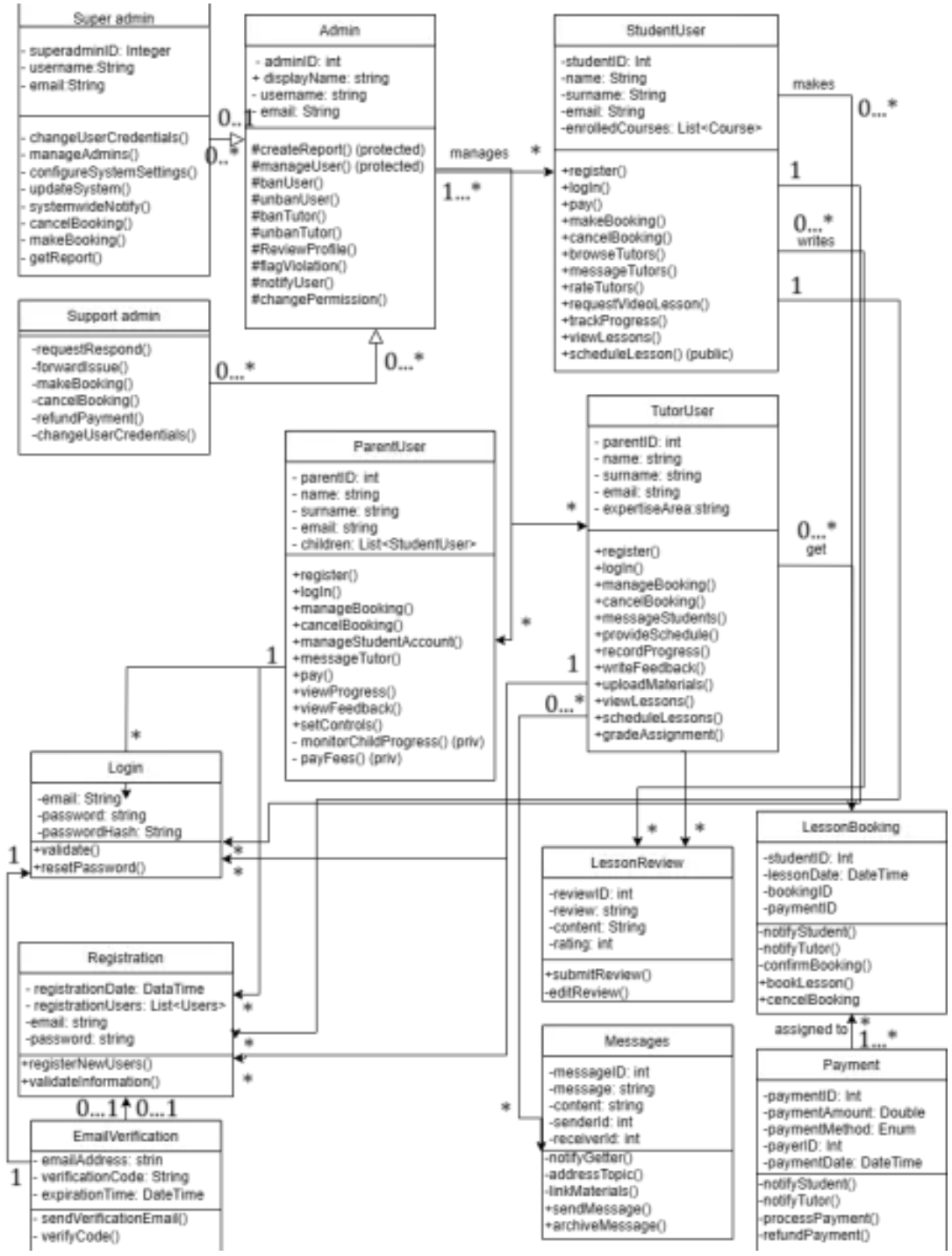

Figure 6: A Class Diagram representation

Above, the hierarchical structure of user roles has been presented. Each of the "blocks" has its attributes (variables), as well as operations (methods) that show the functionalities and privileges of each actor. Moreover, the individual symbols has been shown as follows:

- Access Levels (Class Visibility)
    - "+" → denotes **public** attributes or operations
    - "-" → denotes **private** attributes or operations
    - "#" → denotes **protected** attributes or operations

- Relationships between classes

In our diagrams the classes are involved in one or more relationships with other classes. The types indicated are **association** (denoted with a black arrow) and **inheritance** (denoted with a white arrow).

- Cardinality

The numbers (1, 0…*,1…*,0…1) on the diagram represent cardinality in the context of class relationships, indicating the multiplicity of instances that can be associated with another class. They interact with each other in a following manner:

    - **1** → represents a ***one-to-one*** relationship. It means that an instance of one class is associated with exactly one instance of another class. For example, if a *Login* class is connected to a *StudentUser* class, this would imply that each login instance is associated with exactly one student user.

    - **0…*** → represents a ***zero-to-many*** relationship. It indicates that an instance of one class can be associated with zero or more instances of another class. So, if a *TutorUser* class is connected to a *Messages* class, it would mean that a tutor might have zero or more possible message instances associated with them.

    - **1…*** → is similar to 0…* but indicates a ***one-to-many*** relationship. It means that at least one instance of a class is associated with multiple instances of another class. If a *Payment* class is connected to a *LessonBooking* class, it suggests that there must be at least one or more instances within attributes or operations associated with them.

    - **0..1** → indicates an ***optional*** relationship. An instance of one class may be associated with zero or one instance of another class. In other words, it is a way to denote that the existence of an instance on one end of the relationship is not obligatory. For example, if an *EmailVerification* class is connected to a *Registration class*, it would mean that an *EmailVerification* might not have an attribute or operation associated with it at all or could have only one of them.

5. UML State Diagrams

The UML State Diagram represents the various states and transitions that a specific user can go through in a system. Each Diagram has been presented for each actor present on our tutoring platform, and they are the following:

- Student

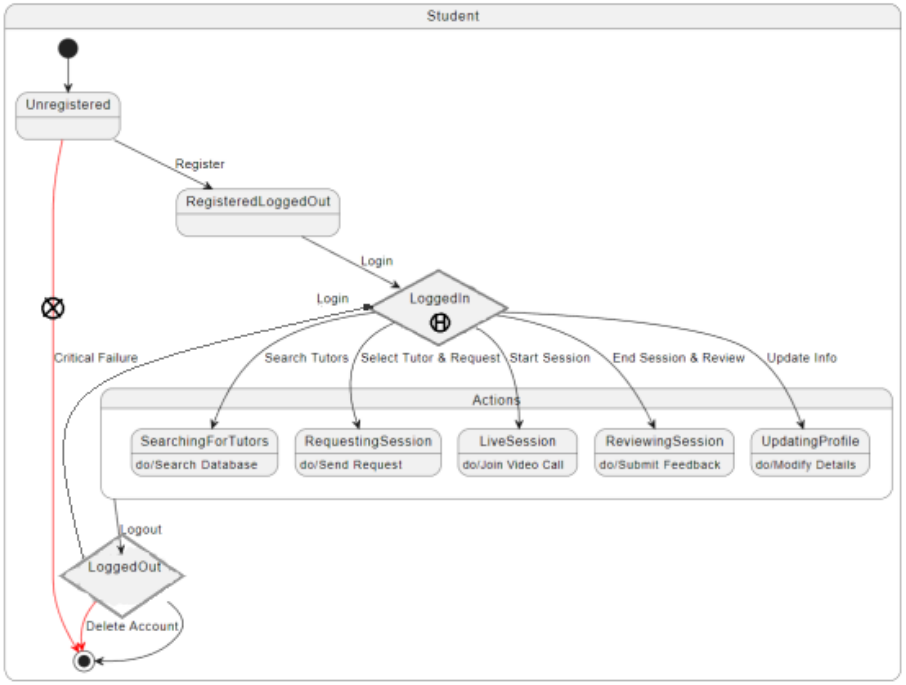

Figure 7: A State Diagram representation for Student

The arrows between states represent transitions, and the actions or activities that cause these transitions are labeled on the arrows. The rounded rectangles represent the states a student can be in, and the ovals at the beginning (*Unregistered*) and end (*Delete Account*) represent the start and end points of the student's journey in this system.

Description of states:

- **Unregistered** - This is the initial state where a student hasn't registered to the system.
    - From this state, the student can move to the *RegisteredLoggedOut* state by performing the action *Register*.
    - There's also a transition indicating *Critical Failure*, which seems to be an unexpected system error or exception state. It can happen if, for instance, a person can access some platform resources, however their account does not exist.
- **RegisteredLoggedOut** - Once a student has registered, they are in this state. In this state, the student is not logged in to the system.
    - They can *Login* to move to the *LoggedIn* state (more precisely, the decision node).

- **LoggedIn** - After logging in, the student enters this state. Several actions can be performed from this state:
    - **Search Tutors/Select Tutor & Request** - This leads to the *SearchingOfTutors* and *RequestingSession* state.
        - **SearchingOfTutors**: Within this action, the student searches the database for tutors.
        - **RequestingSession**: Here, the student sends a request for a tutoring session.
    - **Start Session** - Moves the student to the *LiveSession* state where they can join a video call.
    - **End Session & Review** - After ending the live session, they can move to the *ReviewingSession* state.
        - **ReviewingSession**: Here, the student can submit feedback about the session.
    - **Update Info**: Allows the student to update their profile information, which leads to the *UpdatingProfile* state.
        - **UpdatingProfile**: In this state, the student can modify their account details or information.
- **LoggedOut** - This decision node represents the scenario where a student has specifically logged out of their account, but their account remains in the system (they haven't deleted it).
    - From it, a student can choose to *Login*, which will transition them back to the *LoggedIn* decision node.
    - They can also *Delete Account*, which is an end state, meaning the student has removed their account from the system.

The instance of Shallow History is indicated on the diagram as well. We decided to place it inside the *LoggedIn* state, since if a student logs out and then logs back in, they would return to the most recent action they were taking (like *SearchingForTutors* or *UpdatingProfile*). Moreover, in the current structure there are no nested states within substates, therefore Deep History is not applicable.

- Tutor

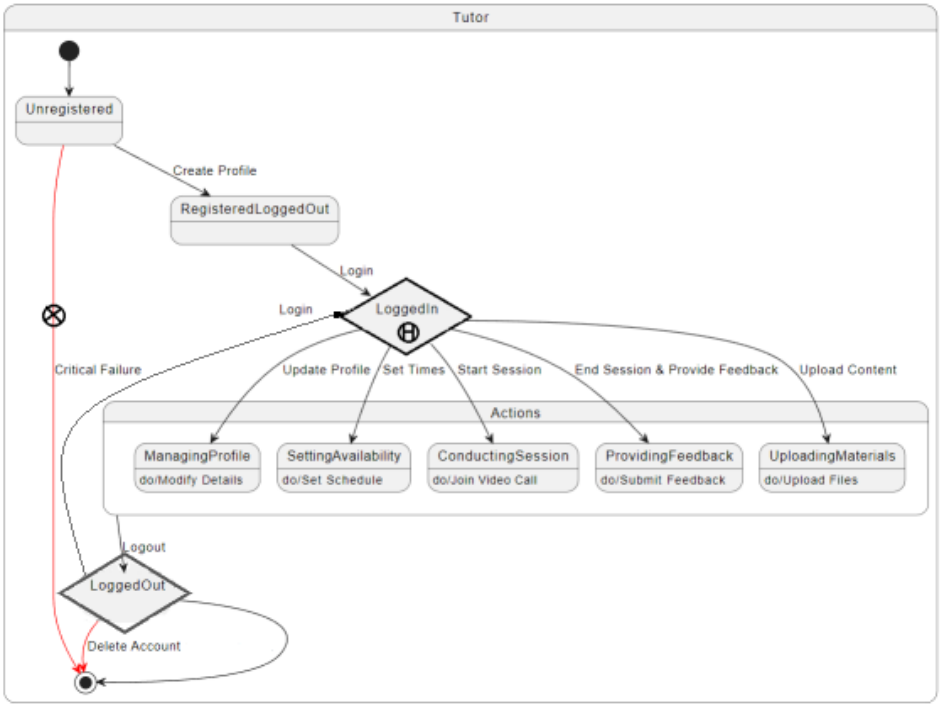

Figure 8: A State Diagram representation for Tutor

The arrows between states represent transitions, and the actions or activities that cause these transitions are labeled on the arrows. The rounded rectangles represent the states a tutor can be in, and the ovals at the beginning (*Unregistered*) and end (*Delete Account*) mark the start and end points of the tutor's journey in this system.

Description of states:

- **Unregistered** - This is the starting point where a tutor is not yet registered in the system.
    - From here, the tutor can transition via *Create Profile* to the *RegisteredLoggedOut* state.
    - The *Critical Failure* transition implies an unexpected error or critical condition in the system, likely leading to an abrupt ending of a session. For example, this could be invoked if a tutor's account is permanently banned or deleted due to policy violations or by their own choice.
- **RegisteredLoggedOut** - After creating a profile, the tutor finds themselves in this state, indicating they are registered but not yet logged in.
    - From this state, the tutor can perform a *Login* action, which leads them to the *LoggedIn* state (more precisely, the decision node).
- **LoggedIn** - Upon logging in, the tutor arrives in this state, where they have access to several actions:
    - **Update Profile** - Begins a transition to the *ManagingProfile* state.

        - **ManagingProfile**: Here, the tutor can modify their account details or information.
    - **Set Times** - Initiates a transition to the *SettingAvailability* state.
        - **SettingAvailability**: Here, the tutor defines their teaching availability.
    - **Start Session** - Directs the tutor to the *ConductingSession* state.
        - **ConductingSession**: Represents the actual tutoring session, with the tutor joining a video call.
    - **End Session & Provide Feedback** - After concluding the session, this leads the tutor to the *ProvidingFeedback* state.
        - **ProvidingFeedback**: In this phase, the tutor offers feedback about the concluded session.
    - **Upload Content** - Allows the tutor to move to the *UploadingMaterials* state.
        - **UploadingMaterials**: Within this state, the tutor uploads instructional materials or resources.
- **LoggedOut** - This decision node represents the scenario where a student has specifically logged out of their account, but their account remains in the system (they haven't deleted it).
    - From this node, a student can choose to *Login*, which will transition them back to the *LoggedIn* decision node.
    - They can also *Delete Account*, which is an end state, meaning the student has removed their account from the system.

As in the Student case, Shallow History has been placed inside the *LoggedIn* state. If a tutor logs out while updating their profile or setting availability, upon returning, they would be taken back to that specific action. Similarly, Deep History would only be relevant if there were nested states within the current elements.

- Parent/ Legal Guardian

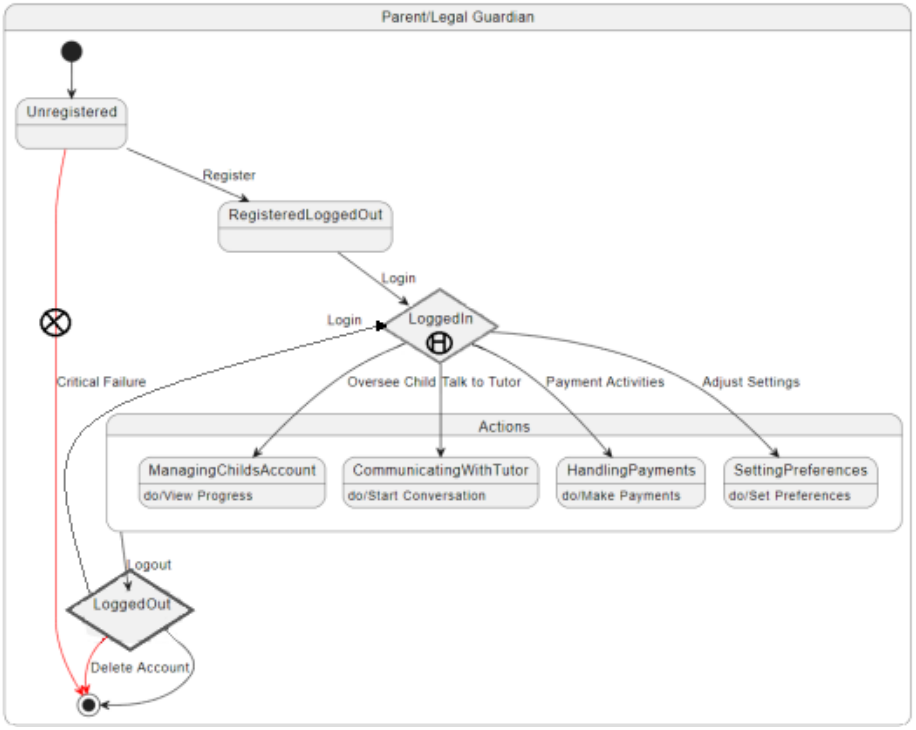

Figure 9: A State Diagram representation for Parent/ Legal Guardian

The arrows between states denote transitions, and the actions or activities prompting these transitions are labeled on the arrows. The rounded rectangles symbolize the states a parent or legal guardian can find themselves in, while the ovals at the beginning (*Unregistered*) and end (*Delete Account*) define the start and conclusion of the parent or legal guardian's journey in this system.

Description of states:

- **Unregistered** - This initial state is for parents or legal guardians who haven't yet registered in the system.
    - From this point, they can *Register*, leading them to the *RegisteredLoggedOut* state.
    - An additional transition, labeled as *Critical Failure*, signifies a potential unexpected situation or error within the system, likely causing the guardian to experience a disruption in their ongoing activities. This can be invoked if the parent decides to permanently remove their and their child's association from the platform and try to enter some functionalities after the completed activity.
- **RegisteredLoggedOut** - After registration, the parent or legal guardian enters this state, suggesting they're registered but not currently logged in.
    - From here, they can *Login*, which moves them to the *LoggedIn* state
- **LoggedIn** - Once logged in, the parent or legal guardian has multiple actions available:
    - **Oversee Child -** Begins a transition to the *ManagingChild'sAccount* state.

        - **ManagingChild'sAccount**: Represents activities related to overseeing the child's account progress and performance.
    - **Talk to Tutor** - Transitions to the *CommunicatingWithTutor* state.
        - **CommunicatingWithTutor**: Within this state, the parent can oversee the child's interactions with the tutor and possibly initiate or engage in conversations themselves.
    - **Payment Activities** - Takes the guardian to the *HandlingPayments* state.
        - **HandlingPayments**: Here, the parent or legal guardian can manage and make payments for the tutoring sessions.
    - **Adjust Settings** - Directs the user to the *SettingPreferences* state.
        - **SettingPreferences**: In this state, the parent can tweak or set their preferences regarding the tutoring system.
- **LoggedOut** - This decision node emerges after the parent has logged out of their account.
    - From this node, a student can choose to *Login*, which will transition them back to the *LoggedIn* decision node.
    - Moreover, the singular indicated action is *Delete Account*, leading to an end state. This suggests the parent or legal guardian has chosen to remove their account from the system entirely.

As in the Student and Tutor case, Shallow History has been placed inside the *LoggedIn* state to remember if they were, for example, last *ManagingChild'sAccount* or *HandlingPayments* activities. Same as above. Deep History is not as applicable given the current state structure.

- Administrator

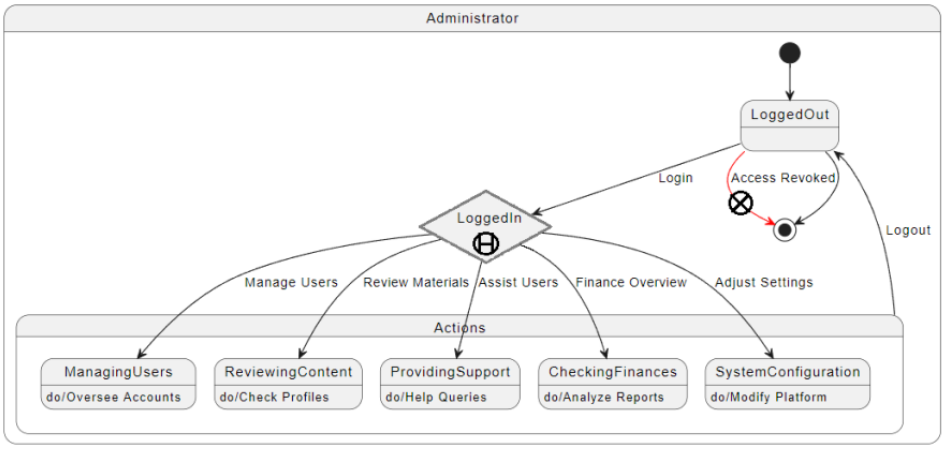

Figure 10: A State Diagram representation for Administrator

The transitions between states are represented by arrows, with each transition's corresponding action or activity labeled on it. Rounded rectangles illustrate the states an administrator can be in, while the oval at the end (*LoggedOut)* defines the potential concluding state of the administrator's engagement in this system.

Description of states:

- **LoggedOut -** This is the starting point for administrators who are not currently logged into the system.From here, they can choose to *Login*, which transitions them to the *LoggedIn* state.
- **LoggedIn** - Once an administrator logs in, they gain access to multiple functionalities:
    - **Manage Users** - This transitions them to the *ManagingUsers* state.
        - **ManagingUsers**: Here, administrators can oversee accounts, which may entail adding, removing, or modifying user profiles and roles.
    - **Review Materials** - Leads to the *ReviewingContent* state.
        - **ReviewingContent**: In this state, administrators can assess and potentially approve or deny content, check user profiles, or take other content-related actions.
    - **Assist Users** - Directs them to the *ProvidingSupport* state.
        - **ProvidingSupport**: Administrators can assist users by addressing their queries, offering technical help, or managing other support-related tasks.
    - **Finance Overview** - Takes them to the *CheckingFinances* state.
        - **CheckingFinances**: This state allows administrators to examine financial reports, oversee transactions, and manage other monetary aspects of the platform.
    - **Adjust Settings** - Navigates to the *SystemConfiguration* state.
        - **SystemConfiguration**: Administrators can modify platform settings, adjust functionalities, or implement system-wide changes.
- **LoggedOut**: This state emerges after an administrator logs out. Based on the diagram, no direct actions are indicated from this state, suggesting they would need to log in again to perform further operations.

From the *LoggedIn* state, an *Access Revoked* transition indicates an administrator's access can be permanently revoked, leading to a termination point. Additionally, the *Logout* action brings the administrator back to the *LoggedOut* state.

In this diagram, Shallow History was presented inside *LoggedIn* to recall the last activity the administrator was doing. As in the previous examples, nested tasks are not present, thus Deep History is not relevant for this specific case.

Differences and Similarities between diagrams:

All diagrams follow a similar pattern of transitioning from an unregistered or logged-out state to a logged-in state, performing various actions, and then logging out. The **Student** and

Tutor diagrams are more focused on the learning and teaching aspects, respectively, while the **Parent/Legal Guardian** diagram emphasizes overseeing and managing the child's education. The **Administrator** diagram, however, is unique in its focus on managing the platform, providing support, and overseeing financial aspects. Moreover, internal actions provide clarity on what happens during each state, making the processes more understandable.

As can be seen, diagrams collectively provide a comprehensive view of the different interactions and states for each actor in the tutoring platform.

6. UML Activity Diagrams

UML Activity diagrams are a middle point between flowcharts and Petri nets. They can be utilized to visualize execution paths within a single method, a flow of activity through the system or an algorithm.The Diagrams have been represented as follows:

- Student

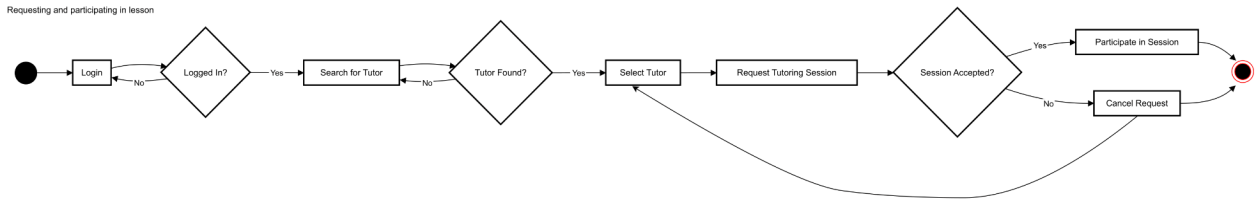

Figure 11: An Activity Diagram representation for Student

This Activity Diagram represents the process a student goes through when requesting and participating in a tutoring session. Throughout the diagram, we can see arrows indicating the flow of activities and decisions, ensuring clarity on the steps a student takes from the start to the end of the process.

Parameters for activities:

1. **Login**: Username, Password
2. **Search for Tutor**: Search criteria (e.g., subject, location, availability)
3. **Select Tutor**: Chosen tutor's details
4. **Request Tutoring Session**: Tutor selection, preferred date and time
5. **Participate in Session**: Active session details
6. **Cancel Session**: Session ID or reference

Detailed description:

1. **Login**: The process starts with the student attempting to login.
2. **Logged In?** After the login attempt, there's a decision point to determine if the login was successful.

    - If **Yes**, the student is logged in and can proceed.
    - If **No**, the student must try to login again.
3. **Search for Tutor**: Once logged in, the student searches for a tutor.
4. **Tutor Found?** After searching, there's another decision point to check if a suitable tutor was found.
    - If **Yes**, the student can proceed to select the tutor.
    - If **No**, the student goes back to the step of searching for another tutor.
5. **Select Tutor**: When a suitable tutor is found, the student selects that tutor.
6. **Request Tutoring Session**: After selecting a tutor, the student sends a request for a tutoring session.
7. **Session Accepted?** There's a decision point to check if the tutor has accepted the session request.
    - If **Yes**, the student can participate in the session.
    - If **No**, the student has the option to cancel the request and probably look for another tutor or reschedule.
8. **Participate in Session**: If the tutoring session request is accepted by the tutor, the student then participates in the session.
9. **Cancel Request**: If the session request is not accepted, the student has an option to cancel the request.

The process terminates when the student either completes the session (denoted by the circle with a thick border) or cancels the request.

- Tutor

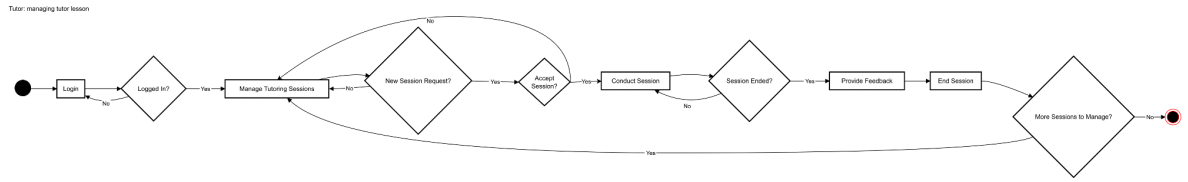

Figure 12: An Activity Diagram representation for Tutor

This Activity Diagram represents the process a tutor experiences when receiving and conducting a tutoring session. Throughout the diagram, there are arrows indicating the flow of activities and decisions, ensuring clarity on the steps a tutor takes from the start to the end of the process.

Parameters for activities:

- **Login**: Username, Password
- **Accept Session**: Session request details (e.g., student, date, time)
- **Conduct Session**: Active session details
- **Provide Feedback**: Student ID, feedback, performance data
- **Manage Tutoring Sessions**: Session list, session details

Detailed description:

- **Login:** The process starts with the tutor attempting to login.
- **Logged In?** After the login attempt, there's a decision point to determine if the login was successful.
    - If **Yes,** the tutor is logged in and can proceed.
    - If **No,** the tutor must try to login again.
- **Await Session Request:** Once logged in, the tutor awaits a session request from a student.
- **Session Request Received?** After waiting, there's a decision point to check if a session request has been received.
    - If **Yes,** the tutor can proceed to review the request details.
    - If **No,** the tutor continues to wait for a session request.
- **Review Session Request:** Upon receiving a session request, the tutor reviews the request details and the student's profile.
- **Accept Session?** After reviewing, the tutor decides whether to accept or decline the session request.
    - If **Accepted**, the tutoring session is scheduled to be conducted.
    - If **Declined**, the tutor manages a session, for example by providing a reason and/or notifying the student.
- **Conduct Session:** If the session request is accepted, the tutor then conducts the tutoring session at the scheduled time.
- **Session Feedback:** After the session, the tutor provides feedback about the student's performance and any follow-up activities.
- **Session Conclusion:** The tutor concludes the session, and there's an option to schedule another session or provide additional resources to the student.

The process concludes when the tutor either completes the session (denoted by the final state symbol) or declines a session request.

- Parent/Legal Guardian

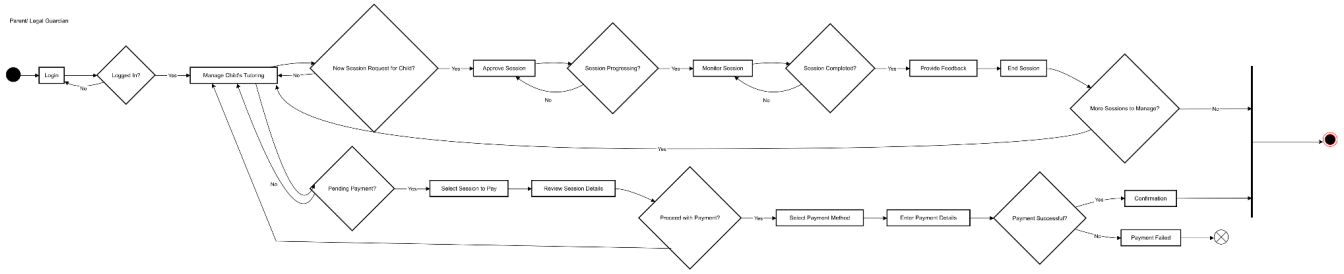

Figure 13: An Activity Diagram representation for Parent/Legal Guardian

The diagram offers a comprehensive view of how a parent or legal guardian can manage tutoring sessions for their child, approve sessions, monitor them, provide feedback, and make necessary payments. Throughout the diagram, there are arrows indicating the flow of activities and decisions, making clarity on the steps a parent takes from the start to the end of the indicated process.

Parameters for activities:

1. **Login**: Username, Password
2. **Approve Session**: Session request details for the child (e.g., tutor, date, time)
3. **Monitor Session**: Active session details, monitoring preferences
4. **Provide Feedback**: Feedback for the child's session

Detailed description:

- **Login**:
    - The process begins when the Parent or Legal Guardian tries to login.
    - If they are successfully logged in, they proceed. If not, the process starts again.
- **Manage Child's Tutoring**:
    - Once logged in, the parent can manage their child's tutoring. This seems to be the main hub of the activity diagram.
- **New Session Request for Child**:
    - The parent checks if there is a new session request for their child.
    - If yes, they move to the *Approve Session* activity.
    - If not, they can choose other activities to manage.
- **Approve Session**:
    - Here, the parent can approve the tutoring session.
    - If the session is progressing, they can monitor the session. If not, they revert to managing other sessions or activities.
- **Monitor Session**:
    - In this stage, the parent can actively monitor the ongoing tutoring session.
    - After monitoring, they check if the session has been completed.
- **Session Completed**:
    - If the session is completed, the parent provides feedback on the session. After giving feedback, they proceed to *End Session*.
    - If the session is not completed, they revert to monitoring the session.
- **End Session**:
    - Once the session is ended, the parent checks if there are more sessions to manage.
    - If there are more sessions, the parent can go back to the *Manage Child's Tutoring* activity.
    - If no more sessions are to be managed, the process ends.
- **Pending Payment**:
    - The parent checks if there's any pending payment for the sessions.

    - If yes, they proceed to *Select Session to Pay*. If not, they move to other activities.
- **Select Session to Pay**:
    - Here, the parent chooses which session they want to make a payment for and then reviews the session details.
- **Review Session Details**:
    - After reviewing, the parent decides whether to proceed with the payment.
    - If they decide to proceed, they select a payment method. If not, they can choose other activities.
- **Select Payment Method**:
    - The parent chooses a payment method and then enters the payment details.
- **Enter Payment Details**:
    - After entering the details, the system checks if the payment is successful.
    - If successful, a confirmation is displayed to the parent. If not, a *Payment Failed* activity is triggered.
- **End of Process**:
    - If the parent has no more sessions to manage or activities to perform, they exit the system (here, indicated by the Synchronization node - merging multiple parallel transitions into one transition).

- Administrator

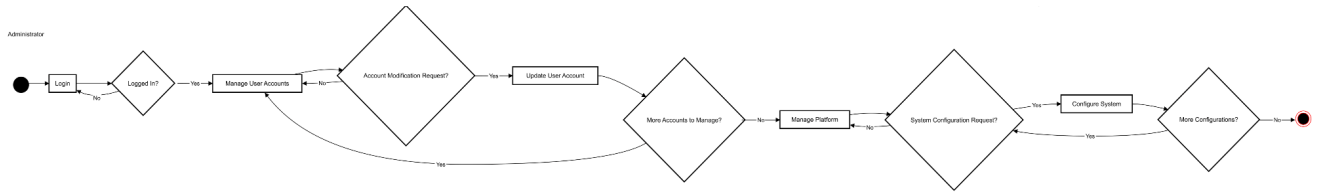

Figure 14: An Activity Diagram representation for Administrator

This Activity Diagram illustrates the process an administrator undergoes to manage user accounts and system configurations on the tutoring platform. The diagram is interconnected with arrows demonstrating the sequence of activities and decisions, offering a clear understanding of the tasks an administrator undertakes from beginning to end.

Parameters for activities:

1. **Login**: Administrator credentials
2. **Update User Account**: User account details, modifications
3. **Manage User Accounts**: User account list, user account details
4. **Configure System**: System settings, configuration details
5. **Manage Platform**: Platform settings, options

Detailed description:

1. **Login:** The procedure commences with the administrator attempting to login.
2. **Logged In?** Following the login attempt, there's a juncture to verify if the login was successful.
    - If **Yes,** the administrator successfully gains access and can move forward.
    - If **No,** the administrator must retry the login process.
3. **Manage User Accounts:** After successful login, the administrator can opt to manage user accounts on the platform.
4. **Account Modification Request?** There's a decision node to determine if there's any pending account modification request.
    - If **Yes,** the administrator proceeds to update the user account.
    - If **No,** the administrator moves to manage other aspects of the platform.
5. **Update User Account:** In the event of a pending account modification request, the administrator carries out the necessary changes to the user account.
6. **More Accounts to Manage?** After updating an account, the administrator checks if there are more accounts that need attention.
    - If **Yes,** the administrator returns to the *Manage User Accounts* step.
    - If **No,** they move on to manage the platform's configurations.
7. **Manage Platform:** The administrator has the option to manage other aspects of the platform or delve into system configurations.
8. **System Configuration Request?** The system checks if there's a need for configuration adjustments.
    - If **Yes,** the administrator initiates the configuration process.
    - If **No,** the administrator can finish their tasks.
9. **Configure System:** If there's a system configuration request, the administrator tweaks the system settings accordingly.
10. **More Configurations?** Post-configuration, there's a check to ascertain if further configurations are needed.

- If **Yes,** the administrator returns to the *Configure System* step.
- If **No,** they conclude their tasks for the session.

The process terminates when the administrator either ends their session after managing user accounts or after adjusting system configurations, indicated by the final state symbol.

7. Sequence Diagrams

UML sequence diagrams are essential tools that illustrate the flow of messages between actors (both humans and systems such as servers or software) in an interaction. They detail the sequence in which these messages are exchanged, clarifying the timing of message reception and ensuring that certain messages, like a message A, do not precede a message C. This establishes a clear order of communication within the interaction. Our Sequence Diagrams present in the following manner:

User Registration and Login

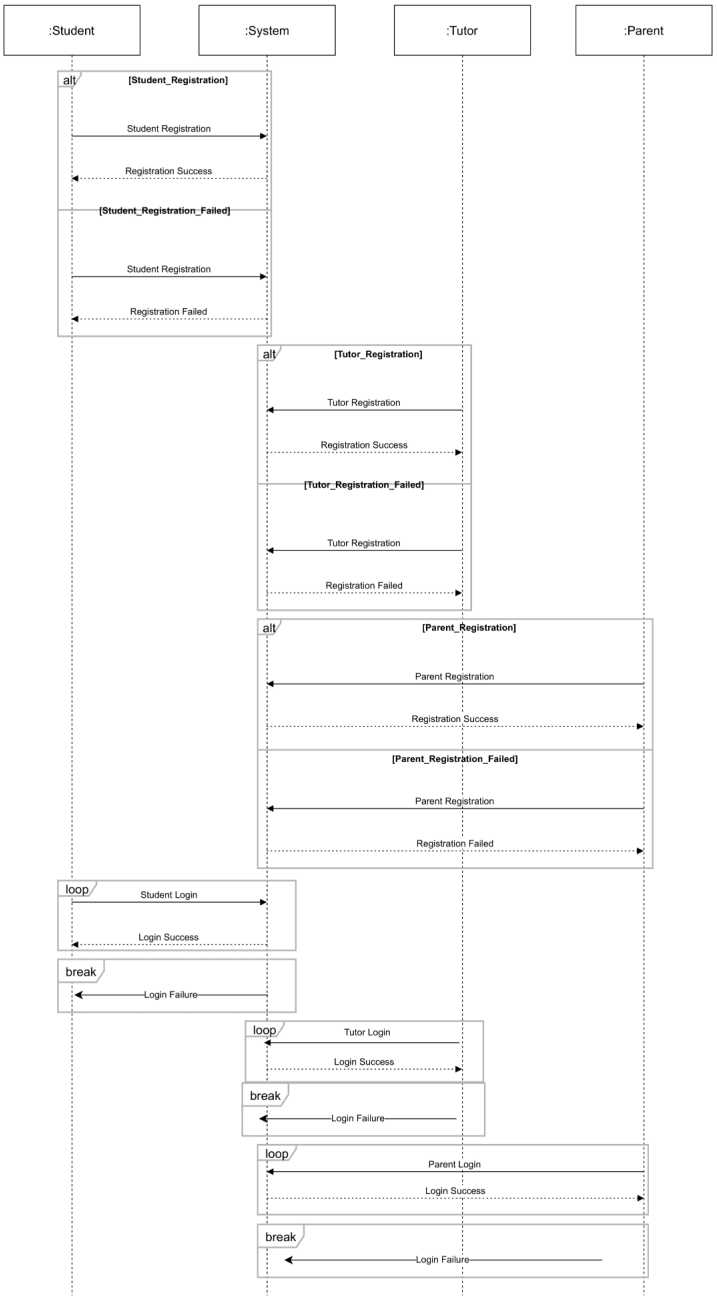

Figure 15: A Sequence Diagram representation for registration and logging in

This sequence diagram outlines the interactions between different user roles—students, tutors, and parents—and a system during the registration and login processes. The key actions depicted include registration initiation, system processing, and both successful and unsuccessful login acknowledgments.

Student Process:

- The student initiates the registration process by communicating with the system.
- The system responds with a *Registration Success* message, indicating the successful completion of the student's registration.
- Subsequently, the student logs into the system, and the system confirms a successful login.
- In case the registration process fails, the student immediately receives an appropriate message on the platform.

Tutor Process:

- Similar to the student process, the tutor undergoes both registration and login procedures.
- The tutor initiates registration, and upon successful completion, the system responds with a *Registration Success* message.
- The tutor then logs into the system, receiving a confirmation of a successful login.
- In case of a failed registration process, the tutor immediately receives an appropriate message on the platform.

Parent Process:

- The parent engages in both registration and login processes.
- The system acknowledges the success of parent registration with a *Registration Success* message.
- The parent proceeds to log into the system, receiving a confirmation of a successful login.
- When the registration process goes unsuccessful, the parent immediately receives an appropriate message on the platform.

Additionally, the *loop* block in the diagram is used to represent the login attempts for each actor. This suggests that the actors can attempt to log in multiple times. The *break*, however, indicates the point at which the loop terminates, which, in this context, would be a login failure - no further attempts can be made at that moment.

Homework Assignment Lifecycle

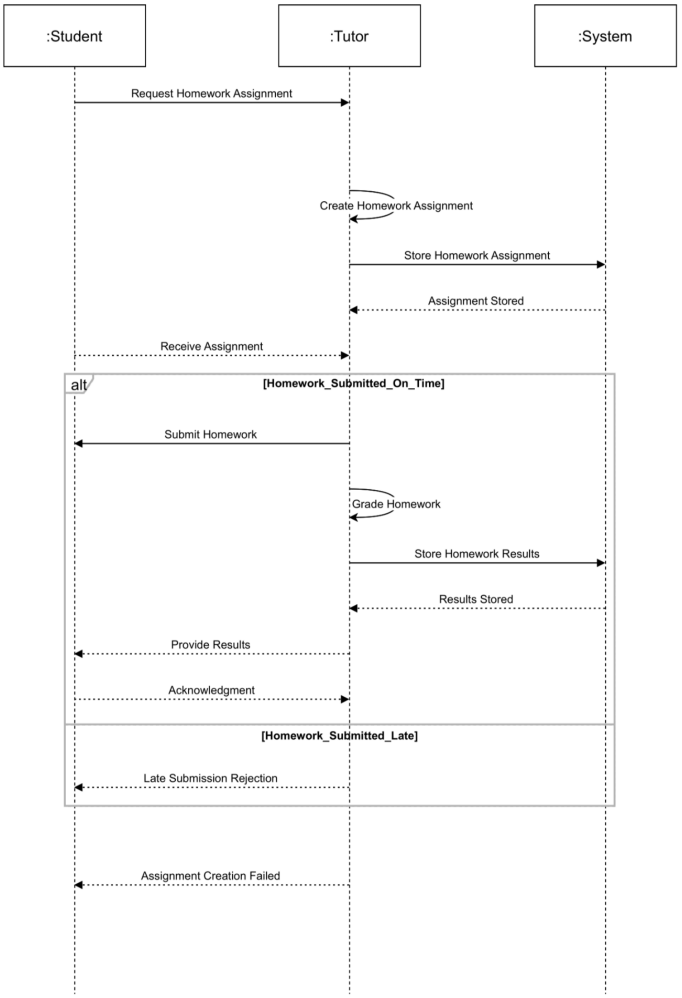

Figure 16: A Sequence Diagram representation for homework assignment

This sequence diagram shows the process of managing a homework assignment between a student, a tutor, and the underlying system. The sequence unfolds with the student initiating a request for a homework assignment from the tutor, leading to a structured series of actions and communications:

Student Request and Tutor Assignment Creation:

- The student triggers the sequence by requesting a homework assignment from the tutor.
- In response, the tutor generates a new homework assignment.

Tutor Storage in the System:

- The tutor, having crafted the assignment, securely stores it within the system.
- The system promptly acknowledges the successful storage of the assignment.

Student Submission (on time) and Tutor Grading:

- With the assignment stored, the system notifies the tutor of the successful storage.
- The student, having received the assignment, submits the completed homework back to the tutor.
- The tutor, in turn, grades the submitted homework.

Results Storage in the System:

- The tutor securely stores the graded results within the system.
- The system confirms the successful storage of the homework results.

Tutor Communicates Results to Student:

- The tutor shares the graded results with the student.

Student Acknowledgment:

- The student acknowledges the received results, confirming the completion of the assignment lifecycle.

Student Submission (after deadline):

- The system rejects late submissions of the homework, as indicated by the appropriate notification directed towards the student.

Failure of Assignment Creation:

- When the creation of the assignment fails, the sequence diagram indicates a suitable message, which would be communicated to the tutor involved in the assignment creation process.

Tutoring Session Request and Confirmation

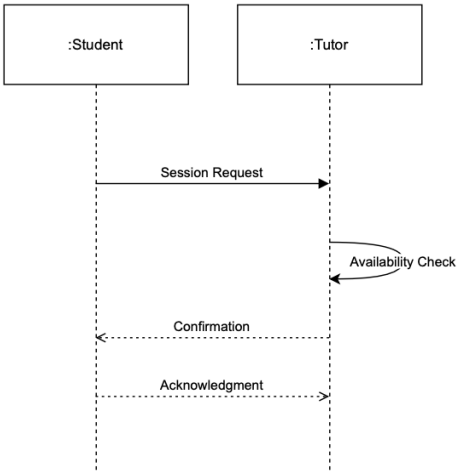

Figure 17: A Sequence Diagram representation for student lesson request

This sequence diagram illustrates the interactions between a student and a tutor during the initiation and confirmation of a tutoring session. The sequence unfolds seamlessly, capturing the essential steps involved in the scheduling process:

Student Initiates Session Request:

- The sequence commences as the student sends a tutoring session request to the tutor, expressing the intent to engage in a learning session.

Tutor Availability Check:

- Upon receiving the request, the tutor conducts an availability check to assess whether they can accommodate the proposed session within their schedule.

Tutor Confirmation to Student:

- Following the availability check, the tutor communicates back to the student with a confirmation, indicating their readiness and ability to conduct the tutoring session.

Student Acknowledgment:

- The student acknowledges the tutor's confirmation, affirming their understanding and acceptance of the scheduled session.

Payment and Receipt Generation

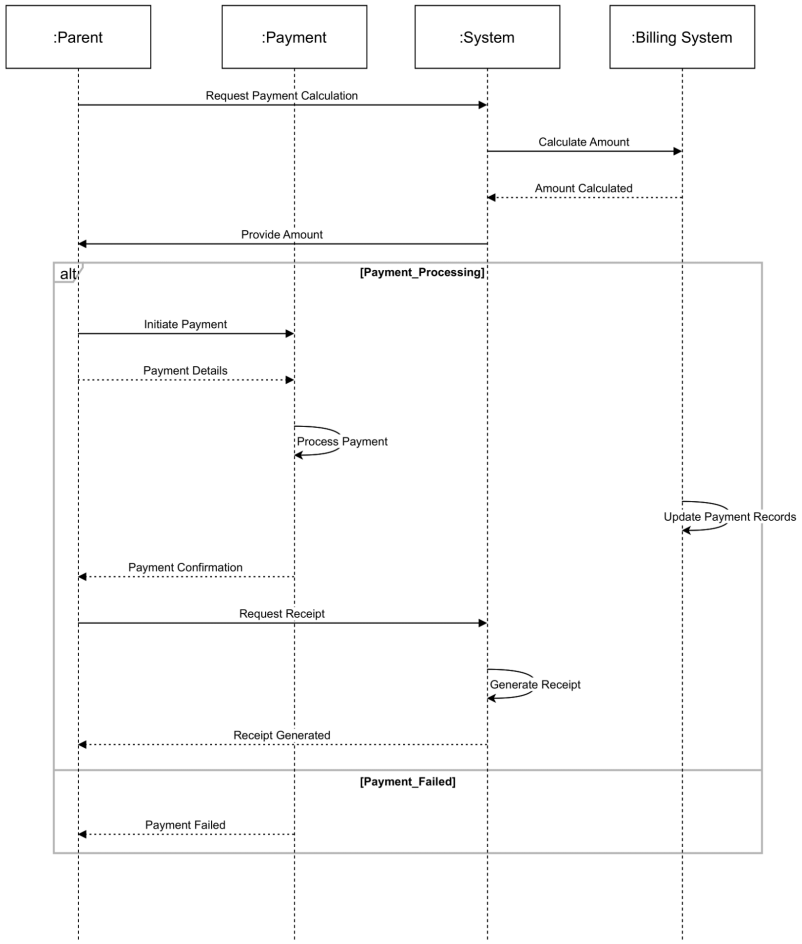

Figure 18: A Sequence Diagram representation for payment processing

This sequence diagram outlines the process involving a parent, the payment system, and the underlying billing and payment infrastructure within a larger system. The sequence captures the interactions related to payment calculation, initiation, processing, and the subsequent generation of a receipt:

Payment Calculation Request:

- The sequence begins as the parent initiates a request to the system for the calculation of a payment amount.

Amount Calculation by Billing System:

- The system delegates the task of calculating the payment amount to the billing system.
- The billing system successfully computes the amount and communicates it back to the system.

Amount Provision to Parent:

- The system, having received the calculated amount, provides this information to the parent, ensuring transparency in the payment process.

Payment Initiation:

- The parent, armed with the payment details, initiates the payment process by engaging with the payment system.

Payment Processing and Record Update:

- The payment system processes the transaction, utilizing the provided payment details.
- Simultaneously, the billing system updates payment records to maintain an accurate financial history.

Payment Confirmation to Parent:

- The payment system confirms the successful processing of the payment to the parent, offering reassurance regarding the financial transaction.

Receipt Generation Request:

- Subsequent to the payment confirmation, the parent requests the system to generate a receipt for the completed transaction.

Receipt Generation:

- The system generates a comprehensive receipt, summarizing the transaction details.
- The generated receipt is promptly provided to the parent, ensuring documentation and clarity.

Payment Failure:

- The system generates the information that the transaction did not go through.

Admin User Management

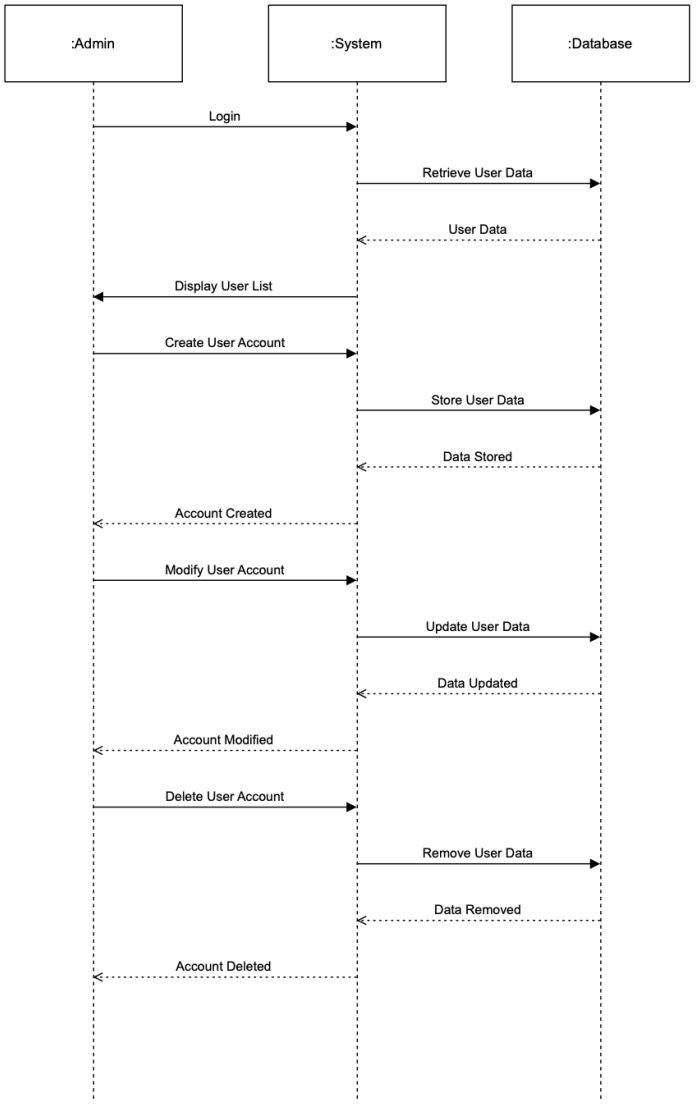

Figure 19: A Sequence Diagram representation for general system management

This sequence diagram illustrates the process of admin user management within a system, involving interactions between the administrator (Admin), the system, and the underlying database.

Login and User List Display:

- The admin initiates the sequence by logging into the system.

- The system, upon receiving the login request, retrieves user data from the database.
- The database responds by providing the user data, allowing the system to display the user list to the admin.

User Account Creation:

- The admin requests the creation of a new user account.
- The system, in response, stores the provided user data in the database.
- The database confirms the successful storage of data, and the system communicates the account creation status back to the admin.

User Account Modification:

- The admin, having logged in, initiates the modification of an existing user account.
- The system updates the user data within the database.
- The database confirms the successful update, and the system notifies the admin of the completed account modification.

User Account Deletion:

- The admin, logged into the system, requests the removal of a user account.
- The system communicates with the database to remove the corresponding user data.
- The database acknowledges the successful removal, and the system informs the admin that the account has been deleted.

Report Generation and Distribution

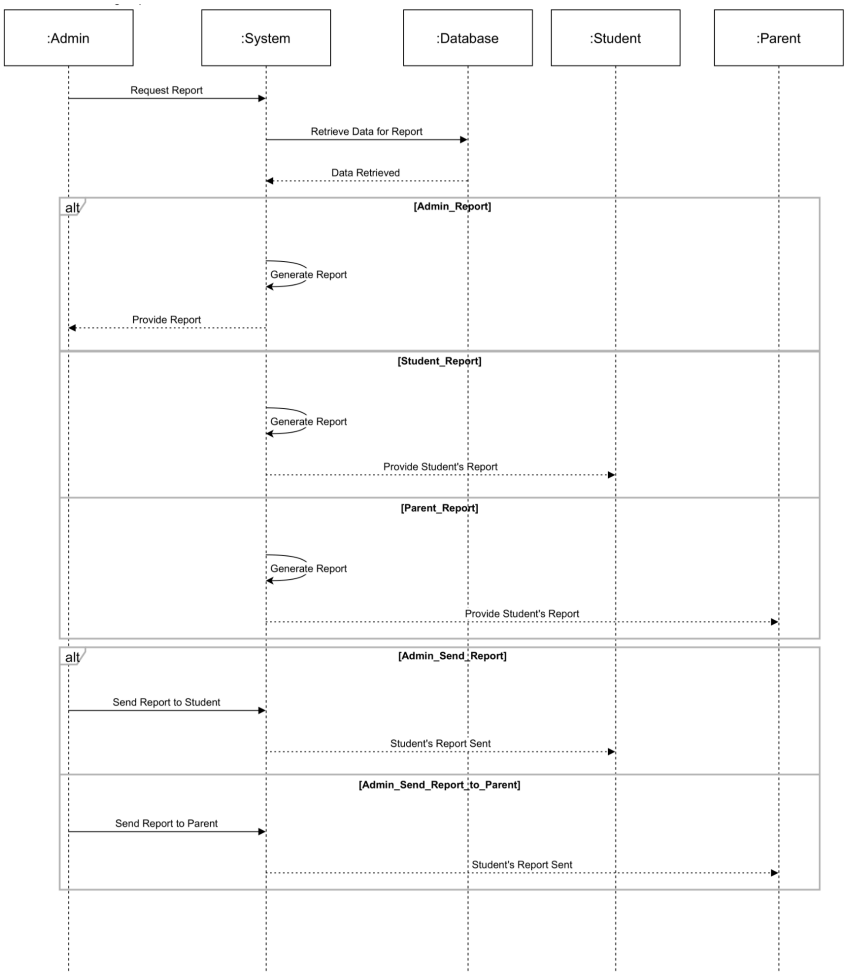

Figure 18: A Sequence Diagram representation for generating reports

This sequence diagram illustrates the process of report generation and distribution within an educational system involving an administrator (Admin), students, parents, the system, and a database:

Admin Requests and Receives Report:

- Admin initiates a request for a report.
- System retrieves relevant data from the database.
- System generates the report and provides it to the admin.

Student Requests and Receives Report:

- Student requests a report from the system.
- System retrieves the student's report data from the database.
- System generates the report and provides it to the student.

Parent Requests and Receives Student's Report:

- Parent requests the student's report from the system.
- System retrieves the student's report data from the database.
- System generates the student's report and provides it to the parent.

Admin Sends Student's Report to Student:

- Admin initiates sending the student's report through the system.
- System sends the student's report to the student.

Admin Sends Student's Report to Parent:

- Admin initiates sending the student's report to the parent.
- System sends the student's report to the parent.

8. API

Each user module (Student, Tutor, Parent, Administrator) is deployed as a web, desktop or mobile application. The application fetches and sends data to the server using HTTP requests (GET, POST, PATCH, DELETE). The proposed technical for the platform are as follows:

- User modules

Programming Language:	C#
Web Application Framework:	ASP.NET Core
Web Application Hosting :	Azure App Service
Desktop Application Framework:	WPF
Mobile Application Framework:	Xamarin

Table 5: Technological solutions for user modules.

The server module, however, is responsible for responding to user requests received through the REST API, managing the database, as well as automatically sending email messages (notifications) to users.

- Server (database) module

Programming Language:	C#
Application Framework:	ASP.NET Core
Web Hosting	Azure App Service
Database Platform	Microsoft SQL Server
ORM Framework	Entity Framework Core

Authorization Service	Azure Active Directory
E-mailing Platform	GetResponse

Table 6: Technological solutions for a server module.

In brief, to simulate an application's operation from the user's side (client, restaurant, administrator modules), it is sufficient to prepare a desktop application (with a GUI) that supports the sending of HTTP requests. The server-side (module base) should contain a hosted web application with exposed API endpoints, connected to a database. To simplify the simulated module base, the database can be replaced with a class aggregating objects corresponding to entries in the database. For simplicity, it is also permissible to exclude sending email messages. The module base decides on the user's right to use the endpoint based on the API key, as well as on the possible result of the request. Below, we may observe how it presents in practice.

API Documentation of a module was prepared with the use of *RAML* specification:

```
#%RAML 1.0

title: Tutoring Platform “Study Buddy”

version: 1.0.0

baseUri: https://api.studybuddy.com/v1

description: API for managing interactions between students, tutors, parents, and
administrators on a tutoring platform.
```

- Security

```
securitySchemes:

  BearerAuth:

    description: JWT Authorization header using the Bearer scheme. Example: "Authorization:
Bearer {token}"

    type: OAuth 2.0

    describedBy:

      headers:

        Authorization:

          description: JWT Authorization header using the Bearer scheme

          type: string

    settings:
```

```
    authorizationUri: # Need to add the authorization URI
    accessTokenUri: # Need to add the access token URI
    authorizationGrants: [ "client_credentials" ]
```

- Object Types
    - User

```
User:
  type: object
  properties:
    id:
      type: integer
      format: int64
      example: 1001
      description: Unique identifier for the user.
    username:
      type: string
      example: "username"
      description: Username
    email:
      type: string
      format: email
      example: "name.surname@example.com"
      description: E-mail address
    role:
      type: string
      enum: [admin, tutor, student, parent]
      description: Role which we can choose on the platform.
      required:
    - id
    - username
    - email
```

```
      - role
```

- Student

```
Student:
    type: object
    properties:
      name:
        type: string
        example: "Eve"
        description: Name
      surname:
        type: string
        example: "Smith"
        description: Surname
      email:
        type: string
        format: email
        example: "name.surname@example.com"
        description: E-mail address
      password:
        type: string
      grade:
        type: string
        description: Current grade of the student
    required:
      - name
      - surname
      - email
      - password
      - grade
```

- Tutor

```
Tutor:
       type: object
```

```
properties:

name:

type: string

example: "Martha"

description: Name

surname:

type: string

example: "May"

desciption: Surname

email:

type: string

format: email

example: "name.surname@example.com"

description: E-mail address

password:

type: string

subjects:

type: array

items:

type: string

description: Subjects the tutor is qualified to teach

qualifications:

type: string

description: A short qualification description of the tutor

required:

- name

- surname

- email

- password

- subjects

- qualifications
```

- Parent

```
Parent:
        type: object
        properties:
        name:
        type: string
        example: "Jacob"
        description: Name
        surname:
        type: string
        example: "Smith"
        description: Surname
        email:
        type: string
        format: email
        example: "name.surname@example.com"
        description: E-mail address
        password:
        type: string
        required:
        - name
        - surname
        - email
        - password
```

- Administrator

```
Administrator:
        type: object
        properties:
        username:
        type: string
        example: "example_username"
        description: Username
        email:
```

```
        type: string
        format: email
        example: "username@example.com"
        description: E-mail address
        password:
        type: string
        required:
        - username
        - email
        - password
```

- Session

```
Session:
        type: object
        properties:
        sessionId:
        type: string
        format: uuid
        description: Unique identifier for a session.
        tutorId:
        type: UserId
        studentId:
        type: UserId
        topic:
        type: string
        scheduledTime:
        type: datetime
        duration:
        type: integer
        status:
        type: string
        enum: [scheduled, completed, cancelled]
        required:
```

```
    - sessionId
    - tutorId
    - studentId
    - topic
    - scheduledTime
    - duration
    - status
```

- Assignment

```
Assignment:
    type: object
    properties:
    AssignmentId:
    type: string
    format: uuid
    description: Unique identifier for an assignment
    title:
    type: string
    example: Assignment no. X
    comment:
    type: string
    dueDate:
    type: datetime
    submitted:
    type: boolean
    grade:
    type: string
    description: Assuming grade is a part of the assignment object
    required:
    - AsignmentId
    - title
    - comment
    - dueDate
```

```
        - submitted

        - grade
```

- Report

```
        Report:

        type: object

        properties:

        reportId:

        type: string

        format: uuid

        userId:

        type: UserId

        content:

        type: string

        required:

        - reportId

        - userId

        - content
```

- Receipt

```
Receipt:

        type: object

        properties:

        receiptId:

        type: string

        format: uuid

        paymentDetails:

        type: string

        dateIssued:

        type: datetime

        required:

        - receiptId

        - paymentDetails

        - dateIssued
```

- PaymentCalculation

```
PaymentCalculation:

       type: object

       properties:

       sessionId:

       type: string

       format: uuid

       duration:

       type: integer

       format: int32

       description: Duration in minutes.

       hourlyRate:

       type: number

       format: float

       description: Hourly rate for the session.

       required:

       - sessionId

       - duration

       - hourlyRate
```

- CalculatedAmount

```
CalculatedAmount:

       type: object

       properties:

       calculatedAmount:

       type: number

       format: float

       required:

       - calculated Amount
```

- Error

```
Error:
```

```
      type: object
      properties:
      code:
      type: integer
      format: int32
      message:
      type: string
      required:
      - code
      - message
```

- API Endpoints

{GET}: Retrieves information from the server (*Read*).

{POST}: Sends data to the server to create a new resource (*Create*).

{PUT}: Updates existing information or creates it if it does not exist (*Update*).

{DELETE}: Removes existing information (*Delete*).

Each endpoint may have different responses based on the HTTP status codes, indicating success (e.g., 200 OK, 201 Created), client errors (e.g., 400 Bad Request, 403 Forbidden, 404 Not Found), or server errors (e.g. 500 Internal Server Error). The RAML also specifies that some endpoints are protected by the *BearerAuth* security scheme, meaning they require a valid authorization token to access.

- Universal Paths
  - */public-info*

```
/public-info:
  get:
      description: Retrieves publicly accessible information.
```

  - */auth/login*

```
/auth/login:
  post:
      description: Authenticate a user and retrieve a token.
```

```
        body:
        application/x-www-form-urlencoded:
        type: object
        properties:
        username: string
        password: string
        required: [username, password]
        responses:
        200:
        body:
        application/json:
                type: object
                properties:
                token: string
                user: User
        401:
        body:
        application/json:
                type: Error
        description: Unauthorized.
```

- Student Paths
  - */students*

```
/students:
  post:
        description: Register a new student account.
        body:
        application/json:
        type: object
        properties:
        username: string
```

```
      email: string
      password: string
      required: [username, email, password]
      responses:
      201:
      description: Student successfully registered.
      400:
      body:
      application/json:
            type: Error
      description: Invalid input.
  get:
      description: Retrieve a list of all student accounts (admin only).
      responses:
      200:
      body:
      application/json:
            type: array
            items: User
      403:
      body:
      application/json:
            type: Error
      description: Forbidden. Only admins can access this endpoint.
```

- */students/{studentId}*

```
/students/{studentId}:
  uriParameters:
      studentId:
      type: UserId
```

```
get:
      description: Get a specific student's details.
      responses:
      200:
      body:
      application/json:
             type: User
      404:
      body:
      application/json:
             type: Error
      description: Student not found.


put:
      description: Update a student's details.
      body:
      application/json:
      type: object
      properties:
      username: string
      email: string
      required: [username, email]
      responses:
      200:
      description: Student updated successfully.
      400:
      body:
      application/json:
             type: Error
      description: Invalid input.
```

```
  delete:
        description: Delete a student's account.
        responses:
        200:
        description: Student deleted successfully.
        404:
        body:
        application/json:
               type: Error
        description: Student not found.
```

- */students/{studentId}/assignments*

```
/students/{studentId}/assignments:
  get:
        description: Request homework assignment and submit it.
        securedBy: [BearerAuth]
        responses:
        200:
        body:
        application/json:
               type: array
               items: Assignment
```

- */students/{studentId}/sessions*

```
/students/{studentId}/sessions:
  post:
        description: Schedule a tutoring session.
        body:
        application/json:
        type: Session
        responses:
```

```
      201:
      description: Session successfully scheduled.
404:
  body:
      application/json:
            type: Error
      description: URL not found. Session scheduling unsuccessful.
      securedBy: [BearerAuth]
```

- *`/students/{studentId}/reports`*

```
/students/{studentId}/reports:
  get:
      description: Retrieve a student's report.
      responses:
      200:
      body:
      application/json:
            type: array
            items: Report
      403:
      description: Forbidden. Only the student can access this report. # Here, the
concept that Student and Parent receive separate report from Tutor
      404:
      body:
      application/json:
            type: Error
      description: Student or report not found.
      securedBy: [BearerAuth]
```

- Tutor Paths
  - *`/tutors`*

```
/tutors:
  post:
```

```
      description: Register a new tutor account.
      body:
      application/json:
      type: object
      properties:
      username: string
      email: string
      password: string
      required: [username, email, password]
      responses:
      201:
      description: Tutor successfully registered.
      400:
      description: Invalid input.
  get:
      description: Retrieve a list of all tutor accounts (admin only).
      responses:
      200:
      body:
      application/json:
             type: array
             items: User
      403:
      body:
      application/json:
             type: Error
      description: Forbidden. Only admins can access this endpoint.
```

- */tutors/{tutorId}*

```
/tutors/{tutorId}:
  uriParameters:
```

```
      tutorId: UserId
get:
      description: Get a specific tutor's details.
      responses:
      200:
      description: Tutor details.
      404:
      description: Tutor not found.
put:
      description: Update a tutor's details.
      body:
      application/json:
      type: object
      properties:
      username: string
      email: string
      required: [username, email]
      responses:
      200:
      description: Tutor updated successfully.
      400:
      description: Invalid input.
delete:
      description: Delete a tutor's account.
      responses:
      200:
      description: Tutor deleted successfully.
      404:
      description: Tutor not found.
```

- */tutors/{tutorId}/sessions*

```
/tutors/{tutorId}/sessions:

  uriParameters:

        tutorId: UserId

  post:

        description: Schedule a new tutoring session based on the availability
(availability check).

        body:

        application/json:

        type: Session

        responses:

        201:

        description: Session successfully scheduled.

        400:

        description: Invalid input.

  get:

        description: Retrieve a list of all sessions (admin and tutor specific).

        responses:

        200:

        body:

        application/json:

                type: array

                items: Session

        403:

        description: Forbidden. Only admins and specific tutors can access this endpoint.
```

- */tutors/{tutorId}/sessions/{sessionId}*

```
/tutors/{tutorId}/sessions/{sessionId}:

  uriParameters:

        tutorId:

        type: UserId

        description: The unique identifier for a tutor.

        sessionId:
```

```
      type: SessionId
      description: The unique identifier for a session.
get:
      description: Get details of a specific session.
      responses:
      200:
      body:
      application/json:
             type: Session
      description: Details of the session retrieved successfully.
      404:
      body:
      application/json:
             type: Error
      description: Tutor or session not found.
put:
      description: Update details of a specific session.
      body:
      application/json:
      type: Session
      responses:
      200:
      description: Session details updated successfully.
      body:
      application/json:
             type: Session
      400:
      description: Invalid input provided.
      body:
      application/json:
```

```
            type: Error
      404:
      description: Tutor or session not found.
      body:
      application/json:
            type: Error
  delete:
      description: Delete a specific session.
      responses:
      200:
      description: Session deleted successfully.
      404:
      description: Tutor or session not found.
      body:
      application/json:
            type: Error
```

- */tutors/{tutorId}/assignments*

```
/tutors/{tutorId}/assignments:
  uriParameters:
      tutorId:
      type: UserId
      description: The unique identifier of the tutor.
  post:
      description: Create a new assignment and have the ability to grade it.
      body:
      application/json:
      type: Assignment
      responses:
      201:
      description: Assignment created successfully.
      body:
```

```
      application/json:
            type: Assignment
      400:
      description: Invalid input.
      body:
      application/json:
            type: Error
  get:
      description: Retrieve a list of all assignments (tutor and student specific).
      responses:
      200:
      description: List of assignments retrieved successfully.
      body:
      application/json:
            type: array
            items: Assignment
      403:
      description: Forbidden. Only specific tutors and students can access this endpoint.
      body:
      application/json:
            type: Error
```

- */tutors/{tutorId}/assignments/{assignmentId}*

```
/tutors/{tutorId}/assignments/{assignmentId}:
  uriParameters:
      tutorId:
      type: UserId
      description: The unique identifier of the tutor.
      assignmentId:
      type: AssignmentId
      description: The unique identifier of the assignment.
```

```
get:
      description: Get details of a specific assignment.
      responses:
      200:
      body:
      application/json:
             type: Assignment
      description: Details of the assignment retrieved successfully.
      404:
      body:
      application/json:
             type: Error
      description: Assignment not found.
put:
      description: Update an assignment (tutor specific).
      body:
      application/json:
      type: Assignment
      responses:
      200:
      description: Assignment updated successfully.
      body:
      application/json:
             type: Assignment
      400:
      body:
      application/json:
             type: Error
      description: Invalid input provided.
      404:
```

```
      body:
      application/json:
            type: Error
      description: Assignment not found.
  delete:
      description: Delete an assignment (tutor specific).
      responses:
      200:
      description: Assignment deleted successfully.
      404:
      body:
      application/json:
            type: Error
      description: Assignment not found.
```

- Parent Paths
  - */parents*

```
/parents:
  post:
      description: Register a new parent account.
      body:
      application/json:
      type: Parent
      responses:
      201:
      description: Parent successfully registered.
      body:
      application/json:
            type: Parent
      400:
      description: Invalid input.
```

```
      body:
      application/json:
            type: Error
  get:
      description: Retrieve a list of all parent accounts (admin only).
      responses:
      200:
      description: List of parent accounts.
      body:
      application/json:
            type: array
            items: User
      403:
      description: Forbidden. Only admins can access this endpoint.
      body:
      application/json:
            type: Error
```

- */parents/{parentId}*

```
/parents/{parentId}:
  uriParameters:
      parentId:
      type: UserId
      description: The unique identifier of the parent.
  get:
      description: Get a specific parent's details.
      responses:
      200:
      description: Parent details.
      body:
      application/json:
```

```
            type: User
      404:
      description: Parent not found.
      body:
      application/json:
            type: Error
put:
      description: Update a parent's details.
      body:
      application/json:
      type: Parent
      responses:
      200:
      description: Parent updated successfully.
      body:
      application/json:
            type: User
      400:
      description: Invalid input.
      body:
      application/json:
            type: Error
delete:
      description: Delete a parent's account.
      responses:
      200:
      description: Parent deleted successfully.
      404:
      description: Parent not found.
      body:
```

```
      application/json:
            type: Error
```

- */parents/{parentId}/payments*

```
/parents/{parentId}/payments:
  uriParameters:
      parentId:
      type: UserId
      description: The unique identifier of the parent.
  post:
      description: Initiate a payment process for a student's tutoring session.
      body:
      application/json:
      type: Payment
      responses:
      201:
      description: Payment initiated successfully.
      400:
      description: Invalid input.
      body:
      application/json:
            type: Error
```

- */parents/{parentId}/payments/{paymentId}*

```
/parents/{parentId}/payments/{paymentId}:
  uriParameters:
      parentId:
      type: UserId
      description: The unique identifier of the parent.
      paymentId:
      type: string
      format: uuid
```

```
      description: Unique identifier for the payment transaction.
  get:
      description: Retrieve the payment details for a parent.
      responses:
      200:
      description: Payment history for the parent.
      body:
      application/json:
            type: array
            items: PaymentDetails
      404:
      description: Parent not found.
      body:
      application/json:
            type: Error
```

- */parents/{parentId}/payments/{paymentId}/receipt*

```
/parents/{parentId}/payments/{paymentId}/receipt:
  uriParameters:
      parentId:
      type: UserId
      description: The unique identifier of the parent.
      paymentId:
      type: string
      format: uuid
      description: Unique identifier for the payment transaction.
  get:
      description: Request a receipt for a specific payment.
      responses:
      200:
      description: Receipt for the payment.
```

```
      body:
      application/json:
            type: Receipt
      404:
      description: Payment not found or receipt unavailable.
      body:
      application/json:
            type: Error
```

- */parents/{parentId}/payments/{paymentId}/calculate-payment*

```
/parents/{parentId}/payments/{paymentId}/calculate-payment:
  uriParameters:
      parentId:
      type: UserId
      description: The unique identifier of the parent.
      paymentId:
      type: string
      format: uuid
      description: Unique identifier for the payment transaction.
  post:
      description: Calculate the cost of a tutoring session before payment.
      body:
      application/json:
      type: PaymentCalculation
      responses:
      200:
      description: Calculated cost of the tutoring session.
      body:
      application/json:
            type: CalculatedAmount
      400:
```

```
        description: Invalid input or calculation error.

        body:

        application/json:

                type: Error
```

- */parents/{parentId}/reports*

```
/parents/{parentId}/reports:

  uriParameters:

        parentId:

        type: UserId

        description: The unique identifier of the parent.

  get:

        description: Retrieve reports for a parent's child (accessible by parent).

        queryParameters: # The studentId is more likely a query parameter rather than a
path parameter

        studentId:

        type: UserId

        description: The unique identifier of the student.

        required: true

        responses:

        200:

        description: Report details for the parent's child.

        body:

        application/json:

                type: array

                items: Report

        403:

        description: Forbidden. Only the parent can access this report.

        body:

        application/json:

                type: Error

        404:
```

```
        description: Parent, student, or report not found.
        body:
        application/json:
                type: Error
```

- Administrator Paths
  - */admins*

```
/admins:
  post:
        description: Register a new administrator account.
        body:
        application/json:
        type: NewAdmin
        responses:
        201:
        description: Administrator successfully registered.
        400:
        description: Invalid input.
        body:
        application/json:
                type: Error
  get:
        description: Retrieve a list of all administrator accounts.
        responses:
        200:
        description: List of administrator accounts.
        body:
        application/json:
                type: array
                items: User
        403:
```

```
      description: Forbidden. Only registered admins can access this endpoint.
      body:
      application/json:
            type: Error
```

- */admins/{adminId}*

```
/admins/{adminId}:
  uriParameters:
      adminId:
      type: UserId
      description: The unique identifier of the administrator.
  get:
      description: Get a specific administrator's details.
      responses:
      200:
      description: Administrator details.
      body:
      application/json:
            type: User
      404:
      description: Administrator not found.
      body:
      application/json:
            type: Error
  put:
      description: Update an administrator's details.
      body:
      application/json:
      type: AdminDetails
      responses:
      200:
```

```
      description: Administrator updated successfully.
      body:
      application/json:
            type: AdminDetails
      400:
      description: Invalid input.
      body:
      application/json:
            type: Error
      404:
      description: Administrator not found.
      body:
      application/json:
            type: Error
  delete:
      description: Delete an administrator's account.
      responses:
      200:
      description: Administrator deleted successfully.
      404:
      description: Administrator not found.
      body:
      application/json:
            type: Error
```

- */admins/{adminId}/user-management*

```
/admins/{adminId}/user-management:
  uriParameters:
      adminId:
      type: UserId
      description: The unique identifier of the administrator.
```

```
  post:
        description: Create a new user account (admin only).
        securedBy: [BearerAuth]
        body:
        application/json:
        type: UserCreation
        responses:
        201:
        description: User account created successfully.
        400:
        description: Invalid input.
        body:
        application/json:
               type: Error
        403:
        description: Forbidden. Only registered admins can create user accounts.
        body:
        application/json:
               type: Error
```

- */admins/{adminId}/user-management/{userId}*

```
/admins/{adminId}/user-management/{userId}:
  uriParameters:
        adminId:
        type: UserId
        description: The unique identifier of the administrator.
        userId:
        type: UserId
        description: The unique identifier of the user to be modified.
  put:
        description: Modify an existing user account (admin only).
```

```
securedBy: [BearerAuth]

body:

application/json:

type: UserModification

responses:

200:

description: User account modified successfully.

400:

description: Invalid input.

body:

application/json:

       type: Error

403:

description: Forbidden. Only admins can modify user accounts.

body:

application/json:

       type: Error

404:

description: User not found.

body:

application/json:

       type: Error
```