Alexis Lappe, Kyle Sargeant, Michelle Chew Poi Yan, Kevin Emery, & Heather Baranek
Dr. Dave
CS 485
December 6th, 2021

Final Report

TriScan

TriScan, is an all-in-one plagiarism and grammar checker. No longer will faculty members have to spend hours scouring the internet and checking references to ensure students are not plagiarizing. Instead, students can upload their assignments to our software and faculty members can let TriScan do the hard-work for them.

Through login credentials, the application will determine if the user is a student or faculty member. Based on this, different functionality will be available. For example, students will be able to upload assignments, quizzes, and exams and see plagiarism and grammar reports sent to them by faculty members. Faculty members will be able to scan assignments, quizzes, and exams for plagiarism, similarities, and grammatical errors and generate reports detailing said categories. The faculty members can also send these reports to specific, selected students.

TriScan works by using data mining. It will process the student's uploaded assignment, visit each reference link, and scan the content of that reference to check for plagiarism or similarities. Sections that appear to be plagiarism or show similarity to the original source will be highlighted and have an associated link to the original reference. Faculty members can then easily navigate to the reference and investigate more deeply to see if plagiarism occurred. Aside from plagiarism, students can also use the application to check for grammatical errors.

Tools and Technologies:

Tools and Platforms: Github and Visual Studio/Visual Studio Code

We decided to use Github because it is commonly used in industry for software development and version control. Additionally, our team has used Github before and it easily integrates with Visual Studio. We decided to use Visual Studio because it allows us to build and run our project without having to use multiple IDES.

Languages:

Front-End: HTML, CSS, and Javascript

We decided to use HTML, CSS, and Javascript to build our front-end because they are the languages most commonly used for front-end development. Additionally, multiple resources and guides are available for these languages if problems occur.

Database: MongoDB

We decided to use MongoDB because it's good for file storage and it's document based.

Back-End: Python

We decided to use Python for our back-end because Python is one of the essential languages needed for data mining. An adaptable and widely-known language, Python is easy to learn and has many resources and guides online that can help us as we learn how to use data mining in software applications.

Functional Requirements

1. Users shall be able to create an account.
    - 1.1. The user shall input first name, last name, an email, and a password to create an account.
        - 1.1.1. Passwords must be at least 8 characters, have at least 1 capital character, 1 lower case character, and 1 number.
        - 1.1.2. Email must be valid.
        - 1.1.3. Passwords and emails cannot contain any special characters or spaces.
    - 1.2. The user can select to be a registered student or a faculty.
    - 1.3. All input fields must be fulfilled in order to register.
    - 1.4. Once registered, a user should be logged in.
    - 1.5. Description: This functional requirement walks through the process of creating an account. To register for an account, users must input their name, email and password. Passwords must follow the guideline of the characteristics mentioned above. Users will have the option to register for an account as a student user or a faculty user. Users must fill in all required input fields to proceed with registration.
2. Users shall login to the system with their credentials.
    - 2.1. The user shall input their email and password to login.
    - 2.2. The user shall be shown to the dashboard page after inputting successful credentials.
    - 2.3. If the user inputs incorrect credentials, they should input new credentials and be shown why their credentials failed.
    - 2.4. Description: This functional requirement describes the process of logging into the system. Registered users will be able to log into the system with their email and password. A dashboard page will appear if login is successful and if login fails, the system will indicate incorrect input and prompt again the login page.
3. Users shall be able to change their login password.
    - 3.1. Once logged in, the user shall be able to input their current password and their new desired password that matches the requirements in the registration process.
    - 3.2. If users forget their current password, users should click a button and the button will prompt for email which will be sent to users.
    - 3.3. Description: This functional requirement describes the steps to change account password. Users will have to enter their current password then only can they change to a new password. If users forget their current password, they have to click a button “forgot password?” and enter their email address in order to change account password via email credentials.
4. Faculty users shall be able to create and sort educational assessments in categories.
    - 4.1. Categories will consist of Quizzes, Exams and Assignments.

4.1.1. Categories will allow faculty members and students to manage and organize documents.

4.2. Description: This functional requirement outlines the ability for faculty to be able to create and sort educational assessments within the application.

5. Faculty users shall be able to add additional assessment categories such as quizzes, exams, and assignments to existing categories.
   5.1. Description: This functional requirement allows faculty members to add additional assessment categories to those already listed on the dashboard.
6. Faculty users shall be able to view student submitted assignments.
   6.1. Faculty shall be able to select which course the student is in.
   6.2. Faculty shall be able to select the student's name from a list that contains the class roster.
   6.3. Faculty shall be able to select which category the student's assignment is in (Quizzes, Exams, Assignments, etc).
   6.4. Description: This functional requirement walks through the process of a faculty user viewing a student assignment. First, the faculty user must select the course the student is in. Next, the faculty user must select the course section the student is in. Then, the faculty user must select the student's name from the class roster. In addition, the faculty member must select which category the student's assignment is in. Finally, the faculty member will then be able to select the student's assignment and view it.
7. Faculty users shall be able to view a report of a student's assignment that details instances of plagiarism, similarities, and grammatical errors.
   7.1. The system shall display the results of its scan of the assignment to the faculty user.
   7.2. The report shall feature the percentage of the assignment that was plagiarised.
   7.3. The report shall feature the percentage of the assignment that had similarities to other references.
   7.4. The report shall feature the percentage of the assignment that had grammatical errors.
   7.5. The report shall have a list of reference links that align with the highlighted instances of plagiarism and similarities.
   7.6. Description: This functional requirement describes the report provided to the faculty user. The report will detail instances of plagiarism, similarities, and grammatical errors. It will list a percentage of the amount of plagiarism, similarities, and grammatical errors that occur in the assignment. It will also feature a list of reference links that align with the instances of plagiarism and similarities in the assignment.
8. Faculty users shall be able to send the report of the student's assignment to the student.
   8.1. For group projects, faculty users shall be able to send reports to the entire group or to selected students in the group.
   8.2. Description: This functional requirement walks through the process of sending the report to students. Faculty users have control over who gets to view the report based upon students assignment submission. If it is a group project, faculty users have the option to send the report to group members collectively or individually.

9. Student users shall be able to upload typed or handwritten assignments, quizzes, and exams.
    9.1. Student user shall use a file option to choose the appropriate category for the right upload
    9.2. Description: This functional requirement walks through the process of uploading the right documents into the system. Students will have the options to pick from a variety of file formats and there will be a button to upload the file.
10. When assignments are uploaded, the system will automatically scan the assignment for plagiarism, similarities, and grammatical errors and create a corresponding report for faculty to view.
    10.1. The system will save the assignment and submission time to the student's submission history
    10.2. After completion of the document scan, the system will post the results to the faculty account corresponding to the course
    10.3. Faculty will be able to view the report immediately after its post to their account
    10.4. Description: This functional requirement describes the process when assignments are uploaded into the system. To begin, the system will collect the submission time of the student which will be recorded in the student's submission history. Once the system scans the document, it will generate a report and send it to the faculty account corresponding to the course. Faculty users will be able to view the report summary of the student's submission.
11. Student users shall be able to view a report of their assignment sent to them by a faculty member.
    11.1. Students will receive a message after logging in if they received a report from the faculty
        11.1.1. Message shall include the assignment, the course and professor as well as provide a link for students to access the report
    11.2. Students can also navigate to the submission of any assignment to see if they received permission to view the report
    11.3. Description: This functional requirement walks through the process of students viewing the reports after the submissions. When students log into their accounts, a notification will appear to them if the faculty user gives permission to the student to view the report.
12. Student users shall be able to view their previous submission history.
    12.1. The history shall include all submitted assignments listed by most recent upload date
    12.2. Listed assignments have an association with the course it was submitted for
        12.2.1. Assignments will be labeled with their associated course
    12.3. Description: This functional requirement walks through the ability for students to view their submission assignment history. From the homepage, students can select the history tab to view their submission history. Within this tab, the previous submissions will be listed by most recent date to least recent date.

Non-Functional Requirements

1. Application shall always be available.

1.1. Description: This non-functional requirement describes the availability of the system

2. System shall register new users within 2 seconds of submission.
    2.1. The system shall return any conflicts that a new user may trigger.
        2.1.1. Users cannot have multiple accounts with the same email.
        2.1.2. Users must have a secure password.
            2.1.2.1. Passwords must contain at least one capital letter, at least one number (1-10) and at least one symbolic character (!,@,#,$,%, etc.)
        2.1.3. Users must provide a first and last name.
        2.1.4. Users must choose a valid role. (Student/Faculty)
            2.1.4.1. If the user selects faculty, they must be approved.
    2.2. Description: This non-functional requirement that describes how quickly a user will be registered after submitting the form and how the system will handle user input.
3. System shall allow a user to login within 2 seconds of an attempt.
    3.1. System will return a valid session and all needed data for site usage on a correct login.
    3.2. Description: This non-functional requirement that describes the process of logging into an account and how quickly that should be handled.
4. System shall update user passwords within 2 seconds of the change.
    4.1. The system will hash passwords with a secure hashing algorithm.
    4.2. Description: This non-functional requirement that describes the process that a user takes to change/update their login password.
5. System shall load the application interface within 5 seconds after login.
    5.1. Application interface should be easily accessible to both faculty users and student users.
        5.1.1. Application user interface will be well organized to decrease confusion and increase accessibility.
    5.2. Description: This non-functional requirement describes how the system will load the application after a user logs in successfully.
6. System shall update educational assessment category options within 3 seconds of a new category submission.
    6.1. Description: This non-functional requirement describes how the system updates new categories for submission within a specific timeframe.
7. Faculty shall be able to view assignments within 5 seconds of being selected.
    7.1. The system will load Faculty courses within 3 seconds.
    7.2. The system will display a class roster within 3 seconds.
    7.3. Description: This non-functional requirement describes how quickly the assignment will be loaded after selection and how quickly the system will load various data such as courses, course sections, and class rosters.
8. Faculty users shall be able to view a report of a student's assignment within 3 seconds of the assignment being selected.
    8.1. The system will load scan results within 3 seconds of the assignment being opened.
    8.2. The system shall display the percentage of the assignment that was plagiarised within 3 seconds.

8.3. The system shall display the percentage of the assignment that has similarities to other references within 3 seconds.
8.4. The system shall display the percentage of the assignment that had grammatical errors within 3 seconds.
8.5. The system shall display a list of reference links that align with the highlighted instances of plagiarism and similarities within 3 seconds.
8.6. Description: This non-functional requirement describes how quickly the system will load and display the report of a student's assignment after selection. It also describes how quickly elements of the report, such as percentages and references, will be displayed.

9. The system shall color-code the report to distinguish between instances of plagiarism, similarities, and grammatical errors.
9.1. The report shall use green to show plagiarism.
9.2. The report shall use blue to show similarities.
9.3. The report shall use yellow to show grammatical errors.
9.4. Description: This non-functional requirement describes how the report will be color-coded to distinguish between instances of plagiarism, similarities, and grammatical errors. Plagiarism will be shown in green, similarities in blue, and grammatical errors in yellow.

10. System shall scan the upload and produce a report including plagiarism, similarities and grammatical errors within 120 seconds.
10.1. Description: This non-functional requirement describes the process and time limit of creating a report. After the student uploads their assignment the system will scan and produce the report for viewing on the faculty account within 120 seconds.

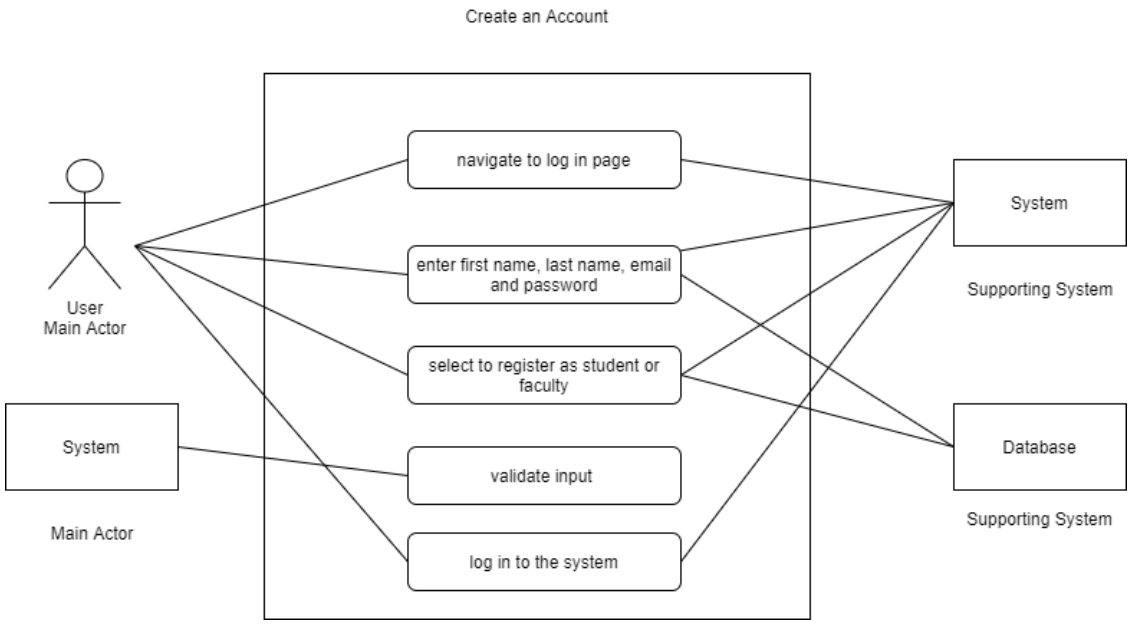
11. System shall provide messages pertaining to report availability within 3 seconds of a student user login.
11.1. Description: This non-functional requirement outlines the time limit for messages to post to the student account. Message alerts for available reports will notify the student within 3 seconds of them logging in.

12. System shall load report for student viewing within 2 seconds after student selects it.
12.1. Description: System shall load report for student within 2 seconds.

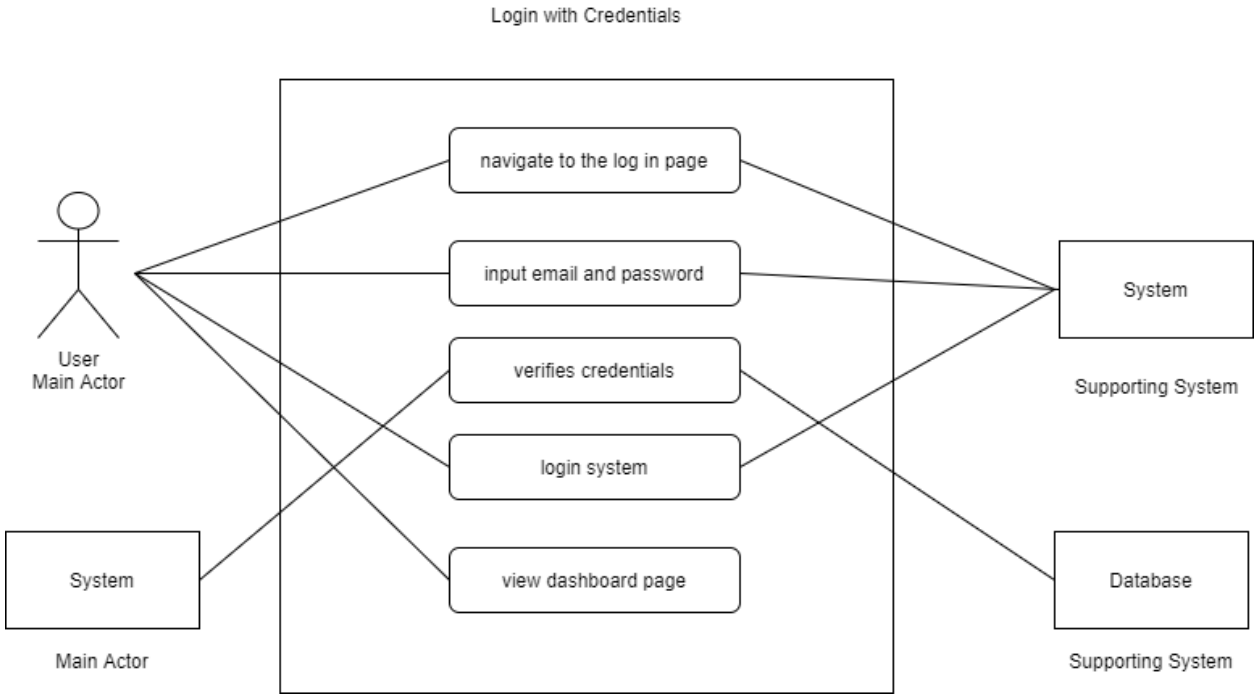
13. The system shall populate the submission history within 10 seconds.
13.1. Assignments will be ordered by most recent submission date.
13.2. Description: This non-functional requirement outlines the loading time for students to view their assignment submission history. The loading time is limited to 10 seconds maximum.

1. Use Case Diagrams

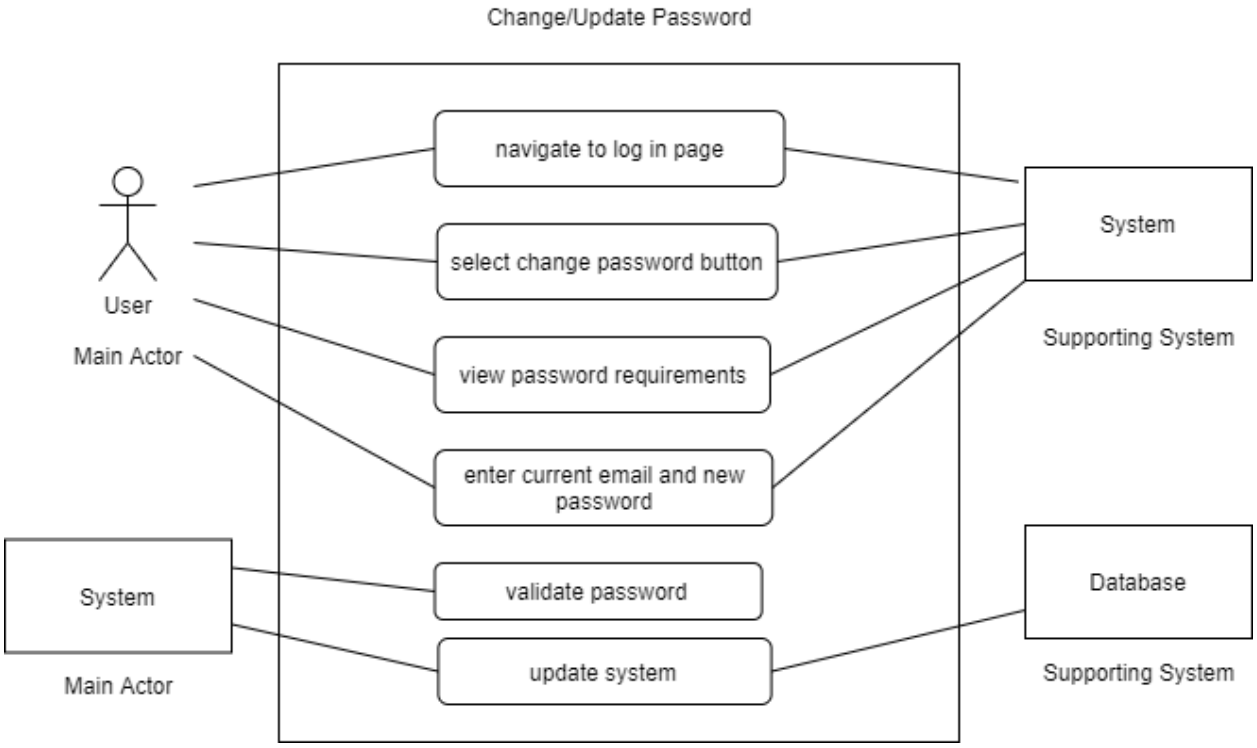
1.1. Main Use Case Diagrams



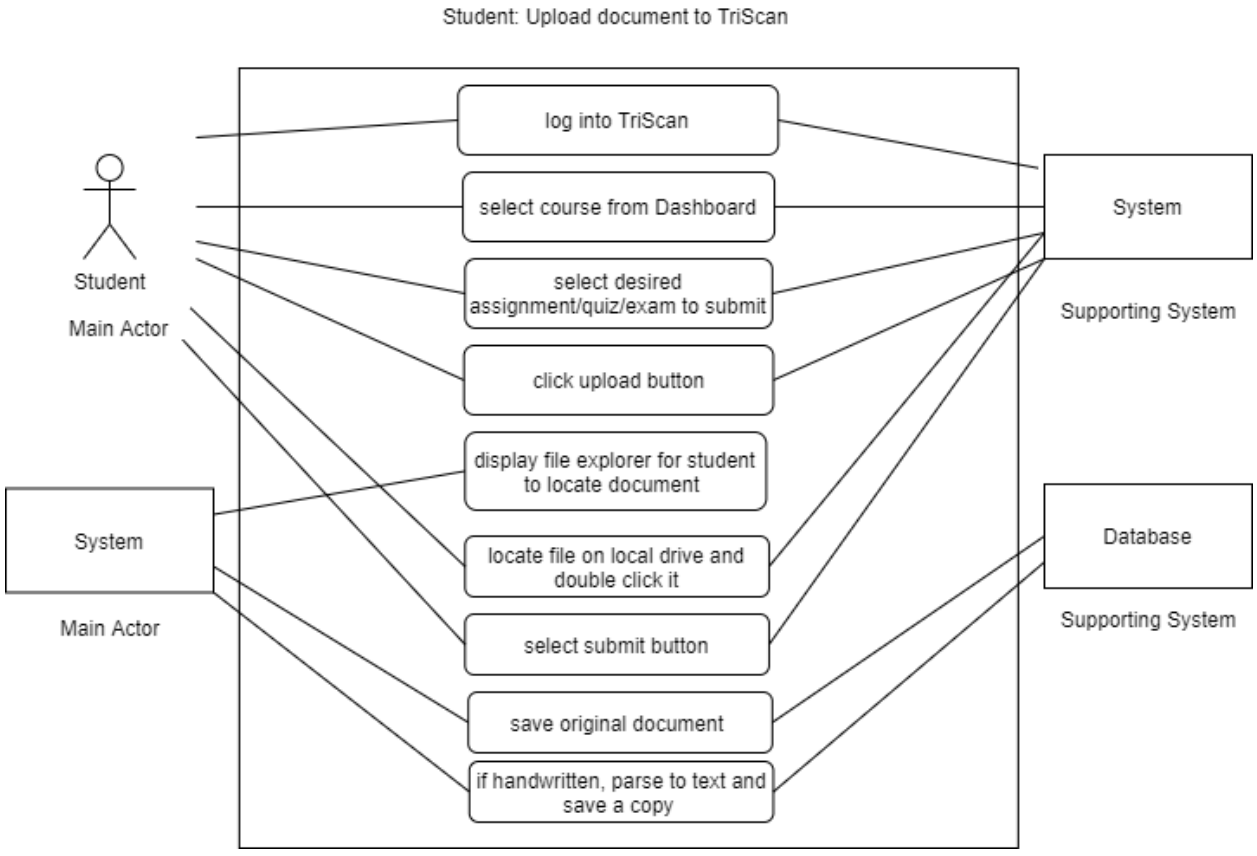
M1. The main actors for the Create an Account Use Case are the user and system. The supporting systems are the system and database. The user will navigate to the login page hosted by the system. The user then enters first name, last name, email address and password in the input fields. The user has the option to register a new account as student or faculty. Next, the system will validate the password and email input fields and check if both meet the system requirements. Assuming the inputs meet the requirements the system will store the registration information in the database for data access in the future. After registration, the user is able to log in to the system



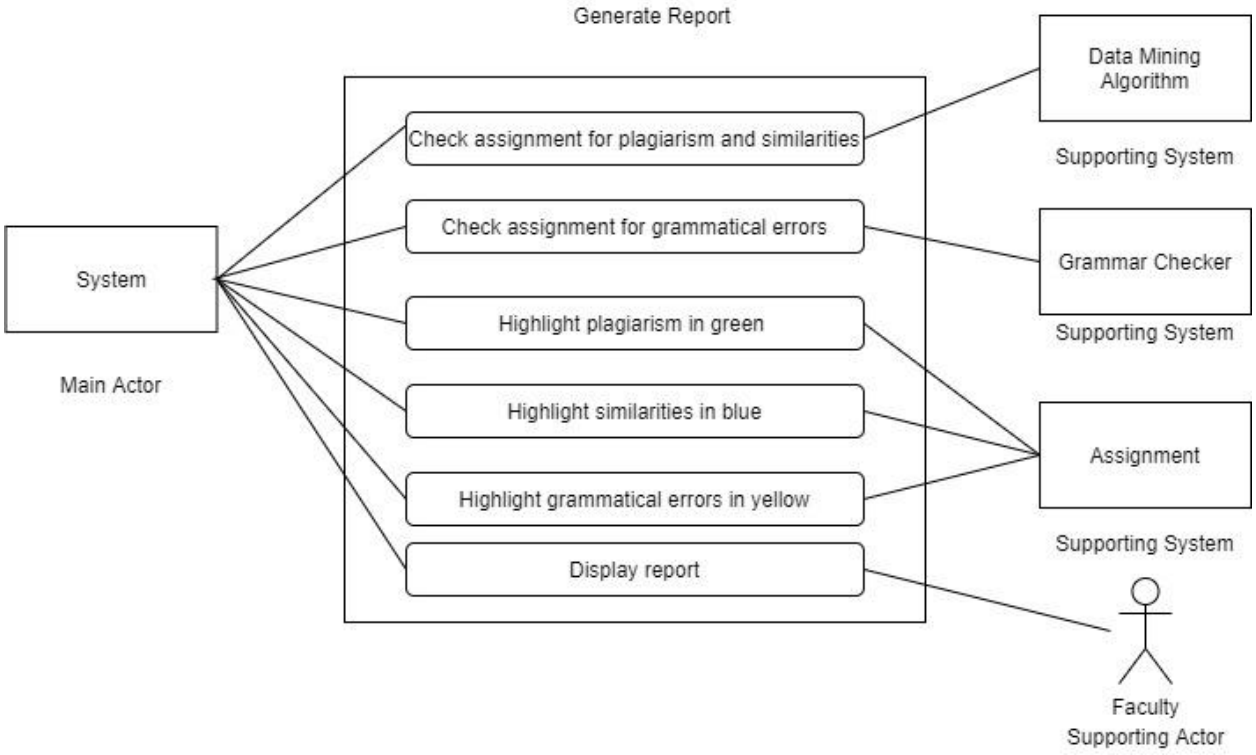
M2. The main actors for the Login with Credentials Use Case are the user and system. The supporting systems are the system and the database. The user will navigate to the login page hosted by the system and enter email address and password. The system will process the user's email address and password and look for the right match in the database. Assuming the credentials are correct and there is an existing account, the user will be able to log into the system and view the dashboard page.



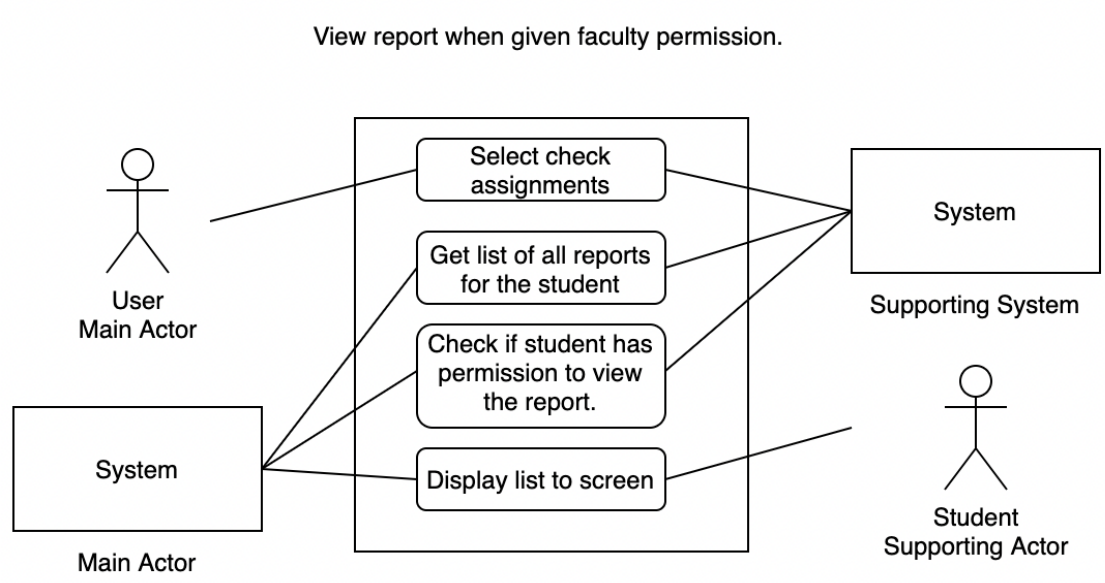
M3. The main actors in the change or update password use case are the user and the system. The supporting systems are the system and the database. In this use case the user navigates to the login page hosted by the system. Then, the user selects the change password button located on the page. Next, they are able to view the password requirements displayed by the system and can enter a new password with their email. The system will process this user provided password and validate it to ensure it meets the password requirements. Assuming the password is valid, the system will update the system by replacing the password for the given user in the database.



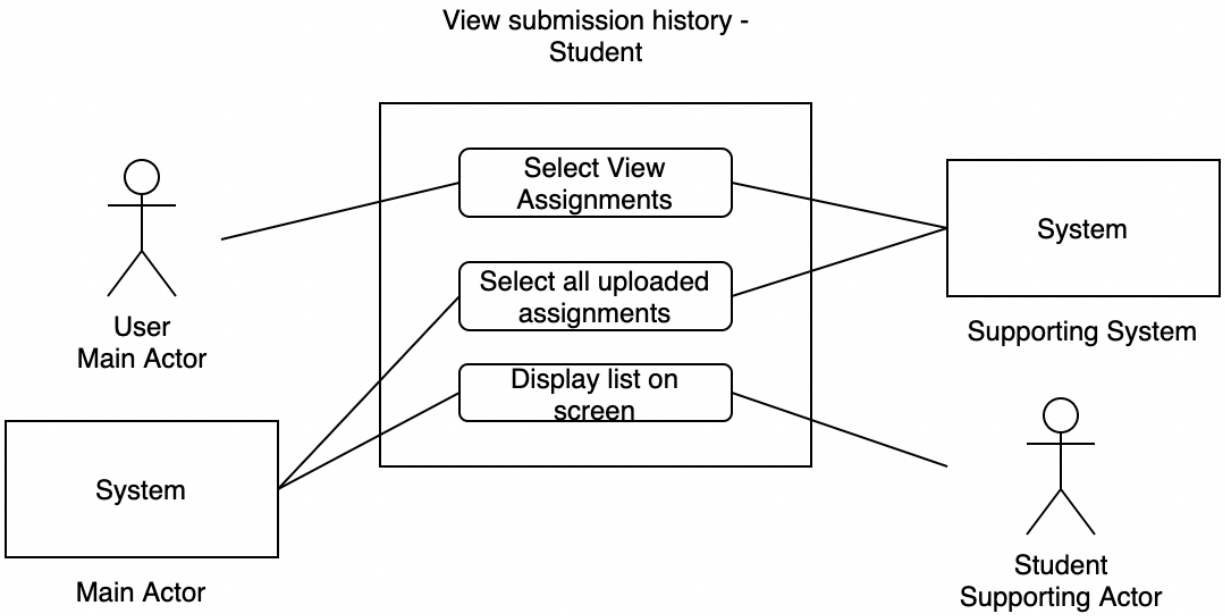
M4. The main actors in the upload document to TriScan use case are the student and the system. The supporting systems are the system and the database. In this use case, the student logs into TriScan and selects a course from the homepage dashboard. Next, they select the assignment, quiz, or exam they wish to submit. Then, the student will click the upload button and the system will display a file explorer for the student to use to locate their file on their local drive. After selecting the file, the student will click the submit button to complete the file upload. Lastly, the system will save the document to the database.



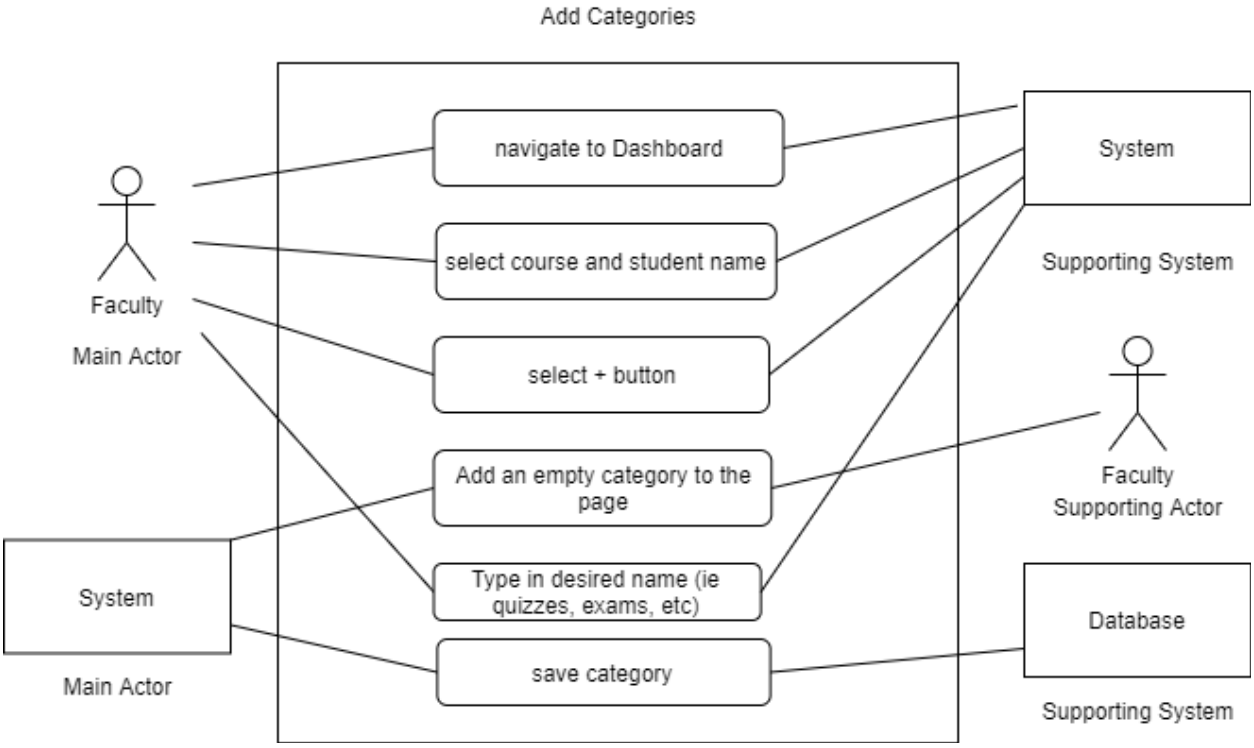
M5. The main actor in the Generate Report use case is the system. The supporting actors are a data mining algorithm, a grammar checker, the assignment, and faculty. The system checks the assignment for plagiarism, similarities, and grammatical errors. It then highlights the plagiarism in green, the similarities in blue, and the grammatical errors in yellow. Finally, it displays the report.



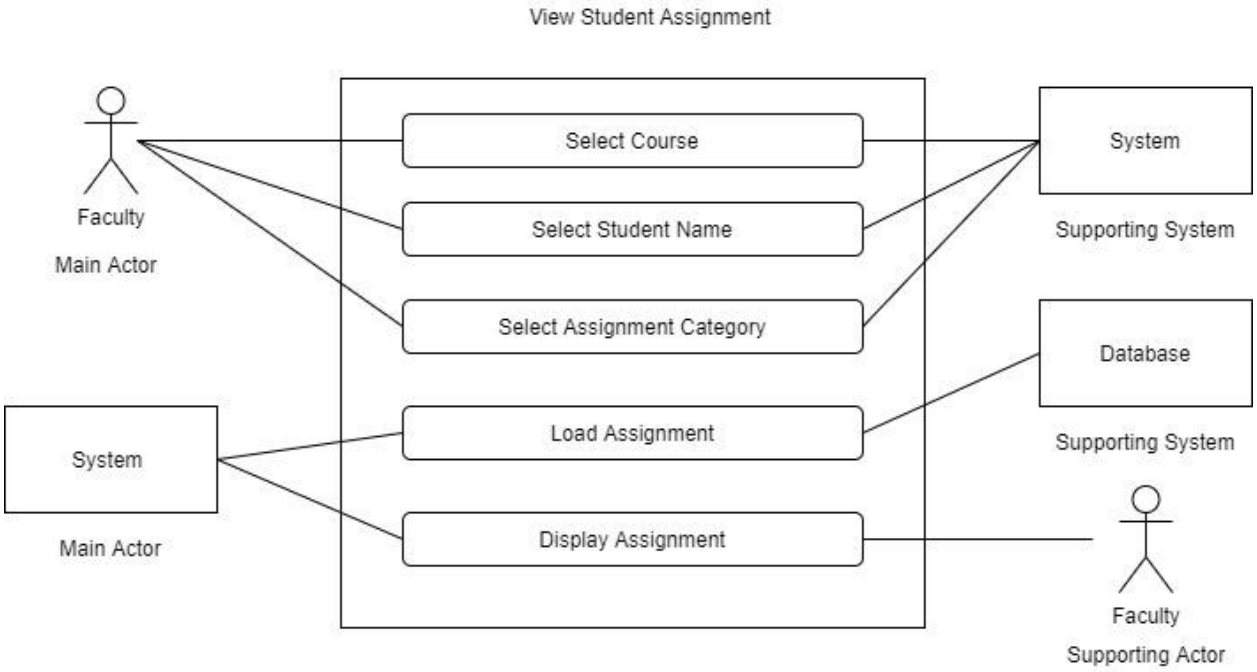
M6. The main actors in the view report use case are the student and the system. The supporting system is the system. In this use case, a student will request to see all of their available reports. The system will then get a list of all the reports generated for the student. It will see if the student was given permission to view each specific report. Once that list is generated, the system will return this list to the user.



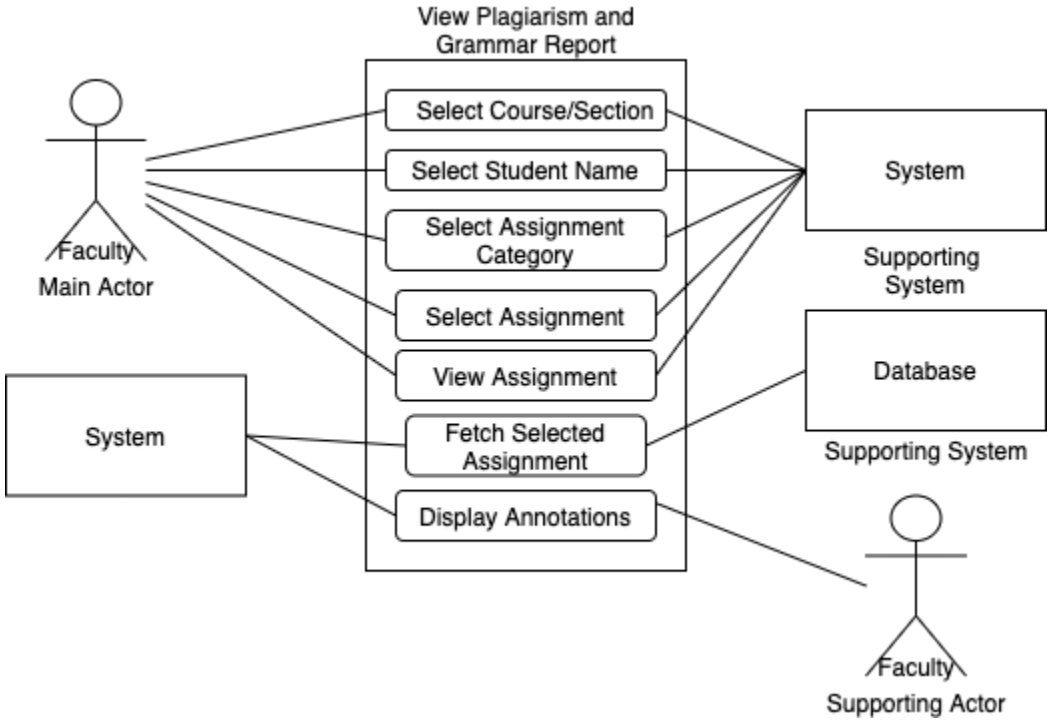
M7. The main actors in the view submission history use case are the student and the system. The supporting system is the system. In this use case, a student will request a list of their past submitted assignments. After that, the system will select all of the assignments uploaded by that student. Then, the system will return this list to the user.



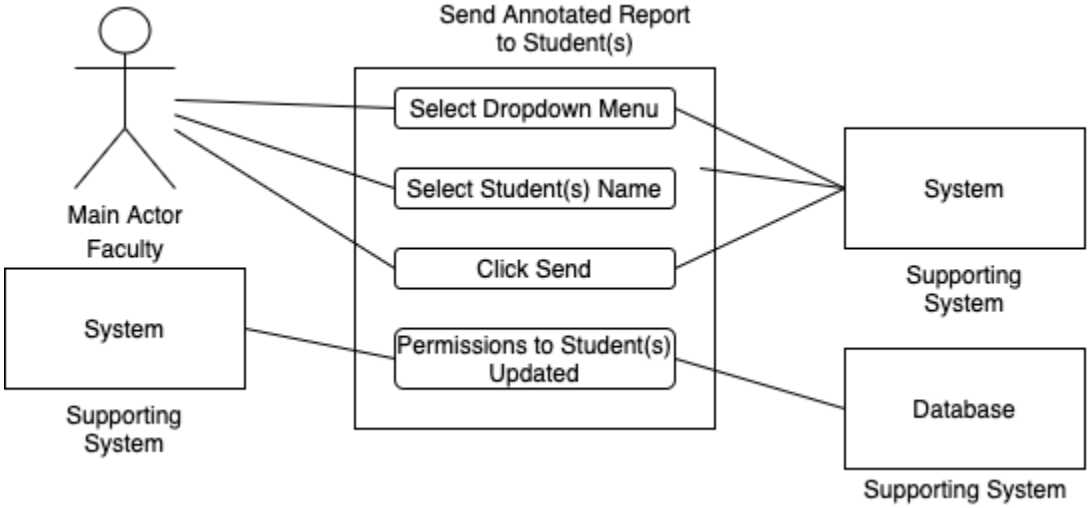
M8. The main actors in the add categories use case are the faculty and the system. The supporting actor is the faculty and the supporting systems are the system and the database. In this use case, the faculty member navigates to the TriScan dashboard. Then, they select a course and student name. Inside the student page, they can click on the "plus" button at the top of the page. After the button has been selected, the system adds an empty category inside the page. Then, the faculty member enters their desired name for that category. Lastly, the information is saved to the database.



M9. The main actors in the View Student Assignment use case are the faculty member and the system. The supporting actors are the system, the database, and the faculty member. The faculty member will select the course, the student name, and the assignment category of the assignment they want to view. The system will load the assignment from the database and then display it.

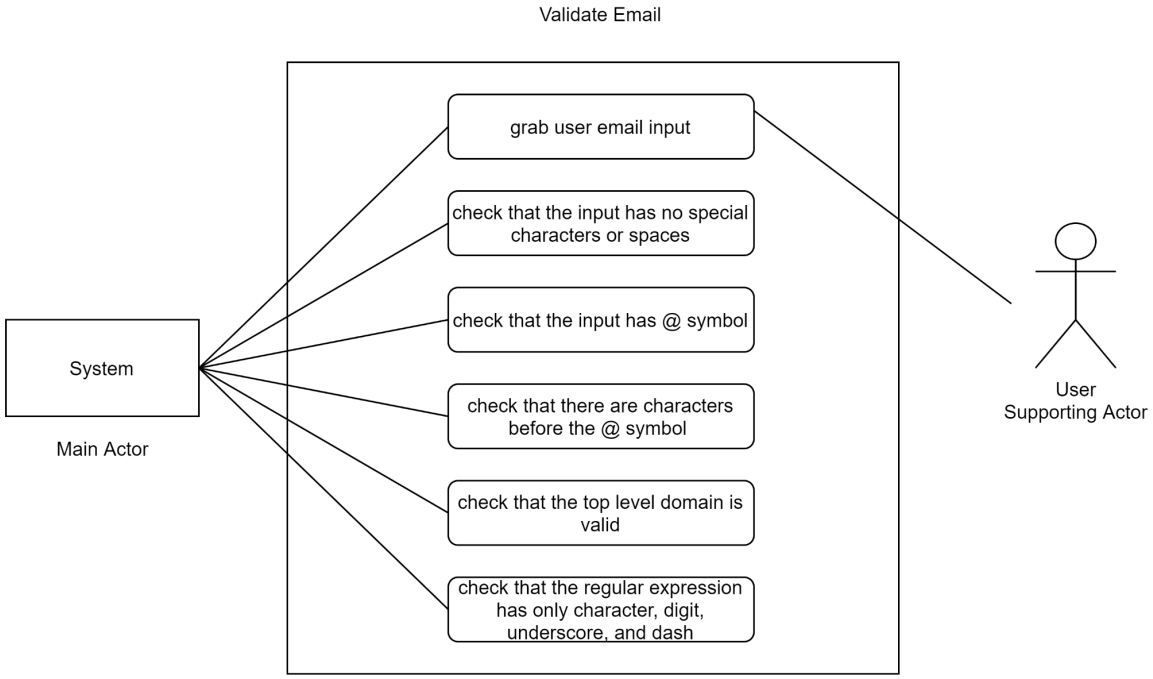


M10. The main actors in sending plagiarism and grammatical error reports to specific students use-case are the faculty member and the system. The supporting system is the system and the database. The faculty member will be able to select a drop down menu that will display a list of all the students in the class and section selected in the categories. The faculty member will then be able to select one or more of the students to send the annotated report to.

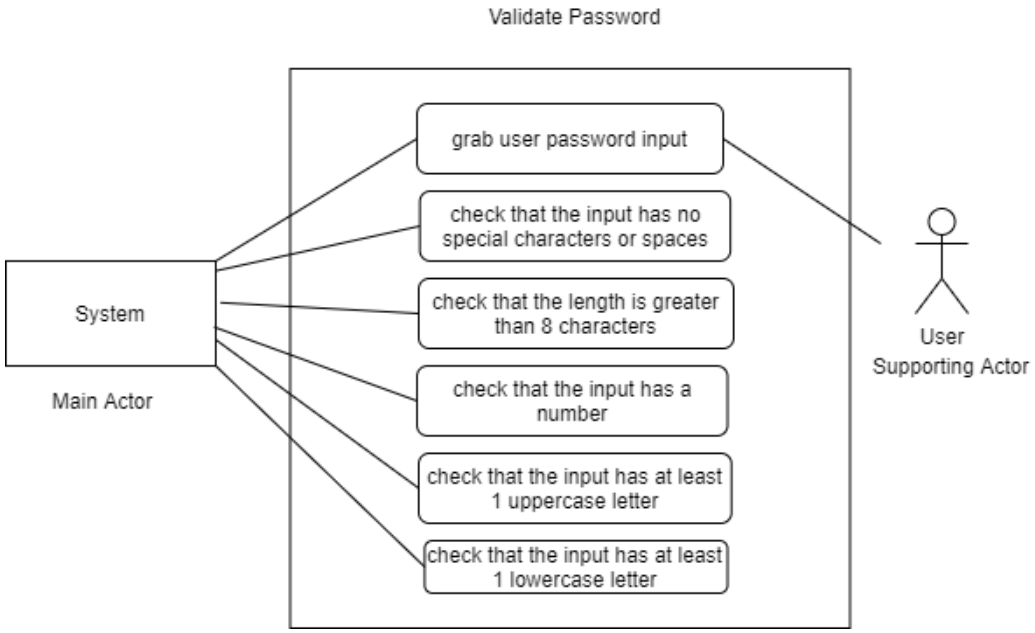


M11. The main actors in the view plagiarism report and grammar errors use case are the faculty member and the system. The supporting system is the system and the database. The supporting actors include the faculty member. The faculty member will select the course, the student name, and the assignment category of the assignment they want to view. The system will load the assignment from the database and then display it along with highlighted sections with. Green highlighted portions will represent instances of plagiarism with a link to the source plagiarized. Blue highlighted portions will represent instances of similarity to other sources that will also be linked. Finally, yellow highlighted portions will signify grammatical errors in the submission.

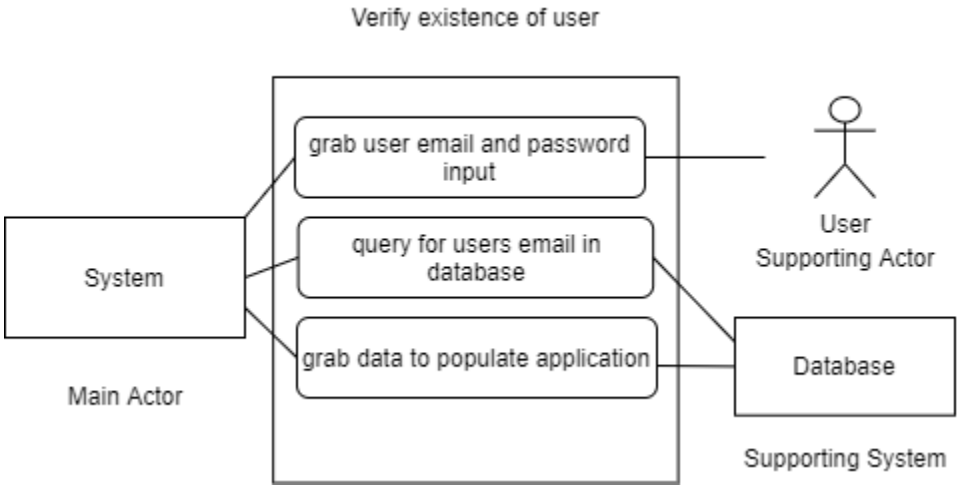
1.2. Alternative Use Case Diagrams



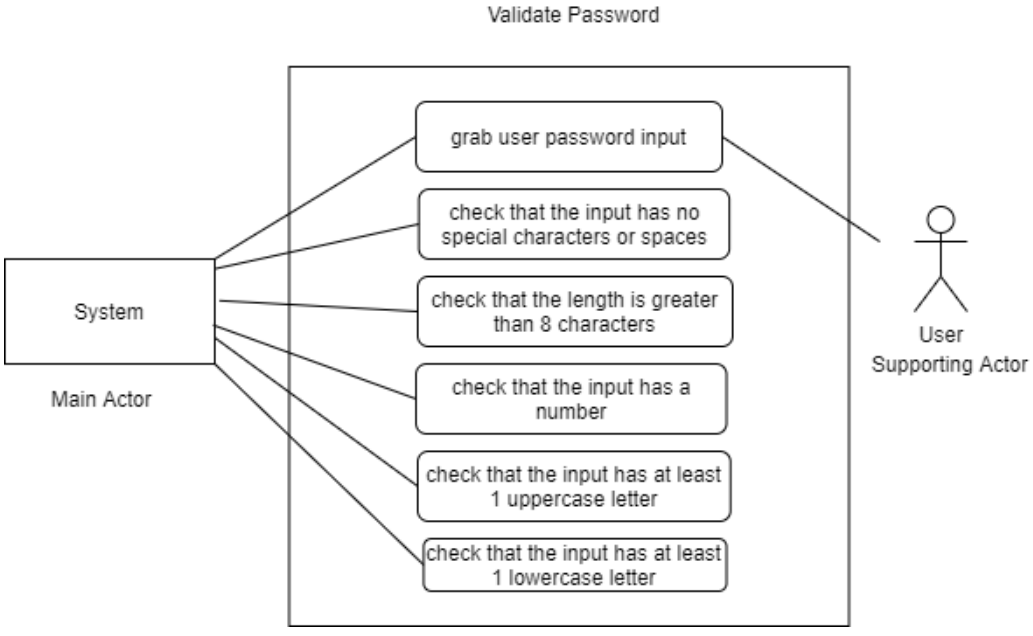
M1A1. The main actor in the validate password use case is the system. The supporting actor is the user. In this use case, the system grabs the input provided by the user. Then, the system checks that input to verify that it has no special characters or spaces. Next, it will check if the input has the @ symbol and that there are characters before the symbol. Then, the system will check the input if it has a valid top level domain. Finally, the system will also check the input for regular expression that only contains character. digit, underscore, and dash.



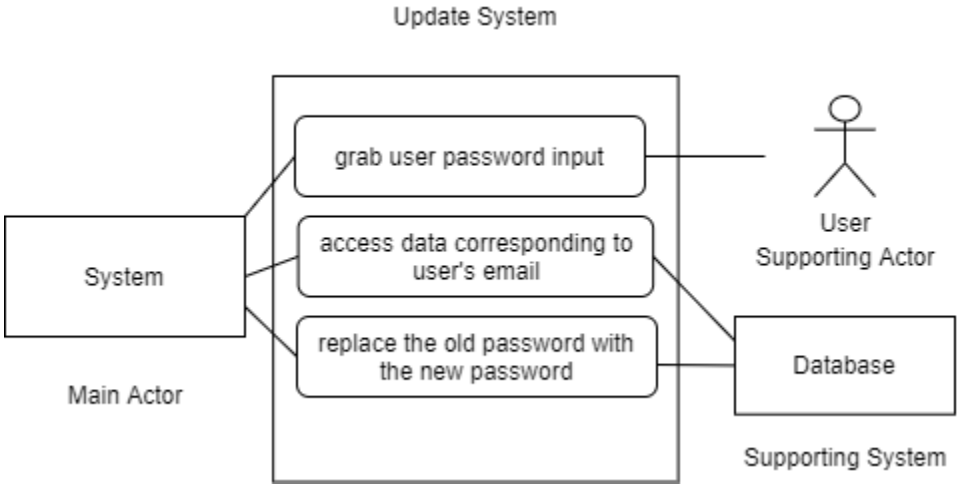
M1A2. The main actor in the validate password use case is the system. The supporting actor is the user. In this use case, the system grabs the input provided by the user. Then, the system checks that input to verify that it has no special characters or spaces. Next, it will check if the length of the input is at least 8 characters long. Then, the system will check the input for the presence of a number, an uppercase letter and a lowercase letter. If the password passes these checks, then it is a valid password



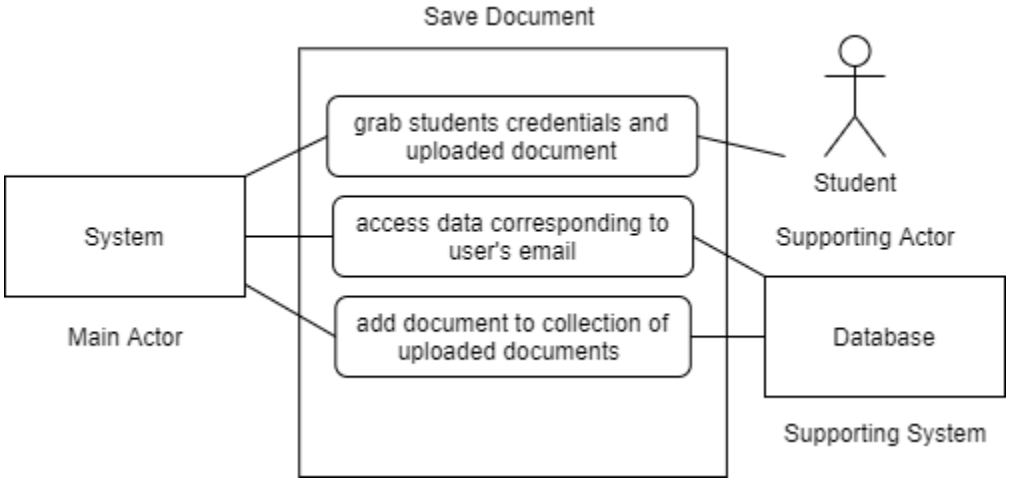
M2A1. The main actor in the verify existence of user use case is the system. The supporting actor is the user and the supporting system is the database. In this use case, the system grabs the input provided by the user. Then, the system checks the email against the database for the existence of an account in the database. Lastly, assuming the account exists, the system gathers the data corresponding to that account to populate the screens in the application.



M3A1. The main actor in the validate password use case is the system. The supporting actor is the user. In this use case, the system grabs the input provided by the user. Then, the system checks that input to verify that it has no special characters or spaces. Next, it will check if the length of the input is at least 8 characters long. Then, the system will check the input for the presence of a number, an uppercase letter and a lowercase letter. If the password passes these checks, then it is a valid password

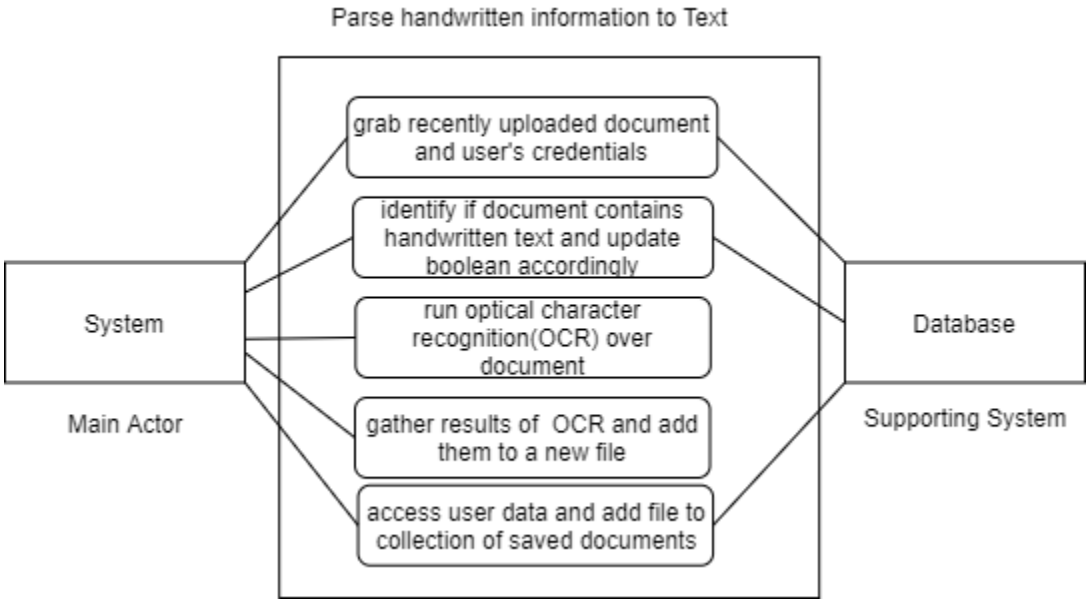


M3A2. The main actor in the update system use case is the system. The supporting actor is the user and the supporting system is the database. In this use case, the system grabs the password that the user entered and accesses the data connected to the user's email. Then, the system updates the database by replacing the old password with the new password.

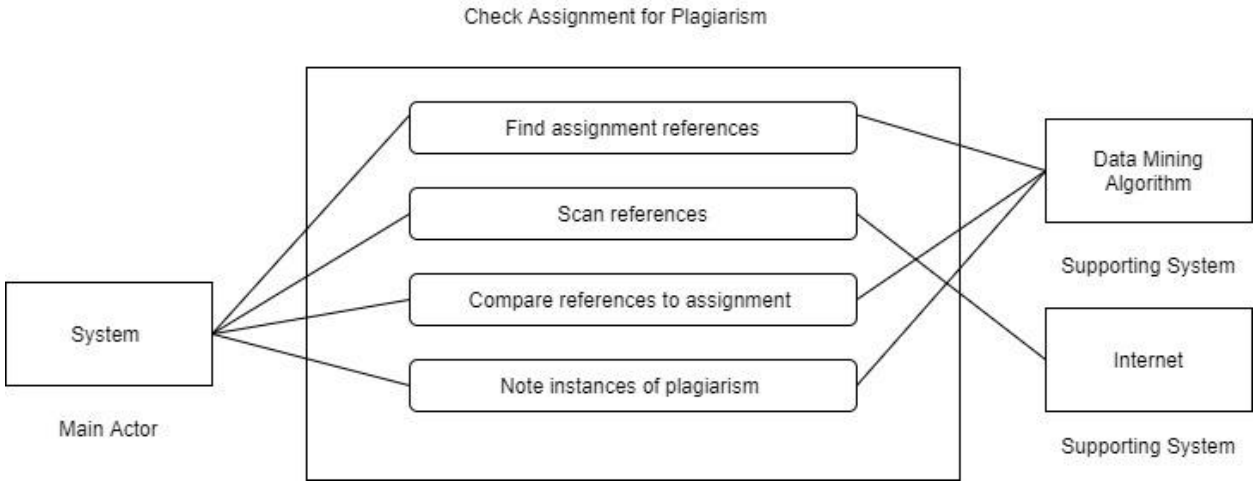


M4A1. The main actor in the save document alternative use case is the system. The supporting actor is the student and the supporting system is the database. In this use case, the system receives the student's credentials and uploaded document. Next, the system

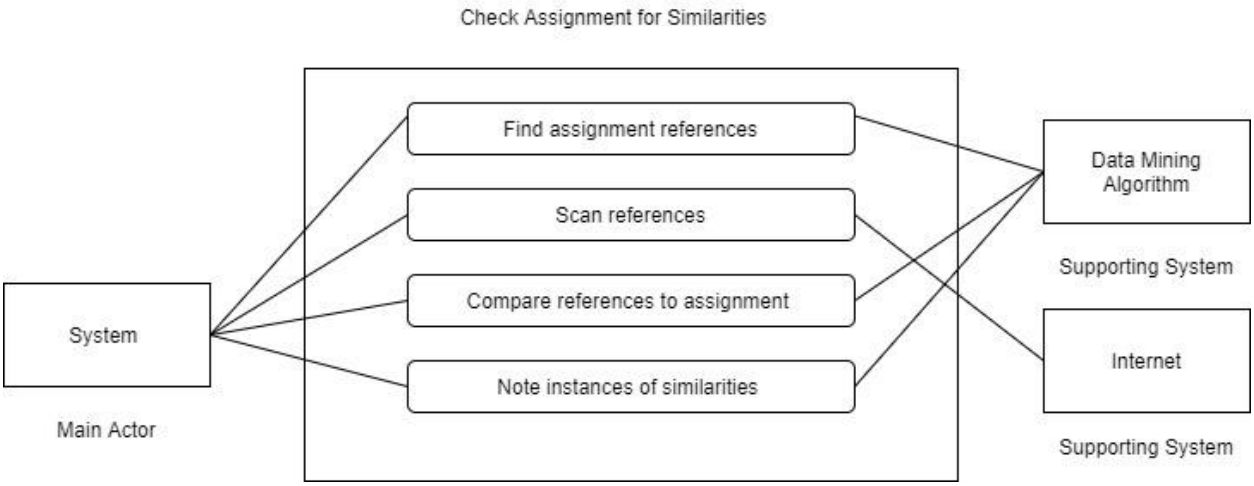
uses the student's email to access their data within the database. Lastly, the system adds the document to the collection of uploaded documents within the user's data.



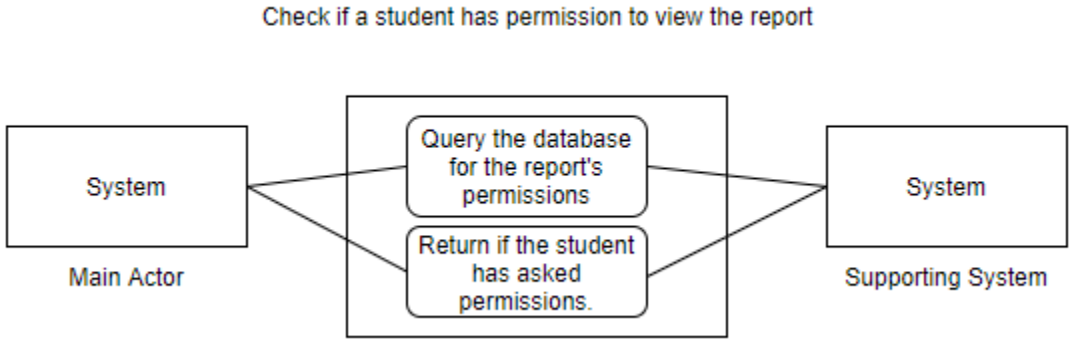
M4A2. The main actor in the parse handwritten information to text is the system. The supporting system is the database. In this use case, the system grabs the recently uploaded document from the database and the user's credentials. Then, the system scans the document to identify any handwritten information and updates the boolean to reflect the presence of handwriting. Next, the system runs optical character recognition (OCR) over the document to extract the text. The system gathers the results of OCR and appends them to a new file. Lastly, the system accesses the user's data and adds the file to the collection of saved documents.



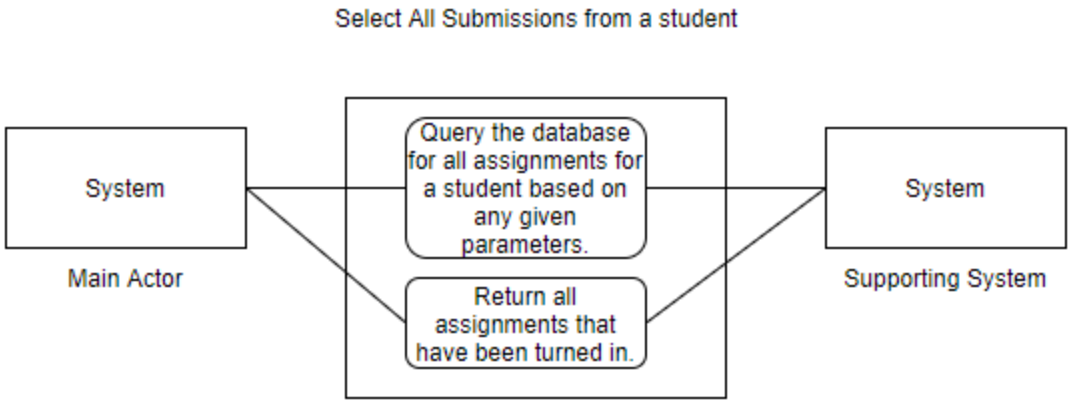
M5A1. The main actor in the Check Assignment for Plagiarism alternate use case is the system. A data mining algorithm and the Internet are supporting systems. The system will find the assignment's references and then scan them. It will compare the references to the assignment and note instances of plagiarism.



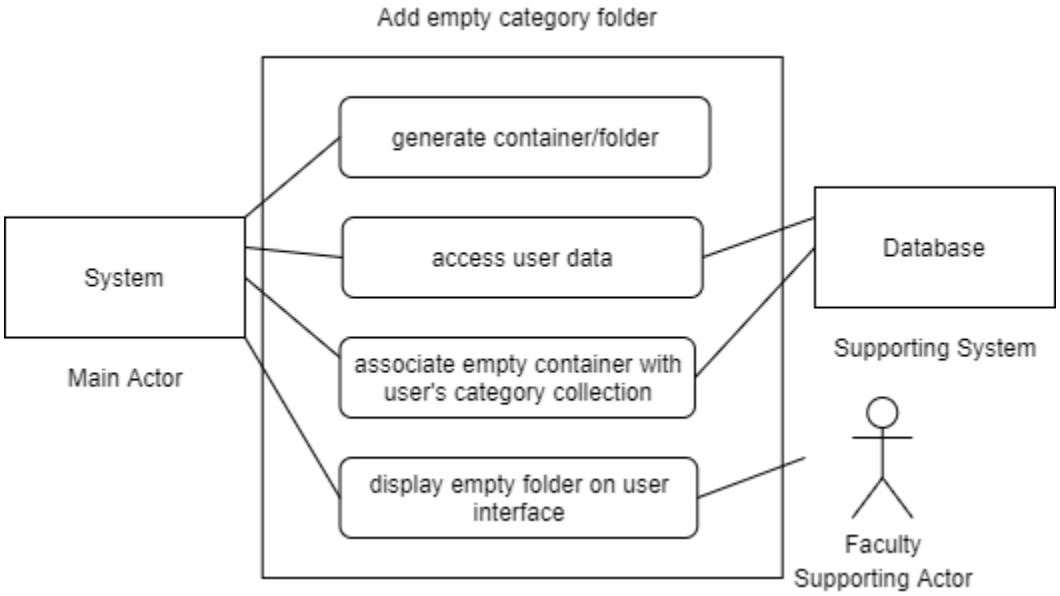
M5A2. The main actor in the Check Assignment for Similarities alternate use case is the system. The supporting systems are a data mining algorithm and the Internet. The system finds the assignment's references and scans them. It then compares the references to the assignment and notes instances of similarities.



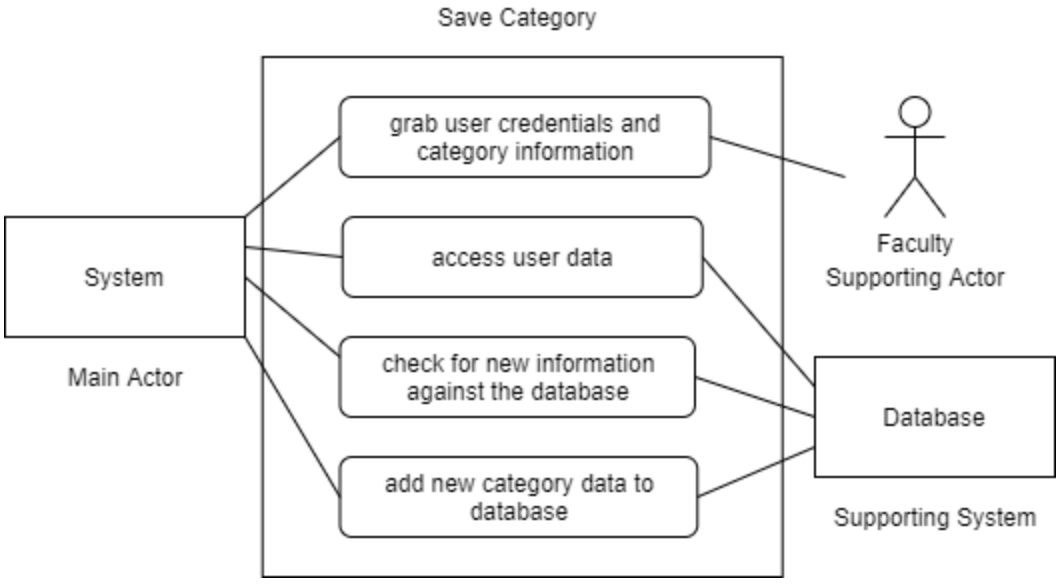
M6A1. The main actor in the Check if a student has permission to view the report alternate use case is the system. The supporting system is the system. The system gives a user and a report and then looks through the report's permissions to see if that user has been approved by a staff member to view whichever report they're requesting.



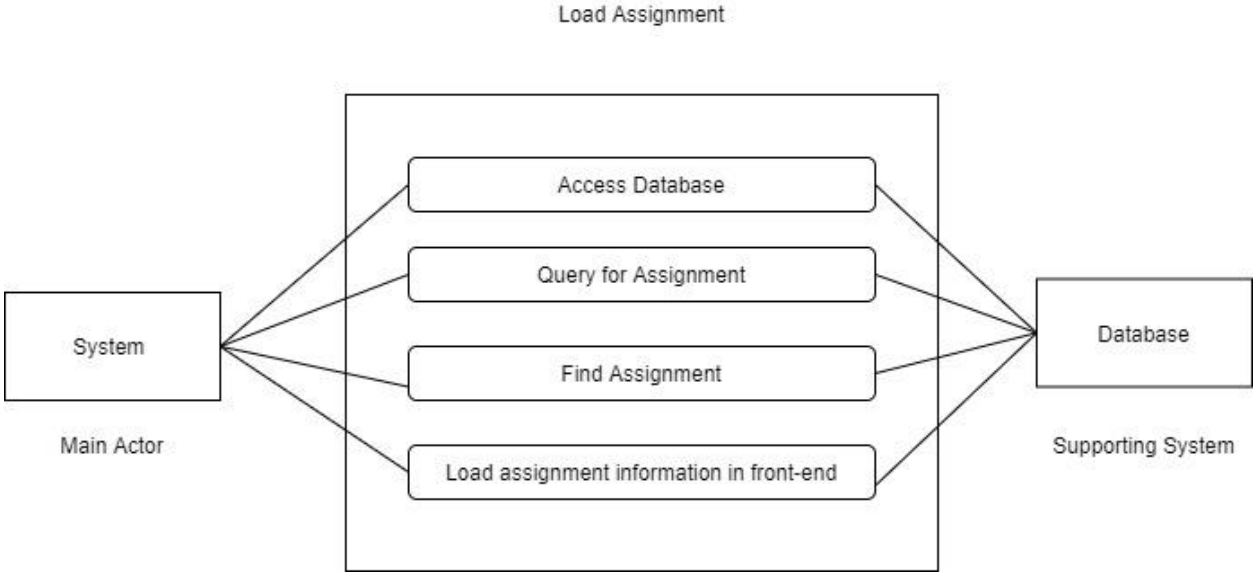
M7A1. The main actor in the Select All Assignments alternate use case is the system. The supporting system is the system. The system finds all submissions in the database that are completed by the user. It ensures that each submission isn't deleted or hidden from the student's view then gives back all of the valid submissions.



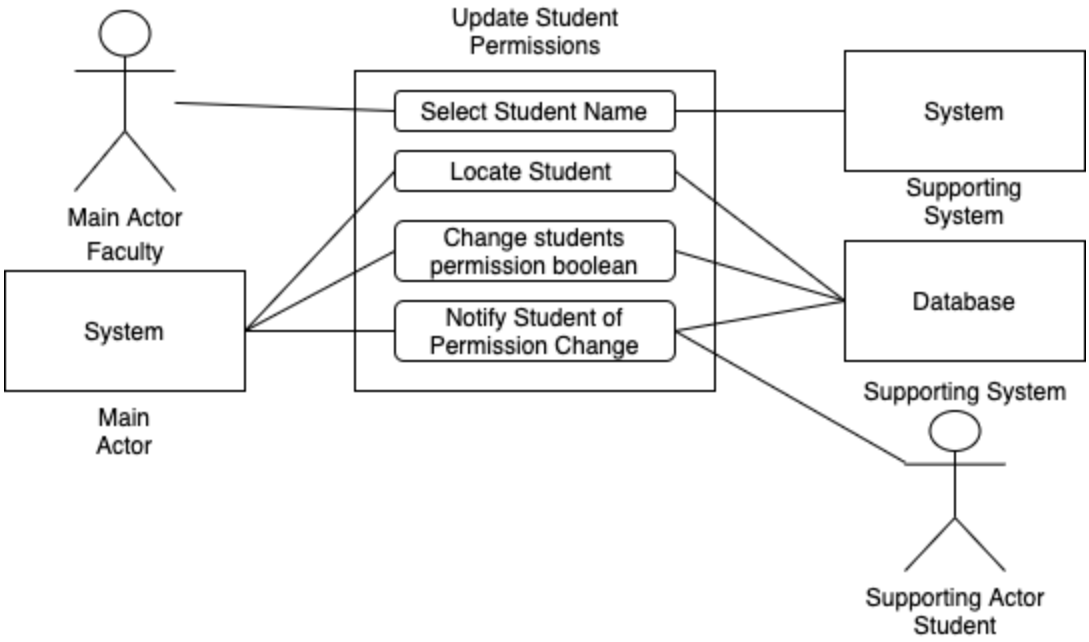
M8A1. The main actor in the add empty category folder alternate use case is the system. The supporting system is the database and supporting actor is the faculty member. In this use case the system generates the new folder. Then, it accesses the user's information and adds the new folder to the user's category collection. Lastly, the system displays the empty folder to the faculty on the user interface.



M8A2. The main actor in the save category alternative use case is the system. The supporting actor is the faculty and the supporting system is the database. In this use case, the system grabs the user credentials from the faculty to access the database and the category information. Next, the system accesses the user's data and checks against the database for updated information about a given category. Lastly, the system adds the new category data to the database.



M9A1. In the Load Assignment use case, the system is the main actor and the database is the supporting system. The system will access the database, query for the assignment, find the assignment, and load it into the front-end.



M11A1. In the Update Student Permissions use-case, the system and the faculty member are the main actors. The system, database and student are supporting systems and actor. The faculty member will select the students name from the drop down menu. The system will communicate with the database to find the student based on the selected name. The system will change the permission to view report permission boolean for the student. The system will then send a notification to the student that their permissions have been changed.

1.3. Exceptional Scenarios

1.3.1. M1

1.3.1.1. The register page is down or non-viewable to users

1.3.1.2. The system fails to register the right user account

1.3.1.3. The system continues with the login process with invalid input

1.3.1.4. The system fails to login after user registration

1.3.2. M2

1.3.2.1. The login page is down or non-viewable to users

1.3.2.2. The system fails to scan user's input fields

1.3.2.3. The system fails to verify credentials hence the user logs into the system with incorrect credentials

1.3.2.4. The dashboard page is down or non-viewable to users after successful login

1.3.3. M3

1.3.3.1. The TriScan login page is down or non-viewable to users

1.3.3.2. The system fails to update the password

1.3.3.3. The system updates the password with an invalid one

1.3.3.4. The system fails to locate an email match in the database

1.3.4. M4

1.3.4.1. The document upload is not saved to the database

1.3.4.2. The document is saved in the incorrect location within the database
1.3.4.3. The file explorer does not load properly and prohibits a file submission
1.3.4.4. The system does not display all available courses and assignments for the student to upload to

1.3.5. M5
1.3.5.1. Instances of plagiarism are not highlighted.
1.3.5.2. Instances of plagiarism are highlighted in an incorrect color.
1.3.5.3. Grammatical errors are missed.
1.3.5.4. The system doesn't display the report.

1.3.6. M6
1.3.6.1. Staff revokes access while the student can click to view.
1.3.6.2. A report is shown to the wrong user.
1.3.6.3. Attempts to access a report they're not authorized for.
1.3.6.4. The system can't display the report.

1.3.7. M7
1.3.7.1. The wrong submissions are loaded.
1.3.7.2. A submission may not be accessible, but is shown.
1.3.7.3. Attempts to access a submission they're not authorized for.
1.3.7.4. The system is down.

1.3.8. M8
1.3.8.1. The system does not add a new category when requested
1.3.8.2. The user is allowed to create new categories but they are not saved to the system
1.3.8.3. The incorrect list of student names is associated with a given course
1.3.8.4. The user is unable to type in a new name for the category

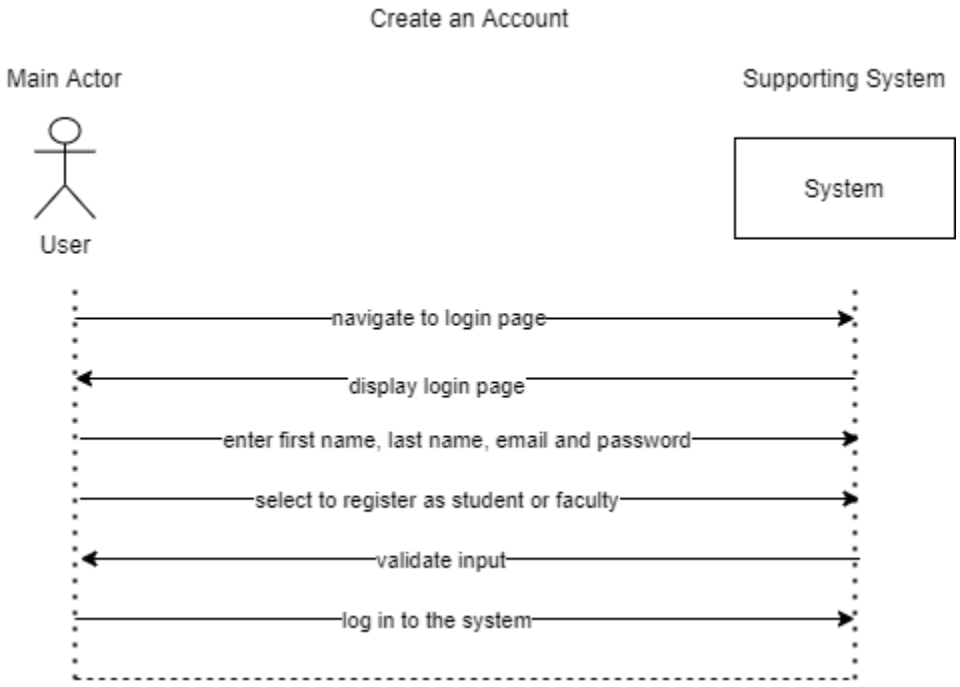
1.3.9. M9
1.3.9.1. Incorrect courses are displayed.
1.3.9.2. Not all student names are listed.
1.3.9.3. The system doesn't load the assignment.
1.3.9.4. The system loads the incorrect assignment.

1.3.10. M10
1.3.10.1. Annotations do not display upon viewing submission.
1.3.10.2. Attempts to access a category that is not selectable.
1.3.10.3. Plagiarism or similarities generate without a clickable source.
1.3.10.4. System incorrectly highlights plagiarism, similarity or grammar error occurrences.

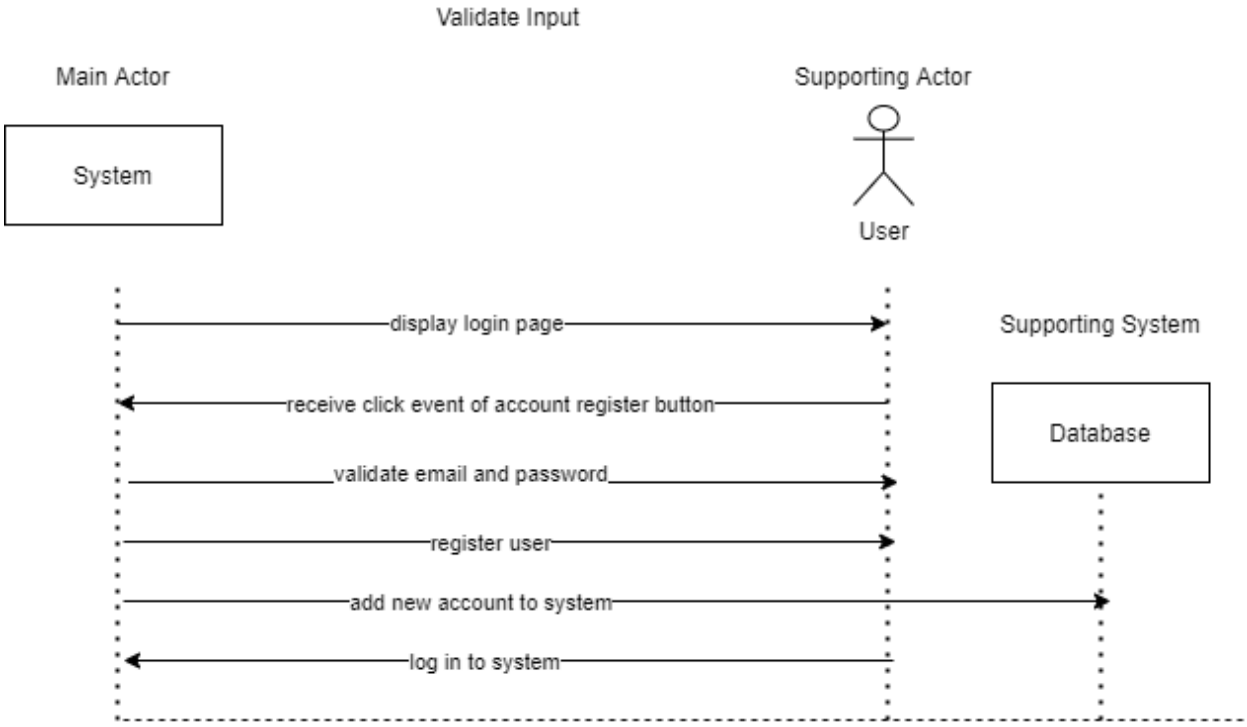
1.3.11. M11
1.3.11.1. Student name list does not accurately reflect the class roster in relation to class and section.
1.3.11.2. Faculty members cannot select more than one student to send the annotated report to.
1.3.11.3. Report is stated as being sent but never received by the student.
1.3.11.4. Students' permission to access the annotated report never gets changed.

2. System Sequence Diagrams

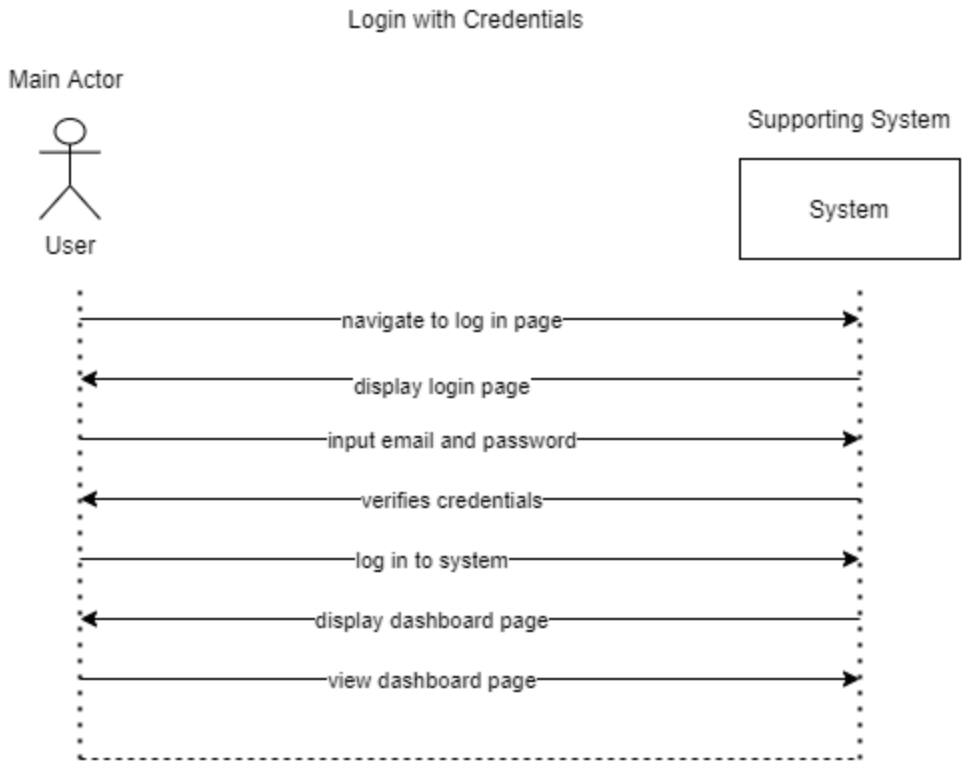
2.1. Main Sequence Diagrams



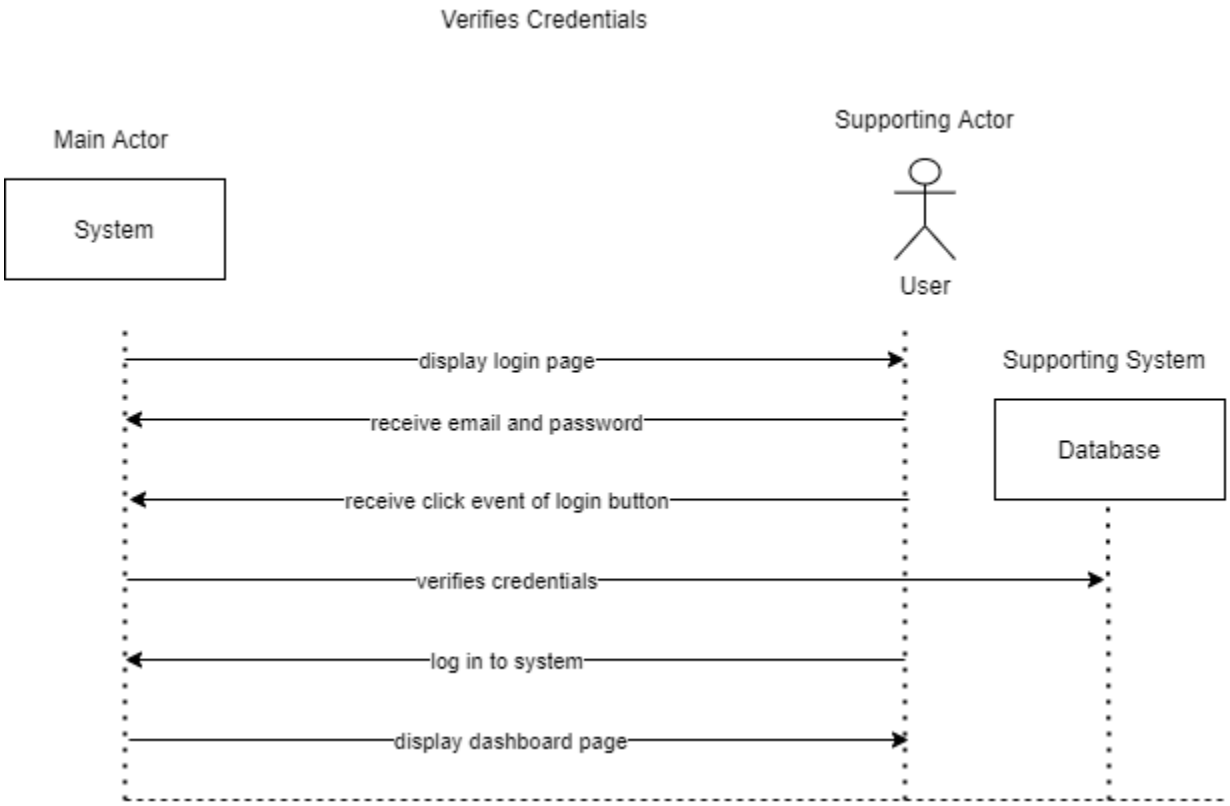
M1SD1. The main actor in the create account system sequence diagram is the student. The supporting system is the system. In this sequence diagram, the user navigates to the login page. Afterwards, the system displays the login page to the user and they enter their name, email and password. The student also registers their status as a student or faculty. Then, the system validates the input the student enters. Lastly, the student logs onto the system.



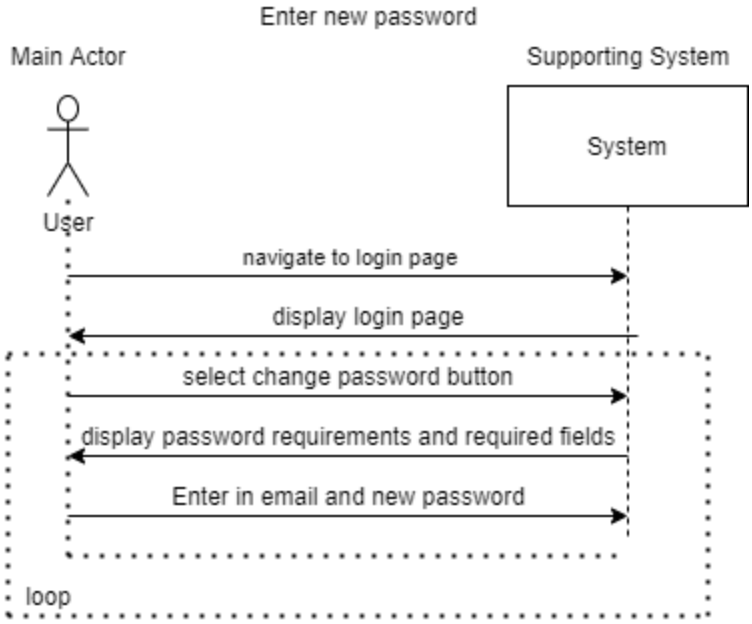
M1SD2. The main actor in the validate input system sequence diagram is the system. The supporting system is the database and the supporting actor is the user. In this sequence diagram, the system displays the login page to the user and receives the click event of the account register button. Next, the system receives the users input and validates their information. Then, the system adds the new account to the database and the user is able to log in to the system.



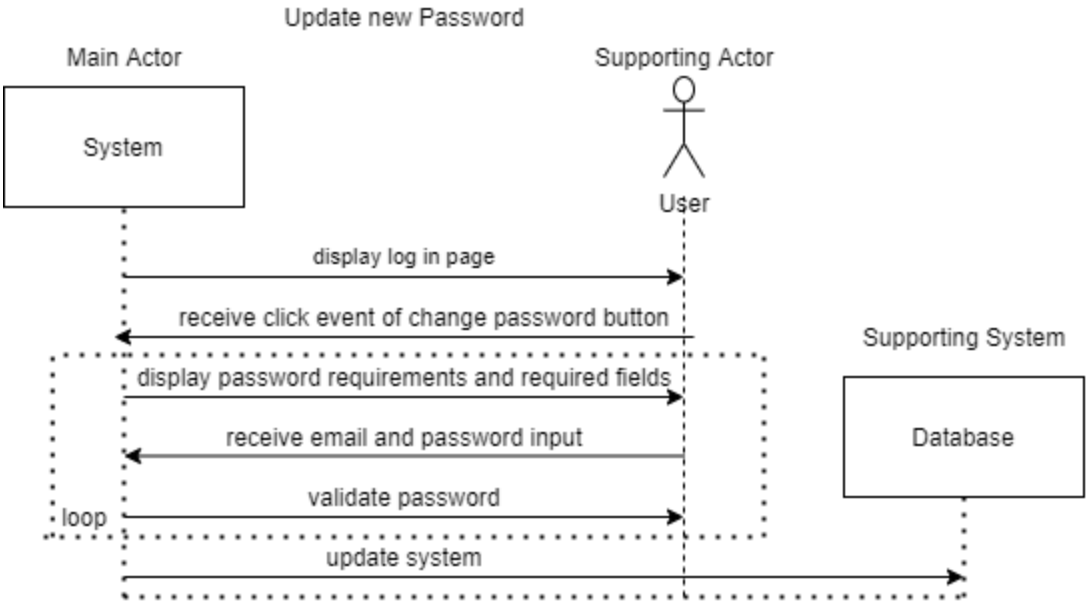
M2SD1. The main actor in the login with credentials system sequence diagram is the user. The supporting system is the system. In this sequence diagram, the user navigates to the login page. Afterwards, the system displays the login page and the user inputs their email and password. Then, the system verifies the user credentials, logs the user into the system, and displays the dashboard page for the user to view.



M2SD2. The main actor in the verifies credentials system sequence diagram is the system. The supporting system is the database and the supporting actor is the user. In this sequence diagram, the system displays the log in page to the user and receives their email and password. The system also receives the click event of the log in button. Next, the system verifies the user's credentials and the user logs onto the system. Lastly, the system displays the dashboard page to the user.

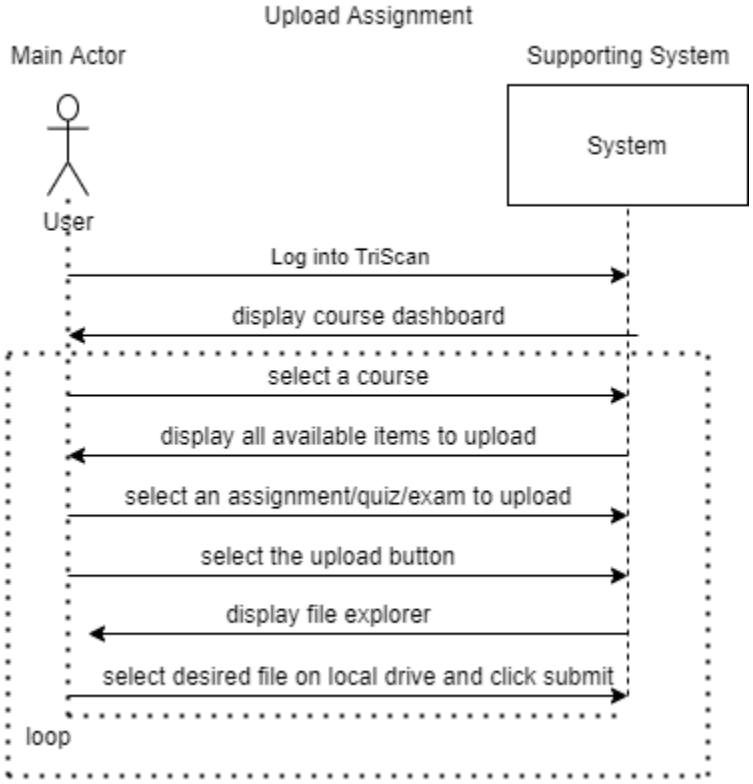


M3SD1. The main actor in the enter new password system sequence diagram is the student. The supporting system is the system. In this sequence diagram, the user navigates to the login page. Afterwards, the system displays the login page to the user and they select the change password button. Next, the system displays the password requirements and required fields for the student to enter and then they enter their email and new password into the system. This process loops through the desired number of times a student wants to change their password.

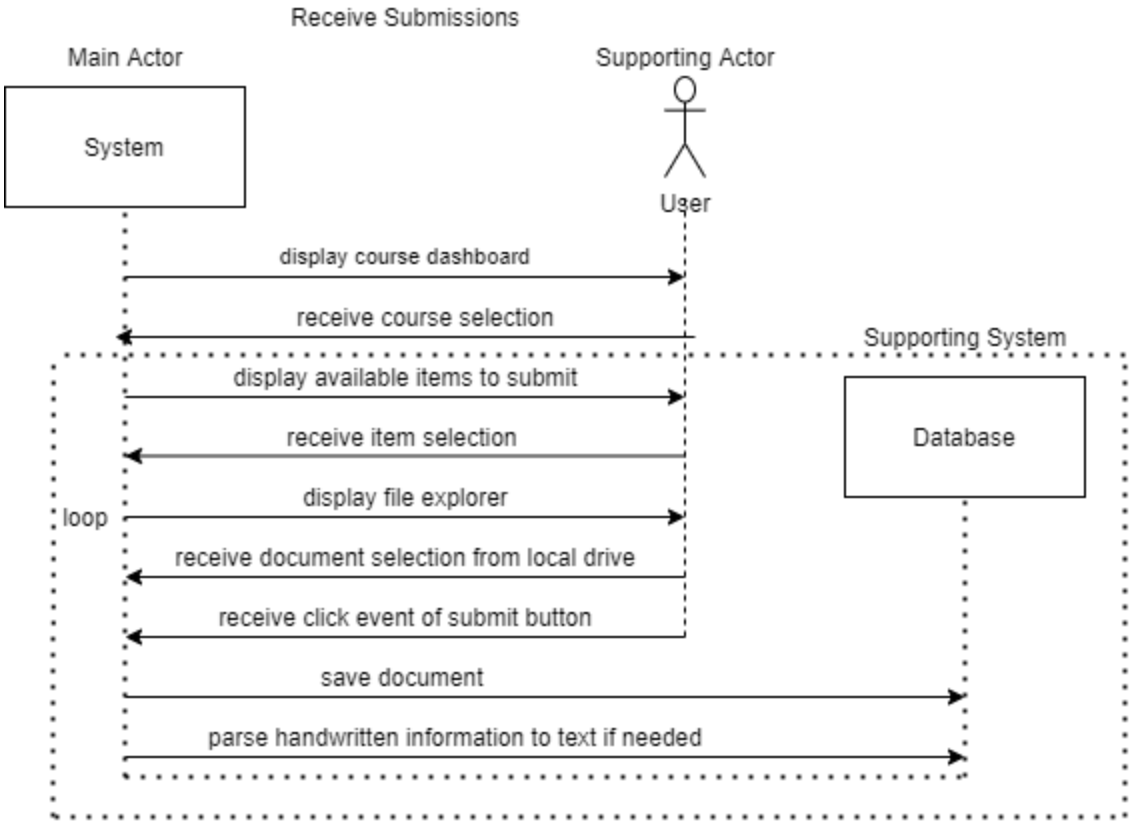


M3SD2. The main actor in the update new password system sequence diagram is the system. The supporting actor is the user and the supporting system is the database. In this

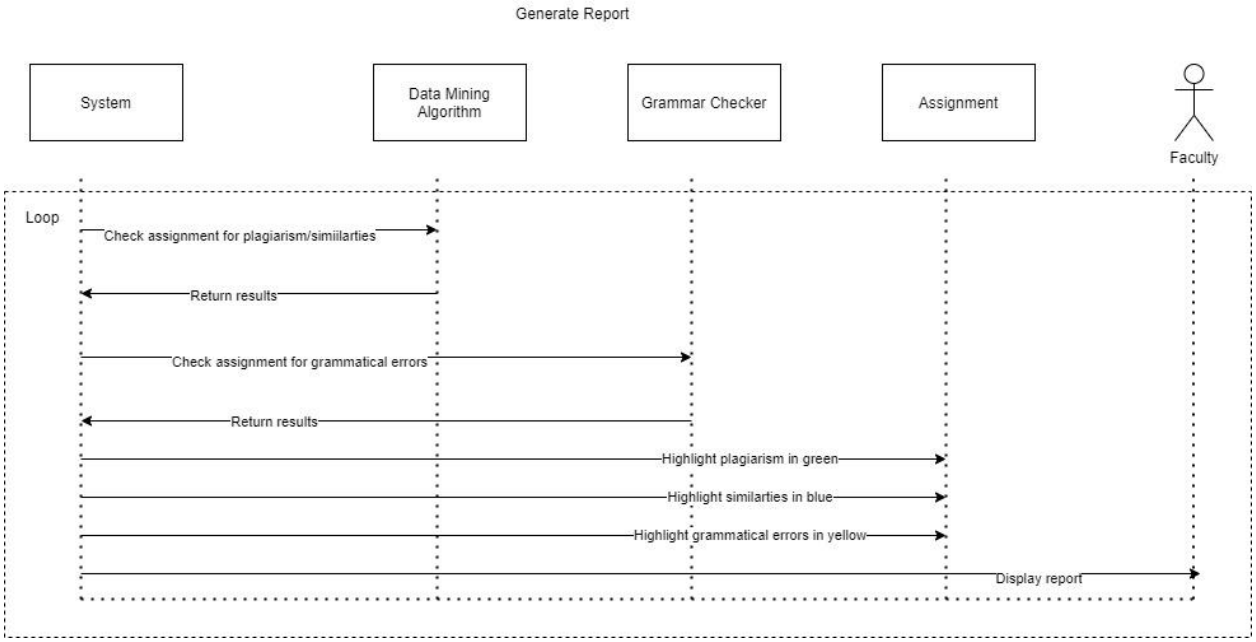
system sequence diagram, the system displays the login page to the user. Next, the system receives the event of the user clicking on the change password button. Then, the system displays the password requirements and required fields for the user to view. The system receives the user's email and desired new password as input. Lastly, the system validates the password and updates the system via the database. The system will loop through validating and receiving a new password until the user enters one that meets the requirements.



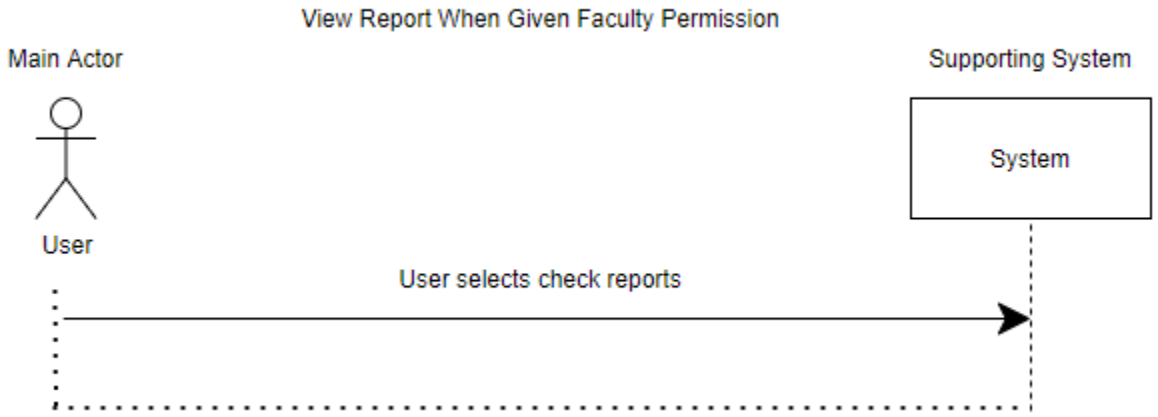
M4SD1. The main actor in the upload assignment system sequence diagram is the student. The supporting system is the system. In this sequence diagram, the user logs into TriScan and the system displays the course dashboard to the user. Next, the student selects a course and the system displays all available items that the student can submit. Then, the student selects the item they desire to submit and clicks the upload button near the assignment. Afterwards, the system displays the file explorer to the student and they local



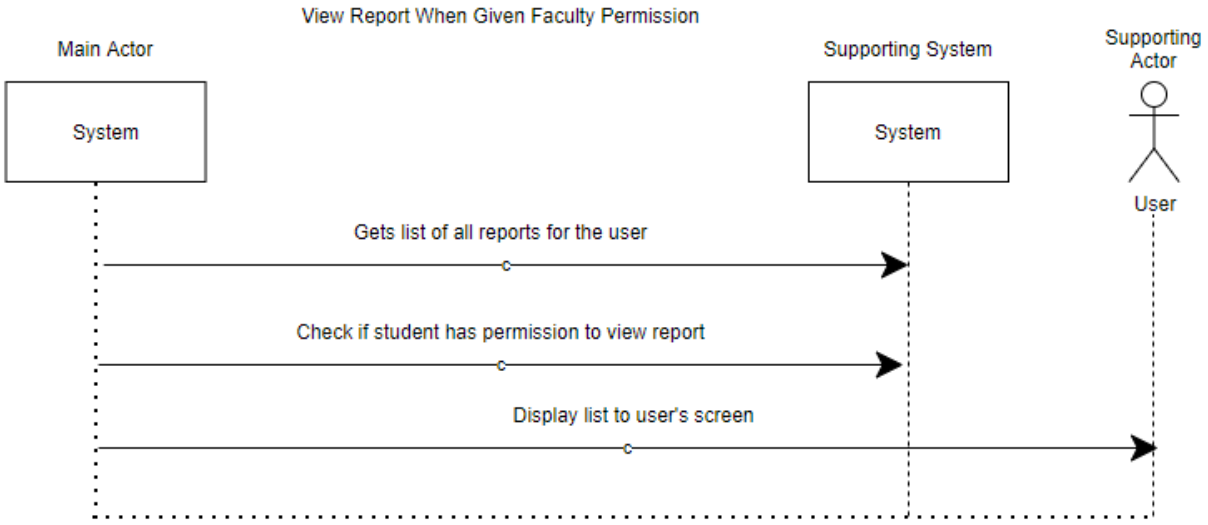
M4SD2. The main actor in the receive submissions system sequence diagram is the system. The supporting actor is the student and the supporting system is the database. In this sequence diagram, the system displays the course dashboard to the student and receives their course selection. Next, the system displays the items available for the student to submit in a given course. After receiving the item that the student selects, the system displays the file explorer. The user utilizes the file explorer to locate their file on their local computer and select submit. Lastly, the system saves the document to the database and also parses any handwriting information to text as additional data. Note, this functionality loops through all items that a student wants to submit



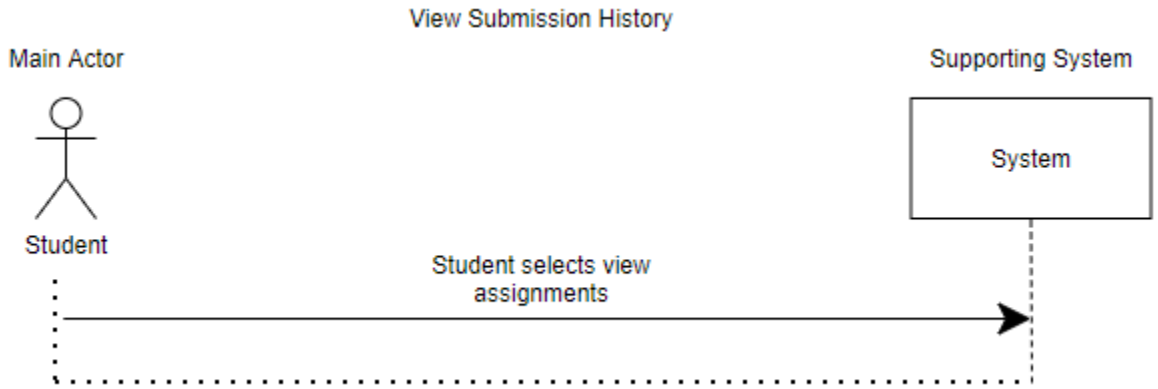
M5SD1. The main actor in the Generate Report SSD is the system. The supporting systems are a data mining algorithm, a grammar checker, the assignment, and a faculty member. The system checks the assignment for plagiarism, similarities, and grammatical errors. It then highlights the instances of plagiarism in green, the similarities in blue, and grammatical errors in yellow. Finally, it displays the report to the faculty. It completes this in a loop until all reports have been generated.



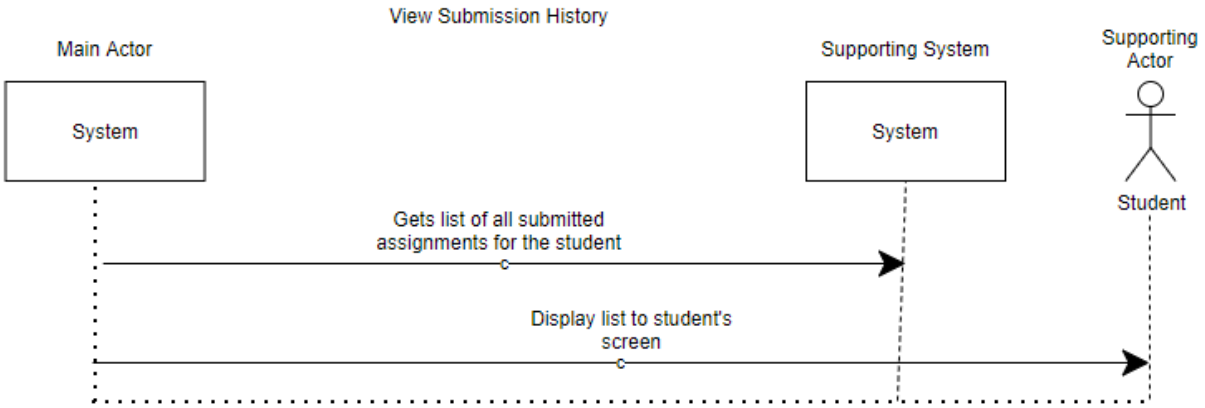
M6SD1. The main actor in the View Report When Given Faculty Permission (1) is the user. The supporting system is the system itself. The user selects to check reports and asks the system for information back.



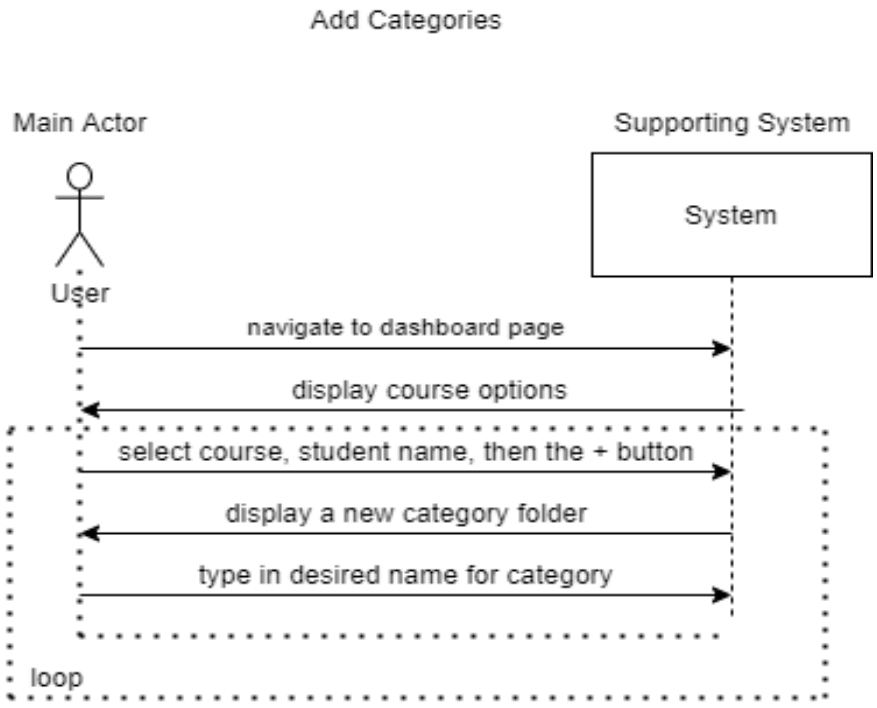
M6SD2. The main actor in the View Report When Given Faculty Permission (2) is the System. The supporting systems are the system itself and the user. After taking in a request to check reports, the system requests a list of reports that involve a user. It then checks to see if the user has permission to view each particular report. Then the system displays that list to the user's screen.



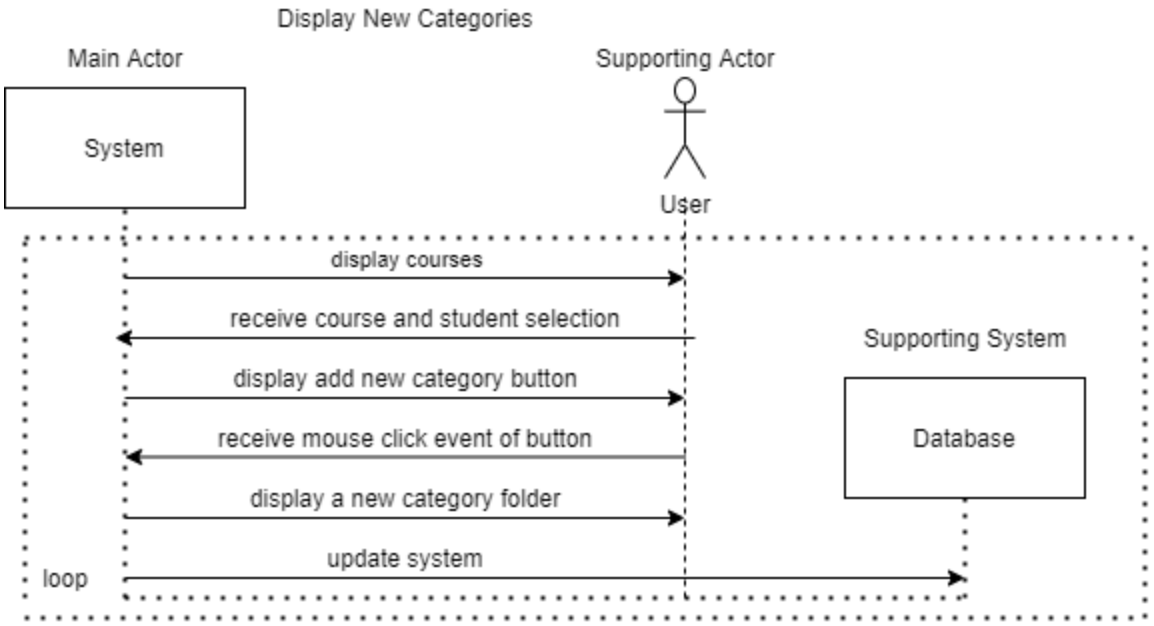
M7SD1. The main actor in the View Submission History (1) is the student. The supporting system is the system itself. The student selects to view assignments and asks the system for information back.



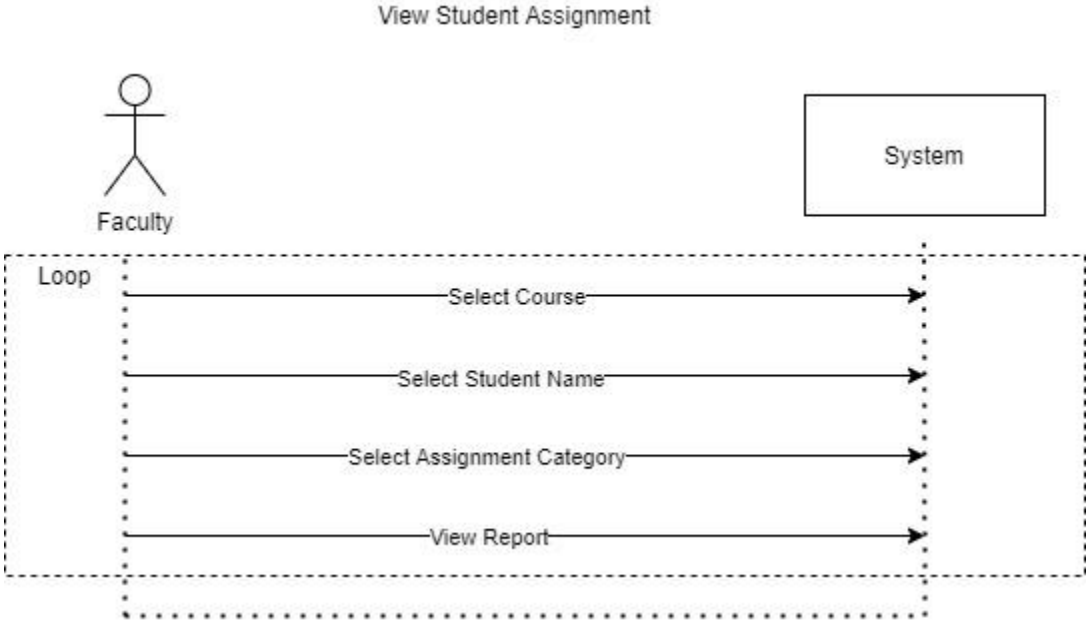
M7SD2. The main actor in the View Submission History (2) is the System. The supporting systems are the system itself and the student. After taking in a request to check reports, the system requests a list of all assignments turned in by the student. Then the system displays that list to the student's screen.



M8SD1. The main actor in the add categories system sequence diagram is the faculty. The supporting system is the system. In this sequence diagram, the user navigates to the dashboard and the system displays their course options. Then, the faculty selects a course, a student and the plus button to receive a new category. After the plus button is selected, the system displays a new category folder to the user and they can enter in a name for it. This process loops for as many categories the faculty desires to add.

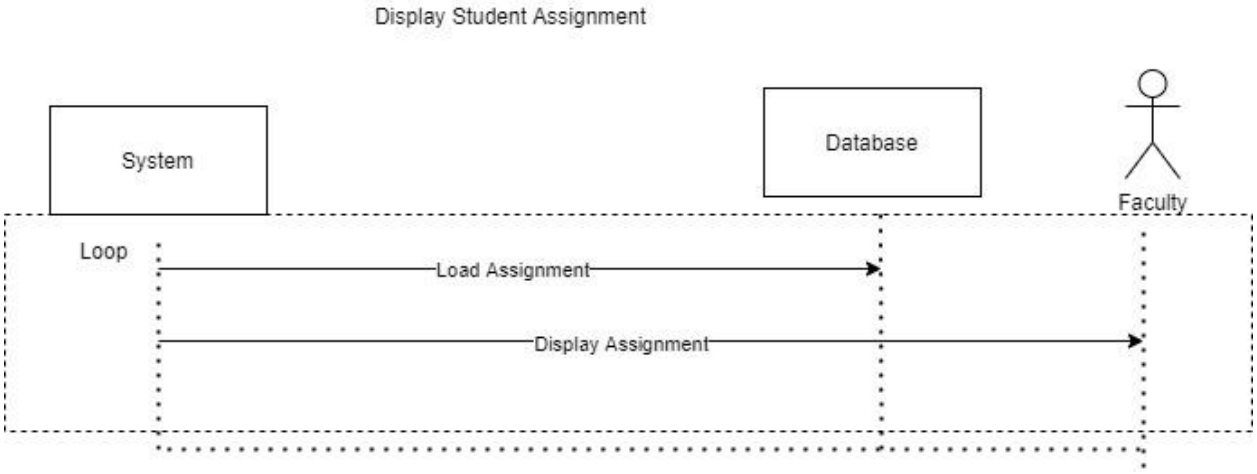


M8SD2. The main actor in the display new categories system sequence diagram is the system. The supporting actor is the faculty and the supporting system is the database. In this sequence diagram the system displays the courses to the faculty and receives the faculty's course and student selection. The system also displays the add new category button and receives the event of the button selection from the user. After the button has been selected, the system displays the new category folder to the faculty and updates the database. This process loops until the user has added their desired number of categories.

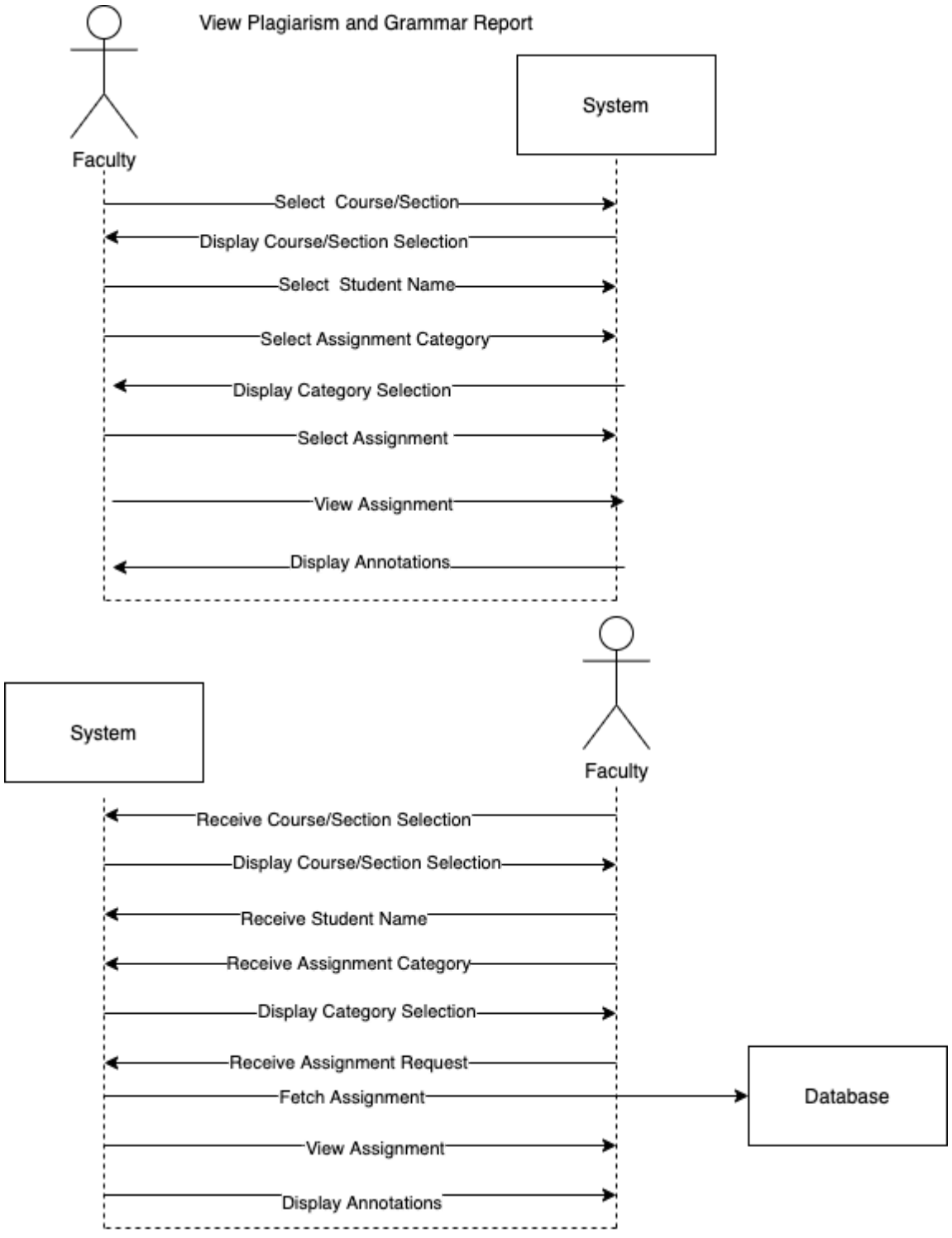


M9SD1. The main actor in the View Student Assignment SSD is faculty. The supporting system is the system. The faculty selects a course, a student's name, and the assignment

category. They then can view the report of the student's assignment. This can be continued in a loop until the faculty has completed viewing all student's reports.

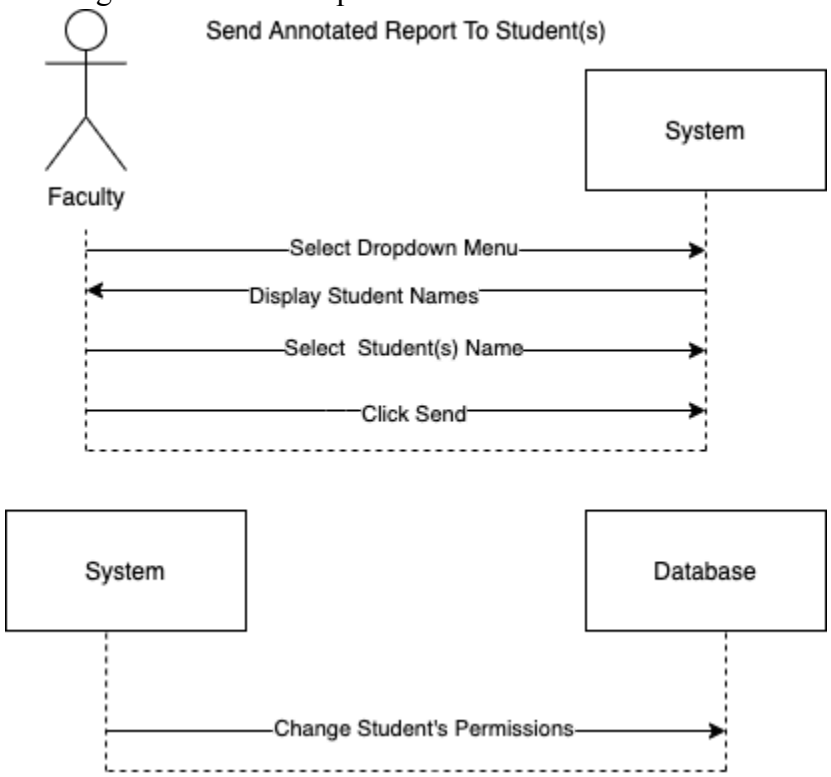


M9SD2. The main actor in the Display Student Assignment SSD is the system. The faculty member and the database are supporting systems. The system will load an assignment from the database and display it to the faculty member. This will continue in a loop until all assignments that are selected are displayed.



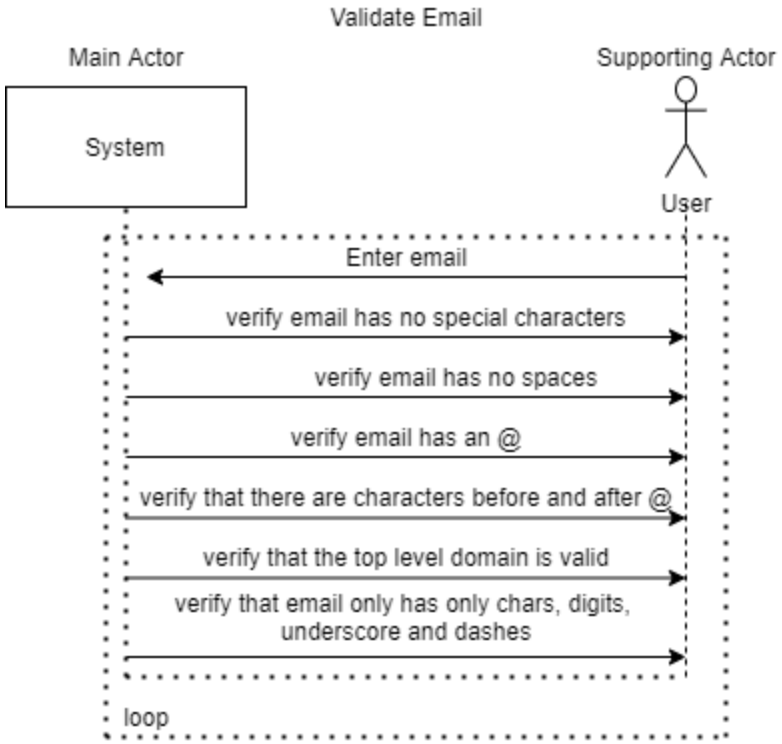
M10SD1. The main actors in the View Plagiarism and Grammar Report SSD are the faculty member and the system. The supporting systems in the SSD are the faculty, system and database. The Faculty member will select course, section, student name, category, and the assignment they want to view. The system will respond to each selection in displaying a different page after selecting each. Once the assignment is

selected the system will interact with the database to retrieve the assignment and display the assignment with the report.

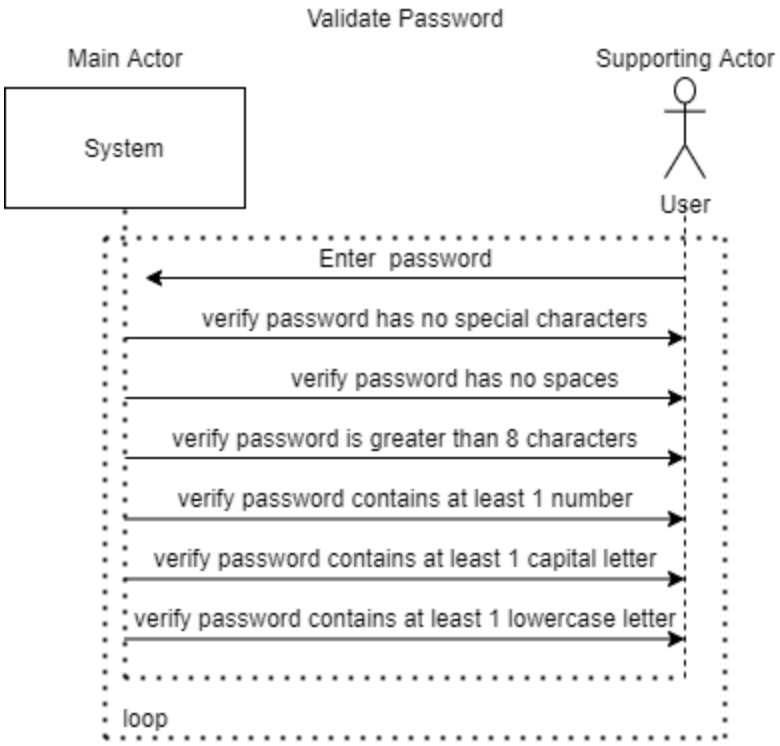


M11SD1. Faculty and the system are the main actors in the Send Annotated Report to Student(s) SSD. The supporting systems are the system and the database. The Faculty member will select the drop down menu and be provided a list of students in that class section/group. The faculty member will then be able to select one or more students to send the annotated report to. The system will then update the permissions for those students in the database.

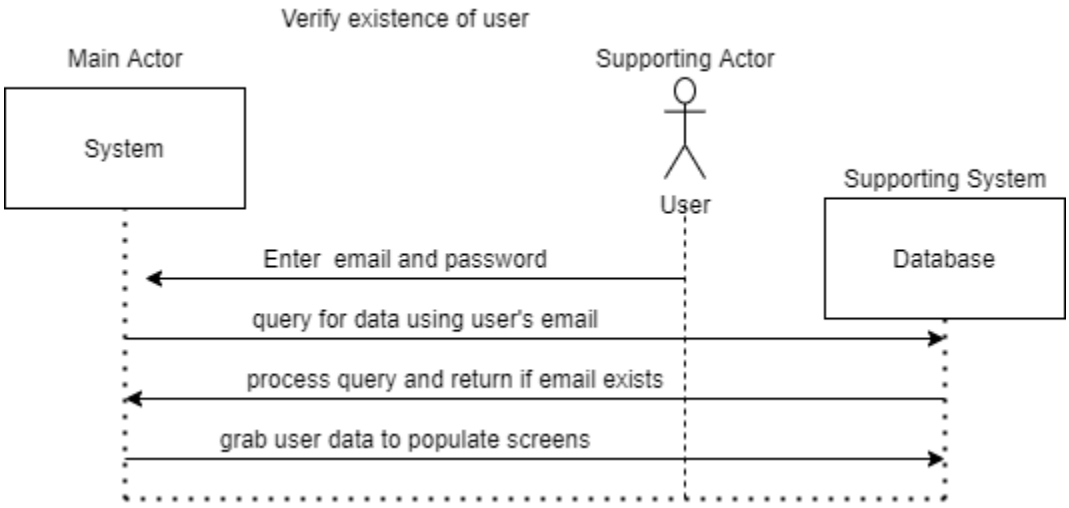
2.2. Alternative Sequence Diagrams



M1A1SD1. The system is the main actor in the validate email alternative system sequence diagram. The supporting actor is the user. In this sequence diagram, the user enters their email to the system. Then, the system grabs that input and verifies that the email has no special character, no spaces, and only characters, digits, underscores and dashes. Additionally, the system verifies that the email has an @ character, characters before and after the @ sign, and that the domain is valid. This functionality loops until the user enters a valid email while creating an account

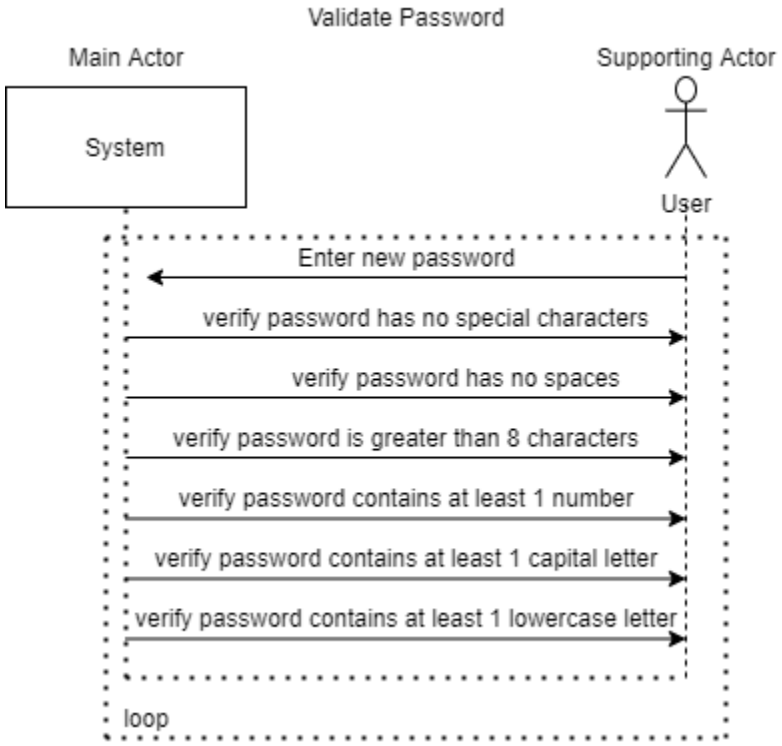


M1A2SD1. The system is the main actor in the validate password alternative system sequence diagram. The supporting actor is the user. In this sequence diagram, the user enters their new password to the system. Then, the system grabs that input and verifies that the password has no special character, no spaces, a length of at least 8 characters, at least 1 number, at least 1 capital letter, and at least 1 lowercase letter. This functionality loops until the user enters a valid password.

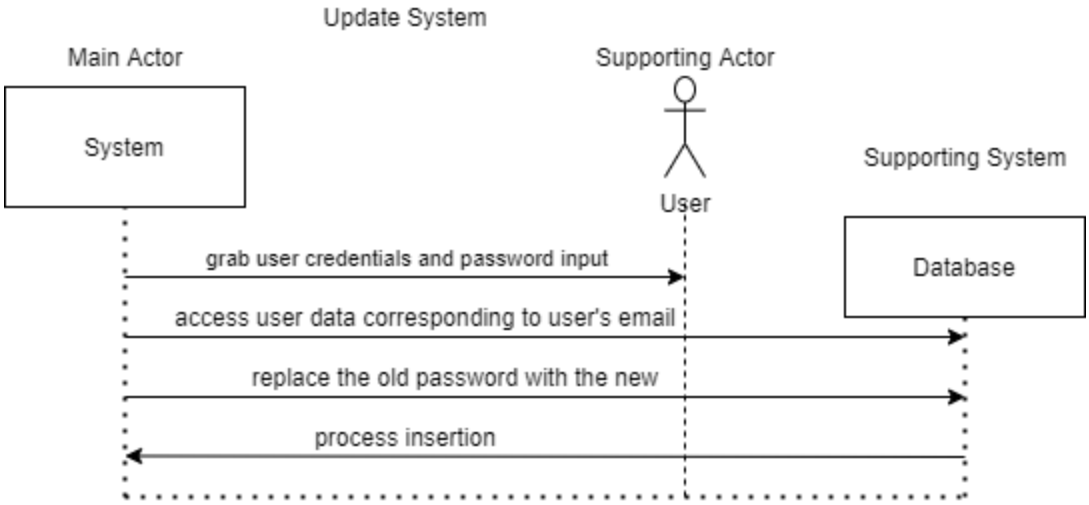


M2A1SD1. The system is the main actor in the verify credentials alternative system sequence diagram. The supporting actor is the user and the supporting system is the database. In this sequence diagram, the user enters their email and password to the

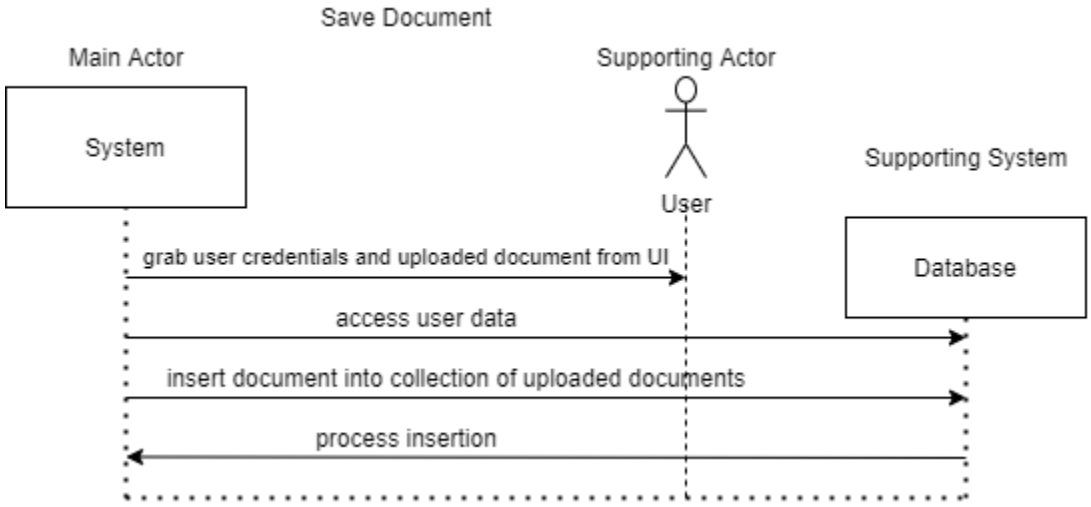
system. Then, the system grabs that input and queries for the existence of that user in the database using the user's email. Next, the database will process the query and return if the email exists in the database. Lastly, the system gathers the user data from the database to populate the user interface screens after login.



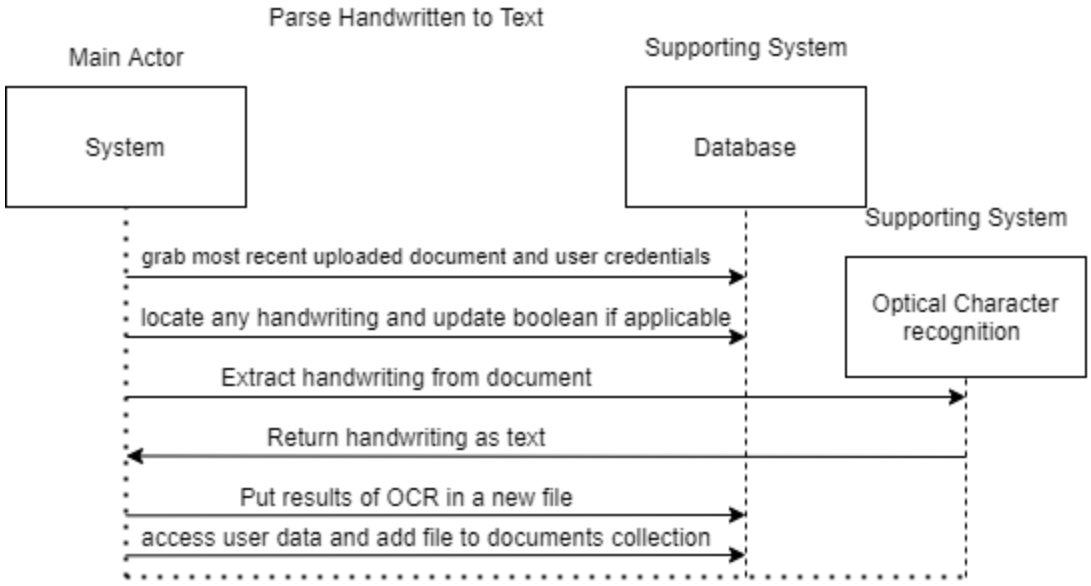
M3A1SD1. The system is the main actor in the validate password alternative system sequence diagram. The supporting actor is the user. In this sequence diagram, the user enters their new password to the system. Then, the system grabs that input and verifies that the password has no special character, no spaces, a length of at least 8 characters, at least 1 number, at least 1 capital letter, and at least 1 lowercase letter. This functionality loops until the user enters a valid password.



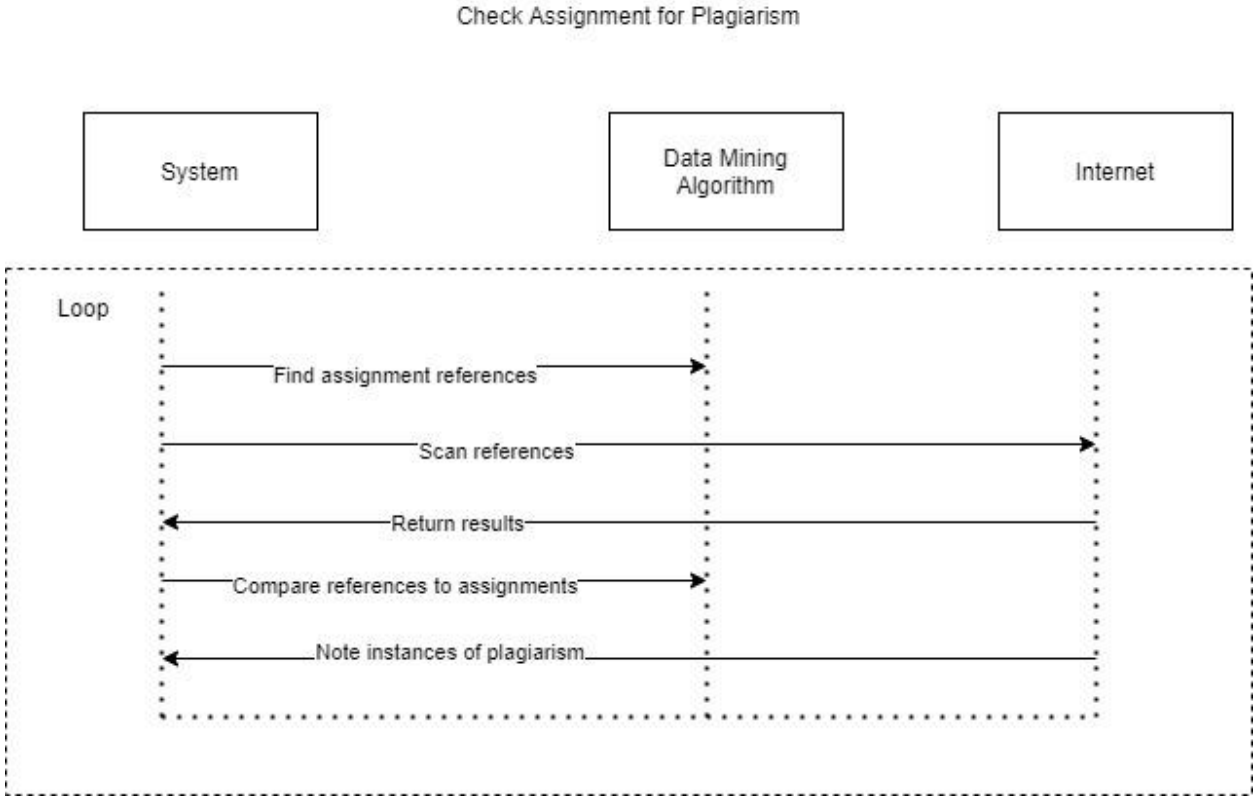
M3A2SD1. The system is the main actor in the update system alternative system sequence diagram. The supporting actor is the user and the supporting system is the database. In this sequence diagram, the system grabs the user's credentials and password input from the user interface. Next, the system accesses the user's data in the database using their email and replaces the prior password with the new one. Lastly, the database processes the insertion.



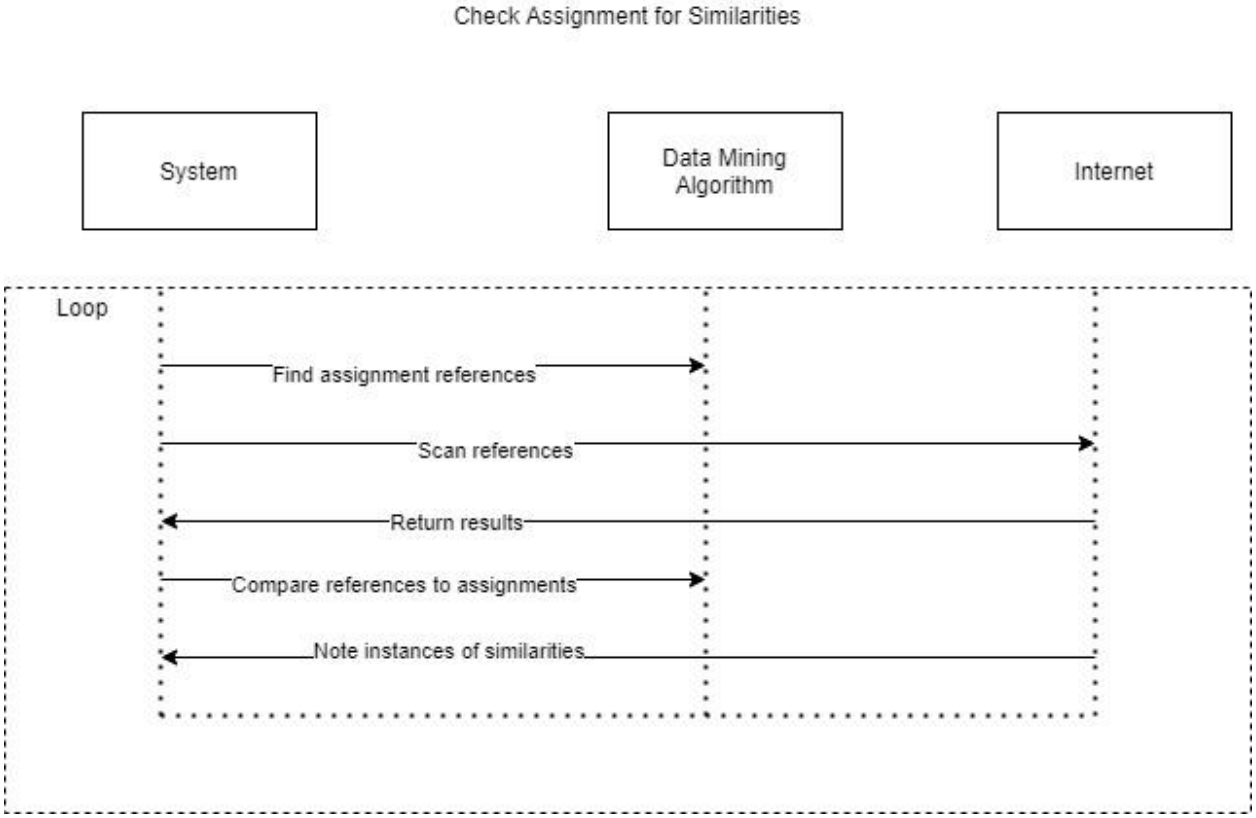
M4A1SD1. The system is the main actor in the save document alternative system sequence diagram. The supporting actor is the user and the supporting system is the database. In this sequence diagram, the system grabs the user's credentials and uploaded document from the user interface. Next, the system accesses the user's data in the database using their email and inserts the newly uploaded document into their collection of uploaded documents. Lastly, the database processes the insertion.



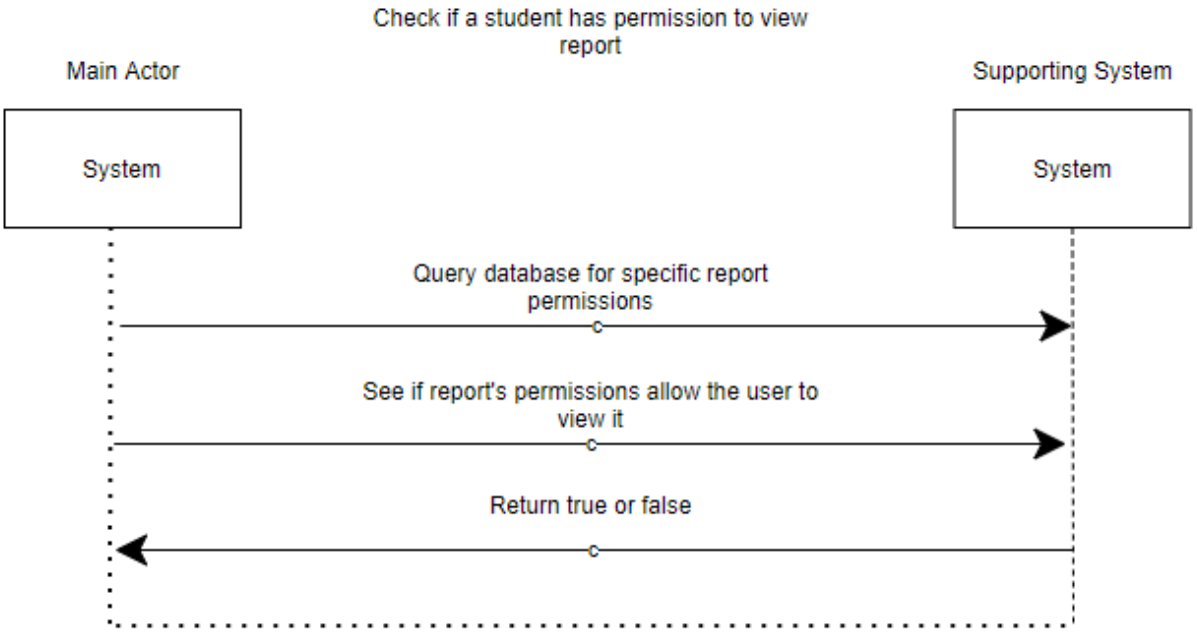
M4A2SD1. The system is the main actor in the parse handwriting to text alternative system sequence diagram. The supporting systems are the database and optical character recognition algorithm. In this sequence diagram, the system grabs the most recently uploaded document and the corresponding user's credentials from the database. Next, the system scans the document for handwriting and updates the flag in the data to be true if it finds any. Then, the system extracts the handwriting from the document by running it through the optical character recognition (OCR) algorithm. The OCR returns the extracted text and the system places those results in a file. Lastly, the system accesses the user's data and adds the new file to their collection of documents.



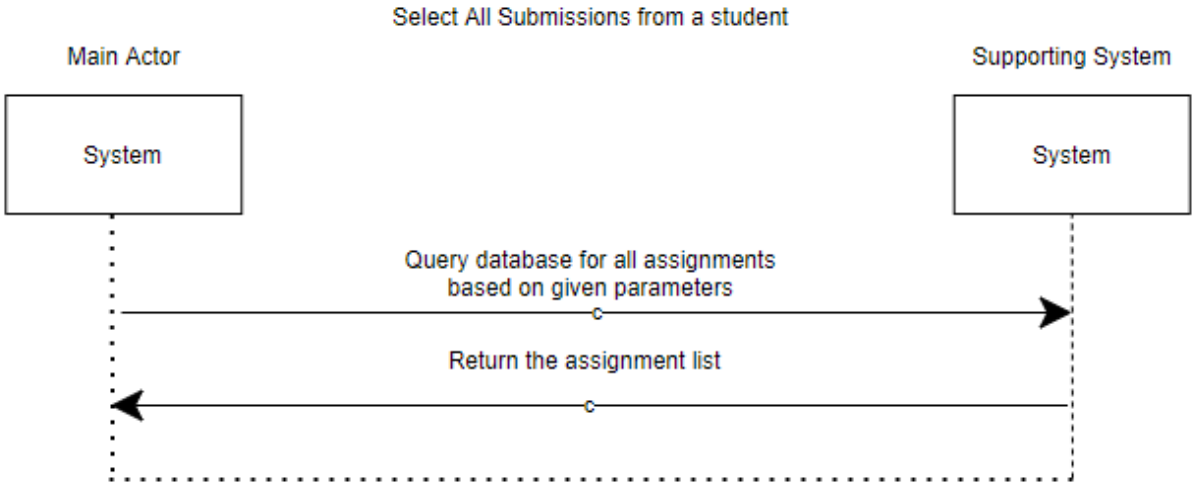
M5A1SD1. The system is the main actor in the Check Assignment for Plagiarism ASD. A data mining algorithm and the Internet are the supporting actors. The system finds the assignment's references, scans them, and compares them to the assignment. It then notes instances of plagiarism.



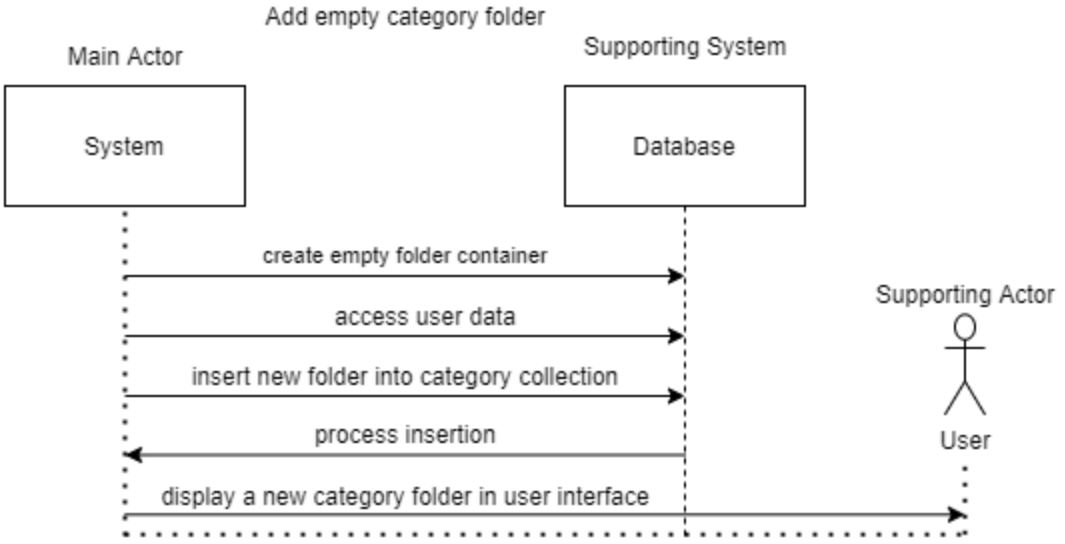
M5A2SD1. The system is the main actor in the Check Assignment for Similarities SSD. A data mining algorithm and the Internet are the supporting actors. The system finds the assignment's references, scans them, and compares them to the assignment. It then notes instances of similarities.



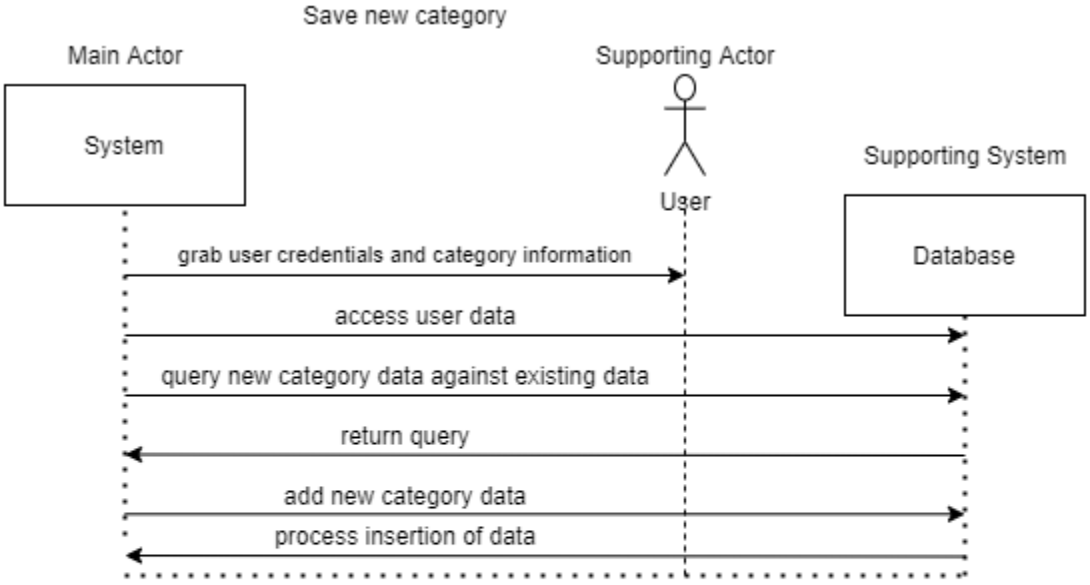
M6A1SD1. The system is the main actor in the Check if a student has permission to view report SSD. The supporting actor is the system. The system queries the database to view a report's permissions . It then checks to see if the user it's requesting for is in the permissions for said report. It returns a boolean if the user can or cannot access the report.



M7A1SD1. The system is the main actor in the Select All Submission from a student SSD. The system is the supporting system. The system queries the database for all assignments for a given student with optional parameters (classes, dates, sorting, etc). It then returns the list to the student.

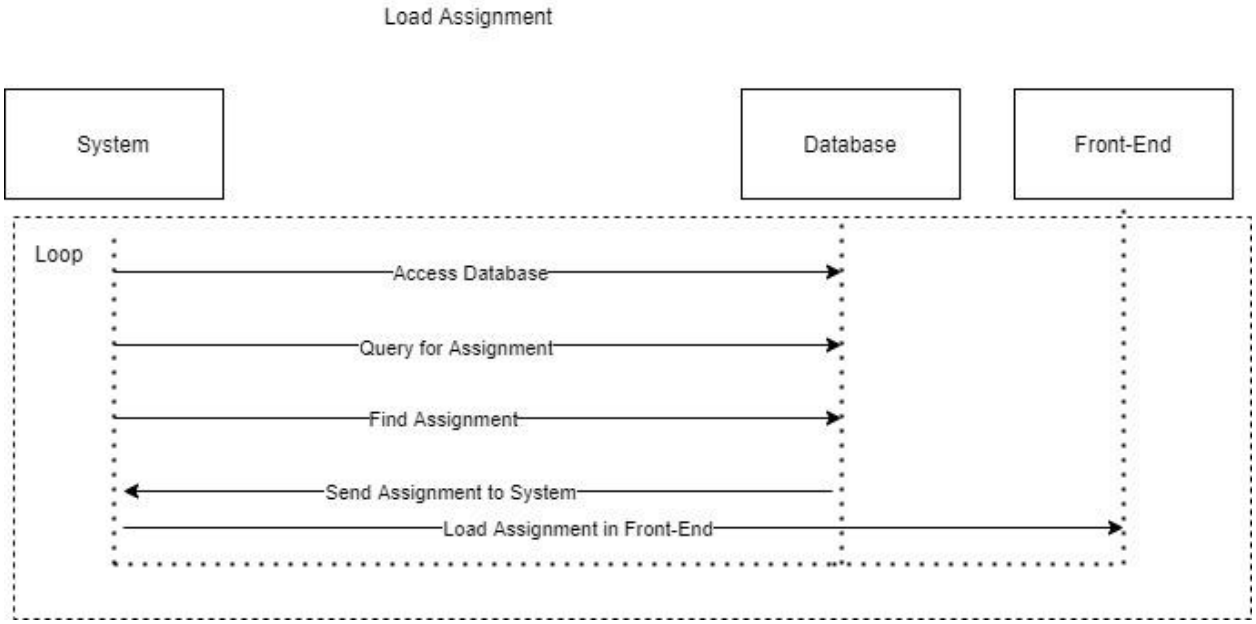


M8A1SD1. The system is the main actor in the add empty category alternative system sequence diagram. The supporting system is the database and the supporting actor is the faculty. In this sequence diagram, the system creates an empty category folder. Then, the system accesses the user's data and adds the new category folder to their list of categories. The database processes the insertion and then the system displays the new category to the faculty in the user interface for them to rename.

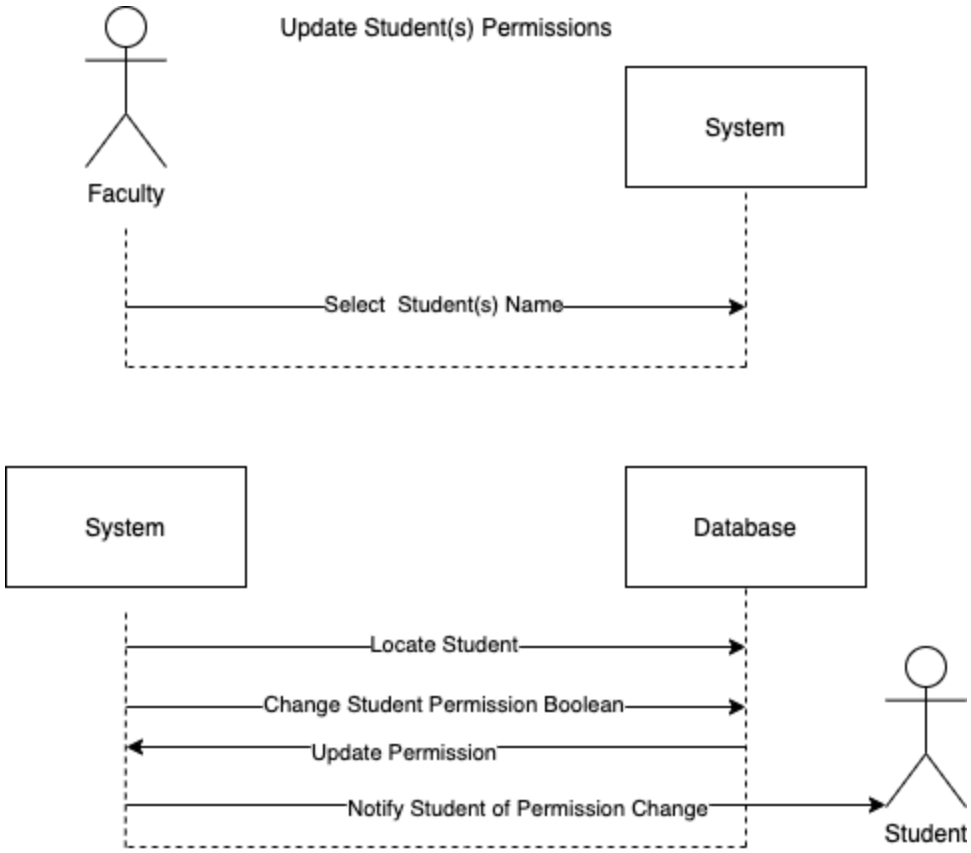


M8A2SD1. The system is the main actor in the add empty category alternative system sequence diagram. The supporting system is the database and the supporting actor is the faculty. In this sequence diagram, the system grabs the user credentials and category information. Then, the system accesses the user's data and queries the new category data

against the existing data. The database returns the query and the system adds the new data in. The database processes the insertion of data.



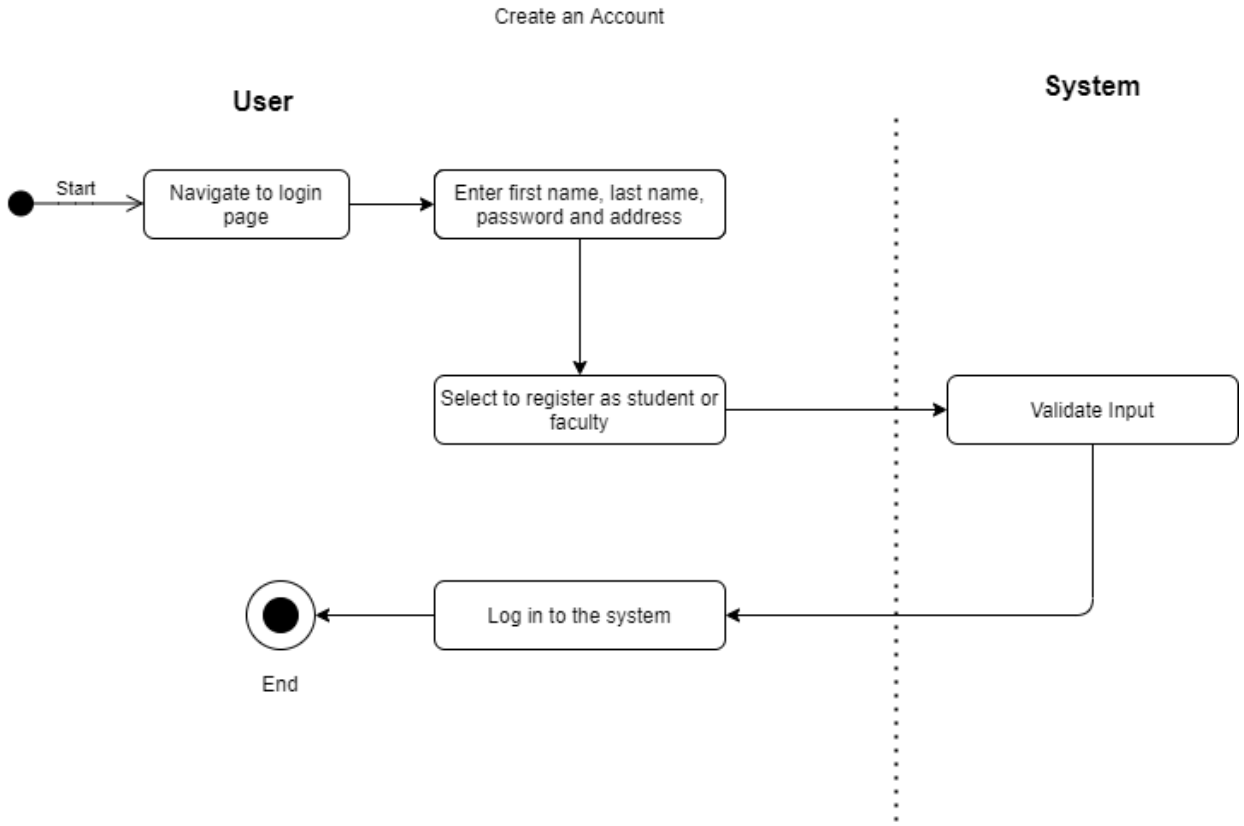
M9A1SD1. The system is the main actor in the Load Assignment ASD. The database and front-end are supporting systems. The system will access the database, query for the assignment, receive the assignment, and load it into the front-end. This will continue in a loop until all assignments have been loaded.



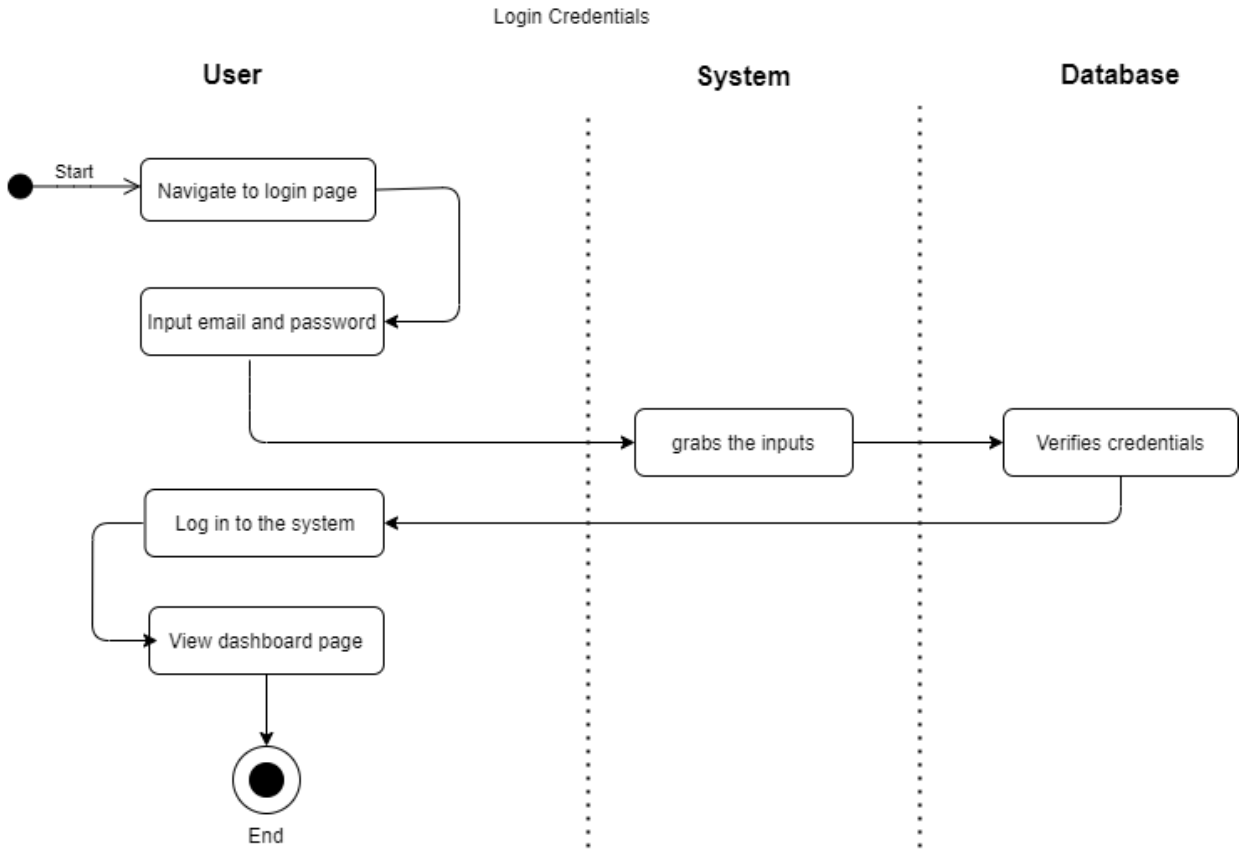
M11A1SD1. In the Update Student(s) Permissions ASD, the main actors are the faculty and the system. The supporting systems are the system and the database. Once the faculty member has selected the student(s) name(s) and has clicked "send" the system will contact the database to locate the student(s) and change the permission boolean for those students. Then the database will respond with the updated permissions allowing the system to notify the student.

3. Activity Diagrams

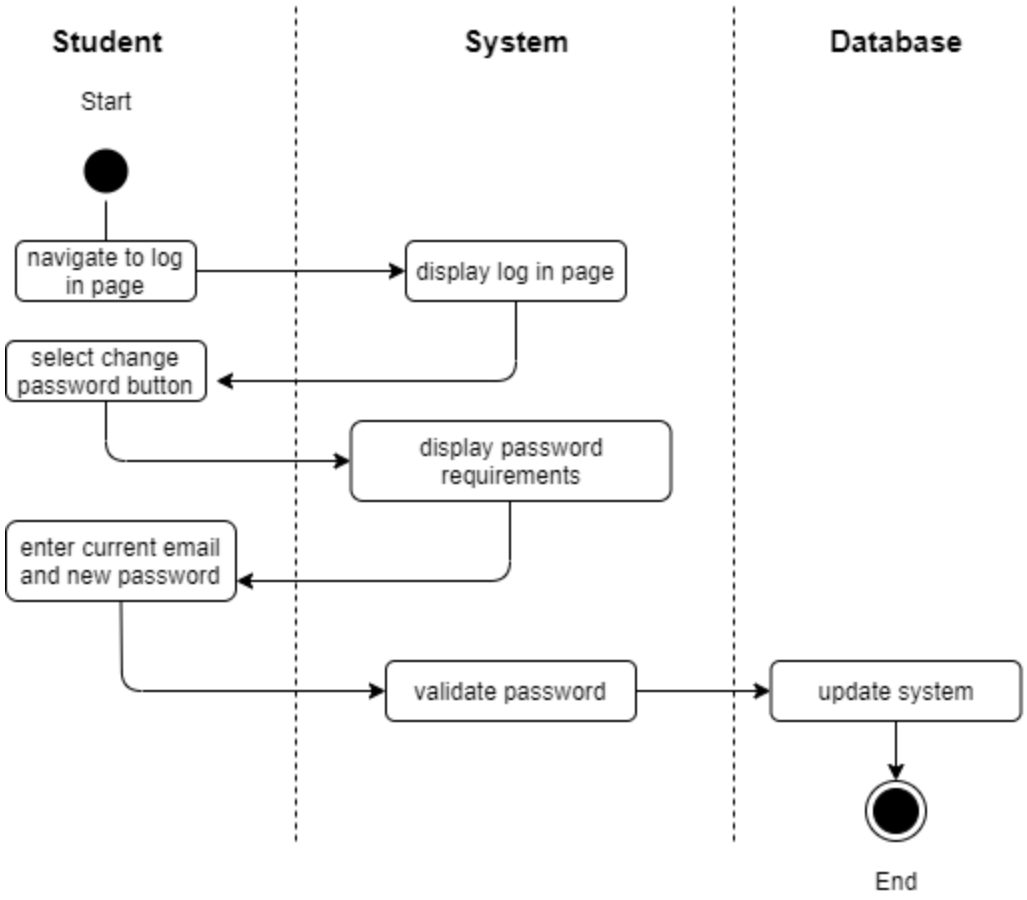
3.1. Main Activity Diagrams



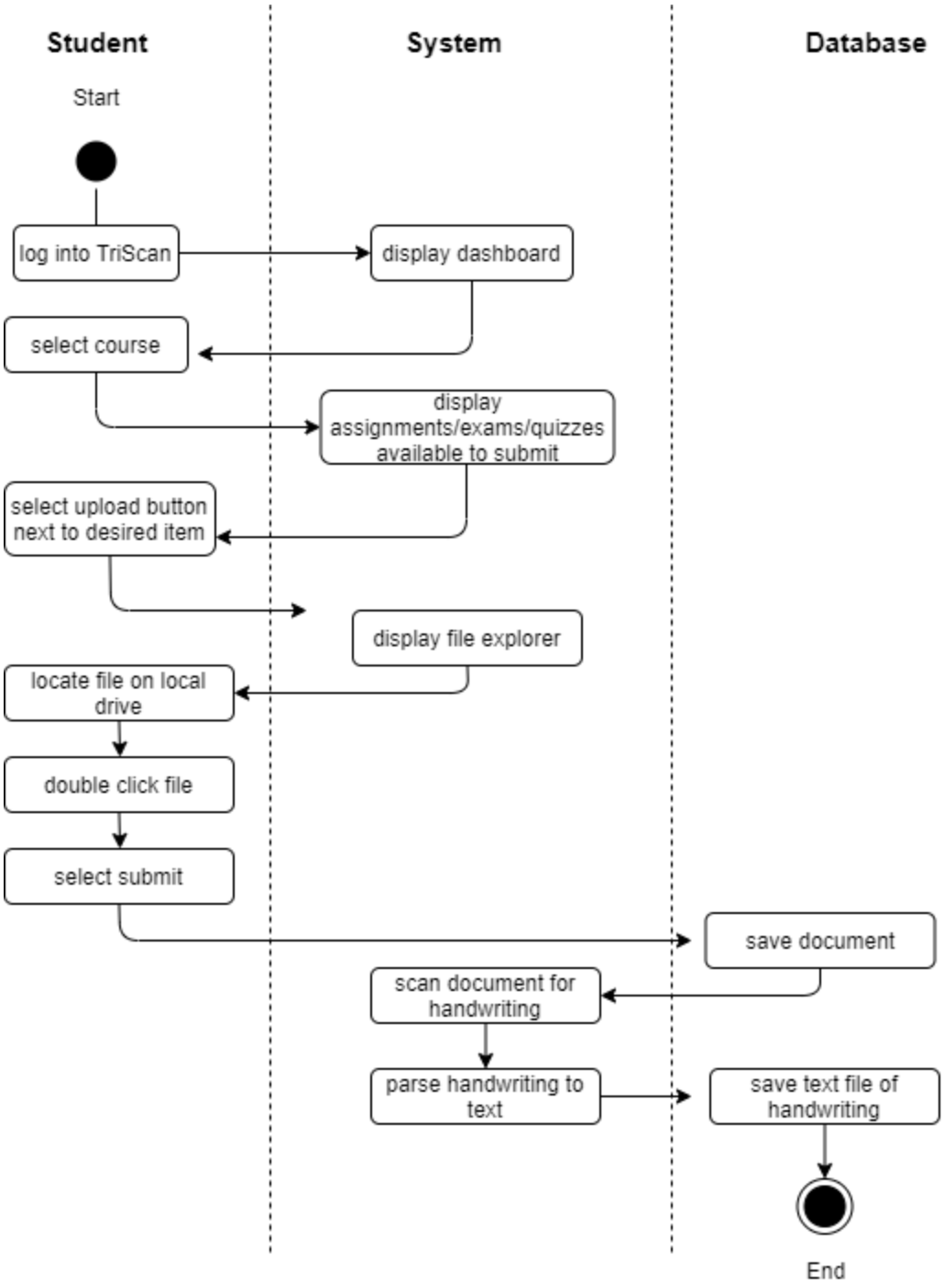
M1. In this activity diagram, the student goes to the login page, clicks on the create account button, and enters their first name, last name, email address, and password. Next, the user selects their registration as a student or faculty. Then, the system validates the user's input and the user is able to log into the system.



M2. In this activity diagram, the student goes to the login page and enters their email address and password. The system then gathers the user's input and verifies it against the database. Aftwards, the user is able to log into the system and view the dashboard page.

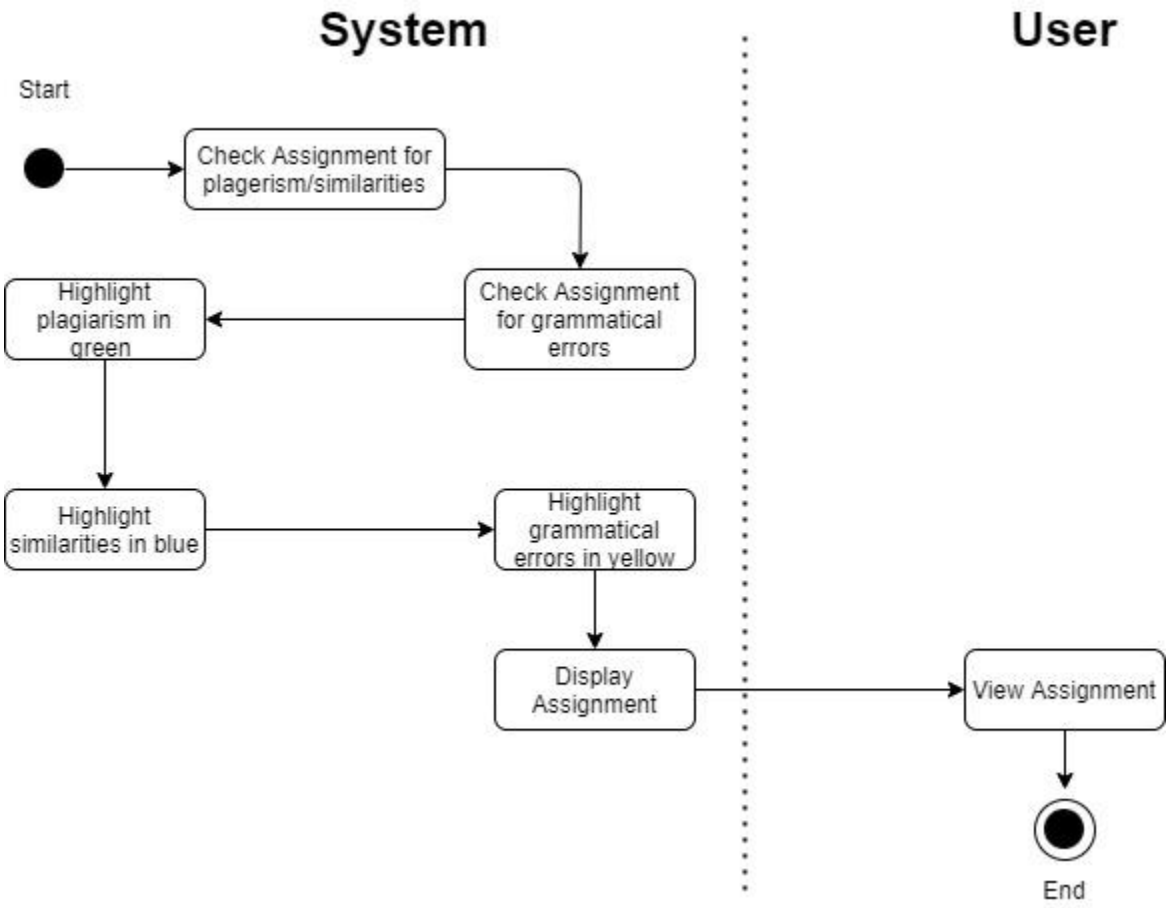


M3. In this activity diagram, the student goes to the login page and clicks the change password button. The system displays the requirements for a new password and the student is able to enter their current email and new password. After the system validates the password, the database is updated to reflect the changes.

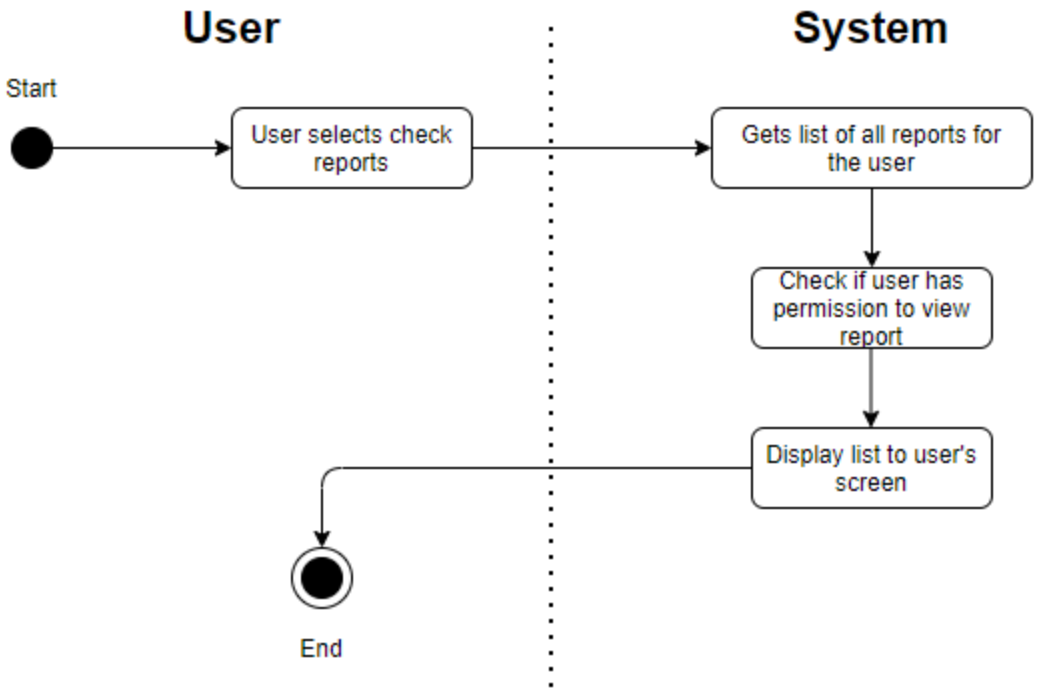


M4. In this activity diagram, the student logs into TriScan and selects a course from their dashboard. Next, the system displays all available assignments, exams, and quizzes to submit. Then, the student selects an upload button next to the item they wish to upload and the system provides a file explorer. The student locates the file on their local drive, double clicks it and selects submit. Then, the file is saved to the database and the system

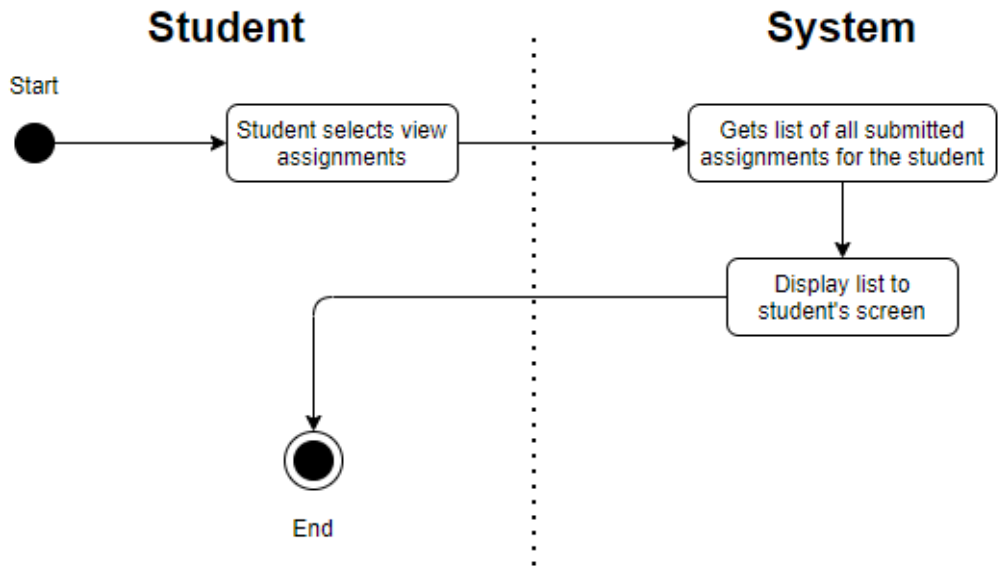
scans the document for handwriting and parses any handwritten information to text if found. Lastly, the parsed handwriting is also saved as a text file to the database as well.



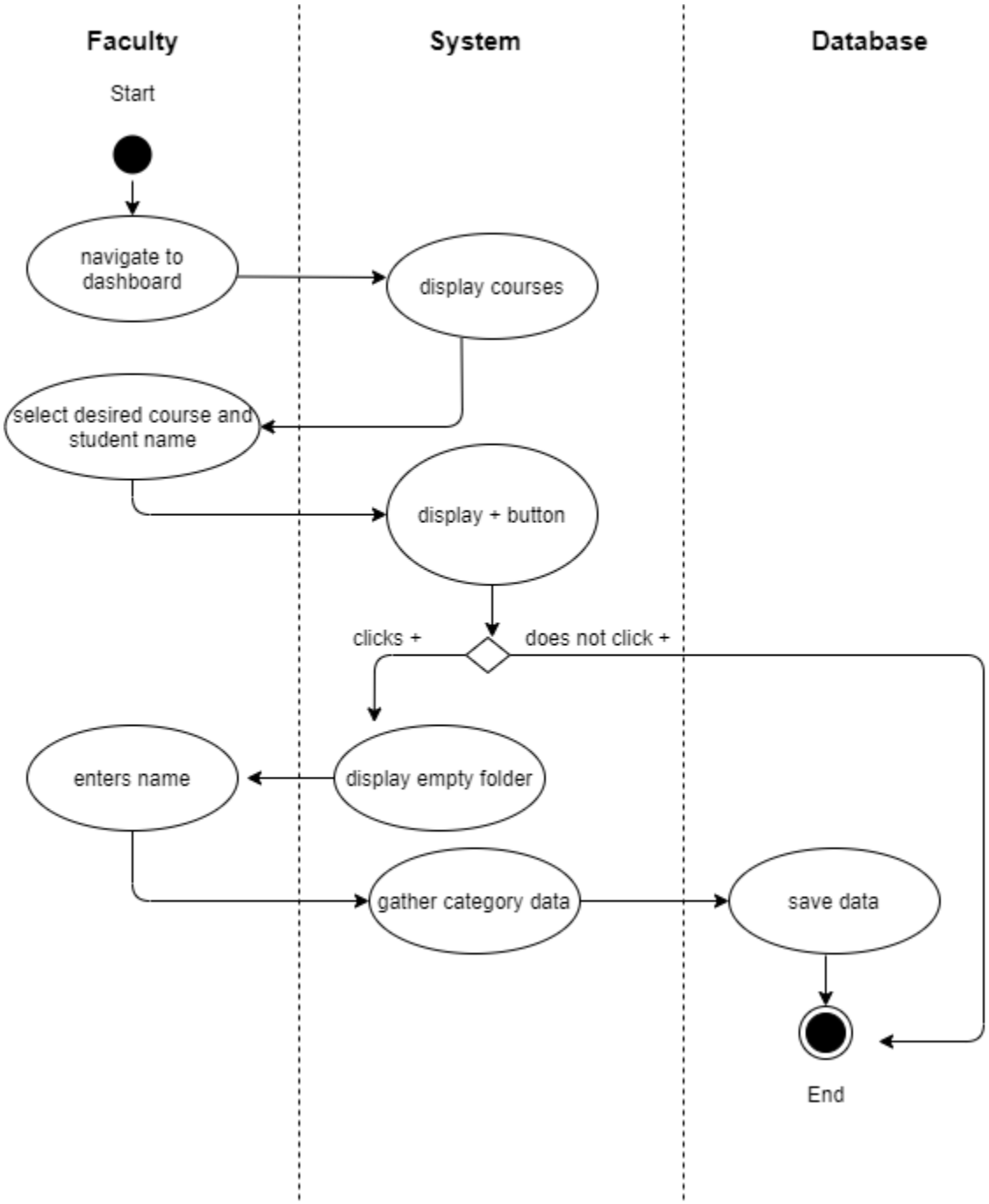
M5. In this activity diagram the system checks the assignment for plagiarism, similarities, and grammatical errors. It then highlights the plagiarism in green, the similarities in blue, and the grammatical errors in yellow. It then displays the assignment to the user who can view it.



M6. In this activity diagram the user requests to see a list of reports they can view. The system gets a list of all the reports for a given user, checks if they have permissions to access the report and then displays the reports list to the student.

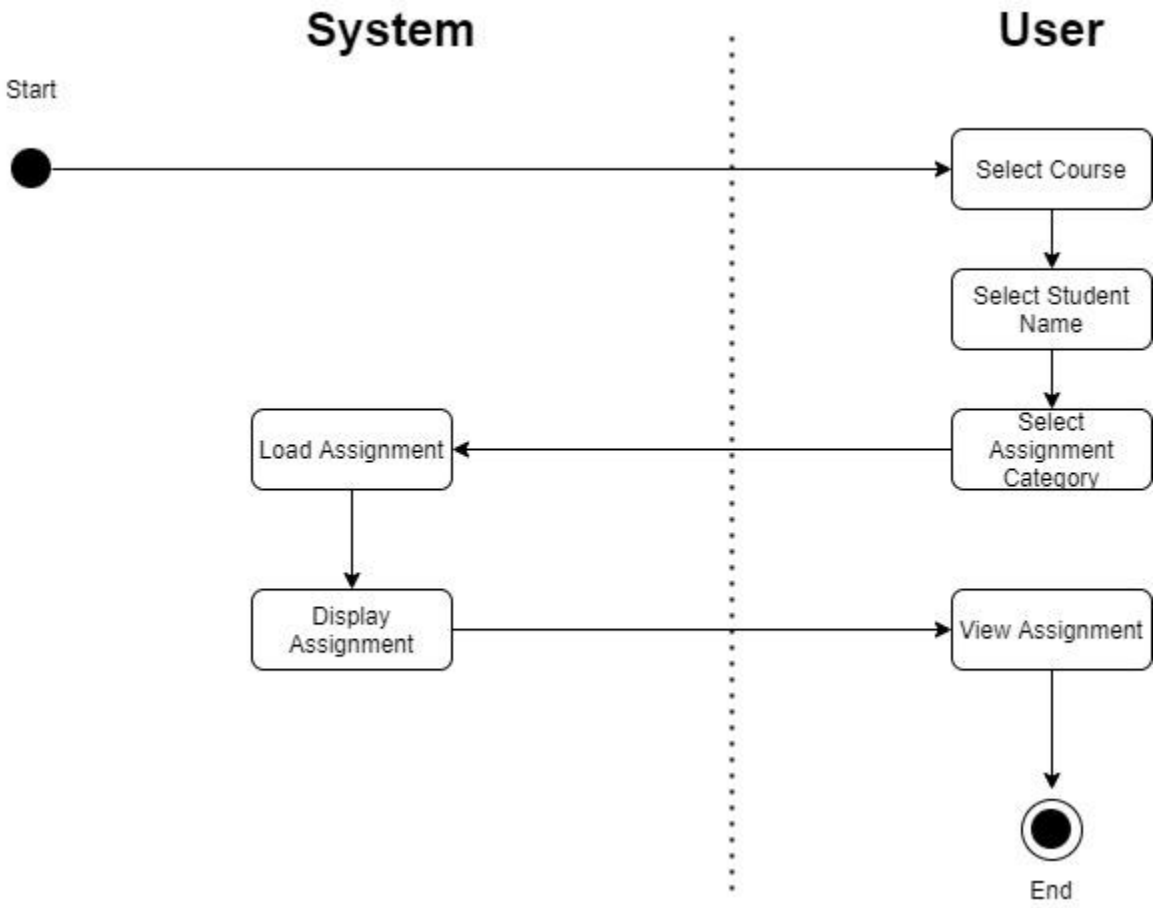


M7. In this activity diagram the student requests to see a list of all previous submissions. The system gets a list of all the assignments given to a user, checks if they have submitted a response to each assignment, and then displays the assignments with submissions back to the student.

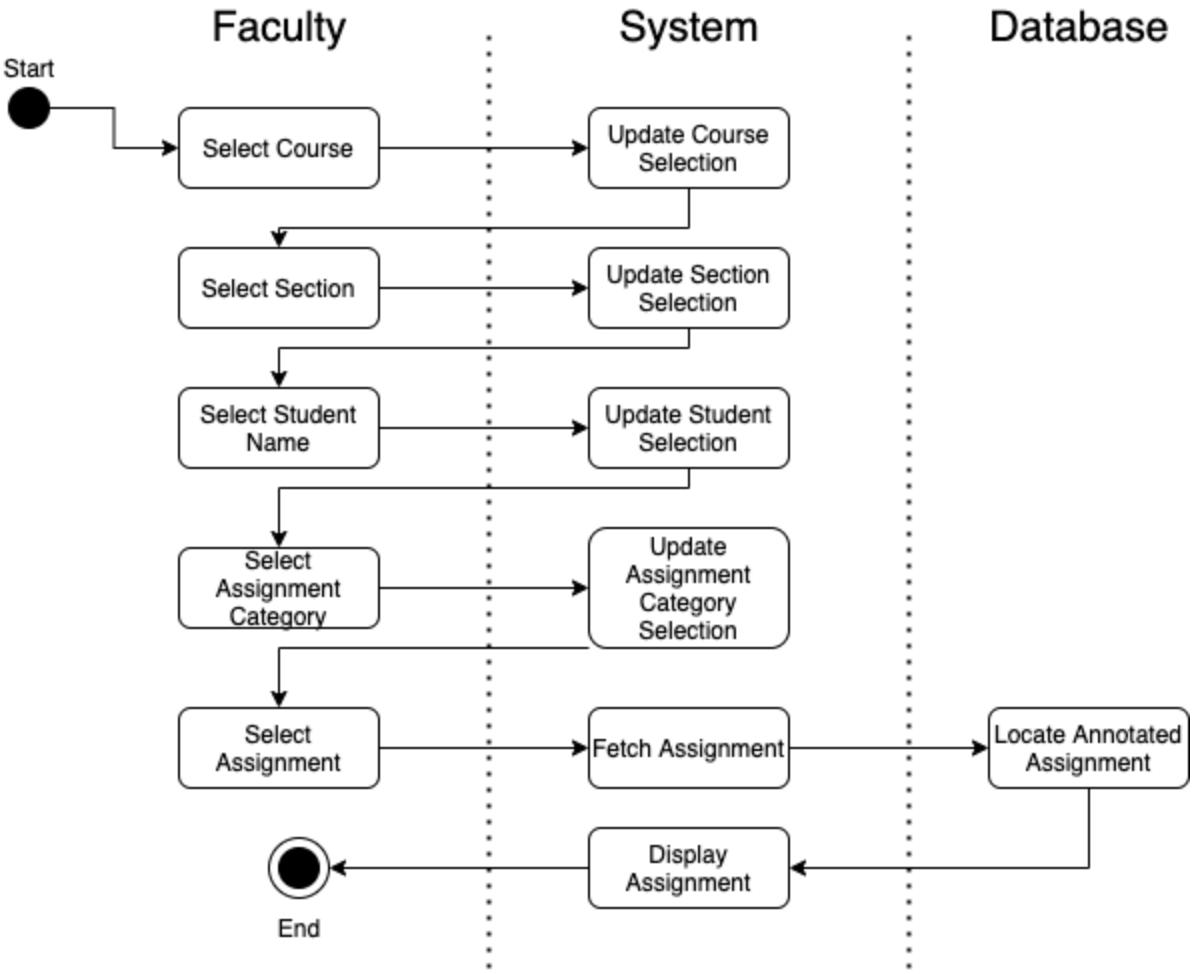


M8. In this activity diagram, the faculty goes to their dashboard and selects a course and student name. The system displays the add category button and if the faculty clicks it, the

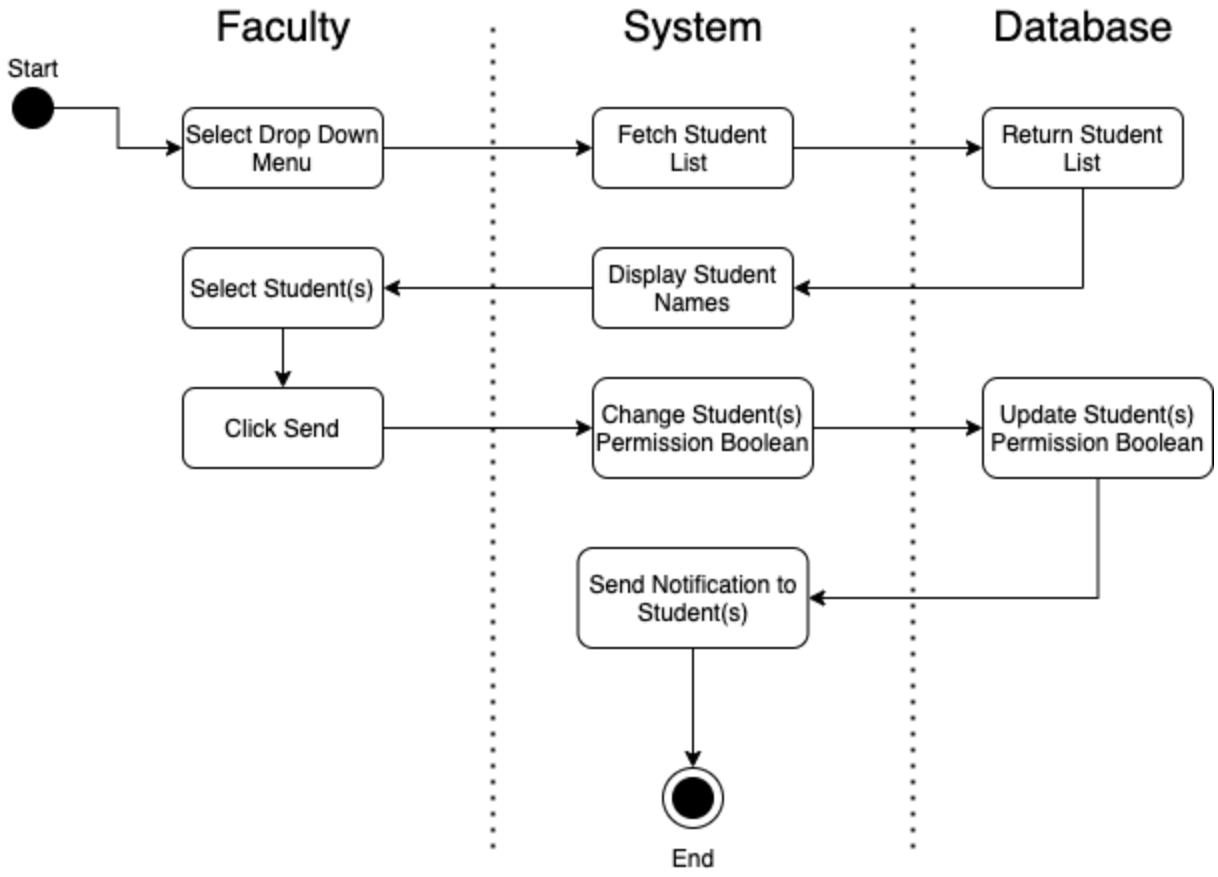
system will display an empty folder that the faculty member can rename. Lastly, the system will gather the category data and save it to the database.



M9. In this activity diagram the user selects the course, student, and assignment category of the assignment. The system then loads the assignment and displays it to the user who can view it.

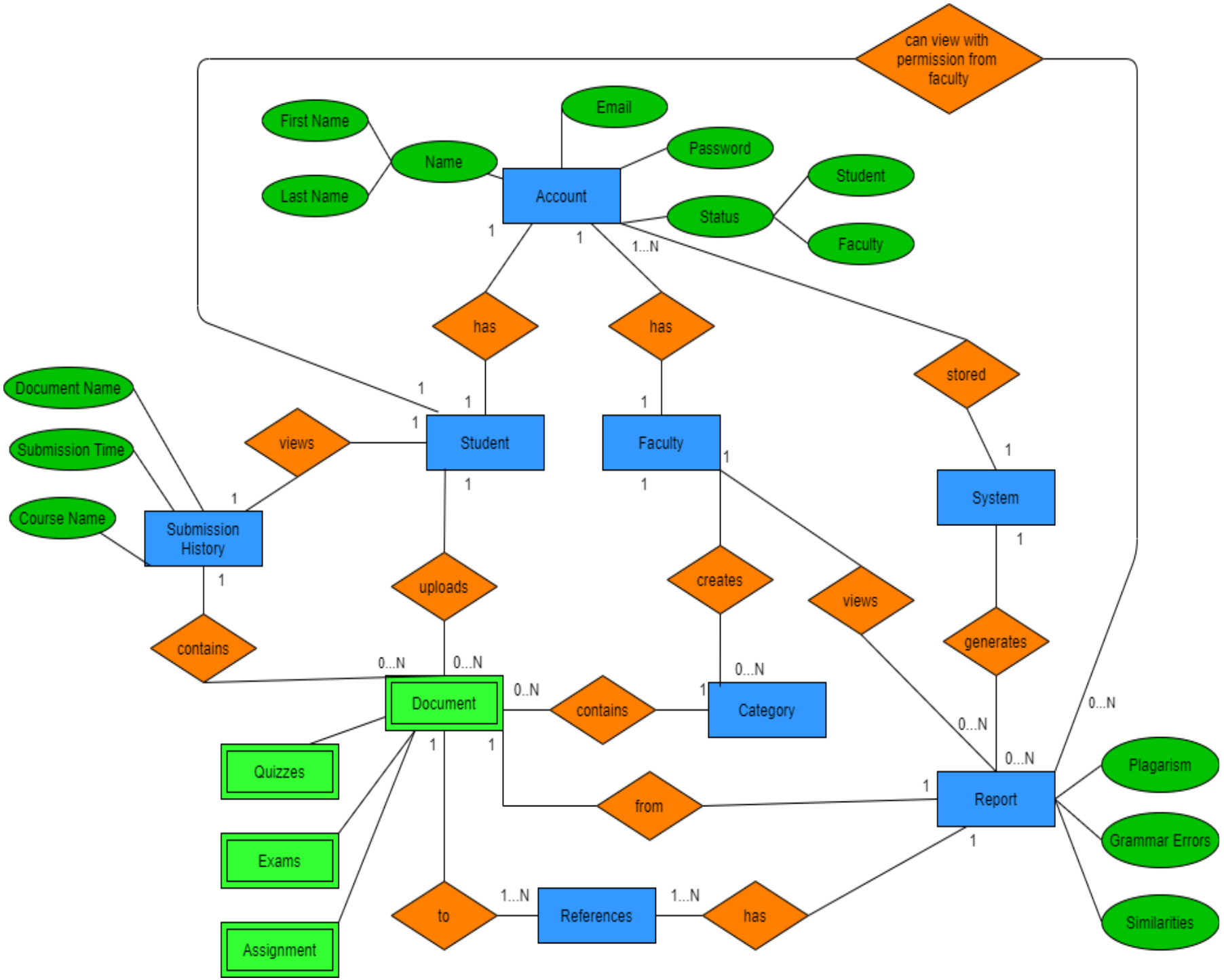


M10. In this activity diagram the faculty member will select a course, section, student name, assignment category and finally the assignment name. The system will keep track of each selection and fetch the assignment from the database with the selected “filters”. The database will then return the selected annotated assignment and the system will then display it for the faculty member.



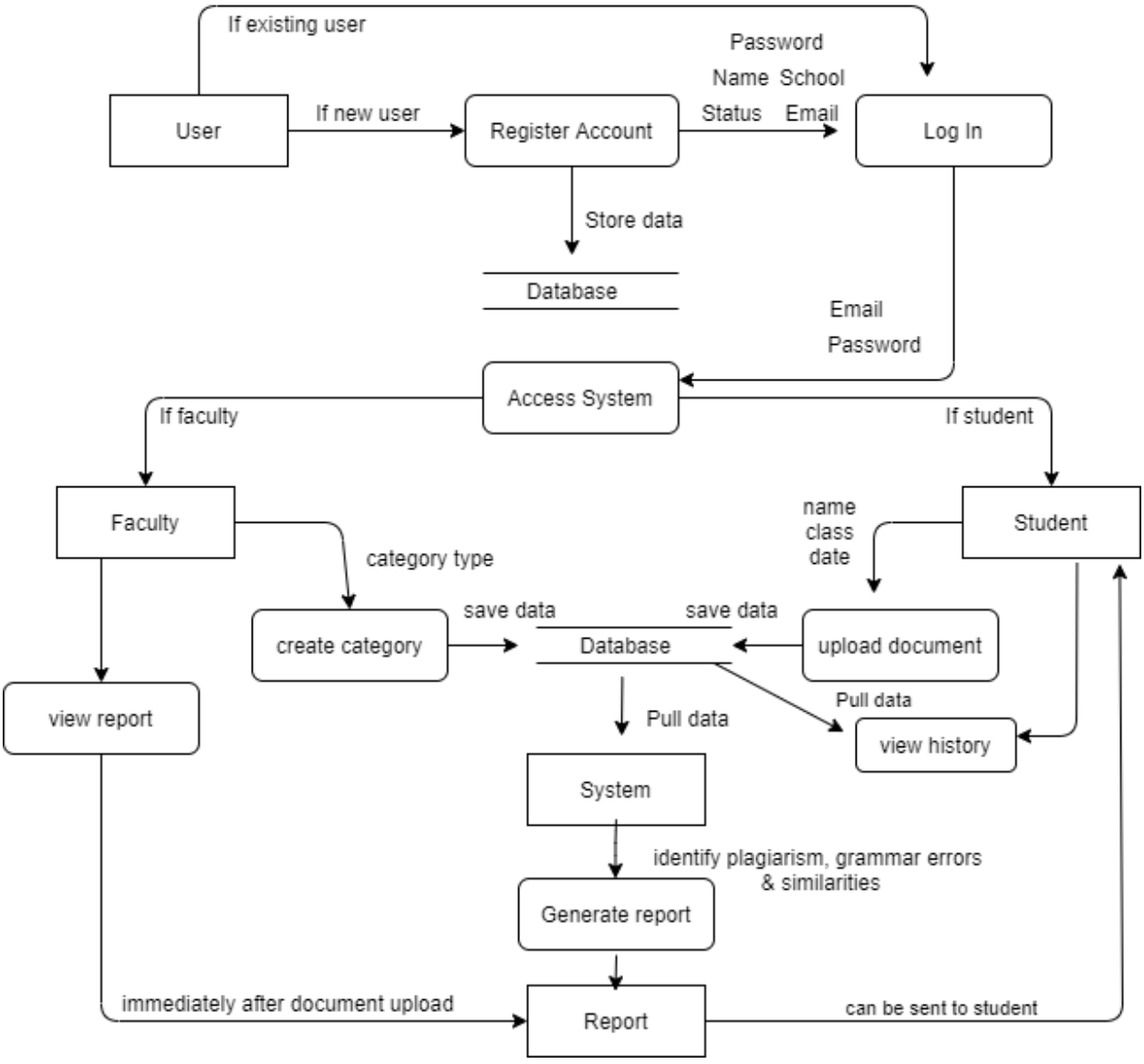
M11. In this activity diagram, the Faculty member will select the dropdown menu. The system will fetch the list of names from the database of students who are in the class and section. The system will then display the names of the students in the drop down menu where the faculty member will then select the students they would like to send the report to. When they have made their selection they will click send and the system will change the permissions of each of those students in the database. Once the database has been updated the system will send a notification to the students that their permissions have been changed.

4. Entity-Relationship Diagrams


can view with permission from faculty
First Name
Last Name
Name
Email
Password
Account
Student
Status
Faculty
1
1
1...N
has
has
stored
Document Name
Submission Time
Course Name
views
Student
Faculty
System
Submission History
uploads
creates
views
generates
contains
0...N
0...N
Document
0..N
contains
Category
Quizzes
Exams
Assignment
from
Report
Plagarism
Grammar Errors
Similarities
to
1...N
References
1...N
has

This Entity Relationship diagram illustrates the interaction between the student, faculty and system. A student has an account, can upload a document, view their submission history, and can conditionally view a report when given faculty permission. A faculty member has an account, can create a category, and view the reports. The system will store the account data and will generate the report from the documents. A report will contain the references as well.

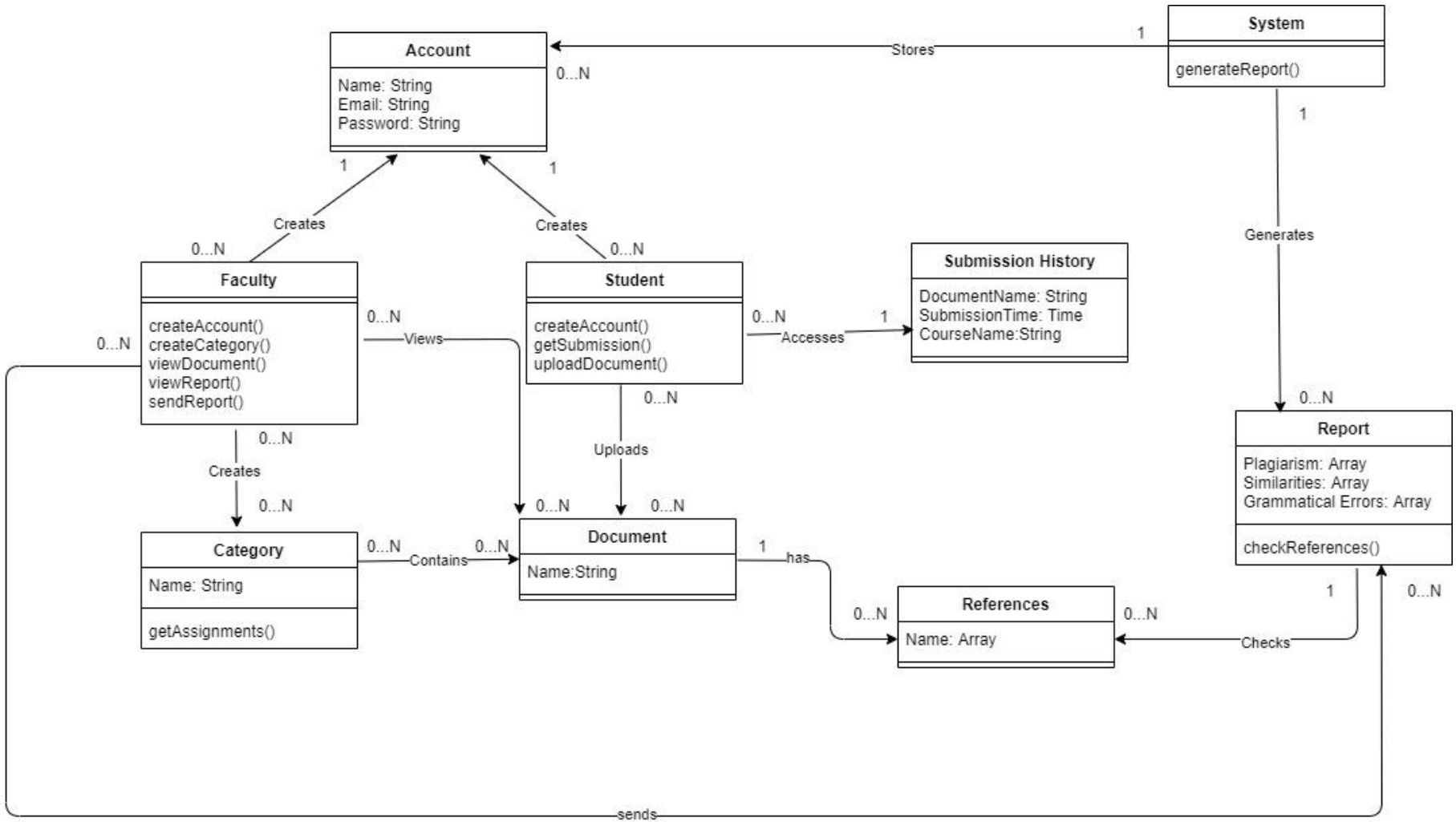
5. Dataflow Diagram



In this dataflow diagram, the flow begins with the user. If they are a new user, they must register an account which stores their email, password, name, school and status. Next, they can log in using their email and password to access the system. Then, the flow splits between the two statuses: student and faculty. If the user has the status of a student, they can upload a document containing the name of the document, the course it is assigned to

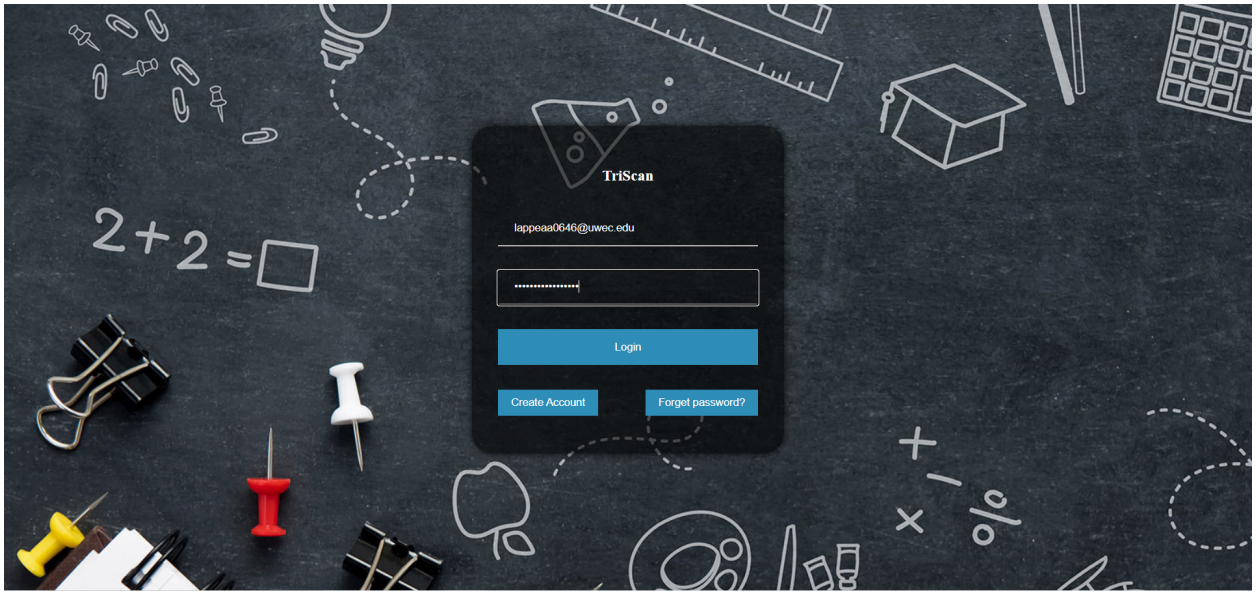
and the date. Students can also view their history where the data is pulled from the database to show to the student. If the user is faculty, they are able to create new categories which are stored in the database. Faculty can also view reports immediately after a document is uploaded by a student. Once a document is uploaded and saved to the database, the system pulls that information and identifies plagiarism, grammar errors, and similarities to generate a report. Lastly, the report can be sent to the student for them to view.

6. Class Diagram

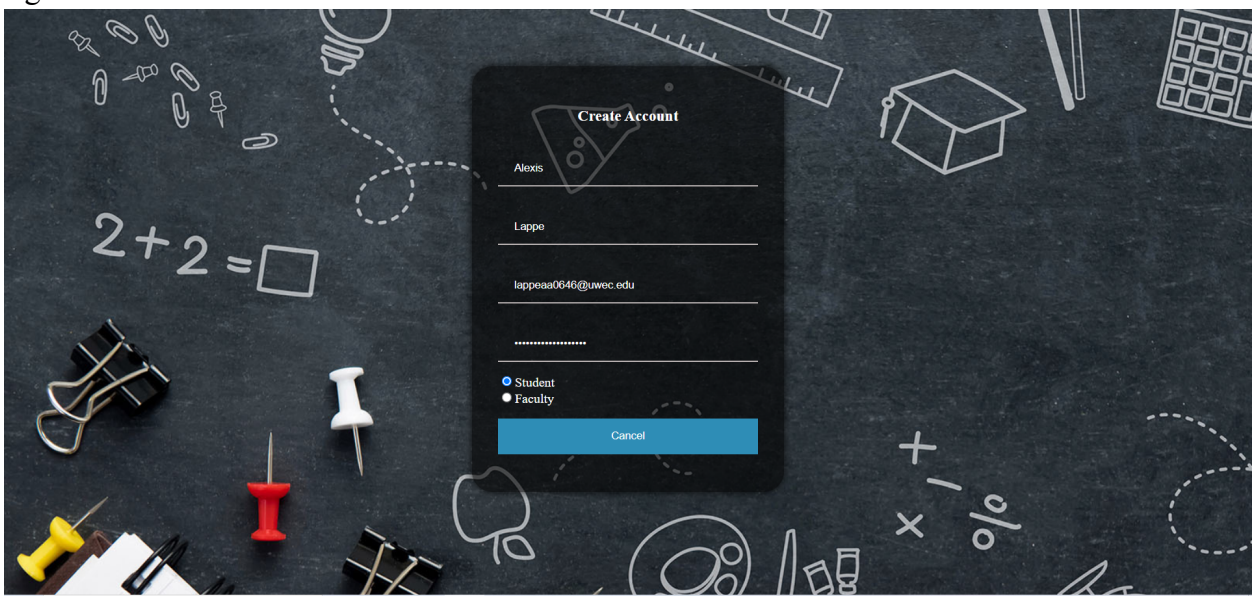

Account
Name: String
Email: String
Password: String
System
generateReport()
Stores
1
0...N
Creates
Creates
Generates
Faculty
createAccount()
createCategory()
viewDocument()
viewReport()
sendReport()
Student
createAccount()
getSubmission()
uploadDocument()
Submission History
DocumentName: String
SubmissionTime: Time
CourseName:String
Accesses
Views
Uploads
Report
Plagiarism: Array
Similarities: Array
Grammatical Errors: Array
checkReferences()
Category
Name: String
getAssignments()
Contains
Document
Name:String
has
References
Name: Array
Checks
sends

The class diagram has relationships similar to those displayed in the ER diagram. The Student class can create an account, see their submission history, and upload a diagram. The faculty class can create an account, create assignment categories, view documents, and send reports. The category class can get the assignments associated with that category. The report class can check references and the system class can generate reports.

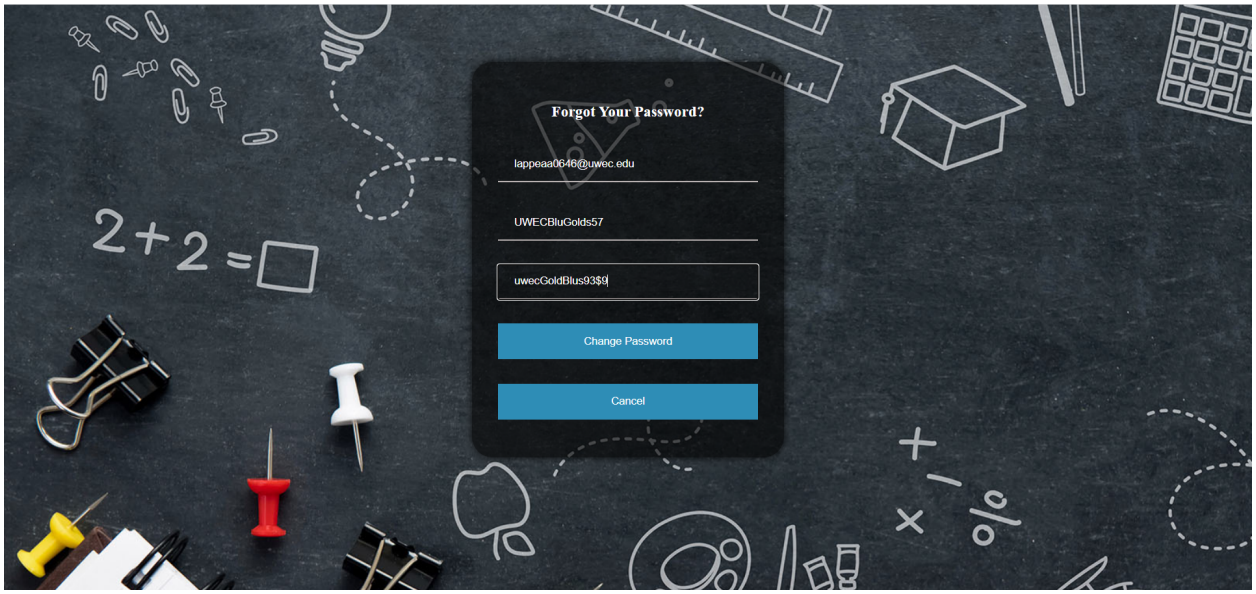
7. Front End Screenshots



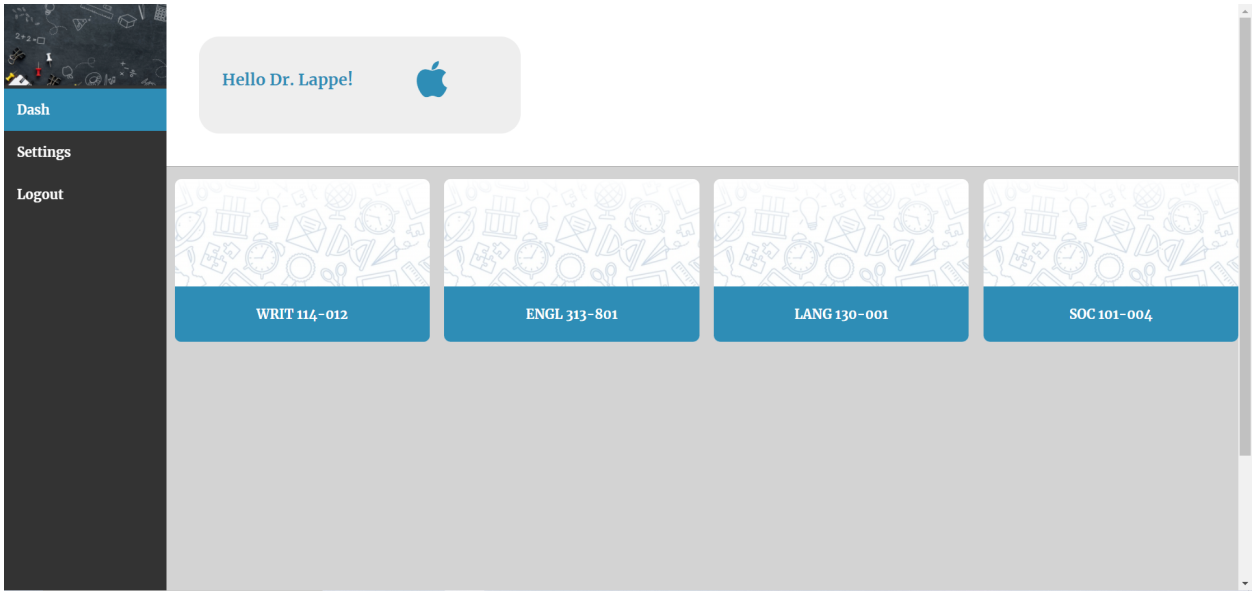
Login Page: This is the Login Page for Triscan. Users must input their email and password to login.



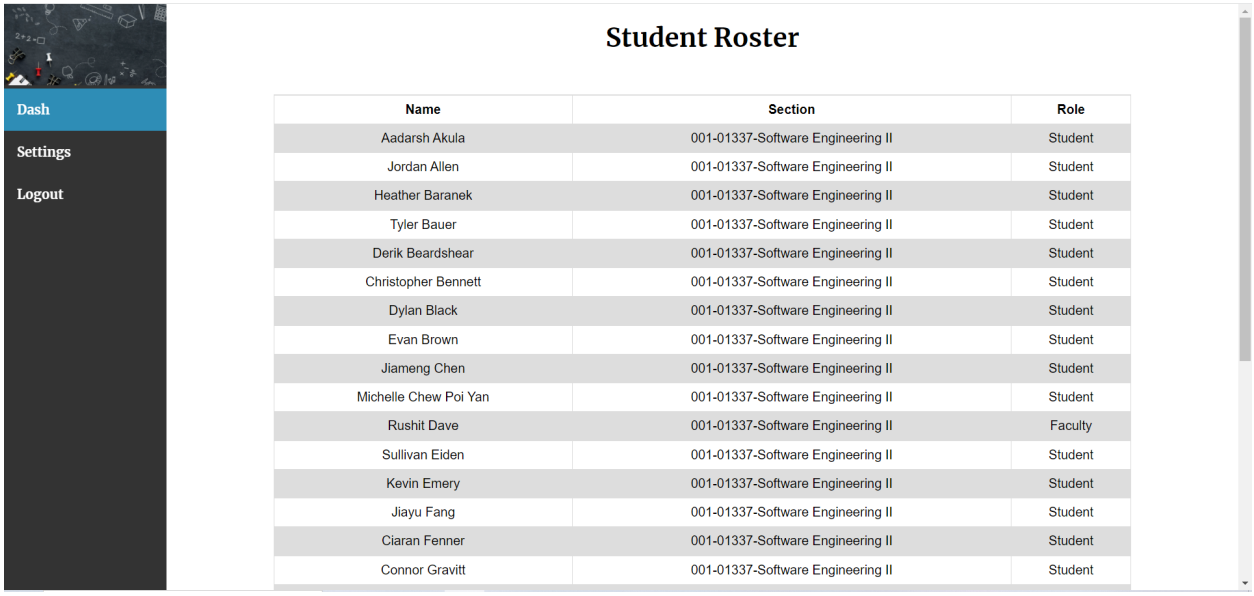
Create Account: Users can create an account using the Create Account Page. They must input their name, email, and password and select whether they are a student or faculty user.



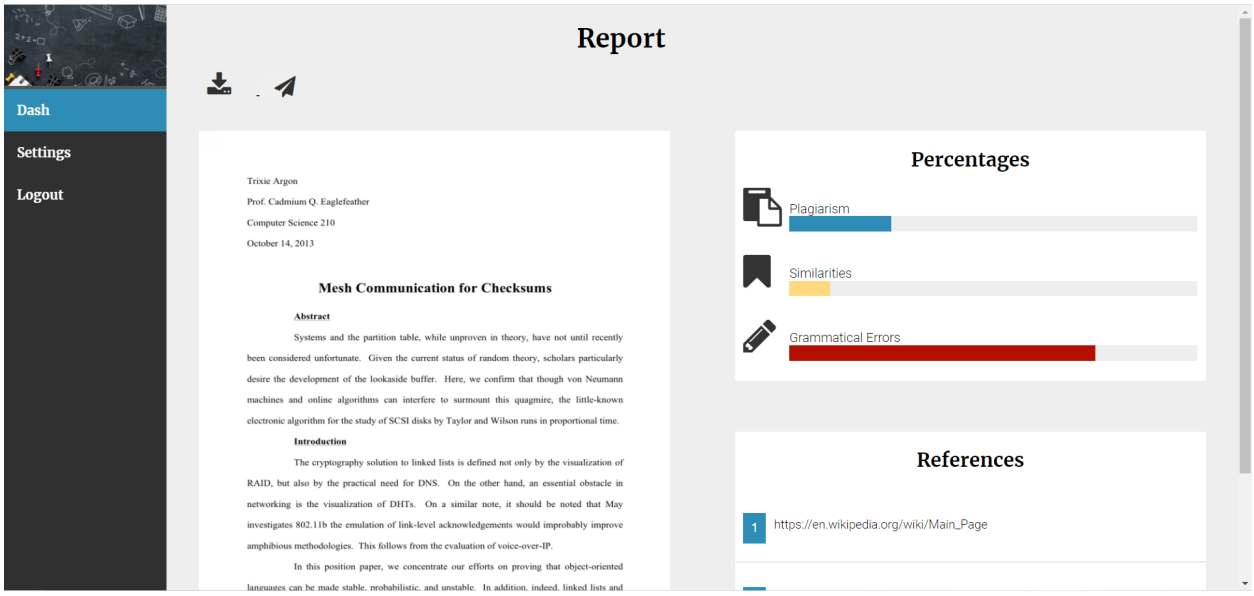
Change Password: Users can change their password using the Change Password Page. They can input their email, old password, and new password and click Change Password to associate the new password with their account.



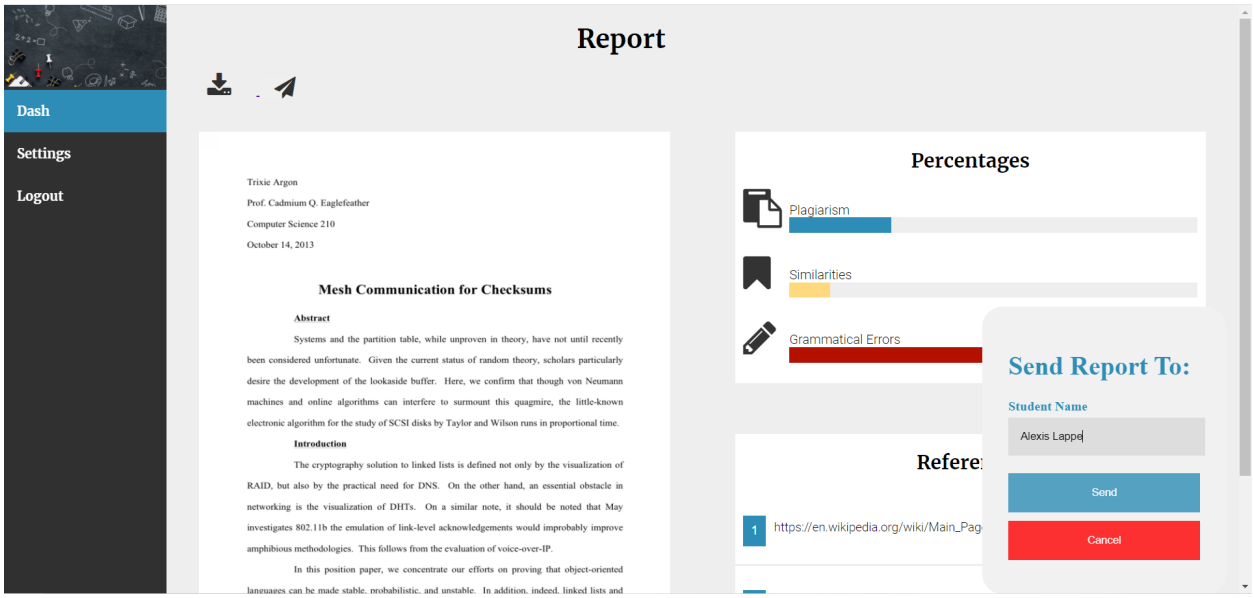
Faculty Dashboard: Faculty members can access their courses on the Faculty Dashboard.



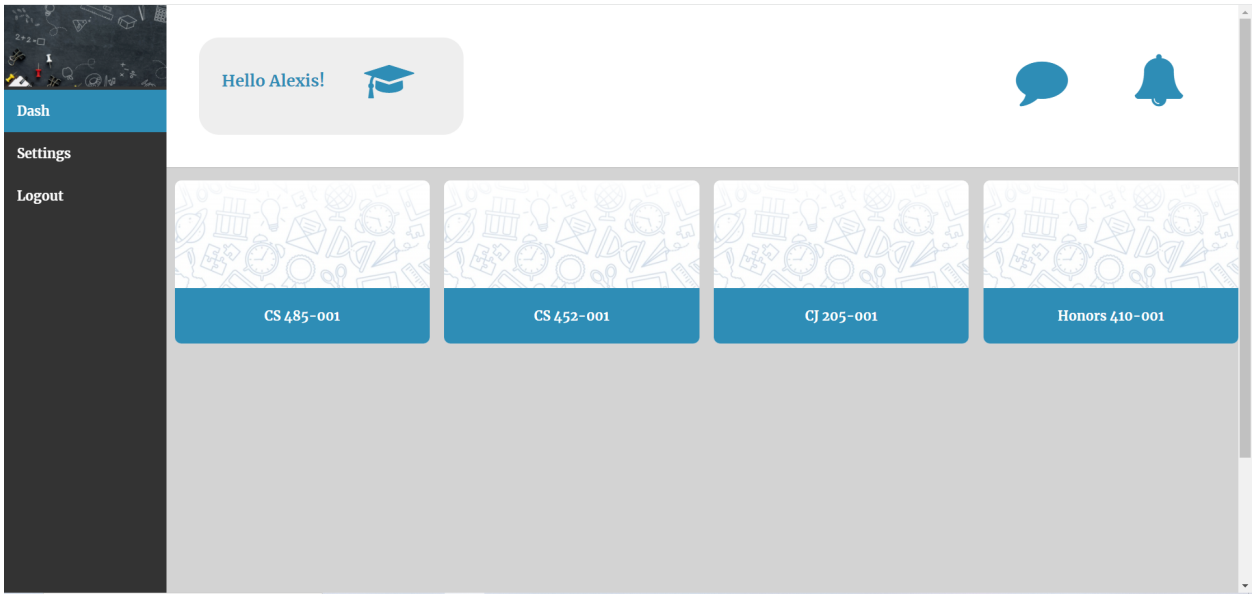
Student Roster: The Student Roster Page is where faculty members can see what students are enrolled in their course. When they click on a student's name, they will be taken to their assignments.



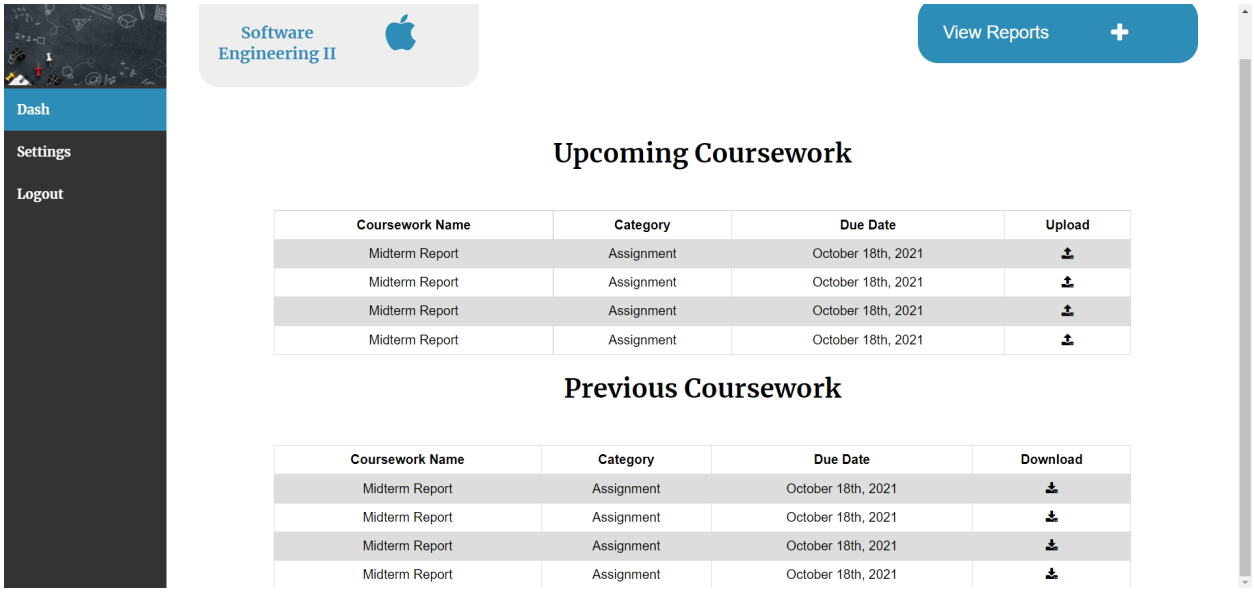
Report: Faculty members can create and view a report of the student's assignment that checks for plagiarism, similarities, and grammatical errors.



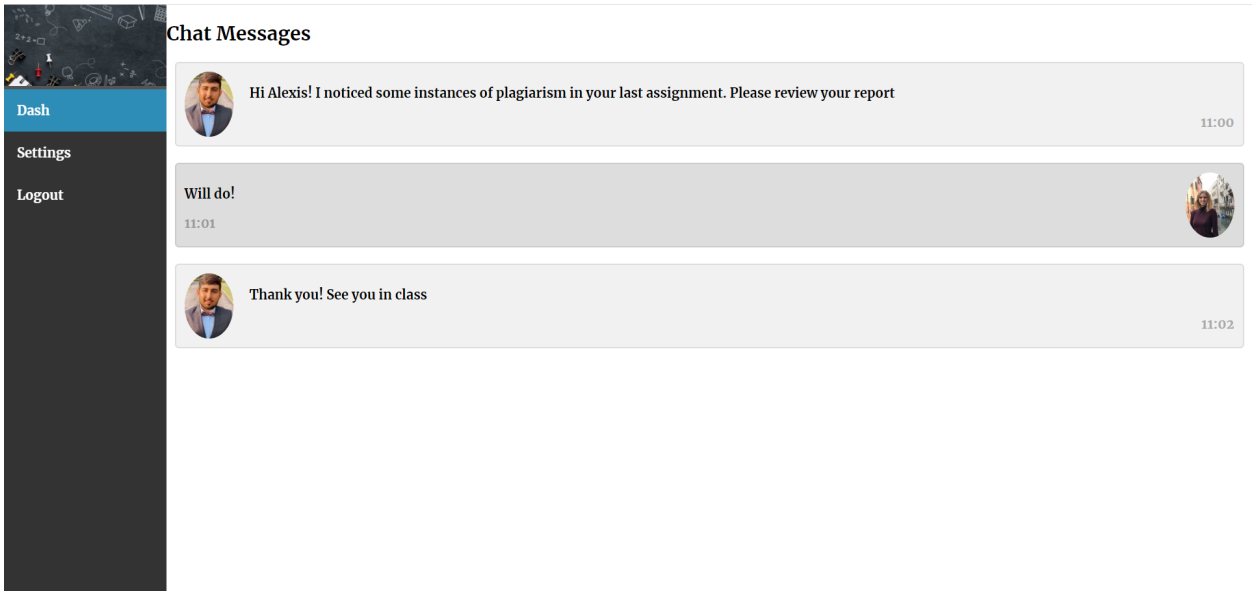
Send Report: When faculty members click on the paper plane icon on the Report Page, they can send the report to a student or students.



Student Dash: Students can access their courses, messages, and see if they have reports on the Student Dash page.



Coursework Page: Student's can view their upcoming and previous coursework on the Coursework Page. They can download and view previous assignments using the download icon and upload upcoming assignments using the upload icon.

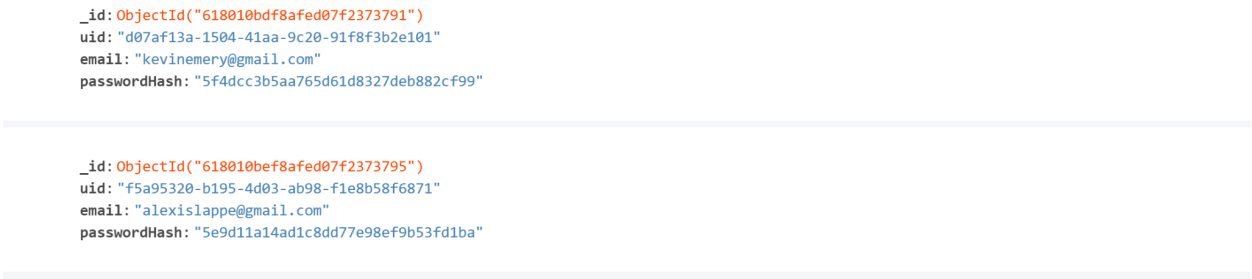


Messages: Students can view messages from faculty members on the Messages page.

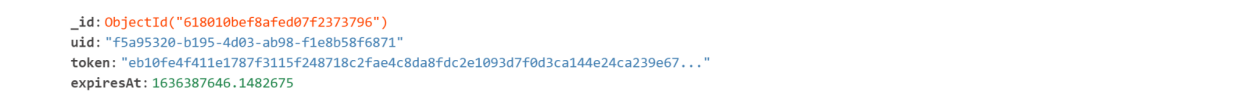
8. Database Explanations

The Mongo Database has thirteen collections to support the functionality of TriScan. The collections are as follows: assignments, categories, chats, courses, credentials, files, fs.chunks, fs.files, messages, reports, sessions, submissions, and users.

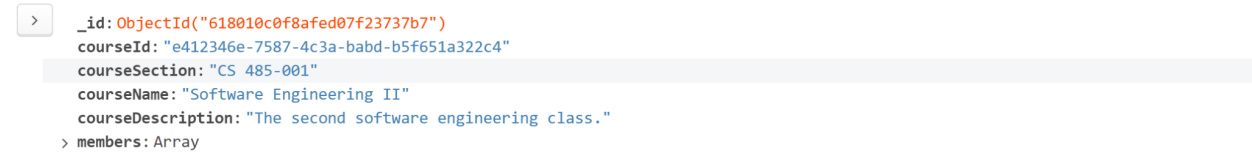
Users: The users collection supports the create account functionality. A new user is added into the collection when the form is submitted on the create account page. A document in the users collection contains the unique MongoDB object id, the system generated user id, the role in the TriScan system, the first name, the last name, the full name and the potential of a profile picture. The default account has no profile picture upload capabilities and functionality to upload a photo will be added in future iterations. The uid is utilized to map the other collections to a specific user.



Credentials: The credentials collection supports the create account, login, and change password functionality. A new set of credentials is inserted into the collection at the same time a new user is added to the users collections. A document in the credentials collections contains a unique MongoDB object id, the system generated user id corresponding to the user, the email entered into the create account form, and a hashed version of the user's password from the create account form. Note that the hashed passwords are also unique.



Sessions: The sessions collection supports the ability for students and faculty to login and maintain authorized sessions. Each time a user logs in, a new session is inserted into the database. A document in the session collection contains the unique MongoDB object id, the logged in user's uid, a token which randomly generated string saved in the client's browser, and the timestamp at which the session will expire. This allows the backend to communicate with the client to allow for the ability to verify a given user's authenticity and to also allow for timing out of sessions that are unused for a given amount of time.



Courses: The courses collection supports the student and faculty dashboards and report querying functionality in TriScan. New courses are inserted on the backend of the application by an administrator. A document in the courses collection contains a unique MongoDB object id, the system generated course id unique to a given course, the course section, the course name, and the members of the course stored in an array. The course id is utilized to link to the assignments, reports, and submission collections. Additionally, the elements of the members array are the uids corresponding to the users. Note that members are both faculty and students.

```
_id: ObjectId("61a45d0175620905ac0b6f41")
courseId: "e412346e-7587-4c3a-babd-b5f651a322c4"
courseCategories: Array
    0: "Assignments"
    1: "Quizzes"
    2: "Exams"
    3: "Reports"
```

```
_id: ObjectId("61a45d0175620905ac0b6f42")
courseId: "87aeca62-cf89-4efe-8710-d9e1529d6dea"
courseCategories: Array
```


Categories: The categories collection supports functionality on the faculty dashboards. A document in the categories collection contains a unique MongoDB object id, the system generated course id corresponding to a course in the courses collection, and the array courseCategories of a given course. Each course receives the categories assignments, quizzes, and exames. Faculty users are able to add additional categories for a given course. However, the functionality to delete them is not currently supported. Every assignment for a given course is also assigned to a specific category for querying purposes.


```
_id: ObjectId("618010c0f8afed07f23737b9")
assignmentId: "7d9925b2-9e7c-4f39-8e8c-ebe50389dcd4"
courseId: "e412346e-7587-4c3a-babd-b5f651a322c4"
assignmentName: "Initial Report"
assignmentDescripti… : "Only submit doc, docx or pdf files."
dueAt: 1638687202
category: "Assignments"
```

```
_id: ObjectId("618010c0f8afed07f23737ba")
assignmentId: "14d31629-8951-461a-bf06-0797f6d8108b"
courseId: "e412346e-7587-4c3a-babd-b5f651a322c4"
assignmentName: "Exam 1"
assignmentDescripti… : "Based of Chapters 1 to 1000"
dueAt: 1636387649
category: "Exams"
```


Assignments: The assignments collection supports the functionality for students to upload submissions and well as report querying capabilities. Assignments correspond to a specific course and contain a category. A document in the assignments collection contains a unique MongoDB object id, a system generated course id corresponding to a course in the courses collection, a system generated assignment id used to link to a submission and report, the assignment name, the assignment description, when it is due (converted to a unix time) and the category of the assignment. The assignment collection is used as a frame for student submission as students submissions go into the submissions collection not the assignments collection.

```
_id: ObjectId("61a92af24de1aa2a230c4edf")
submissionId: "93788b0c-e77e-4b1a-ab92-27e3f7679642"
uid: "d07af13a-1504-41aa-9c20-91f8f3b2e101"
courseId: "e412346e-7587-4c3a-babd-b5f651a322c4"
assignmentId: "7d9925b2-9e7c-4f39-8e8c-ebe50389dcd4"
files: Array
submittedAt: 1639081331
references: Array
```


Submissions: The submissions collection supports the student upload capabilities of TriScan. The submission id corresponds to the report generated. The uid links the submission to the user who submitted. Additionally, the course Id links to the course in the course collections. Lastly, the assignment id links the submission to its original assignment that the user submitted it to. A document in the submissions collection contains a unique MongoDB object id, a system generated course id, a system generated assignment id, a system generated user uid, a system generated submission id, a file array containing the submission file, the time the submission occurred (converted to unix time), and the array of references - also contained in the reports collection.


```
_id: ObjectId("618010c1f8afed07f23737cd")
fileId: "15085f71-2b69-4c42-95b2-be496c34f596"
fileName: "Ut.docx"
```

```
_id: ObjectId("618010c1f8afed07f23737ce")
fileId: "179000b9-24c6-439b-b6f3-7d35835abea4"
fileName: "Modi.docx"
```

```
_id: ObjectId("618010c2f8afed07f23737cf")
fileId: "3acd4936-2a6e-45b4-b708-08d366470e2b"
fileName: "Tempora.docx"
```


Files: The files collection supports the file upload functionalities for student users on TriScan. This collection stores the MongoDB object id, a randomly generated fileId, and a fileName string to prevent abuse of the GridFS system. The use of the collection is when seeking a file from GridFS, it will require knowing a fileId and not just a fileName. This prevents any bad actor from attempting to guess file names at the download endpoint where file protections are lessened.

```
_id: ObjectId("6196e1c32f0467690b1fb69c")
files_id: ObjectId("6196e1c32f0467690b1fb69b")
n: 0
data: Binary('eW8NCg==', 0)
```

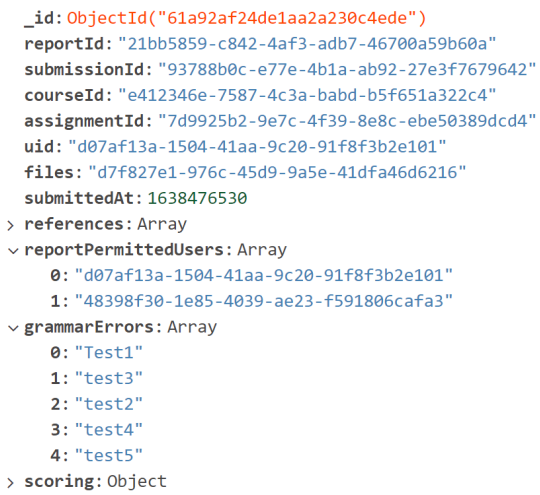
```
_id: ObjectId("619d38d4de1801d303e3a00a")
files_id: ObjectId("619d38d4de1801d303e3a009")
n: 0
data: Binary('UEsDBBQABgAIAAAAIQDfpNJsWgEAACAFAAATAAgCW0NvbnRlbnRfVHlwZXNdLnhtbCCiBAIooAACAAAAAAAAAAAAAAAAAAAAAAAA...', 0)
```


Fs.chunks: The fs.chunks collection supports the file upload functionalities for student users on TriScan. This collection comes with GridFS and is used to store the data of a file split into chunks, hence the name. The collection holds the MongoDB object id of itself, the MongoDB object of the file in relation to the Fs.files collection, the nth place in terms of reading file in order, and the data for said chunk. The n is enumerated throughout file upload when split into separated chunks. Each chunk allows for 16MBs to be held within itself due to limitations within MongoDB, but with the enumeration of the ids, allows for mass selection or streaming of the data.


```
_id: ObjectId("6196e0ca9172814a880470fc")
filename: "a6fc226f-533b-4bf5-baea-6aabc35525fb"
contentType: null
md5: "c3c17ca291db7701c8a338213dceb75c"
chunkSize: 261120
length: 4
uploadDate: 2021-11-18T23:24:59.678+00:00
```

```
_id: ObjectId("6196e138b90c1d5c47b18d34")
filename: "a4a27303-b584-4ab0-bf40-39117d288641"
contentType: null
md5: "c3c17ca291db7701c8a338213dceb75c"
chunkSize: 261120
length: 4
uploadDate: 2021-11-18T23:26:48.613+00:00
```


Fs.files: The fs.files collection supports the file upload functionalities for student users on TriScan. This collection comes with GridFS and stores the file metadata for the uploaded files. This collection holds the MongoDB object id, file name, content type, md5 hash, chunk size, length, and upload date of the file. This allows GridFS to more easily query the data and also allows our system to hook the Files collection to the Fs.files collection to secure downloading.



Reports: The reports collection supports the plagiarism, similarities, and error detection of TriScan. A document in the reports collection contains a unique MongoDB object id, a system generated course id linking the report to a specific course, a system generated assignment id linking the report to a specific assignment, a system generated user uid linking the report to a user, a system generated submission id linking the report to a specific submission, the file id for a given report called files, the time the submission the report was generated from was submitted (in unix time), an array of references to files that plagiarism was detected from, an array of users permitted to view the report, an array of grammar errors detected in the submission with the submission id, and an array of scores indicating the percentages of plagiarism, similarities, and the number of grammar errors.

```
_id: ObjectId("618010bff8afed07f23737af")
chatId: "99bf7d26-4d67-47fa-a7e3-8f842890ee30"
members: Array
  0: "7aac7a79-5782-48a6-a412-8fe12c84f39c"
  1: "f5a95320-b195-4d03-ab98-f1e8b58f6871"
chatName: "Alexis & Heather"
lastMessageAt: 1638340367
lastMessageId: "994d3766-d0fe-4841-8eb9-dc5e935be088"
```

```
_id: ObjectId("618010c0f8afed07f23737b4")
chatId: "fd7d74e2-8954-47d2-bff8-2792ee7aff27"
members: Array
  0: "48398f30-1e85-4039-ae23-f591806cafa3"
  1: "7aac7a79-5782-48a6-a412-8fe12c84f39c"
  2: "f5a95320-b195-4d03-ab98-f1e8b58f6871"
lastMessageAt: 1638427335
lastMessageId: "89a7c059-eef8-4735-9e4d-3a94f9697904"
chatName: "Kevin, Heather, & Dr. Dave"
```

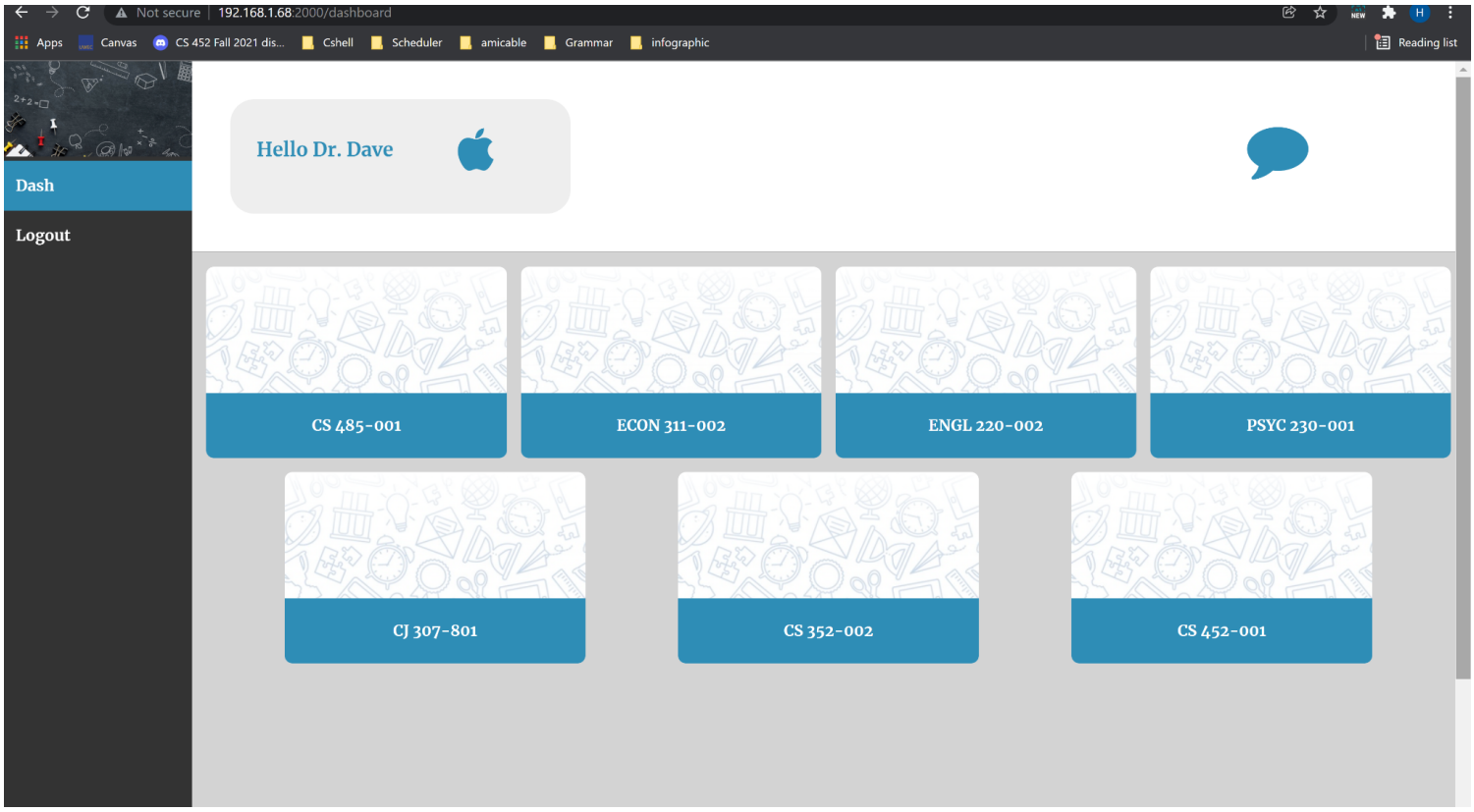

Chats: The chats collection supports the chat functionality on TriScan. A document in the chats collection contains a unique MongoDB object id, a system generated course id linking the report to a specific course, a system generated assignment id linking the report to a specific assignment, a system generated user uid linking the report to a user, a system generated submission id linking the report to a


```
_id: ObjectId("61a944836a59b6ecac4c1968")
messageId: "2875a45d-7ab0-4659-a99d-ed425f92755e"
chatId: "7651fbf1-7bfd-43e1-9fb9-d67f148633e9"
uid: "13932682-e83e-4bec-969d-3ffd9c9e05e3"
sentAt: 1638483075
content: "Hello did you finish your homework for SE 2?"
attachments: Array
```

```
_id: ObjectId("61a94e2baefd8ed8b5590bd2")
messageId: "990d4847-68a3-464c-adfb-299218de57d2"
chatId: "7dcbff9d-7009-42bf-9cd5-f43d0816bb80"
uid: "48398f30-1e85-4039-ae23-f591806cafa3"
sentAt: 1638485547
content: "DR DAVE DONT FAIL US"
attachments: Array
```


Messages: The messages collection supports the chat functionality on TriScan through its housing of individual user messages. A document in the messages collection contains a unique MongoDB object id, a system generated course id linking the report to a specific course, a system generated assignment id linking the report to a specific assignment, a system generated user uid linking the report to a user, a system generated submission id linking the report to a

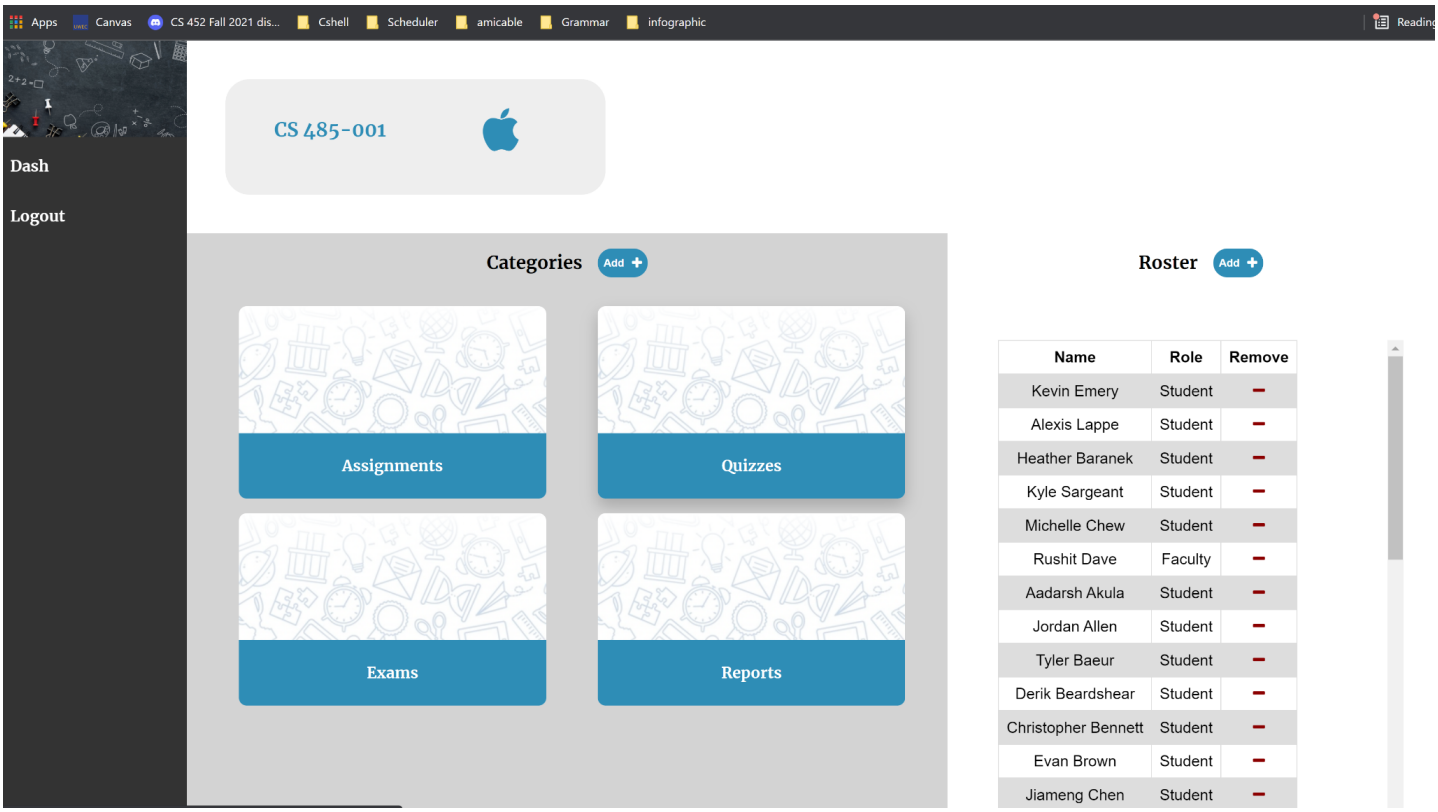
9. Main Back End Functionality



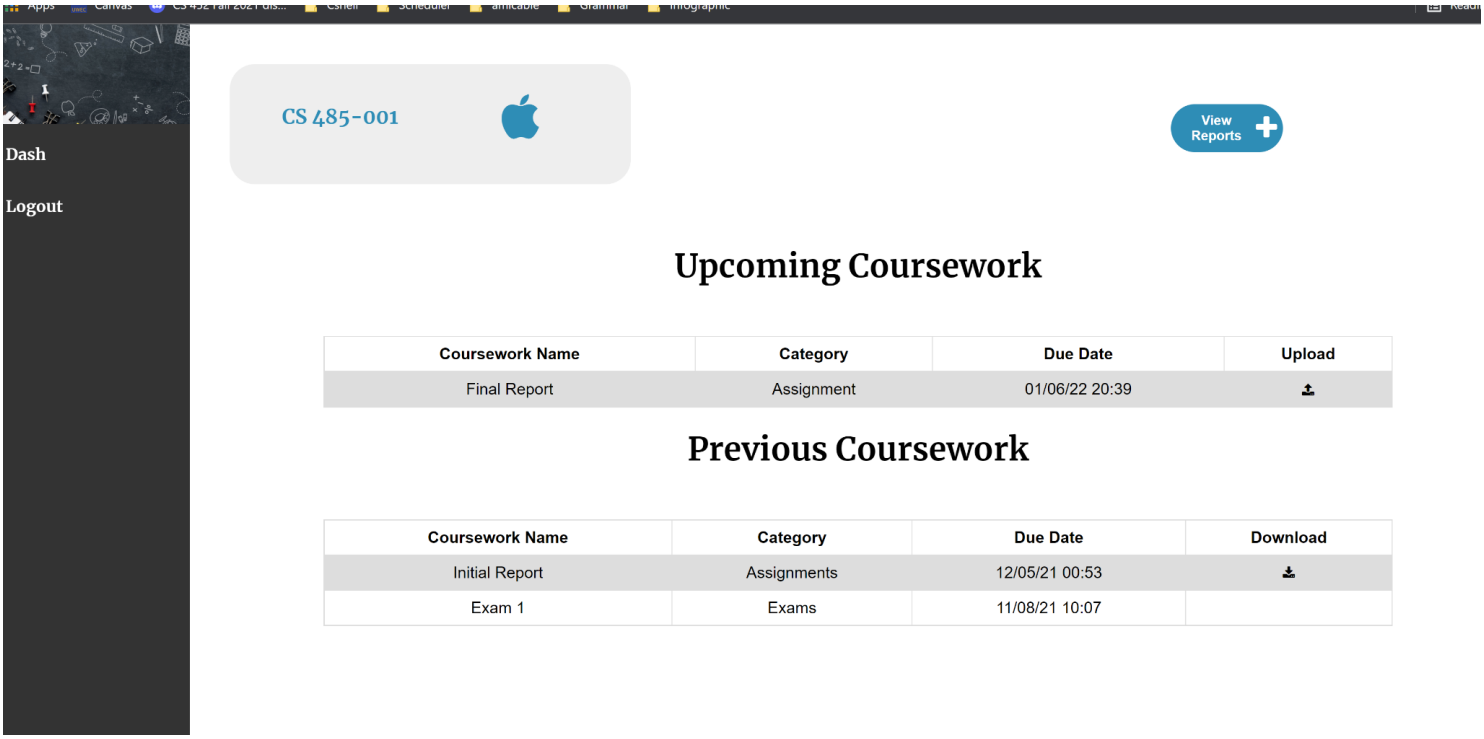
Dashboard UI: Classes are displayed dynamically in the dashboard based on the user.

```
@app.route("/dashboard")
def dashboard_page():
    if 'Authorization' in request.cookies.keys():
        uid = get_uid_from_session(request.cookies['Authorization'])

        if uid is not None:

            user = get_user_from_uid(uid)

            role = get_role_from_uid(uid)
            if role is not None:

                courses = get_courses(uid)

                if role == 'Student':

                    student_name = user["firstName"]
                    return render_template('student.html', student_name=student_name, courses=courses)

                if role == 'Faculty':

                    faculty_name = user["lastName"]
                    return render_template('faculty.html', faculty_name=faculty_name, courses=courses)

    response = make_response(redirect('/login'))
    response.set_cookie('Authorization', '', max_age=0)
    return response
```

The function dashboard_page() uses the session information to determine which user is logged in. Next, it queries for the user's role and creates an array of all of the courses the user is a member of. Lastly, it grabs the user's name to pass to the UI and displays the dashboard depending on the user's role with their courses.



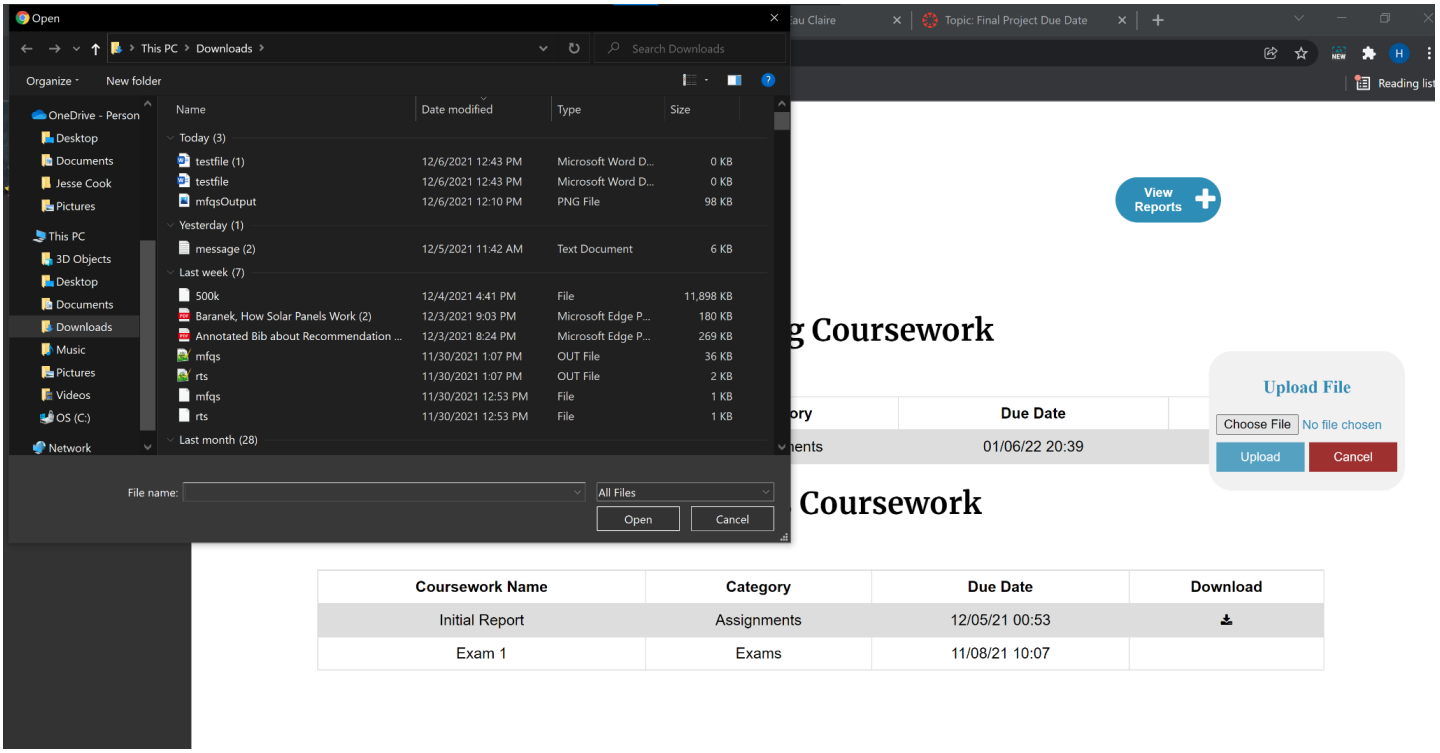
Faculty Class Selection: The above UI displays when a faculty member selects a class from their dashboard. The class section populates in the top left, the categories for the course display on the left and the class roster displays on the right in a scrollable region.



Student Class Selection: The above UI displays when a student selects a course from their dashboard. The class section displays in the top left corner. The button to view any available reports for the specific class is in the right hand corner. Lastly two centered tables display the lists of upcoming student coursework and previous coursework they have submitted. Student users are able to upload a document for a given assignment by selecting the upload button. Similarly, if a report is available for an assignment, they can click the download button next to it to download it. Note that the assignment/coursework information is passed as a parameter to the loading url for it to be dynamically displayed in the .html template.

```python
@app.route("/course/<courseId>")
def list_courses(courseId):
    uid = get_uid_from_session(request.cookies['Authorization'])
    role = get_role_from_uid(uid)
    if uid is not None:
        if role == 'Student':
            course = get_course_section(courseId)
            assignments = get_assignments(uid, courseId)
            print(courseId)
            user = get_user_from_uid(uid)
            name = user['fullName']
            return render_template("/class.html", name=name, assignments=assignments, course=course, courseId=courseId, url_root=request.base_
        if role == 'Faculty':
            students = get_students(courseId)
            course = get_course_section(courseId)
            categories = get_categories(courseId)
            print(courseId)
            return render_template("/faculty_course_view.html", course=course, students=students, categories=categories, courseId=courseId)
```

The function list_courses() although not terrifically named, provides the conditional back end logic to list the proper course information for a user depending on their role. It gathers all of the information to be dynamically displayed via multiple loops in the UI html templates based on the user's role.



Student - Upload File: The above UI demonstrates a students ability to submit a file. Student users are able to navigate to a specific course and click the upload button next to a given assignment. Then, the upload file form displays, and upon clicking choose file, the file explorer will pop-up for the user to locate a local file to submit.

```python
app.config["ALLOWED_FILE_EXTENSIONS"] = ["PDF", "DOC", "DOCX"]

# Function that checks for file extensions
# return True if file extensions is allowed
#return False if no file extensions were given or file extensions not allowed
def allowed_file(filename):

    if not "." in filename:
        return False

    # take the first element from the right and store in the extension var
    ext = filename.split(".", 1)[1]

    # compare ext with allowed_file_extensions
    if ext.upper() in app.config["ALLOWED_FILE_EXTENSIONS"]:
        return True
    else:
        return False
```

The allowed_file(filename) function takes in filename as argument and checks if the file has appropriate file extension that is specified in app.config[“ALLOWED_FILE_EXTENSIONS”]. The function returns True if the submitted file has the right file extensions (i.e PDF, DOC, DOCX) and returns False if the file does not have the right file extensions. This function is later being used in upload_file() function.

```python
@app.route("/upload-file/<courseId>", methods=["GET", "POST"])
def upload_file(courseId):

    if request.method == "POST":

        if 'Authorization' in request.cookies.keys():
            uid = get_uid_from_session(request.cookies['Authorization'])

            if request.files and uid is not None:

                # Get the file object
                file = request.files["file"]

                if file.filename == "":
                    print("File must have a filename")

                    return redirect('/course/'+courseId)

                elif not allowed_file(file.filename):
                    print("The file extensions is not allowed")

                    return redirect('/course/'+courseId)
                else:
                    # extra step to secure the file
                    filename = secure_filename(file.filename)
                    assignmentId = request.form['assignmentId']
                    if len(assignmentId) > 0 and len(courseId) > 0:
                        file_content = file.read()
                        print(filename)
                        print(courseId)
                        print(assignmentId)

                        submissionId = str(uuid.uuid4())

                        file_data = upload_file_to_database(filename, file_content, file, courseId, submissionId, assignmentId, uid)
                        print(file_data)

                        submissions_object = {
                            'submissionId': submissionId,
                            'uid': uid,
                            'courseId': courseId,
                            'assignmentId': assignmentId,
                            'files': [file_data['fileId']],
                            'submittedAt': math.ceil(time.time()) + (60 * 60 * 24 * 7),
                            'references': []
                        }
                        mongo.db.submissions.insert_one(submissions_object)

    return redirect('/course/'+courseId)
```

The upload_file() function handles the backend functionality of submitting an assignment. It checks for a post request and assuming one is found, it grabs the user's uid from the session data. After verifying a file has been selected, it performs validation on the file's extension before inserting a new submission object to the database. The function upload_file() also calls upload_file_to_database() which handles the report logic and is described below.

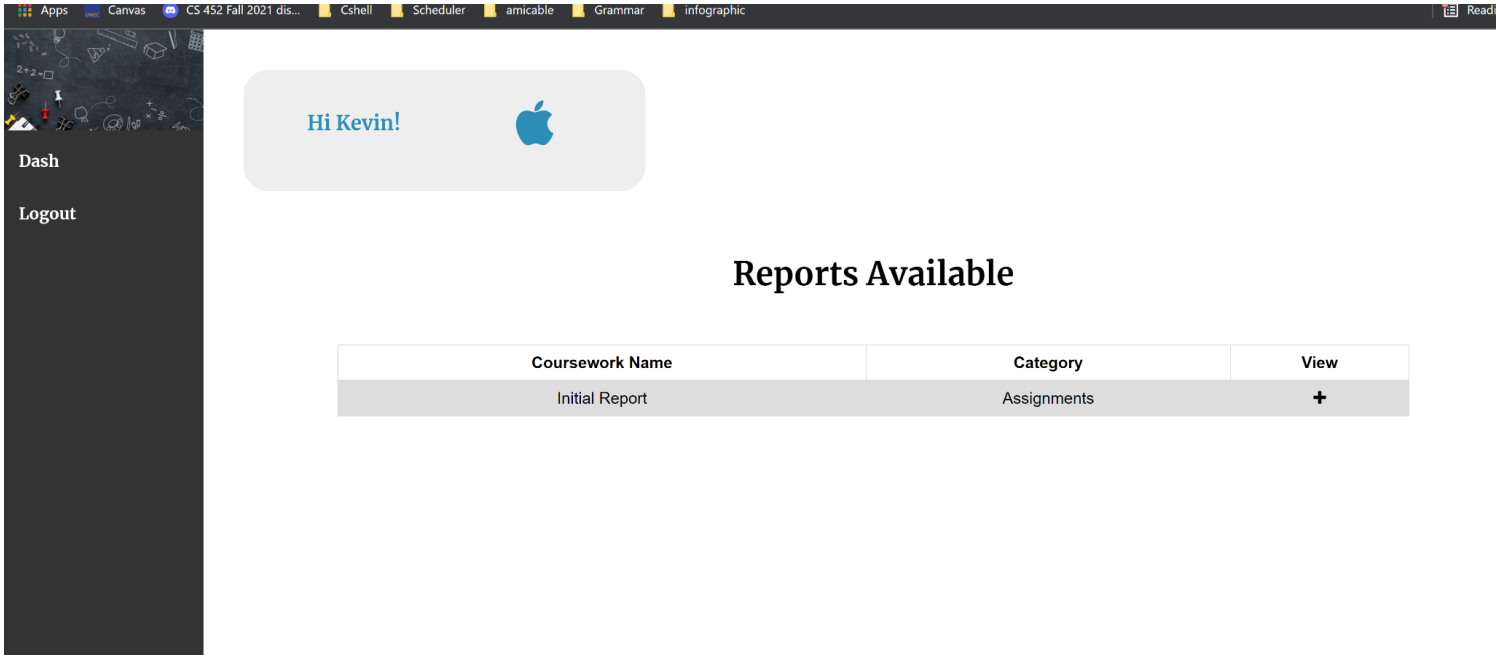
```python
def upload_file_to_database(file_name, file_contents, file_obj, courseId, submissionId, assignmentId, uid):
    fileId = str(uuid.uuid4())
    mongo.save_file(fileId, file_obj)

    file = {
        "fileId": fileId,
        "fileName": file_name
    }

    mongo.db.files.insert_one(file)


    if file_name[-4:].lower() == '.doc' or file_name[-5:].lower() == '.docx':

        try:
            os.mkdir('temp_files')
        except:
            pass

        f = open(f'temp_files/{file_name}', 'wb')
        f.write(file_contents)
        f.flush()

        text = textract.process(f'temp_files/{file_name}').decode()


    elif file_name[-4:].lower() == '.pdf':

        file_binary = io.BytesIO(file_contents)

        # creating a pdf reader object
        pdfReader = PyPDF2.PdfFileReader(file_binary)
        print(pdfReader.numPages)

        text = ''

        for page in range(0, pdfReader.numPages):
            pageObj = pdfReader.getPage(page)
            text += '\n' + pageObj.extractText()
```

Scroll for rest of function below.

```
tokenized_text = sent_tokenize(text)

#MAKE REPORT
reportId = str(uuid.uuid4())
report = {
    'reportId': reportId,
    'submissionId': submissionId,
    'courseId': courseId,
    'assignmentId': assignmentId,
    'uid': uid,
    'files': fileId, # supposed to be an array, but we've got code tha
    'submittedAt': math.floor(time.time()),
    'references': [],
    'reportPermittedUsers': [uid], # need to make it so the teacher is
    'grammarErrors': [],
    'scoring': {'plagiarism': 0, 'grammar': 0, 'similarities': 0}
}
mongo.db.reports.insert_one(report)

# send each sentence to grammar checker
for d in tokenized_text:
    grammar_check_file(d, reportId)

return file
```

```
def grammar_check_file(text, reportId):
    gingered_dict = GingerIt().parse(text)
    print(gingered_dict)
    for fix in gingered_dict['corrections']:
        mongo.db.reports.update_one({'reportId': reportId}, {'$push': {'grammarErrors': fix}})
```

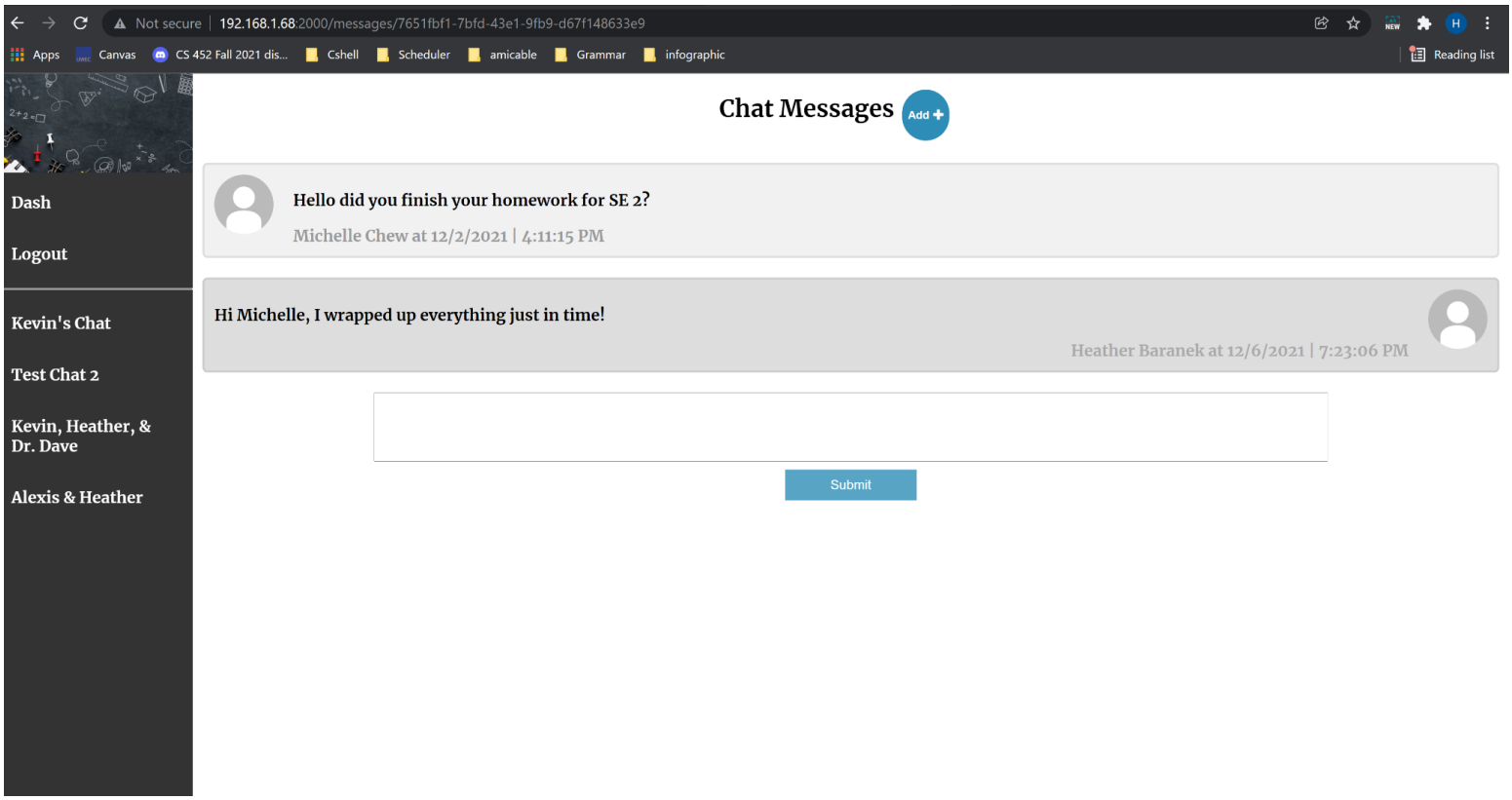
The upload_file_to_database() and grammar_check_file() function support the functionality of creating a report and parsing the file for any grammar errors. In upload_file_to_database() the submitted file is inserted into the files collection. Next, it is opened based on its file type and using textract, it is parsed into smaller chunks on a page by page basis to be fed into grammar_check_file(). Additionally, upload_file_to_database() also inserts the report document for the submission in the database. In grammar_check_file(), using Gingerit, a grammar error detector, the tokenized text is parsed and the errors are pushed to the grammarErrors array in the reports collection in Mongo.



Student - View all reports available: Users are able to view available reports in multiple ways. Student users can view all of their available reports as well as all available reports for a specific class. This screenshot above emcompasses the display of all available reports for a student user and they can view this screen by clicking on the bell on the dashboard. Note that faculty users are able to view reports via category or on a student basis by clicking student names or categories after they have selected a course. Also, note that the report displaying querying is altered only slightly for each of the pages so only this code will be reviewed to avoid redundancy.

```python
@app.route("/reports/<student_name>")
def reports(student_name):
    uid = get_uid_from_session(request.cookies['Authorization'])
    if uid is not None:
        courses = get_courses(uid)
        reports = []
        assignmentsList =[]
        submissions= []
        for course in courses:
            assignments = get_assignments(uid, course["courseId"])
            previousAssignment = assignments['previous']
            for assign in previousAssignment:
                most_recent_submission = list(mongo.db.submissions.find({'uid': uid, 'assignmentId': assign['assignmentId']}).sort('_id', -1))
                if len(most_recent_submission) > 0:
                    most_recent_submission = most_recent_submission[0]
                    report = mongo.db.reports.find_one({'submissionId': most_recent_submission['submissionId']})
                    permitted_users = report['reportPermittedUsers']
                    for p in permitted_users:
                        if p == uid:
                            reports.append(report)
                            submissions.append(assign)
                            assignment = mongo.db.assignments.find_one({'assignmentId': assign['assignmentId']})
                            print(assignment)
                            assignmentsList.append(assignment)
    try:
        return render_template("/report.html", courses=courses, student_name=student_name, reports=reports, zip=zip, submissions = submissions,
    except:
        return redirect('/dashboard') #there if it fails to actually run
```

The function reports() handles the backend logic that allows students to view all of their available reports across their classes. First, it gets the user's uid via the session data in Mongo. Next, it defines arrays to store pertinent information that will get passed to the UI for dynamic displaying. For each course a student is in, it grabs all of the assignments that have been submitted or have gone past their due date. From those assignments, it iterates over each one and grabs the most recent submission if there is one. Next, it grabs the report with the match submissionId and also saves a reference to the permitted users who can view the report. Lastly, it iterates over the permitted user array to see if the student is allowed to view their report. If they are, it appends them to the arrays that are passed to the report.html template to display them dynamically to the student user. The logic in this function is also applied to the other report querying pages on TriScan.



Chat/Messaging UI: Users are able to chat with each other on TriScan. The user can navigate to the chat icon to get to this page. On the left side bar, users can switch between their established chats. In the main container, all previously stored messages will load and at the bottom, users can type their new message into a freeform box and click submit to send it to the other user(s). Additionally, users are able to create new chats by clicking the add button. Moreover, they can add more members to an existing chat by also clicking the add button and entering the chat name and the new user they want added.

```python
def create_chat(userId):
    chatId = str(uuid.uuid4())
    sentAt = math.floor(time.time())
    mongo.db.chats.insert_one({'chatName': 'New Chat', 'chatId': chatId, 'lastMessageAt': sentAt, 'lastMessageId': None, 'members': [userId]})
    return chatId


def create_chat_with_user(userId, creatorId, chatName):
    existingChatId = list(mongo.db.chats.find({'chatName': {'$all': [chatName]}}).limit(1))
    if len(existingChatId) > 0:
        mongo.db.chats.update_one({'chatName': chatName}, {'$push': {'members': userId}})
        return existingChatId
    else:
        chatId = str(uuid.uuid4())
        sentAt = math.floor(time.time())
        mongo.db.chats.insert_one({'chatName': chatName, 'chatId': chatId, 'lastMessageAt': sentAt, 'lastMessageId': None, 'members': [userId]})
        mongo.db.chats.update_one({'chatId': chatId}, {'$push': {'members': creatorId}})
        return chatId


def get_recent_chats(userId, limit=5):
    recent_chats = list(mongo.db.chats.find({'members': {'$all': [userId]}}, {'_id': 0}).sort('lastMessageAt', -1).limit(limit))
    return recent_chats


def get_most_recent_chat(userId):
    most_recent_chat = list(mongo.db.chats.find({'members': {'$all': [userId]}}, {'_id': 0}).sort('lastMessageAt', -1).limit(1))
    if len(most_recent_chat) > 0:
        most_recent_chat = most_recent_chat[0]
        return most_recent_chat['chatId']
    else:
        return None


def rename_chat(chatId, chatName):
    mongo.db.chats.update_one({'chatId': chatId}, {'$set': {'chatName': chatName}})


def load_chat_data(chatId):
    chat_data = mongo.db.chats.find_one({'chatId': chatId}, {'_id': 0})
    if chat_data is not None:
        users = list(mongo.db.users.find({'uid': {'$in': chat_data['members']}}, {'_id': 0}))
        dict_users = {}
        for user in users:
            dict_users[user['uid']] = user
        chat_data['members'] = dict_users
        return chat_data
    else:
        return {}
```

The functions above all for all the manipulation of chats. The create_chat() function will create an empty chat with just that user so they can save notes to themselves or add more users later on. The create_chat_with_user() function allows the user to create a new chat with another user and specify a name to go along with it. The get_recent_chats() returns the most recent chats that a user is participating in so that they can view it on the sidebar of the site for easier navigation. The

get_most_recent_chat() function returns the chat that the user most recently received a message from. This is what is loaded when the user first clicks onto the messages tab to give easy access. The rename_chat() function allows a user to rename a given chat. The load_chat_data() returns all information about a chat to the user. This includes full info for all members, most recent message timestamp and id, chat name from the chats collection.

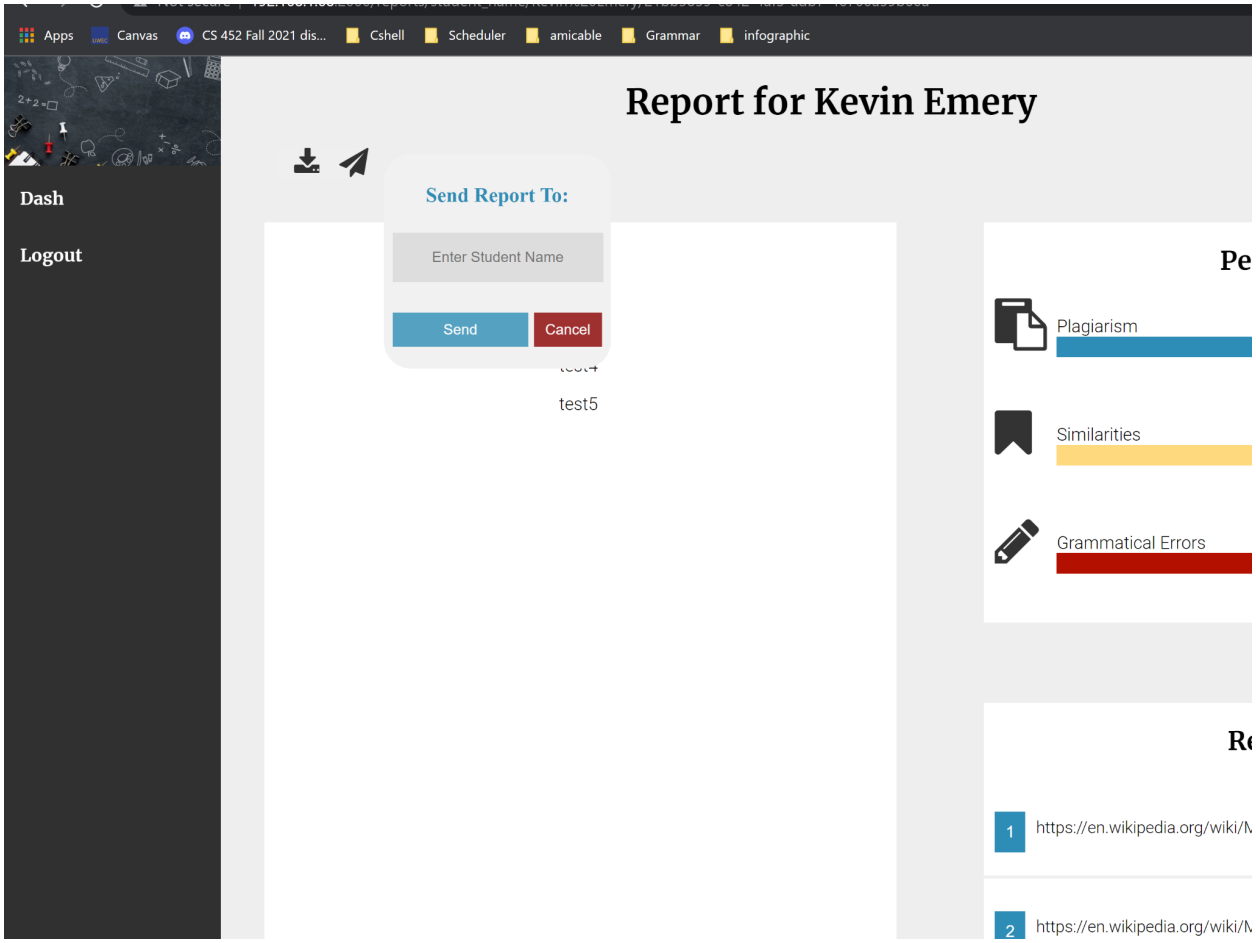
```
def send_message(userId, chatId, message, attachments=[]):
    messageId = str(uuid.uuid4())
    sentAt = math.floor(time.time())
    mongo.db.messages.insert_one({'messageId': messageId, 'chatId': chatId, 'uid': userId, 'sentAt': sentAt, 'content': message, 'attachments': attachments})
    mongo.db.chats.update_one({'chatId': chatId}, {'$set': {'lastMessageAt': sentAt, 'lastMessageId': messageId}})


def load_messages(chatId, limit=5, before=None, after=0):
    if before is None:
        before = time.time()
    messages = list(mongo.db.messages.find({'chatId': chatId, 'sentAt': {'$lt': before, '$gt': after}}, {'_id': 0}).sort('sentAt', -1).limit(limit))

    userids = set()
    for message in messages:
        userids.add(message['uid'])

    users = list(mongo.db.users.find({'uid': {'$in': list(userids)}},{'_id':0}))

    users_dict = {}
    for user in users:
        users_dict[user['uid']] = user

    response = {}
    response['messages'] = messages
    response['users'] = users_dict
    return response
```

The send_message() function allows the user to add a message to a given chat with any attachments. Attachments were not finalized, but would be direct links that would embed into the messaging HTML. The load_messages() returns the messages before and after a given unix timestamp in a chat. This allows for easier pagination for scrolling up and down through messages. The ability to paginate through the message wasn't fully implemented due to time constraints, but would be just a little more work on the Javascript side of things.

```
load_chat_data();
load_messages(start_time,0);
refresh_side_chats();

setInterval(function() { var this_time = Date.now()/1000; load_messages(this_time, most_recent_message); last_refresh = this_time}, 5000);
setInterval(refresh_side_chats, 5000);
```

The code above is the scripting that runs on the client's browser to get up to date messages and side chats. The load_chat_data() function requests JSON from our flask endpoint running the load_chat_data() function from the backend. The load_messages() function requests JSON from our flask endpoint running the load_messages() function from the backend. The refresh_side_chats() function requests JSON from our flask endpoint running the get_recent_chats() function. All of these take the response and parse them into the HTML. A majority of the code for each is a simple XMLHttpRequest object hitting our server and then looping through the results. The setInterval functions run the functions given in the first parameter every 5000 milliseconds. This allows the user to get updated information while also

not spamming the server with requests. In this use case, instant replying isn't needed and 5 seconds is very reasonable for a school conversation platform.



Faculty: Viewing/Sending Reports: Faculty users are able to view reports of students' submissions. On this page, grammar errors from the grammarErrors array of the report collection will populate in the white containers on the right hand side of the page. Additionally, upon integration of the plagiarism/similarity model, the output from that will also populate in that region. Another main functionality of this page is the ability to send reports. Upon clicking the small arrow icon, a form will appear in which the faculty member can enter a student's first and last name. If that student is a user in the database, their user uid will be added to the reportPermittedUsers array for the specific report in the reports collection. Note that the icon to send a report is only present when the user has a role of faculty.

```
def add_to_report(reportId, uid):

    mongo.db.courses.update_one({'reportId': reportId}, {'$push': {'reportPermittedUsers': uid}})
```

```python
@app.route("/reports/<reportId>/student_name/<student_name>/add", methods=["POST", "GET"])
def add_person_to_report(reportId, student_name):
    if request.method == "POST":

        req = request.form
        print(req)
        if 'name' in req.keys():

            uid = get_uid_from_name(req['name'])
            if uid is not None:
                add_to_report(reportId, uid)
            else:
                print("User is not registered")
    return redirect('/reports/student_name/'+student_name+'/'+reportId)
```

The functions add_person_to_report() and add_to_report() handle the back end logic of adding a user to the report list. In add_person_to_report(), the function verifies that a post request has been received at the given url. Then, it goes and grabs the form. After verifying that a name is present in the form, it attempts to get the uid from the name that was entered. Once a uid is found that matches the name, the reportId and uid are sent to the function add_to_report(). Within, add_to_report(), the Mongo Database is updated by pushing the uid that was passed onto the array of reportPermittedUsers for the report with the matching reportId.

```
[ ]
    fs = gridfs.GridFS(db)

    fileName='e39995be-ea71-4dfc-80d8-ff358fb99e40'

    print(fs.exists({"filename":fileName }))
    if(fs.exists({"filename":fileName })):
      for grid_out in fs.find({"filename": fileName},no_cursor_timeout=True):
        data = grid_out.read()
        file=io.BytesIO(data)

      print(file)

        # Look at text str
      pdfFileObj = open('drive/My Drive/CS 485/test/test.docx', 'wb')
      pdfFileObj.write(data)
      pdfFileObj.flush()
      #NEED TO SOMEHOW READ TEXT FROM _io.BytesIO OBJECT
    else:
      print("failed to collect file!")
```

Data Mining Model: The main purpose for the model is to apply K-Nearest Neighbor to classify text to identify instances of plagiarism. The model first connects to the mongoDB database and reads in the student submitted document.


```
[ ]
    fs = gridfs.GridFS(db)

    fileName='e39995be-ea71-4dfc-80d8-ff358fb99e40'

    print(fs.exists({"filename":fileName }))
    if(fs.exists({"filename":fileName })):
      for grid_out in fs.find({"filename": fileName},no_cursor_timeout=True):
        data = grid_out.read()
        file=io.BytesIO(data)

      print(file)

        # Look at text str
      pdfFileObj = open('drive/My Drive/CS 485/test/test.docx', 'wb')
      pdfFileObj.write(data)
      pdfFileObj.flush()
      #NEED TO SOMEHOW READ TEXT FROM _io.BytesIO OBJECT
    else:
      print("failed to collect file!")
```

The reason we are saving the file to google drive is because we could not find an effective way to store a file in memory coming from the database.

```
print(len(full_file_paths))

outputPDF=''
outputDocx=''

dirname=''

for dir in full_file_paths:

  print(os.path.basename(dir))




  if(dir.find(".pdf")!=-1):#is a pdf


    #convert .pdf to .txt

    dirname = os.path.dirname(dir)
    pdfFileObj = open(dir, 'rb')
    # creating a pdf reader object
    pdfReader = PyPDF2.PdfFileReader(pdfFileObj)

    count = pdfReader.numPages
    #for loop to read pages
    for i in range(count):

      pageObj = pdfReader.getPage(i)
      outputPDF += pageObj.extractText()

    outputPDF = outputPDF.lower()


        # closing the pdf file object
    pdfFileObj.close()

    x_train.append(outputPDF)
    y_train.append(dir)

  elif(dir.find(".docx")!=-1):#is a pdf

    import docx
    doc = docx.Document(dir)


    for para in doc.paragraphs:
        outputDocx+=para.text+"\n"
    #print(outputDocx)
    x_train.append(outputDocx)
    y_train.append(dirname)
    dirname = os.path.dirname(dir)
```

Next the model pre-process the resource data i.e. the text documents we are comparing to see if there has been plagiarism.

```
#convert to numpy
x_train = np.array(x_train)
y_train = np.array(y_train)
print(y_train.shape)
print(x_train.shape)
```

```
(41,)
(41,)
```

```
from sklearn.feature_extraction.text import CountVectorizer

count_vect=CountVectorizer()
x_train_counts=count_vect.fit_transform(x_train)

print(x_train_counts.shape)
```

```
(41, 96507)
```

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import  TfidfVectorizer

text_clf = Pipeline([('tfidf', TfidfVectorizer()),('clf', KNeighborsClassifier(n_neighbors=1))])

text_clf.fit(x_train,y_train)
```

```
Pipeline(steps=[('tfidf', TfidfVectorizer()),
                ('clf', KNeighborsClassifier(n_neighbors=1))])
```

```
predictions = text_clf.predict(x_test)
#predictions,i=text_clf.score(x_test2)
print(predictions)
```

```
['drive/My Drive/CS 485/Training Documents/English(Engl 220: creative writing)/Papers/Sample 2.pdf'
 'drive/My Drive/CS 485/Training Documents/Small Group Communication(CJ 307-801)/Papers'
 'drive/My Drive/CS 485/Training Documents/Psychology(Psych 230: Human Development)/Assignments/Case6.pdf'
 'drive/My Drive/CS 485/Training Documents/Psychology(Psych 230: Human Development)/References/LifespanDevelopment.pdf'
```

Once all the data was pre-processed we insert the data into appropriate numpy arrays, vectorize the text of the input data and insert it into the KNN algorithm. The output then states the closest resource text that represents the most plagiarized text source. Unfortunately this did not turn out as planned as for some reason the outputted text had no relation to the student submitted document. Ideally this would have been able to give an accurate output and allowed us to find the euclidean distance to find the similarity between the two documents.