Part 1:

Revise user stories, reflect what is now fully implemented tested and working:

1. As a student of a particular course, I want to view notes uploaded by the professor or other students in this class, so that I can better understand course materials and prepare for homeworks and exams.
    a. Common cases:
        i. A logged-in user can search a class by department name or course number.
        ii. The user can click a particular file that was previously uploaded to view.
    b. Special cases:
        i. A logged-in user enters a department name or course number that does not exist. The system pops up an error message.
2. As a student/professor of a particular course, I want to upload my lecture notes to share with other students in this class, so that I can help others to study.
    a. Common cases:
        i. A logged-in user can upload his class notes in pdf file type by indicating which department and course number the file belongs to.
        ii. The uploaded file is then available to all users to view and download.
    b. Special cases:
        i. A logged-in user enters a department name and course number combo that does not exist. The system pops up an error message.
        ii. A logged-in user enters a valid combo but tries to upload a file that has invalid file type. The system pops up an error message.
3. As a student of a particular course, I want to download notes uploaded by the professor or other students in this class, so that I can view the notes offline.
    a. Common cases:
        i. A logged-in user can search a class by department name or course number.
        ii. The user can click a particular file that was previously uploaded, then choose to download.
    b. Special cases:
        i. A logged-in user enters a course number that does not exist. The system pops up an error message.
4. As a student of a particular course, I want to upvote or downvote a note so that I have a way to tell others students that the file is of high or low quality.
    a. Common cases:
        i. A logged-in user can search a class by department name and course number.
        ii. The user can upvote or downvote this file.
        iii. The upvote and downvote number increments by 1 accordingly.
        iv. The user can undo his action by clicking the button again and the upvote/downvote number decrements by 1 accordingly.

    b. Special cases:
        i. A logged-in user enters a department name and course number combo that does not exist. The system pops up an error message.
        ii. A logged-in user tries to upvote (or downvote) a file that has been downvoted (or upvoted) by himself. The system pops up a message telling the user he has already voted for this file.
5. As a student/professor of a particular course, I want to delete some files uploaded by myself if some mistakes are present in the notes, so that the wrong notes won't mislead other students.
    a. Common cases:
        i. A logged-in user can search a class by department name and course number.
        ii. The user can choose the particular file that he wants to delete and then delete the file.
        iii. The deleted file is then removed from the system.
    b. Special cases:
        i. A logged-in user enters a department name and course number combo that does not exist. The system pops up an error message.
6. As a student of a particular course, I want to post comments and read comments posted by other students to a particular file belonging to that class, so that I can ask questions, discuss my thoughts, and answer questions related to the course materials.
    a. Common cases:
        i. A logged-in user can search a class by department name and course number.
        ii. The user can choose a file to view explicitly.
        iii. The user can read comments in the discussion forum of that file.
        iv. The user can post his new comments to the forum.
        v. The post is then available to view for all users.
    b. Special cases:
        i. A logged-in user enters a department name and course number combo that does not exist. The system pops up an error message.
        ii. The user enters a blank comment and tries to post it. The system detects that the post is blank and pops up an error message.
7. As a student of a particular course, I want to see the comments by me, notes saved by me, and my personal info, and my own notes by going to my personal page.
    a. Common cases:
        i. A logged-in user can search a class by department name and course number.
        ii. The user can go to my profile
        iii. Profile page will render user's info along with sections of saved favorites, notes and comments by the user
8. As a student of a particular course, I want to delete the comments by me, notes saved by me, and my own notes by going to my personal page.
    a. Common cases:

i. A logged-in user can search a class by department name and course number.
ii. The user can go to my profile
iii. Profile page will render user's info along with sections of saved favorites, notes and comments by the user
iv. User can delete any notes/comments by clicking the garbage icon on the top right of each card

b. Special cases:
i. There are no saved notes/comments, user cannot click any icon because no cards with the information of notes/comments are rendered

part 2

Write a test plan that explains the ***equivalence partitions*** and ***boundary conditions*** necessary to unit-test each of the major subroutines in your system (methods or functions, excluding constructors, getters/setters, helpers, etc.) and then implement your plan. Associate the names of your specific test case(s) with the corresponding equivalence partitions and boundaries (if applicable). Your test suite should include test cases from both ***valid*** and ***invalid*** equivalence partitions, and just below, at, and just above each equivalence class boundary (or inside vs. outside the equivalence class when boundary analysis does not apply). Note the same test case might apply to multiple equivalence classes. Say there is a method whose input should be an integer between 1 and 12. Then there is an equivalence class 1-12, an equivalence class <1 (or <=0), and an equivalence class >12 (or >= 13). A test case with input 0 would be just below the lower boundary of the equivalence class 1-12 and also at the high boundary of the equivalence class <1.

Include the link to the folder in your github repository that contains your automated test suite.

---------- Frontend Test Plan ----------

Since most of our pages are made up from React components that don't have any dependencies with other components, we test each component with jest unit tests. Each .test.js file is a unit test for our component. The equivalent class is defined from the user

input. ⌀ indicate input is empty. For Api tests, we mock ‘axios’, ‘Cookie’ etc. for each unit test.

1. AddCourse.test.js
   Description: This is the add course button of our page, only admins have the permission to add a course. For students, this button should not be rendered.

   Input: <Authorization token stored in the cookie>
   Valid equivalent class:
   - <Authorization token with admin permission>

   Invalid equivalent class:
   - <Authorization token with student permission>
   - <⌀>

   Test Plan: We simulated two users with different authentication cookies and checked whether the button is rendered. (student should not see the button)

2. AddCourseView.test.js
   Description: This is the page for teachers to add a new course to the system. This page needs to interact with the backend API so we created a mock HTML form submission for testing.

   Input: <Form Entry #1, Form Entry #2, Form Entry #3, Form Entry #4>
   Valid equivalent class:
   - <CourseNumber, CourseName, DepartmentName, Term>

   Invalid equivalent class:
   - <CourseNumber, CourseName, DepartmentName, ⌀>
   - <CourseNumber, CourseName, ⌀, Term>
   - <CourseNumber, ⌀, DepartmentName, Term>
   - <⌀, CourseName, DepartmentName, Term>
   - etc...
   - <⌀, ⌀, ⌀, ⌀>

   Boundary analysis:
   One outside the boundary test case would be an input string with only spaces, it should be rejected
   - <String with space only, ……>

   And one inside the boundary would be string with leading/trailing spaces, the leading/trailing spaces should be ignored, then rest of it is accepted
   - <String with leading/trailing spaces, ……>

Test Plan: We mocked four form submissions with jest. One submission with valid <CourseNumber, CourseName, DepartmentName, Term>, one with one field taken out (left empty), one submission with a field filled with space, and one with leading/trailing spaces. Then we see if the invalid submissions are rejected.

3. AddNote.test.js
   Description: This component simply renders a button and redirects the user to note upload page when clicked.

   Test Plan: we simply checked whether the component is rendered correctly with jest.

4. CommentEditor.test.js
   Description: This is the component for users to add a comment to a note. It includes a textarea for users to input the comment and a button for users to submit the comment.

   Input: <Text Entry>
   Valid equivalent class:
   - <Non-empty Text Entry>

   Invalid equivalent class:
   - <∅>

   Boundary analysis:
   One outside the boundary test case would be an input string with only space, it should be rejected
   - <String with space only>

   And one inside the boundary would be string with leading spaces, the leading/trailing spaces should be ignored, then rest of it is accepted
   - <String with leading spaces>

   Test Plan: we create 4 test cases, one with some valid comments, one with empty string, one with couple spaces, one with some spaces followed by a valid comment. Then we mocked clicking the submit button and see if the valid cases are submitted and invalid ones are rejected.

5. CommentListView.test.js
   Description: This component simply renders a list of comments for the user to view. We tested the component by mocking the object returned by the API endpoint.
   Input: <(From the API) List of comments in JSON Format>
   Valid equivalent class:
   - <JSON Object with>
   - <Empty JSON Object>

   Invalid equivalent class (the component simply renders nothing):
   - <∅>
   - <Invalid JSON Object>

   Test Plan: we simulated the API request with a mocked API endpoint. Then we checked if all comments in the provided JSON object were correctly rendered on page (by search for keywords).

6. CourseAutoComplete.test.js
   Description: This component is simply a search bar that autocomplete user's input into a course number
   Input: <Text>
   Valid equivalent class:
   - <Valid Course number in the database>

   Invalid equivalent class (nothing shown in the dropdown bar):
   - <Valid Course number not in the database>
   - <Invalid Course number>
   - <Random String>

   Test Plan: we simulated the user entering characters in the search bar and checked whether the items shown in the dropdown menu is correct.

7. CoursePage.test.js
   Description: This component renders a page that shows all courses in the database
   Input: <A serialized JSON string (from the API)>
   Valid equivalent class:
   - <List of courses in JSON string format>

   Invalid equivalent class (nothing rendered):
   - <∅>

Test Plan: we simulated the API request with a mocked API endpoint. Then we checked if all courses in the provided JSON object were correctly rendered on page (by search for keywords).

8. CustomLayout.test.js
   Description: This component renders the header and footer of all pages. It serves as a container for many components.
   No equivalent class for this component, it takes no input from the user/server. It simply serves as a template.

   Test Plan: we simply checked if the elements are correctly rendered (whether the text matches).

9. DeleteNote.test.js
   Description: This component simply renders a delete button on the note page. If the current user is not the uploader for the note, the component won't render the button.
   Input: <Cookie>
   Valid equivalent class:
   - <Cookie with same userID as the uploader>

   Invalid equivalent class:
   - <∅>
   - <Cookie with different userID as uploader>

   Test Plan: we mocked users visiting the page with different cookies and see if the button is rendered. Then we mocked the user clicking the button and checked whether a DELETE request was sent.

10. DepartmentsListView.test.js
    Description: This component renders a list of departments, users can click into a department to view all courses in that department.
    Input: <Serialized JSON String (from the API)>
    No equivalent class for this component. This page only serves the purpose of displaying departments to users

    Test Plan: we simply mocked the API endpoint and checked if the elements are correctly rendered (whether the text matches).

11. DropDown.test.js
    Description: This component renders a dropdown menu with a “Profile” button that takes the user to profile page and a “Logout” button that clears the Cookie.
    Input: ∅
    No equivalent class for this component. This component is mostly for aesthetic purposes.

    Test Plan: we simply mocked a user clicking the items in the dropdown menu and see whether the “Profile” button indeed redirects the user to the profile page and whether the “Logout” button actually clears the Cookie.

12. Favorite.test.js
    Description: This component renders a favorite icon. When clicked by the user, it sends a request to the server and adds the note to the user's favorite list.
    Input: <Click event>
    Valid equivalent class:
    - <onClick event when the note is not in user’s favorite list>
    - <onClick event when the note is in user’s favorite list>

    Invalid equivalent class:
    - <∅> (Failed to send POST request to the API)

    Test Plan: we simply mocked a user clicking the favorite icon and checked whether a “set favorite” POST request was sent to the endpoint. Then we mocked the user clicking the icon again and checked whether a “remove favorite” POST request was sent to the endpoint. If the POST failed, the add/remove favorite action is rejected and nothing is updated in the frontend.

13. NoteDetailView.test.js
    Description: This component serves as a container for Vote.js, Favorite.js, Comment.js, Preview.js
    No equivalent class for this component. This component is just a container.
    Test Plan: we simply checked whether all the inner components are rendered.

14. NoteListView.test.js
    Description: This component serves as a container for Note.js. It renders a list of note info and displays it to the user.
    Input: <Serialized JSON String (from the API)>
    Valid equivalent class:

- <List of notes in JSON string format>
- <Empty List in JSON string format>

Invalid equivalent class:

- <∅> (Failed to fetch from the API)

Test Plan: we mocked an API endpoint sending JSON objects to the component and checked whether all notes are rendered. If it fails to fetch from the API, nothing is rendered.

15. Notes.test.js

    Description: This component renders a card containing information about a certain note.

    Input: <JSON Object>

    Valid equivalent class:

    - <JSON Object fetched from the API>

    Invalid equivalent class:

    - <∅> (failed to fetch from API)

    Test Plan: we mocked an API endpoint sending JSON objects to the component and checked whether all notes details are rendered. If it fails to fetch from the API, nothing is rendered.

16. PersonalPage.test.js

    Description: This component renders the user's profile page. The page contains information such as "name", "email". The page also contains a subsection that displays the user's favorite notes.

    Input: <Cookie>

    Valid equivalent class:

    - <Cookie with user info>

    Invalid equivalent class:

    - <∅> (No Cookie provided)
    - <Cookie containing credential from user that doesn't exist>

    Test Plan: Since this page is mostly about displaying the user's information and doesn't involve too much user interaction. We simply simulated a user with the correct Cookie visiting the page and clicking on the page to transition between subsections and see if correct elements are rendered.

17. Preview.test.js
    Description: This component simply renders a preview for the pdf file
    Input: <URL to the pdf file>
    Valid equivalent class:
    - <URL to the pdf file>

    Invalid equivalent class:
    - <∅> (empty link)
    - <URL to other file format>

    Test Plan: Since our pdf previewer is implemented using the Google doc's pdf previewer, there is not too much we can test about it. We simply checked if the Google doc's previewer is rendered correctly.

18. UploadForm.test.js
    Description: This component renders the note upload form.
    Input: <Form Entry #1, Form Entry #2, Form Entry #3, Form Entry #4, File>
    Valid equivalent class:
    - <FileName, CourseNumber, DepartmentName, Description, PDF File>

    Invalid equivalent class:
    - <∅, CourseNumber, DepartmentName, Description, PDF File>
    - <FileName, ∅, DepartmentName, Description, PDF File>
    - etc...
    - <∅, ∅, ∅, ∅, ∅>
    - <..., Non-PDF File>

    Boundary analysis:
    One outside the boundary test case would be an input string with only space, it should be rejected
    - (For any text field) String with space only

    And one inside the boundary would be string with leading spaces, the leading/trailing spaces should be ignored, then rest of it is accepted
    - (For any text field) String with leading spaces

    Test Plan: All the fields must be non-empty and non-all-spaces, and the uploaded file must be in PDF format. We simply created test cases with some invalid fields. And checked whether the component is able to catch the error and reject it. For the valid test case, we checked whether a valid POST request is sent by the component.

19. Vote.test.js
    Description: This component renders the upvote and downvote button. Clicking the corresponding button will upvote/downvote the note. When the user has upvoted/downvoted, clicking the same vote button will revert the vote. When the user has upvoted/downvoted and the user clicks the other vote button will overwrite the previous vote.
    Input: <ClickEvent>
    Valid equivalent class:
    - <onClick upvote when the user hasn't voted yet>
    - <onClick upvote when the user has upvoted>
    - <onClick upvote when the user has downvoted>
    - <onClick downvote when the user hasn't voted yet>
    - <onClick downvote when the user has upvoted>
    - <onClick downvote when the user has downvoted>

    No Invalid equivalent class, the user can click the button in any way he/she wishes and the action is defined.

    Test Plan: We simply simulated a user clicking the buttons in the ways we defined above and see if the vote count on the note is correct.

20. googleLogin.test.js AND googleLoginService.test.js
    Description: The two components are used to authenticate the user with Google Login service. People with non @columbia.edu email are not allowed to login.
    Input: <Email, Password>
    Valid equivalent class:
    - <Email @columbia.edu , Valid password>

    Invalid equivalent class:
    - <Non @columbia.edu email, Valid password>
    - <@columbia.edu email, Invalid password>
    - <Non @columbia.edu email, Invalid password>

    Test Plan: We simply simulated a user login through Google Login and checked whether the correct Cookie is stored after a valid login. And no Cookie is stored after an invalid login.

21. routes.test.js
    Description: This is the react router module that defines the transition between pages. It serves as a container for all pages.
    No equivalent class for this component. This component is just a container.
    Test Plan: we simply checked whether the component is rendered.

---------- Backend Test Plan ----------

The equivalence partitions and boundary conditions necessary to unit-test each of the major methods in the backend are described below.

1. http://127.0.0.1:8000/api/user/:
    a. Valid Get
        i. Get a user by a valid user id
        ii. Get a user by a valid user email
    b. Invalid Get
        i. Get a user by an invalid or nonexistent user id
        ii. Get a user by an invalid or nonexistent user email
    c. Valid Add
        i. Add a valid new super user
        ii. Add a valid new regular user
    d. Invalid Add
        i. Add a duplicate user
        ii. Add a new user with invalid email address, i.e. non-columbia email
    e. Valid delete
        i. Delete a valid regular user
        ii. Delete a valid super user
    f. Invalid delete
        i. Delete a nonexistent user

2. http://127.0.0.1:8000/api/note/:
    a. Valid Get
        i. Get notes by a valid course number
    b. Invalid Get
        i. Get notes by an invalid or nonexistent course number
    c. Valid Add
        i. Add a note to a valid course
    d. Invalid Add
        i. Add a note to a nonexistent course
        ii. Add a duplicate note
    e. Valid Delete
        i. Delete a valid note
    f. Invalid equivalence class
        i. Delete a nonexistent note

3. http://127.0.0.1:8000/api/comment/:
    a. Valid Get
        i. Get a comment by a valid id
    b. Invalid Get
        i. Get a comment by an invalid or nonexistent id
    c. Valid Add
        i. Add a comment with a valid note id
    d. Invalid Add
        i. Add a comment with an invalid or nonexistent note id
    e. Valid Delete
        i. Delete a valid comment
    f. Invalid Delete
        i. Delete a nonexistent comment

4. http://127.0.0.1:8000/api/vote/:
    a. Valid Get
        i. Get a vote by a valid id
    b. Invalid Get
        i. Get a vote by an invalid or nonexistent id
    c. Valid Add
        i. Add a valid upvote
        ii. Add a valid downvote
    d. Invalid Add
        i. Add upvote with nonexistent user
        ii. Add downvote with nonexistent user
        iii. Add upvote with nonexistent note
        iv. Add downvote with nonexistent note
        v. Add an duplicate upvote
        vi. Add an duplicate downvote
    e. Valid Undo Vote
        i. Undo a valid upvote
        ii. Undo a valid downvote
    f. Invalid Undo Vote
        i. Undo a nonexistent upvote
        ii. Undo a nonexistent downvote

5. http://127.0.0.1:8000/api/course/:
    a. Valid Get

        i. Get course info by a valid course number
    b. Invalid Get
        i. Get course info by an invalid or nonexistent course number
    c. Valid Add
        i. Add a new valid course with valid department name
    d. Invalid Add
        i. Add a new valid course with invalid or nonexistent department name
        ii. Add a duplicate course
    e. Valid delete
        i. Delete a valid course
    f. Invalid delete
        i. Delete a nonexistent course

6. http://127.0.0.1:8000/api/favorite/:
    a. Valid Get
        i. Get a favorite record by a valid id
    b. Invalid Get
        i. Get a favorite record by a invalid id
    c. Valid Add
        i. Add a new favorite record with valid user id and note id
    d. Invalid Add
        i. Add a new favorite record with invalid or nonexistent user id and valid note id
        ii. Add a new favorite record with valid user id and invalid or nonexistent note id
        iii. Add a new favorite record with invalid or nonexistent user id and note id
        iv. Add a duplicate favorite record
    e. Valid Delete
        i. Delete a valid favorite record
    f. Invalid Delete
        i. Delete a nonexistent favorite record

7. http://127.0.0.1:8000/api/department/:
    a. Valid Get
        i. Get department info by a valid department name
    b. Invalid Get

        i. Get department info by a invalid or nonexistent department name
    c. Valid Add
        i. Add a new department
    d. Invalid Add
        i. Add a duplicate department
    e. Valid Delete
        i. Delete a valid department
    f. Invalid Delete
        i. Delete a nonexistent department

Part 3:

Backend coverage reports:

https://github.com/wx2227/4156/tree/master/backend/cover

Coverage achieved 100% for all branches and statements.

Frontend coverage reports:

https://github.com/wx2227/4156/tree/master/frontend/coverageReport

Coverage achieved 100% for all branches.

For DepartmentListView.test.js, even if we run all equivalence tests and boundary tests, and all lines are covered by the test, the Stmts and Funcs did not achieve 100% for some reason. We believe it might be caused by ToggleButtonGroup, a react-bootstrap component used to achieve switching-tab effect in the department page. This component is wrapped by the react-bootstrap library, and we cannot test the implementation detail in this case. As a result, this is the best coverage test we can do.

Part 4:

Travis CI Configuration:

https://github.com/wx2227/4156/blob/master/.travis.yml

Travis CI Build History Report:

https://travis-ci.com/github/wx2227/4156