Nudging

Developer: Matthew Chung

Submitted April 15, 2023

Reviewer: Judy Li

Reviewed April 17, 2023

Criteria

Major Positives & Negatives

Positives

- The engineer's intentions are clear through the designs. They are easy to read and follow the same structure for both diagrams.
- The designs follow an MVC architecture, with a model (FormNudge), view (ChoreListToDoTab), and controller (NudgeController).

Negatives

- Logging is not included in the diagrams. This will make it more difficult for the developer to implement this functionality. *
- There is return type for the calls included in the frontend (ChoreListToDoTab) but not the backend components (Controller, NudgeManager, NudgeDAO). The return response returns a 'result' but it's not clear if it's a class and what its values will be.
- The Sendgrid column is included but not used in the success diagram which can hinder readability.
- There is no frontend or backend validation included. The BRD states that the user should be authorized before using the nudge feature and that the user is only to use this feature once a day. *
- The design doesn't follow the team's high level architecture as It doesn't include a service layer. *
- The design shows that all function calls (NudgeController, NudgeManager) are synchronous while calling the SendGrid API asynchronously.[2] This arrangement will work but does not take advantage of the resource

* Design recommendation given

utilization, improved responsiveness, and increased scalability an asynchronous design can bring. *
- The Sendgrid API should not interact with the ChoreListToDoTab in the failure diagram. It is an asynchronous call so we should not require a response.
- The Nudge database entries are not sufficient to cover the functionalities of the feature. *

Unmet Requirements

- *Sending a nudge will be logged to a database in the form of activity and/or errors (if any). [1]*
  - Logging is not included in the diagrams. This will make it more difficult for the developer to implement this functionality.*
- *Only* users with an active authentication session and access to the space can utilize this feature. *[1]*
  - There are no calls to validate the user's claims before allowing them to use the feature in the frontend and backend.*
- *Users can only nudge one assignee per day. [1]*
  - There is no validation so this is an unmet requirement.*
- *The task being nudged must be incomplete. [1]*
  - There are no frontend validations so this is an unmet success case.*

Design Recommendations

- Success and failure logging should be added to the NudgeManager layer to record that a nudge has been sent.
  - Recommendations: Logs should be added to the:
    - Controller: Logging user input, request details, and response information can be added in the controller to capture high-level information about the incoming requests and the overall flow of control.
    - Manager: Logging business logic related to nudging, such as data validation, and error handling. This can help capture information related to the processing of nudging logic and any business-specific events or errors.
    - Service: Logging user authorization can help record

* Design recommendation given

        - Data Access: Logging database operations (inserting) related to nudging, can provide visibility into interactions with the database and errors/exceptions that could occur.
    - Positives: Logging allows the developer to locate the problem quickly when an error/exception arises or if there are any suspicious activities. The data from the logs can also be used in analytics to monitor application health, and provide data to improve the application in the future. This also covers the business rule: Sending a nudge will be logged to a database in the form of activity and/or errors (if any). [1]
    - Negatives: Logging may affect performance of the application and increase code complexity.
- Add frontend and backend validation.
    - Recommendation: Frontend Validation should be added to check if the user is authorized using the global variable, auth. Backend validation should be implemented to call the Authorization function ValidateClaim(UserAccount ua, Claim targetClaim) to double check whether the user is authorized to perform nudging. Backend validation should also be included to check whether a feature is incomplete and if a nudge has already been made in the past 24 hours.
    - Positives: Filtering out unauthorized users with frontend validation can avoid unnecessary API requests to the backend, reducing the load on the backend server and improving performance. Backend validation should always be added to check the authenticity of the frontend request and to check that business rules are met before completing the request. This also covers the business rule: Only users with an active authentication session and access to the space can utilize this feature. [1]
- The current database setup does not allow the required functionality of nudging to be implemented.
    - Recommendation: The Nudge database should also include an entry date, as well as the task being nudged. This will allow for the validation and completion of the business rules: Users can only nudge one assignee per day and The task being nudged must be incomplete. [1] The validation can be completed by querying the Nudge database for the task within 24 hours.
    - Positives: It will cover a business rule, and allow backend validation.
    - Negatives: None

* Design recommendation given

- The diagram indicates that all controller, manager, and data access function calls are synchronous. However the SendGrid API function call is asynchronous.
    - Recommendation: Make the controller, manager, and data access function asynchronous to increase resource utilization, improved responsiveness, and increased scalability, especially in scenarios with high concurrency, where blocking threads can negatively impact performance. In this way, the manager is not just waiting for the SendGrid API to complete.
    - Positives: When you have an asynchronous method at one level of your application, such as in a controller, manager, or service function, it allows that level of the application to continue executing other tasks while awaiting the completion of the asynchronous operation. This can lead to better resource utilization, improved responsiveness, and increased scalability, especially in scenarios with high concurrency, where blocking threads can negatively impact performance.
    - Negatives: Increased code complexity and maintainability.

Test Recommendations

Integration Tests

1. SendGrid API Integration Test: Write integration tests to ensure that the SendGrid API is properly integrated with your feature nudging functionality. This could include tests to verify that emails are being sent correctly with the expected content, recipients, and attachments. You can use a test double or mock to isolate the test from actually sending emails during the test run.
2. Manager Integration Test: Write integration tests for the manager layer, which is responsible for coordinating the interactions between the feature nudging functionality and the SendGrid API. These tests can cover scenarios such as creating nudges, scheduling nudges, and tracking the status of nudges in the system. You can use test doubles or mocks for any dependencies that the manager relies on, such as the DAO layer or the SendGrid API.
3. DAO Layer Integration Test: Write integration tests for the Data Access Object (DAO) layer, which is responsible for interacting with the MariaDB database. These tests can include scenarios such as inserting nudges into the database, retrieving nudges from the database, and updating the status of nudges in

* Design recommendation given

the database. You can use a test database or a test double for the MariaDB dependency during the test run.

Unit Tests

1. Manager Unit Test: Write unit tests for the manager layer to test individual methods or functions in isolation. These tests can cover scenarios such as validating input parameters, handling errors, and checking that correct authorization, and chore's list function calls are made. You can use test doubles or mocks for any dependencies that the manager relies on, such as the DAO layer or the SendGrid API.
2. DAO Layer Unit Test: Write unit tests for the DAO layer to test individual methods or functions in isolation. These tests can cover scenarios such as inserting data into the MariaDB database, retrieving data from the database, and updating data in the database. You can use a test database or a test double for the MariaDB dependency during the test run.
3. MariaDB Unit Test: Write unit tests specifically for testing the interactions with the MariaDB database. These tests can include scenarios such as setting up the database, inserting test data, retrieving test data, and cleaning up the database after the tests are run. You can use a test database or a test double for the MariaDB dependency during the test run.

* Design recommendation given

References

[1] githelsui/WeCasa: @ Team HAGS JP. (2023, Feb 17) *GitHub*
https://github.com/githelsui/WeCasa/blob/main/docs/BRD%20v2.0.pdf

[2] Messages in UML diagrams. (2021, March 5) *IBM*
https://www.ibm.com/docs/en/rsm/7.5.0?topic=diagrams-messages-in-uml

* Design recommendation given