Budget Sharing Bar

Developer: Judy Li

Submitted February 17, 2023

Reviewer: Allison Austin

Reviewed February 17, 2023

Criteria

Major Positives & Negatives

Positives

- Each of the sequence diagrams include logging. This makes it easy for the developer to integrate logging into the feature and know what information goes into the logs.
- Messages, queries, method signatures, and return values are detailed. For method calls, it is clear what the function is named, when it is called, how many/which variables are used in the list of parameters, and the data type that gets returned. Axios calls also include the request, URL, and payload, which specifies how the server will be configured for handling these requests.
- Designs are easy to read, as they follow a consistent format and all have a cascading request/response structure. Each object has a lifeline for its constructor and the return value to the correct dependent object.

Negatives

- The sequence diagrams are missing input validation. In the BRD, the bill has valid characters and length limits for the name, description, and value of each bill. The sequence diagrams do not include each of these checks and which components are responsible for carrying out these checks.
- Design does not entirely follow MVC architecture. Each sequence diagram utilizes a view (BudgetBarView) and a controller (BudgetBarController), but the design doesn't include a model, such as an object that defines the data structure for a bill. Some of the method calls include "method(Bill bill)", which

suggests that bills will be created using a defined model, but the model is not used in any of the sequence diagrams.

- The design for adding a bill has a loop surrounding the query, because each entry to the bills table represents one bill for one member, so members that share a bill will require multiple entries into the database. Not only does this pose the risk of running multiple unnecessary queries, but it can also cause the table to fill up quickly. This design is also not consistent across all of the diagrams. For instance, when updating a bill, it only runs one query, and updates only the bill with a specific bill id. But if the design has multiple entries in the table for the same bill, just different members, all of those entries would also need to be updated so all users view identical bills after it is updated (same applies for bill deletion, where a single bill for one user is deleted, but not for other users that share the same bill).
- The return values of the database layer are not accurate/detailed. The database would not return “rows == 1”, it returns the rows affected if you use command.ExecuteNonQuery(). The GroupDAO compares the number of rows affected by the query and makes sure it matches the expected value. “Return matching records” does not provide the developer enough detail on the type/structure of data that gets returned from executing a select statement from the DAO.
- DeleteOutdatedBills in the Get Budget Bar View sequence diagram needs to include the group_id in the query so bills from other groups don’t get deleted.

Unmet Requirements

- *Budget values have no upper-bound limit, however, progression towards the budget may not exceed the defined budget resulting in a negative budget readout. [1]*
    - The diagrams are missing input validation. Hence, this requirement is not met by the design.

Design Recommendations

- Input validation should be implemented at the frontend (in BudgetBarView), and should include the use case of a user uploading a bill with a value that would exceed the defined budget. The additional check would be the following:

$$(Value\ of\ new\ bill\ +\ Amount\ Spent\ for\ the\ month)\ >\ Total\ Monthly\ Budget$$

    - **Positives**: Checking this before the new bill gets added reduces the number of server requests. All of the values needed to complete these checks can be accessed in the BudgetBarView component.
    - **Negatives**:
- For adding, updating, and deleting a bill, I would update the queries and number of database connections based on the design you want to go with. If a single entry into the bills table is assigned to only one member, and there can be multiple members sharing a bill, then you would need to update all records in the table for each user that shares the bill being updated/deleted. If you want to do this with a single query, you can use the bill_name as a shared column, but you need to add a check when creating a bill that a bill with the same name has not already been created.
    - **Positives**: Changes to the existing design are minimal.
    - **Negatives**: You would have to run potentially several different queries. If one of these queries fails, then the bill might not be consistent across all users.
- Alternatively, you can mimic the design for storing user claims in the database. In this case, a single entry in the bills table would represent one bill, and you would have one column for the users that share the bill, and a list of users (stored as a JSON object) in the users column.
    - **Positives**: Only a single query is required for adding, deleting, and updating a bill. Also, failure at the data access layer will not cause inconsistent budgets/bills across different users, and the number of affected rows returned by the database can be cross-checked with the number of users assigned to the bill. A similar solution already exists in the project for representing claims for a user, so most of the implementation can be copied over to the bill management.
    - **Negatives**: Requires storing the users as a JSON object in MariaDB and converting the returned object to a list of Users using JSON serialization. This is extra work on the backend.
- The Bills table should have the group name included as a column. This allows for bills by the same user in separate groups to be distinct at the data access level.

    - **Positives**: Avoids unnecessary querying when grouping by group_id and username, rather than querying the User table for users in the same group, and using a list of usernames when selecting from the Bills table.
    - **Negatives**: Requires updating the entire design, including the database setup, models, and data access objects.
- When pulling up the BudgetBarView, you are running five queries. This can become excessive for a feature that gets navigated to very often. Also, the deletion of bills after 24 hours should be handled by the database server, not the application. To achieve this, you can create scheduled events in MariaDB that run every 24 hours on the Bills table. These events will be responsible for removing bills whose date_entered is older than one month or whose date_deleted is older than 24 hours.
    - **Positives**: This offloads the responsibility of bill deletion from the application. If the operation fails, the application is not affected, and the number of server requests will be significantly reduced.
    - **Negatives**: Requires researching the process of creating two scheduled events through the MariaDB server, which can be tricky and/or time consuming to set up.

Test Recommendations

Integration Tests

- To test updating/deleting a bill, I would manually create, update, and delete bills and make sure that the bill gets updated and/or deleted for every user who shares that bill.
    - Arrange a test bill to update and delete. Make sure there are at least two users in the same group who share the bill.
    - After updating the bill, call GetBills on the group_id of the two users and check that the Bill objects are the same.
    - Delete the bill and call GetBills on the group_id of the two users.
    - Check that the deleted attribute is the same for both users.
    - In regards to the requirements, this test ensures that bills shared by members of a group are consistent even after they are updated and/or deleted by another member. Without a passing mark for this test, bills shared by users might be displayed differently when different users are

logged in and viewing the Finances tab, which defeats the purpose of sharing bills and negatively impacts overall user experience.

- To test for old/deleted bills being properly removed from the database, you can manually create bills and run the scheduled events.
    - Arrange a month-old bill and a test bill to delete.
    - Run the scheduled job either by (1) changing the schedule to run every minute or (2) manually run the jobs using the command line.
    - Check the system under test that the bills do not get returned from the GetBills() return object.
    - This test ensures that the requirement for removing old and deleted bills is being met by the database server. Without a passing mark for this test, old and deleted bills might not be removed properly from the database and querying the historic bills list might become overwhelming for the system and negatively impact user experience.

Unit Tests

- To test input validation, I would attempt to add a bill with invalid input fields for the bill name, description, and value. I would check for string length and valid characters for each field. The most important assertion is that the error does not come from the data access layer, but the view/controller layers.
    - I would also attempt to assign a bill to a user that is not in the group, and add a bill that would exceed the budget. Tests for these use cases should also not reach the data access layer.
    - These tests ensure that the requirements for user input are met by the system. Without a passing mark for this test, invalid bills might reach the data access layer only to be rejected, which can overwhelm the server and negatively impact user experience.

References

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