DAT076: Web applications

Project Report

Group 7

Kevin Collins

Annelie Hansson

Liam Mayor

Philip Öhman

21 Pages

Contents

1 Introduction

As students, managing finances can be overwhelming, especially when existing tools are either too time-consuming or too complicated, making them feel daunting to use. This was a problem faced by all four members of this group, and although we have tried to use tools like Excel, we struggled with being consistent over longer periods. We want to improve on this, and we thought that this course gave us the perfect opportunity to tackle this problem. That's why we have developed *Budgie* – a simple and efficient budget planner and expense tracker designed for the everyday student.

Budgie enables users to easily plan out their finances by creating budgets as well as monitor how closely they follow this plan, using expense tracking. Expected features like adding, editing and removing budgets and/or expenses are easily accessible. Furthermore, the application also helps the user understand where their money is going by visualising their expenditure through a pie chart highlighting how much the user has spent in each category. The overarching purpose of *Budgie* is to be extremely easy to use so that students can start managing their finances without any roadblocks. Our vision is that *Budgie* can act as a gateway into the complex world of personal finance and that its simplicity encourages the user to stay consistent and develop healthy financial habits.

Link to GitHub repository: https://github.com/Lee4-M/DAT076

2 List of Use Cases

Note: We have not included login as its own use case (but rather, combined it with Use Case 1) as just logging in and not doing anything else does not accomplish much for the user. We considered whether register should be a use case, and although we think there is a stronger argument for register than login, we have decided to omit it from this list.

Note: Use Cases 2-11 require the user to be logged in.

1. As a user, I want to be able to log in and view my budget along with my expenses, so that I can get a quick overview of my finances.
2. As a user, I want to be able to add a budget row, so that I can create a more detailed budget plan.
3. As a user, I want to be able to delete a budget row, so that I can modify my budget plan.
4. As a user, I want to be able to add an expense with a specific cost and category, so that I can view and compare all my costs for different categories.
5. As a user, I want to be able to delete an expense that I logged previously so that I have the flexibility to not log certain expenses (f.e. I get a refund).
6. As a user, I want to be able to view my expenses by category on a pie chart, so that I can get an intuitive understanding of my spending for each category.
7. As a user, I want to be able to edit my existing budget amount, so that I have the flexibility to modify my plan in case my circumstances change.
8. As a user, I want to be able to edit my existing expenses, so that I have the flexibility to quickly input expenses and modify them later (f.e. by adding a description).
9. As a user, I want to be able to rename my existing categories, so that I have the flexibility to change category names in case I regret my current category name
10. As a user, I want to be able to move my existing expenses from one budget to another, so that I have the flexibility to re-assign expenses in case I place them under the wrong budget.
11. As a user, I want to be able to sort my expenses and budgets by ordering them by category, budget amount, total expenses per budget or budget variance, so that I can get a better overview of my finances.

3 User Manual

To begin installing the application, please visit our GitHub repository and clone the repository to a local folder: https://github.com/Lee4-M/DAT076. In the server directory, create a .env file with a session secret in the following style:

```
SESSION_SECRET = "<Enter your secret here>"
```

In two terminal windows, enter both the client and server directories separately and run the command `npm install` in each.

To run a local instance of the database in the server-side application, use the command `run npm dev`. Otherwise, our version of the '.env' file and a certification file 'ca.pem' is required to run `npm dev-online`, which will start an online instance of the database.

To run the front-end, run the command above in the 'client' directory as well. Open http://localhost:5173/ in a browser of your choice. This will take you to the login screen of our Budgie application, where you can register your user and enter the same credentials in the login form to enter the main page of the application:

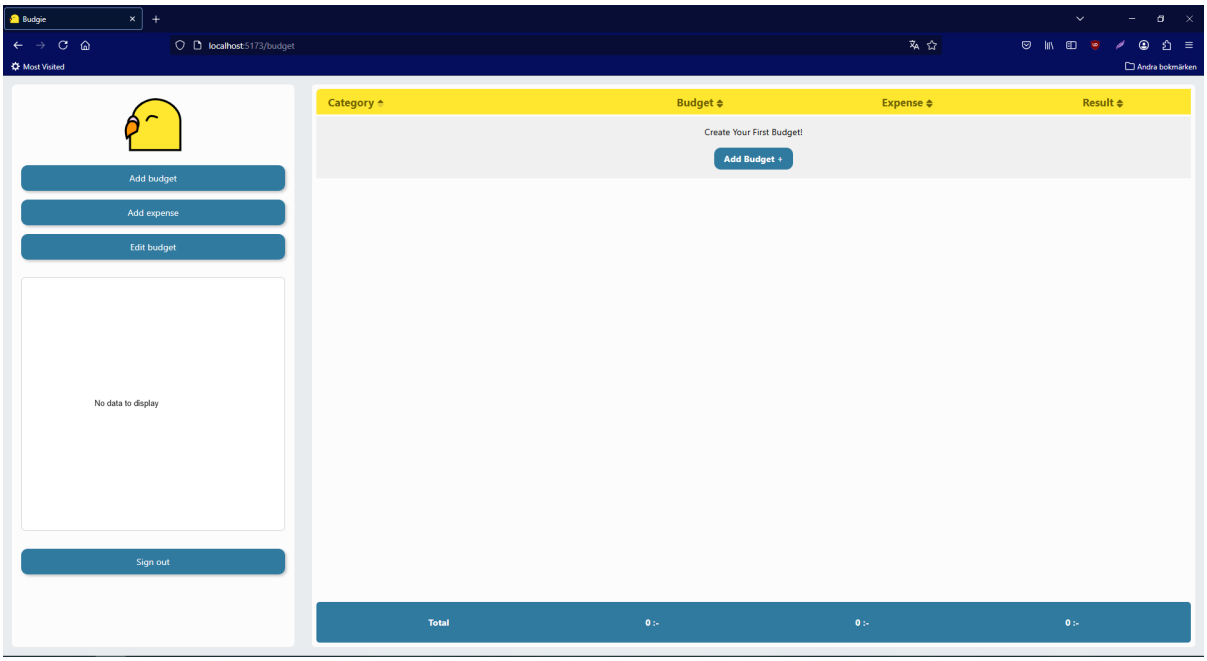

Figure 1: Main Page of *Budgie*

3.1 Use Case 1

Enter username and password and click the "Log In" button. This will take you to the home page with a table of your account's budgets and expenses.

3.2 Use Case 2

Click on the 'Add Budget' button in the sidebar or the budget table (if no previous rows exist currently), which will take you to the budget modal panel. Fill in the fields for a budget in the form and click 'Save Budget' when done. The budget row will now appear in the budget table

3.3 Use Case 3

Click on the budget row you want to delete, after which a panel will drop down with a button 'Delete Budget'. Clicking the button will delete the budget row.

3.4 Use Case 4

Click on the 'Add Expense' button in the sidebar or in the drop-down panel of an existing budget row that has no current expenses, which will take you to the expense modal panel. Pick a category from the drop-down menu if there exists at least one budget category, or enter a category for a new budget row manually. Fill in the other fields accordingly and click 'Save Expense' when done. The new expense will now appear under the budget row it has been assigned to, inside the panel that appears when you click on the budget row.

3.5 Use Case 5

Click on the budget row under which the expense you want to delete is assigned to. A panel will drop down with a button to 'Edit expenses in the bottom left corner'. A trash can icon beside the expense should appear when the button is pressed, clicking the icon will delete the expense.

3.6 Use Case 6

To view the pie chart, add and assign expenses to budget rows, after which a pie chart will appear in the sidebar displaying the chart. Refer to Use Case 4 to understand how to add expenses.

3.7 Use Case 7

Click on the 'Edit budget' button in the sidebar, after which input fields will be present in the budget amount cells to change the values as preferred. To save the changes, press enter on your keyboard or click on the 'Save Changes' button in the sidebar.

3.8 Use Case 8

Click on an existing budget row, after which a panel will drop down with a button 'Edit Expenses'. Input fields will be present in the cost and description cells to change the values as preferred when the button is pressed. To save the changes, press enter on your keyboard or click on the 'Done' button.

3.9 Use Case 9

Click on the 'Edit budget' button in the sidebar, after which input fields will be present in the category cells to rename the category as preferred. To save the changes, press enter on your keyboard or click on the 'Save Changes' button in the sidebar.

3.10 Use Case 10

Click on the two existing budget rows that you want to move the expense(s) from and to, after which panels for the expenses will drop down. Press and hold on the expense that you want to move and drag it to the new budget row that you want to assign it to. Drop the expense in the row, after which the expense will now appear under.

3.11 Use Case 11

Click on the table headers to order the rows by category, budget, expense or result, given that there already exists budget rows in the table. The budget table should now be sorted by the header chosen.

After having done all of the use cases, the application should somewhat look as following:

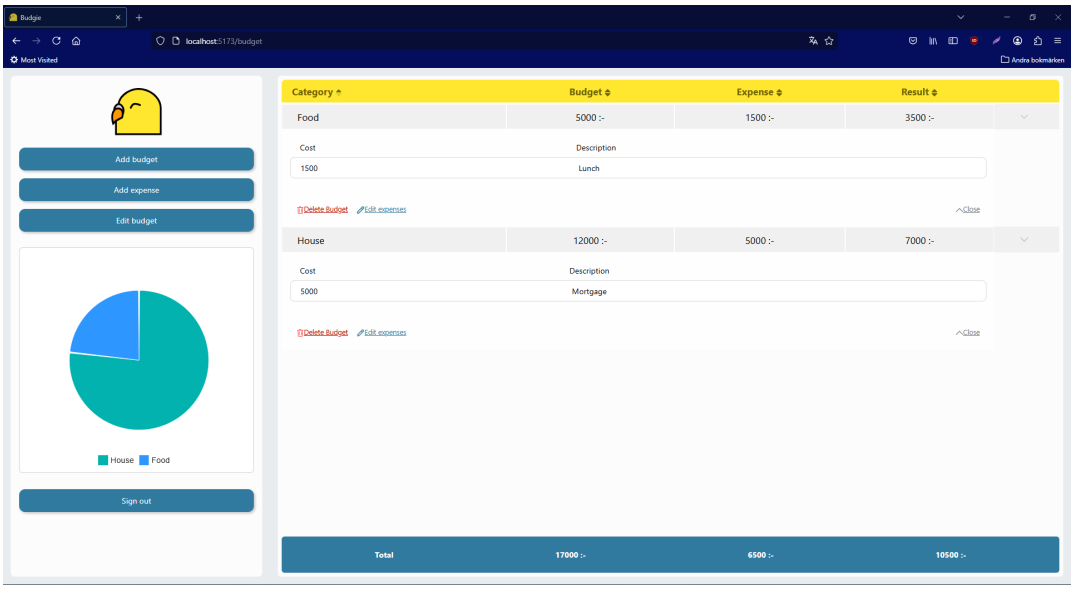

Figure 2: Example Usage of *Budgie*

4 Design

4.1 Frontend

The following sub-titles are written so that the reader can use them as a sort of technical specification, and it should be possible to learn about any one component without having to know about everything else. These are all the components that make up the frontend which the user can interact with (except App).

4.1.1 App

App is the main page of our web application, creating the Sidebar as well as the `BudgetTable` components. It's also responsible for keeping track of many of the application's states, and passing these as well as top-level functions down to child components.

These states include `budgets`, `editedBudgets`, `isEditing` and `expenses`. As one might expect, `budget` is the list of all the budgets displayed in `BudgetTable`, and `expenses` keep track of all expenses. The `expenses` map matches each list of expenses to a `budget_id` as `expenses` are always linked to a `budget`. The state `isEditing` holds a boolean for when the user is currently editing the budgets, and `editedBudgets` is a temporary state used to store changes when the user is in editing mode. Although this implementation requires an extra state, we found this to be better than only using a budgets state as if the user decides to, for example, not save their changes, their edits can simply be discarded (as budgets hold the original budgets). Furthermore, the `editedBudgets` state allows the frontend to be updated dynamically - mirroring user input without any issues - but only sends an API request once the user has saved their edits (rather than, for example, updated budgets in the backend on every keystroke).

Since a lot of functions used in child components are passed down from App, a lot of the actual calls to the API are made here in order to update and synchronize the states. The following API calls are made in App: `getBudgets`, `getExpenses` and `updateBudgetRows`. By passing a `budget_id` into `getExpenses`, this function gets all expenses related to that budget, while `getBudgets` simply returns a list of all the budgets. After a user has edited their budgets and saved, the budgets state is synced with `editedBudgets` and `updateBudgetRows` is used to update the backend with these changes.

4.1.2 Sidebar

The `sidebar` is the other large component of the main page aside from `BudgetTable`. It provides useful tools for the user to add or edit a budget, add an expense as well as presents a visualisation of the user's expenses. There is also a button for signing out. This component is meant to complement the functionality in `BudgetTable`, as well as move some of the functionality out of `BudgetTable`, so that it doesn't visually clog up the important overview found in the table.

There are five properties that `Sidebar` uses. Firstly and secondly, `budgets` and `expenses`, are the states passed down from App as mentioned above. As `Sidebar` contains the 'Edit

budget' button, `isEditing` and `handleSaveBudgetRows` are also passed down from `App`. `IsEditing` is updated as the user clicks 'edit budget' or 'save changes', the latter only appearing when `isEditing` is true and calls `handleSaveBudgetRows` when clicked. Lastly, it also receives `loadBudgets` from `App`, which is further passed down to `ExpenseModal` and `BudgetModal`.

The sidebar maintains two states, `showBudgetModal` and `showExpenseModal`, both booleans responsible for whether or not the `BudgetModal` and `ExpenseModal` will be shown to the user. These are also passed down to their respective components so that their visibility can be toggled.

Since the responsibility of making backend calls for adding expenses and budgets are tasks for the `BudgetModal` and `ExpenseModal`, the sidebar only makes one call to the backend, and that is when the user clicks 'Sign out'. A call is then made to log the user out through `apiLogin`.

4.1.3 BudgetModal

The `BudgetModal` is a modal panel that is displayed when you click the 'Add Budget' button in the sidebar, or the identical button in `BudgetTable` when there are no budgets. It's supposed to take inputs from the user for a new budget (visually represented in `BudgetTable` by a `BudgetRowComponent`), prompting the user for a category name and the amount for that budget. It has an attribute 'show' from the state `showBudgetModal` passed down from `Sidebar`, dictating whether or not it is showing on the screen, as well as functions for closing the modal panel and saving its contents. Both of these functions are handled by their parent, where closing the `BudgetModal` sets `showExpenseModal` to false, and the save function additionally updates the budgets with its parent, Sidebar, calling `loadBudgets` from `App`.

`BudgetModal` makes use of two additional states, `category` and `amount`, to be able to keep track of the user input as the user is typing. When clicking 'Save Budget', it makes a call to the backend through the API, using the function `addBudget` with arguments `category` name and amount for this new budget that the user created. A successful call to the backend will result in adding a new budget (`BudgetRowComponent`) based on the user's input.

4.1.4 ExpenseModal

This component is the input modal panel which pops up after the user clicks the 'Add expense' button. This button exists in the sidebar, as well as in the `ExpenseAccordion` when it is expanded but has no expenses. It is meant to let the user add and save an expense, including the category it belongs to, its cost and, if the user wants, a description of the expense. Having this functionality as its own component in the code base means we don't have to duplicate code since both the `Sidebar` and `ExpenseAccordion` want the same functionality for adding an expense.

The `ExpenseModal`, just like `BudgetModal`, takes in `show`, `handleClose` and `onSave()` from `Sidebar`, and they provide the same functionality as described earlier. Since the

ExpenseModal takes inputs, we want to update these input fields dynamically. It has states for cost, description and expenseCategory. Additionally, it has a state 'budgets', for keeping track of which budgets are available for the user to choose from - shall they want to pick an already existing budget. Additionally, it also maintains the states customCategory and expenseCategory, which work together to allow the user to input a new category to link the added expense to.

Since the user should be able to pick from existing budgets, ExpenseModal makes use of the API call getBudgets. It also uses addExpense so that the user's input becomes an expense.

4.1.5 BudgetTable

This is the main content of the web application, and it shows the user all of their budgets, where one budget is one BudgetRowComponent with a category (for example housing, food, entertainment etc.) and its associated amount. The BudgetTable presents the sum of all expenses per budget as well as the budget variance, written as result (budget amount - total expenses linked to that budget). Additionally, towards the bottom, it shows the sum of all budgets, the sum of all expenses and the sum of budget variances.

The user can sort the BudgetTable by category, budgeted amount, total expenses per budget or the budget variance. Each row in the BudgetTable is represented by a BudgetRowComponent, containing information about that particular budget, as well as the expenses under the same category which are displayed when the user clicks that BudgetRowComponent or the little arrow at the rightmost edge of the BudgetRowComponent.

Among its properties are budgets, expenses, loadBudgets, loadExpenses and isEditing, all passed down from App. The props handleChangeBudgets and handleSaveBudgetRows are also received from App and are then passed down further to BudgetRowComponent. The prop loadBudgets is passed to BudgetModal but loadExpenses is used directly in BudgetTable, to enable the feature where a user can drag and drop an expense from one BudgetRowComponent to another.

BudgetTable keeps track of three additional states: sortBy, sortOrder and showBudgetModal. The state sortBy dictates which one of the headers to use for sorting the budgets (category, budget amount, sum of expenses or budget variance), which is complemented by sortOrder keeping track of whether this should be done in ascending or descending order. showBudgetModal simply holds a boolean on whether the BudgetModal is showing or not. This is the same BudgetModal that can be found by clicking 'Add budget' in the Sidebar. The reason this functionality is also reachable from BudgetTable is that we have added an extra button to prompt the user to add a budget in the BudgetTable itself when no budgets exist.

Only one call to the backend is made from BudgetTable, this is when a user has dragged an expense from a BudgetRowComponent's ExpenseAccordion over to another BudgetRowComponent. BudgetTable then calls updateExpense in the API, which changes that particular expense's budgetId (the field that dictates which budget an expense belongs to).

4.1.6 BudgetRowComponent

`BudgetRowComponent` is the front-end representation of a budget. It shows the category name, the amount budgeted, the sum of all expenses under this category, and the budget variance (budget amount - sum of expenses). Additionally, it allows the user to click on it (or the arrow towards the right of it) to 'expand' the component, showing more detailed information about the associated expenses. Each `BudgetRowComponent` is represented visually by a row in the `BudgetTable`, and each `BudgetRowComponent` has its own `ExpenseAccordion` (an accordion of expenses that you can expand).

The props of `BudgetRowComponent` include a budget, a list of associated expenses, and a boolean `isEditing` with the same functionality as mentioned earlier. In addition to this, its props that are functions include `loadBudgets`, `loadExpenses`, `handleChangeBudgets` and `handleSaveBudgetRows`. The budget is the backend version of a `BudgetRowComponent`, and the list of expenses is what is passed down to each `BudgetRowComponent`'s `ExpenseAccordion`. Both of these properties are passed and connected by `BudgetTable` with the use of each expense's `budget_id`.

Since you can delete a budget by expanding the `BudgetRowComponent`'s `ExpenseAccordion` and clicking the 'delete budget' button, we also need `loadBudgets` to update the front end after the `BudgetRowComponent` has deleted its backend counterpart. `BudgetRowComponent` receives `loadExpenses` as it needs to pass it further down to `ExpenseAccordion`. Lastly, `handleChangeBudgets` and `handleSaveBudgetRows` are used when editing (and saving the edited) budgets.

Clicking on a `BudgetRowComponent` will open up the `ExpenseAccordion`, and since each `BudgetRowComponent` needs to keep track of if the accordion is open or not, it has the state `showExpenseAccordion`, which does just that.

The only call `BudgetRowComponent` makes to the backend, via the API, is `deleteBudget`, which is a call coming from its `ExpenseAccordion` once the user clicks 'delete budget' in the accordion.

4.1.7 ExpenseAccordion

The `ExpenseAccordion` is meant to expand as a user clicks on a `BudgetRowComponent`. They are then presented with more details on each expense in that budget: specifically, the cost of each expense and its description. In the `ExpenseAccordion`, the user also has buttons to delete that entire budget, edit the expenses' cost and description, as well as close the accordion again. While editing the expenses the user can also delete them one by one with a button that looks like a bin.

On top of `budgetId` and `expenses`, it receives the function props `deleteBudget`, `loadExpenses`, `loadBudgets` as well as `handleClose` from `BudgetRowComponent`. As mentioned before, there are buttons for deleting the budget and closing the accordion, which is why both `deleteBudget` and `handleClose` are necessary. `LoadBudgets` is only passed down to `ExpenseModal`, which is only relevant when an `ExpenseAccordion` has no expenses, revealing the button 'add

expense'. Finally, `loadExpenses` is used after deleting an expense, when the user is finished editing expenses. It is also passed down to the `ExpenseModal` as mentioned earlier.

On top of the states `isEditing` and `editedExpenses`, used for keeping track of the functionality for editing expenses, there is also `showExpenseModal`, which holds a boolean for showing or hiding the `ExpenseModal` that can be opened through the `ExpenseAccordion`. When the user clicks the bin to delete an expense, `ExpenseAccordion` calls `deleteExpense` from the backend through the API. Similarly, `ExpenseAccordion` also calls `deleteBudget` from the API when a user clicks 'delete budget'.

4.1.8 LoginScreen

The login screen is where the user finds themselves when navigating to the root url of the web application. Since the application can only be used while logged in, the user is prompted to either log in, or register a new account by going to the register page.

States are used for the input fields to let the user see their inputs, therefore there are states for `username` and `password`. It's also important to give users feedback on what went wrong if they are unsuccessful in their login attempt. This is why there is also a state called error, holding the error message then displayed to the user when that happens.

The `LoginScreen` can navigate the user to other pages (`RegisterScreen` and `/budgets`, the main content of the application). It also makes calls to the backend through `apiLogin` with the method call '`login`'.

4.1.9 RegisterScreen

After clicking the 'register' button in `LoginScreen`, the user is now on the new page created by `RegisterScreen`. The functionality is similar to `LoginScreen`, and states for `username`, `password` and `error` are used. There is also a state with the function `setLoading`, used to aid the user in creating a valid `username` and `password` without reloading the page. Since there are multiple rules for creating a username and password, `RegisterScreen` makes use of this state to check the rules without reloading the page, while giving the user feedback on why their input is invalid.

The only call `RegisterScreen` makes to the API is through `registerUser`, when the user tries to register with a valid `username` and `password`, and it can also navigate the user back to `LoginScreen`.

4.1.10 DraggableExpense

`DraggableExpense` is the front-end representation of an expense that you are able to drag and drop from a budget row to another. It shows the cost of the expense and a description explaining the expense. Inside `DraggableExpense` we have functionality from the library `dnd-kit` to declare the component as a 'draggable', with styling when the component is dragged.

The properties of `DraggableExpense` include an `expense`, a state of `editedExpenses`, a state of the boolean `isEditing`, `handleChangeExpenses`, `handleSaveExpenses`, `handleDeleteExpenses`, which are passed down from `ExpenseAccordion`. The expense is the backend version of an expense, used to display the cost and description of the expense. When 'Edit Expenses' is clicked in `ExpenseAccordion` then we need to display input fields for the cost and description cells and update their values for the state `editedExpenses`. We also change the boolean value of `isEditing`.

Updating values when editing expenses is done using `thehandleChangeExpenses`, which updates the state of `editedExpenses`. To save the updates we call `handleSaveExpenses`, which takes the edited expenses stored in `ExpenseAccordion`, makes an API call with `updateExpense()` with the values from `editedExpenses` and finally calls `loadExpenses()`.

When the 'Edit Expenses' button is clicked in the `ExpenseAccordion`, the `DraggableExpense` should show a clickable trash can icon beside the description cell. Clicking the trash can icon will delete the `DraggableExpense` by calling `handleDeleteExpenses` that makes an API call with `deleteExpense()` using the `id` of the expense.

4.2 Accessibility

As we developed the web application, we tried to be mindful of W3C's Web Content Accessibility Guidelines (WCAG) 2.2, for example not adding unnecessary visual effects and complicated actions [1]. The one exception to this is the previously mentioned `DraggableExpense`, where users without the option to simulate a mouse dragging and dropping an element can't use it. Although this can technically still be achieved by deleting and then recreating that expense in the category where the user wants it, it's still not optimal since the web application doesn't have an easily accessible alternative. Adding such an alternative would improve the web application. However we still decided to keep this feature since it is a quality-of-life upgrade for those that can use it.

To make sure our web application was accessible, we used the IBM Equal Access Accessibility Checker add-on for Firefox. At first we found a lot of violations, especially related to elements' roles being incorrect or non-existent. Having roles on the frontend elements is important when the user is, for example, using a screen reader, therefore we added relevant roles. This is to allow these users to quickly find relevant elements such as a page's main information bulk, navbars, toolbars and so on. Thanks to the add-on we also found low contrasts with text fields on certain background colours and corrected this, something that can make the web application less accessible for visually impaired users.

As we added more features and styling, using this add-on continuously made creating an accessible design a lot easier. Instead of having to rehaul larger portions of the application, we could catch the accessibility violations early.

A bug we discovered was that the category drop-down menu in the expense modal panel is not keyboard-friendly, as the categories are not focused when the menu is opened. We managed to find a solution that worked to make it more accessible. However, this created more bugs and issues for our application that we considered even more detrimental and that

we could not solve in time.

4.3 Model

We decided to have three interfaces for our model: `BudgetRow`, `Expense` and `User`. `BudgetRow` represents an individual row in our budget table, with information about the category and the amount of money allocated. The row is identifiable by its id and another separate id associated to a user (as reflected in the database). To implement the expense tracking part of our application, we use `Expense` to hold fields about the cost of the expense and a description to describe the expense. It is identifiable by its id and another separate id associated with a budget.

The two interfaces are separated to follow the Single Responsibility Principle and are associated by 'assigning' expenses to budget rows by referencing a budget row's id. This allows us to have a separation of concerns, whilst still allowing for features that are connected to both the budget planner and the expense tracker. For example, when we calculate the difference between a budget row's amount and the sum of all expense costs

To allow for sessions and saving data on individual accounts, we create a `User` model to field an id, username and password. To save the data, our rows in the table, for each user we reference them by id in our `BudgetRow` interface.

4.4 Service

We chose to have a separate service for each model to follow the Interface Segregation Principle. Each service also implements its own separate interface to better follow the Dependency Inversion Principle, but we have chosen not to create more classes such as an `expense.memory.ts` service to implement the interfaces. Our main reason for doing this is that we have chosen to solve the issue with in-memory data for testing/development by using 'pg-mem', a library that in-memory emulates an in-memory instance of a Postgres database. The library provides useful methods, such as creating backups and restoring the database, for isolation between test runs and test cases. This means that we do not have to depend on an in-memory service class to provide methods that simulate database operations using an in-memory data structure like an array. However, it could be argued that an in-memory service class would be better for extensibility because we would avoid having to maintain the pg-mem library and adjust our own code base to the library's needs. This is why we have created the interfaces, so that we have the opportunity to implement an in-memory class should that be more suitable for scalability or extensibility. It also allows for the opportunity to implement more services that could depend on the methods provided in the interfaces.

An issue we had when designing our service layer was choosing between separating or merging the budget row and expense layer. Although we wanted to reduce coupling and avoid unnecessary dependencies between expenses and budget rows, we found that the expense service would still need to have some dependencies on the budget service. This is due to the nature of our design, where we have decided that an expense must be 'assigned' to a budget

row. Our concern was that when we would eventually extend our application and implement more features, then the dependencies between the two would grow larger. Ultimately, we decided to keep the two services separate, as we believe the 'expense tracking' part and the 'budget planning' part of our application should be kept separate. Even though there are some dependencies, we are satisfied with our decision as we consider the coupling to be manageable and not detrimental to the quality of our code.

4.5 API Specification

The base URL for API requests is:

```
const BASE_URL = "http://localhost:8080";
```

4.5.1 Creating a New User

Endpoint: `PUT/user`

Creates a new user.

Request Body

- `username`: The desired username for the user
- `password`: The desired password for new the user

Response Body

- `201 Created` - User registered successfully.
- `400 Bad Request` - Invalid `username` or `password` type, or username already exists
- `500 Internal Server Error` - Unexpected server error

4.5.2 Logging In a User

Endpoint: `POST /user/login`

Logs in an existing user and starts a session.

Request Body

- `username`: The username of the user
- `password`: The password for the user

Response

- `200 OK` - User logged in successfully
- `400 Bad Request` - User already logged in or invalid request body
- `401 Unauthorized` - Incorrect or non existing `username` or `password`
- `500 Internal Server Error` - Unexpected server error

4.5.3 Logging Out a User

Endpoint: `POST /user/logout`

Logs out the currently logged-in user.

Request Body

Logout relies on active session information.

Response

- `200 OK` - Logged out successfully
- `401 Unauthorized` - Not currently logged in

4.5.4 Update a Budget Row

Endpoint: `PUT /budget/:id`

Update an existing budget row by ID.

Request Params

- `id` - ID of the budget row to update

Request Body

- `category` - Updated budget row category
- `amount` - Updated budget row amount

Response Body

- `200 OK` - Returns updated budget row
- `400 Bad Request` - Invalid input or missing ID
- `401 Unauthorized` - Not logged in
- `404 Not Found` - Budget row not found
- `500 Internal Server Error` - Unexpected server error

4.5.5 Add a Budget Row

Endpoint: `POST /budget`

Adds a new budget row for logged-in user.

Request Body

- `category` - Budget row name
- `amount` - Amount for a budget row

Response Body

- `201 Created` - Returns the new budget row
- `400 Bad Request` - Invalid `category` or `amount`
- `401 Unauthorized` - User is not logged in
- `500 Internal Server Error` - Unexpected server error

4.5.6 Delete a Budget Row

Endpoint: `DELETE /budget/:id`

Deletes a budget row by ID for a logged-in user

Request Params

- `id` - ID of the budget row

Response Body

- `200 OK` - Budget row deleted
- `400 Bad Request` - Missing or invalid `id`
- `401 Unauthorized` - User not logged in
- `404 Not Found` - Budget row not found.
- `500 Internal Server Error` - Unexpected server error

4.5.7 Retrieve Budget Rows

Endpoint: `GET/budget`

Gets all budget rows for the logged-in user.

Request Body

- No body, depends on user's session

Response Body

- `200 OK` - Returns an array of budget rows
- `401 Unauthorized` - User not logged in or session is invalid
- `500 Internal Server Error` - Unexpected Server Error

4.5.8 Add New Expense

Adds a new expense for a user.

Endpoint: `POST /expense`

Request Body

- `category` - Budget row category for the expense to be placed

- `cost` - Cost of expense
- `description` - (Optional) Description of expense

Response Body

- `201 Created` - Returns the newly created expense
- `400 Bad Request` - Invalid input or missing fields
- `401 Unauthorized` - User not logged in
- `500 Internal Server Error` - Unexpected server error

4.5.9 Retrieve Expenses for a Budget Row

Fetch all expenses associated with a specific budget row.

Endpoint: `GET /expense/:budgetRowId`

Request Params

- `budgetRowId` - ID of the budget row

Response Body

- `200 OK` - An array of expenses for the given `budgetRowId`
- `400 Bad Request` - Invalid `budgetRowId`
- `401 Unauthorized` - User not logged in
- `500 Internal Server Error` - Unexpected Server Error

4.5.10 Delete Expense

Delete an expense by its ID.

Endpoint: `DELETE /expense/:id`

Request Params

- `id` - ID of the expense to delete

Response Body

- `200 OK` - Expense deleted successfully
- `400 Bad Request` - If `id` is missing or invalid
- `401 Unauthorized` - User not logged in
- `500 Internal Server Error` - Unexpected server error

4.5.11 Update Expense

Update an existing expense's values.

Endpoint: `PUT/ expense/:id`

Request Params

- `id` - ID of the expense to delete

Request Body

- `category` - Category of the expense's budget row
- `cost` - amount of expense
- `description` - (Optional) Description of expense

Response Body

- `200 OK` - Returns updated expense object
- `400 Bad Request` - If request body is invalid (missing `cost`)
- `401 Unauthorized` - User not logged in
- `404 Not Found` - Expense `id` does not exist
- `500 Internal Server Error` - Unexpected server error

4.6 ER Diagram

We were a bit unsure about the syntax we should use for the ER diagram as the syntax some members were familiar with (specifically with denoting cardinality of relationships) from a previous course doesn't seem to be used as much these days. As it is not an official UML diagram (and doesn't quite have one correct syntax), after reading up on it, we decided to use the syntax specified here: https://drawio-app.com/blog/entity-relationship-diagrams-with-draw-io/.

This dictates how we denote cardinality in relationships and the idea that a foreign key is written in italics. This syntax also implies that foreign keys are connected directly to the entities and the relationships tend to have more generic verbs (without any attributes in our case).

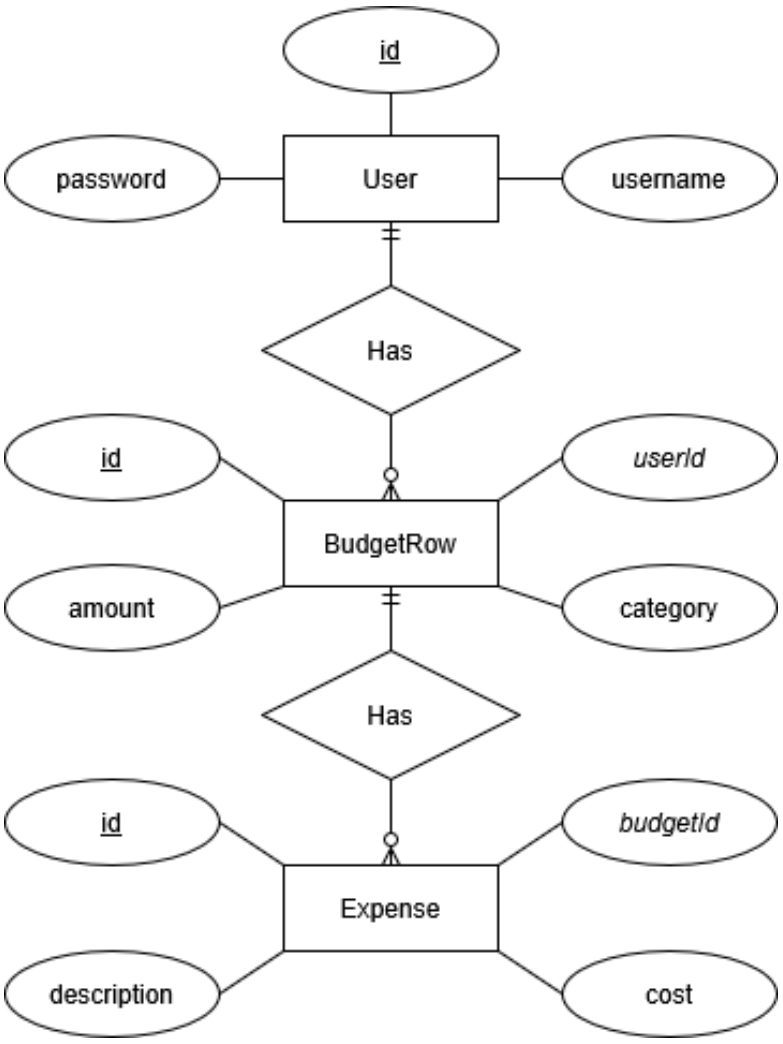

Figure 3: Entity-Relation (ER) Diagram

A User has an id, which is its primary key, as well as a username and password. The id is auto-generated by the system and both the username and password are not allowed to be empty. Furthermore, the username is unique to ensure that two different users don't have the same username.

Each User can have 0 to many BudgetRows, as denoted by the has-relationship in the ER diagram. A BudgetRow has an id, which is its primary key, as well as userId, which is a foreign key that references the user's id. A BudgetRow also has a category and an amount, neither of which are allowed to be null. Additionally, the combination of userId and category is unique. This ensures that the same user cannot create multiple budgetRows with the same category name but multiple users can still use the same category names for their BudgetRows.

Each BudgetRow can also have 0 to many Expenses (the same relationship as users have to BudgetRows). It has an id, which is its primary key, and a budgetId, which is a foreign key that references its parent BudgetRow. It also has a cost attribute, which cannot be null, and a description. Although a description is optional to the user when creating an expense, it is not allowed to be null in the database. Instead, if the user chooses to leave it blank, the value of 'description' in the database becomes the empty string. The reason we made this design decision is because we don't want to force the user to fill out a description, while also differentiating between a null value and an intentionally empty description.

Furthermore, the reason an expense doesn't have a userId attribute is that they are 'inherently' associated through their reference to a budgetRow. If an expense had a direct reference to a userId, this would allow expenses to exist independently from BudgetRows (which we don't want to allow) and can also open doors to creating duplicated data where both a BudgetRow and a user will have their own version of expenses.

4.7 Libraries

In addition to the frameworks provided in the assignments, we also used four libraries: Lodash, MUIX, DND-kit and pg-mem.

4.7.1 Lodash

Since we wanted to add a feature where the user could sort the budgets in BudgetTable, we decided to use a library to do this, since creating our own sorting algorithm in typescript would be time-consuming. We opted for the use of a library to save time as well as make the codebase more compact from a developer's point of view.

4.7.2 MUIX

In order to give the user an alternative overview of their expenses, we wanted to add a pie chart displaying the sum of each budget's expenses. This was done with the help of the Mui Library's x-charts. In Sidebar, we created this component which gave us a good-looking pie chart which could easily be styled with colours, extra fields and helpful animation.

4.7.3 dnd-kit

To add a feature where a user could move an expense from one budget row to another, we used dnd-kit as it is specifically designed for the React library. With a few adjustments to the existing components in BudgetRowComponent, DraggableExpense and BudgetTable, we implemented a simple drag-and-drop feature.

4.7.4 pg-mem

To avoid using the online database hosted on Aiven for development and testing, we opted to use a library which allowed for emulations of in-memory database instances. Pg-mem is a library specifically designed for PostgreSQL, which makes it appropriate to use. This allowed us to implement and test features using in-memory whilst still simulating real database operations. It also allows for a backup and restoration of the database, which simplified our testing process by making it easy to isolate test cases with a fresh instance of the database each run.

5 Responsibilities

Throughout most of the project, the group has worked very closely together. Pair programming as well as group discussions on problem solving and feature implementation have been the working standards since week one. Despite this, we have occasionally had our own areas of responsibility.

At the start of the project, Philip spent most of his time working on the Figma prototypes, as well as creating the first mockup of the sidebar component. Closer to the final versions of the codebase, he worked on implementation such as: switching to an actual database, sorting the BudgetTable, giving users input validation, and a large refactoring effort as we decided to separate budgets and expenses. He also worked on most of the accessibility checking and the project report under the title 'Frontend'.

Annelie worked on the earlier stages of user sessions, as well as the frontend for the login and register pages. At the later stages of the project, she did some frontend clean up, making the webpage more intuitive with icons and arrows, as well as placing new buttons for flow.

Liam worked mainly on the backend side of the application by fixing service and router validation, switching to a database and adding backend tests. He implemented certain use cases in the backend and frontend. On the project report, Liam was mainly responsible for writing the user manual and the design choices under 'Model' and 'Service.

Kevin implemented features of the application such as delete budget and edit budget, which included both front- and back-end parts of the codebase. Kevin was also responsible for writing front-end tests and towards the end of the project, refactoring and cleaning up the codebase. As for the report, Kevin was mainly responsible for the introduction and the ER diagram section.

6 Bibliography

[1] "Web content accessibility guidelines (WCAG) 2.2," WC3 Recommendations. in collab. with A. Campbell, C. Adams, R. Montgomery, Cooper, and A. Kirkpatrick. (2024), [Online]. Available: https://www.w3.org/TR/WCAG22/ (visited on 03/13/2025).