Design Document

for

Project: ManageMe (Group 2)

An All-in-One Self-Management System

Prepared by
Ankur Sharma (2015CS50278)
Lovish Madaan (2015CS50286)
Sudeep Agrawal (2015CS50295)
Siddharth Khera (2015MT60567)

Supervised by
Prof. S.C. Gupta

COL 740 - Software Engineering
Indian Institute of Technology, Delhi
Hauz Khas, New Delhi

Contents

Chapter 1

Introduction

1.1 Purpose

The purpose of this document is to enlist the design specifications for the Self-Management application (ManageMe) in detail. This document serves as a complete guide on the design of the system, listing in detail all the modules and their dependencies with each other. This document also describes components like the server architecture and database system.

1.2 Scope

Scope of this project is building a complete standalone android application which will have different functionalities, all specific to different self-management tasks. The main features for self-management include:

- **Note Taking**: Help users take notes of important things with functionalities like better organization, hide features for notes and filtering according to the tags assigned to individual notes.
- **Expense Tracker**: Help users manage their expenses with the facility to organize expenses according to different categories.
- **Calorie Tracker**: Help users manage their daily food intake and calories with the option to add items already present in the database along with the functionality to add new items and calories manually.

We believe that these three use-cases are highly interdependent on one another and thus helps to have only one application managing the three tasks.

1.3 Overview

This report contains all the information needed to understand the design specifications of ManageMe application. We organize this document as follows:

- Section 2 contains the architectural design specifics with relevant figures for better understanding. It also contains the technologies used.
- Section 3 consists of the detailed design with API function details for each module along with the database design and ER diagram for our system.
- Section 4 highlights the deployment design and the 3-tiered strategy followed by us.

1.4 References

- ManageMe SRS Document.
- Design Format Structure for this document from moodle.

1.5 Definitions, Acronyms and Abbreviations

1.5.1 Definitions

- **ManageMe**: The self-management application detailed in this document.
- **Customer/User**: Here the customer and user are both the same as we are developing this application for the end user directly. Users will be those people who want to personalise and manage their own experience in their own way.
- **Supplier**: We will be the sole suppliers of the application. We will build the app and publish it to the web where users will be able to download and use it.
- **Item**: An item is a data entry which could be of any category i.e. an item could be a note, an expense entry, food entry etc.
- **App Interface**: This refers to the frontend application that the user will be interacting with and broadly acts as a mediator between the user and the backend server.

1.5.2 Abbreviations

- **DB** - Database.

Chapter 2

System Description

2.1 Architectural Design

This section details the architectural design involving the user, app interface, and the backend server. Refer to Figure 2.1 for the design.

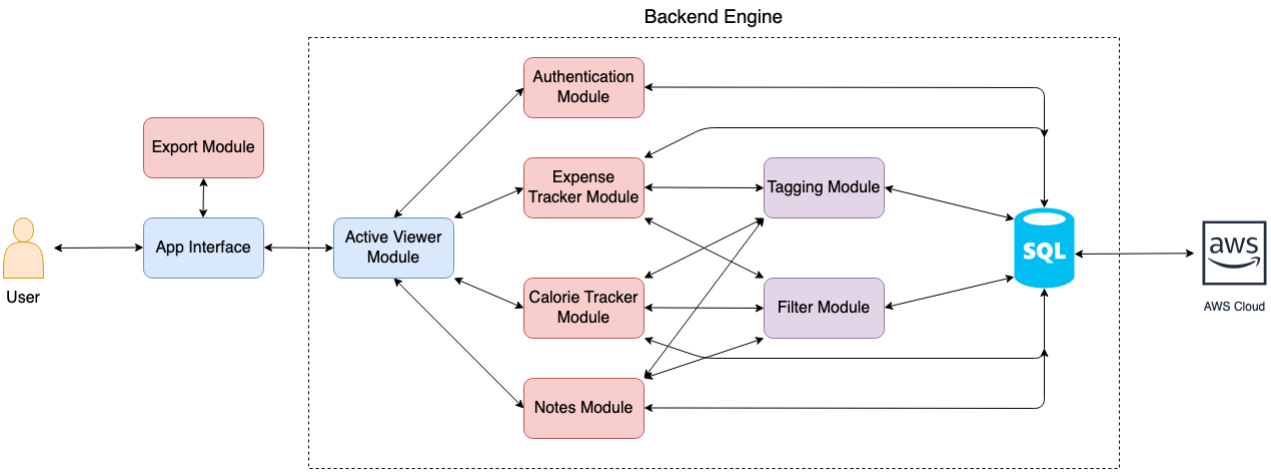


Figure 2.1: Architectural Design of ManageMe

2.2 Module Definitions

In this section, we explain the various components of our architectural design in detail.

2.2.1 User

As discussed in Section 1.5.1, user refers to the end-customer using our application.

2.2.2 App Interface

App Interface refers to the front-end design and the various functionalities that the user can select to navigate the application.

2.2.3 Active Viewer Module

This module acts as the mediator between the App Interface and the Backend Engine. It is responsible for handling and processing all the queries selected by the user and displays the active window and the query results that the user has requested.

2.2.4 Authentication Module

This module is responsible for the Sign-In and Sign-up functionalities of our application. Once a user signs up, his/her encrypted information will be saved on both the local device Database (DB) and synced with the cloud server whenever internet connectivity is available. When a user signs in to the application, his/her previous records are fetched from the DB and displayed on the screen.

2.2.5 Expense Management Module

This module is responsible for all expense-related tasks in the application. It provides functionality for adding, deleting, or modifying expense records. It also provides features for tagging the expenses and filter the expenses according to tags/date range. To do that, it sends the request to either the Tagging Module or the Filter Module which then send back the relevant entries.

2.2.6 Calories Management Module

This module is responsible for tracking a user's daily food intake. This module is also responsible for displaying the food items along with their calories information already present in the database. The user then just has to select the quantity of the food item and his/her food calories information will be displayed automatically. For food items not present in the database, the user can add items and their calorie information manually. The user can give tags to the food items like breakfast/lunch/dinner etc. and filter the entries according to date range.

2.2.7 Notes Management Module

This module is responsible for making notes about important information relevant to the user. The user can hide individual notes, give different colors and can also star them. The starred notes are displayed on the top of the list. Here also the user can give different tags to the notes and filter the notes according to keywords provided.

2.2.8 Tagging Module

This module is responsible for all tag related functionality in the application for the three management modules - expenses, calories, and notes. It organizes the database according to the tags present in the system to make tag-based queries faster.

2.2.9 Filtering Module

This module handles the filter queries from the three management modules. It provides an option to filter the items according to the date range. It is responsible for adding timestamp information in the database so that filter queries can be performed efficiently. For the Notes Management module, it provides the additional functionality of filtering according to keywords provided by the user in the search box.

2.2.10 Export Module

This module handles requests by the user to export any kind of information present in the application (notes, expenses, or calories) in a CSV/JSON format. This helps the user to make hard copies of the information for further analysis and better planning.

2.3 Technology and Tools used

The following tools and technologies have been used in building our project.

2.3.1 Front End Interface

- **XML:** Used for developing layouts in Android application
- **Java:** Manually creating some complex layouts in the app. *Fragment Views* have been used as a building block of our application theme.
- **Kotlin:** Designing activity layouts of some components.
- **Material Design:** Implemented new components like *Floating Action Button* while ensuring backward compatibility.

2.3.2 Backend Server and Database

- **CSV:** Format used for exporting the user's data so as to create a local copy.
- **JSON:** Database format used for storing the notes.
- **SQLite:** Relation database technology used to set up database, which is synchronization with the Amazon Dynamo-DB on the AWS S3.
- **Java:** For writing the backend code.
- **Kotlin:** For coding some specific components in our backend.
- **AWS:** Amazon Web Services used for the deployment of the server. We use Amazon Dynamo-DB with the sync framework to synchronize our SQLite records on the S3 buckets.

2.3.3 Development Tools

- **Android Studio:** For creating the overall mobile application.
- **VS-Code:** Smart code editor used while writing the backend.
- **Git:** Version control system used for sharing and collaboration amongst different members of the group.

Chapter 3

Detailed Design

3.1 Module APIs

This section will provide a brief description of all the APIs used by each module of the system.

3.1.1 Sign-In Module

API Function	Description
registerUser(name, username, email)	Registers a user in the database with initialisation
loginUser(name, password)	Allows the user to log-in into the self-management portal if the password is correct
computeHash(loginPassword)	Computes the hash function for the user entered password
fetchHash(userId)	Fetch the hash value of the actual password of the user with this userId if present
verifyCredentials(givenHash, actualHash)	Returns True if the hash values match otherwise False
fetchDetails(userId)	Fetch the user details from the database upon successful authentication

3.1.2 Expense Management Module

API Function	Description
selectCurrency()	Select currency type from a range of currency options.
addCategory(category)	Add category to the pre defined set of catgories.
addExpense(amount, category, timestamp)	Add expense entry.
modifyExpense(expenseId, expenseEntry)	Select any current entry and modify it's details.
deleteExpense(expenseId)	Delete any current expense entry.
filterExpenses(dateRange)	Select a date range or predefined ranges like day/week/month/year to filter the expense records.

3.1.3 Calories Management Module

API Function	Description
addFoodItem(foodItem, foodCalories)	Add any food and it's calorie information to the existing food database.
addItem(foodItem, quantity)	Add food intake entry.
modifyItem(foodItemId, foodEntry)	Select any current entry and modify it's details.
deleteItem(foodItemId, foodEntry)	Delete any current food entry.
filterItems(dateRange)	Select a date range or predefined ranges like day/week/month/year to filter the calorie intake records.

3.1.4 Notes Management Module

API Function	Description
newNote()	Open the activity to create a new note (initially empty)
setFonts(noteId, fontProperties)	Set the font style and font size of the note
setColorTheme(noteId, colorProperties)	Set the color theme of the notes for better categorization
saveNote(noteId, title, bodyText)	Save changes to the note currently added along with its title
hideNoteBody(noteId)	Hides the notes body in the List View of notes for privacy reasons
searchNotes(keyText)	Searches the keyText across across all the notes (both titles and body) and displays the results matched
starNote(noteId)	Stars the note for you and sorts the list view so as to show the starred notes on the top of the list
updateNote(noteId)	Opens an already existing note for updation
deleteNote(noteId)	Deletes the selected note from the database

3.1.5 Filtering Module

API Function	Description
filterExpenses([expenseId]List, dateRange)	Filter expenses according to the provided date range or select from pre-existing options like day/week/month/year.
filterFoodItems([foodItemId]List, dateRange)	Filter calorie records according to the provided date range or select from pre-existing options like day/week/month/year.
filterNotes([NoteId]List, keyword)	Filter notes according to the keyword provided by the user with only those notes displayed that contain the corresponding keyword.

3.1.6 Tagging Module

API Function	Description
createTagsExpenses(tagText)	Creates a custom tag with the given tagText for Expenses
tagExpenseIntake(expenseId, [tagId]List)	Tags a given expense entry with the specified list of tags
showTagwiseExpenses(tag)	Shows all the expenses corresponding to a given text
deleteTagsExpenses(tagId)	Deletes a tag from the tag list of expenses
createTagsCalories(tagText)	Creates a custom tag with the given tagText for Calories
tagCalorieIntake(foodItemId, [tagId]List)	Tags a given food entry with the specified list of tags
showTagwiseCalories(tag)	Shows all the food items corresponding to a given text
deleteTagsCalories(tagId)	Deletes a tag from the tag list of food items
createTagsNotes(tagText)	Creates a custom tag with the given tagText for Notes
tagNotes(noteId, [tagId]List)	Tags a given note entry with the specified list of tags
showTagwiseNotes(tag)	Shows all the notes corresponding to a given text
deleteTagsNotes(tagId)	Deletes a tag from the tag list of notes

3.1.7 Export Module

API Function	Description
exportExpenses([ExpenseId]List)	Export all expense reports in CSV/JSON format to local device.
exportFoodLogs([FoodItemId]List)	Export all food intake and calorie records in CSV/JSON format to local device.
exportNotes([NoteId]List)	Export all notes in CSV/JSON format to local device.

3.2 Database Design

This application will have 4 different databases.

- **User Database:** It is an SQLite database that will contain the information of all the registered users of this application.
- **Expense Database:** It is also an SQLite database for maintain the expense records with timestamps of all the users along with their custom categories.
- **Calorie Database:** It is also an SQLite database for maintain the food consumption records with timestamps of all the users along with their custom categories.
- **Notes Database:** It is a JSON database for maintaining the user personalised custom notes along with their meta information.

These are explained in detail in the following sub sections.

3.2.1 User Database

Field	Type	Description
UserID	Integer	Unique ID for the user
LastLogin	DateTime	Timestamp of latest login
UserName	String	User defined unique name for login purposes
Name	String	Name of the user
UserEmail	String	Email ID of the User
PasswordHash	String	Computed hash code of the user's password

3.2.2 Expense Database

Field	Type	Description
ExpenseID	Integer	Unique ID for expense entry.
amount	Integer	Amount value for a particular expense entry.
category	String	Category of expense entry
LastUpdated	DateTime	Date and Time when expense entry was last updated.

3.2.3 Calorie Database

Field	Type	Description
FoodItemID	Integer	Unique ID for food entry.
name	String	Name of the item in food entry.
quantity	Integer	Quantity for a particular food entry.
calories	Integer	Calories corresponding to one portion of food item in the entry.
LastUpdated	DateTime	Date and Time when food entry was last updated.

3.2.4 Notes Database

Field	Type	Description
NoteId	Integer	Unique Note ID for each note
Title	String	Title of the note
Body	String	Text body of the note
Colour	String	Color theme of the note for categorization
Tags	String List	List of tags marked by the user for the node
Favoured	Boolean	Flag for indicating whether note is marked as favorite by user
FontSize	14/18/22	Font size of the note text
HideBody	Boolean	Flag used for hiding the body of the note

JSON database updates are fast, light and easier to synchronise with the remote database. Hence, a JSON database is chosen for notes because the text updates/edits are made to the notes much more often than expenses and fitness records.

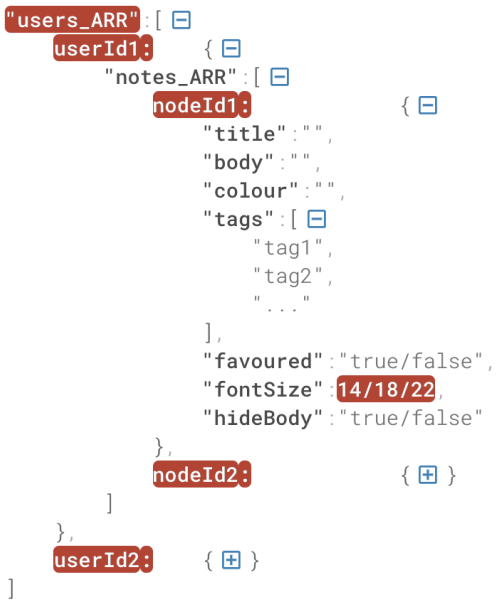


Figure 3.1: Notes JSON Database

3.2.5 ER Diagram

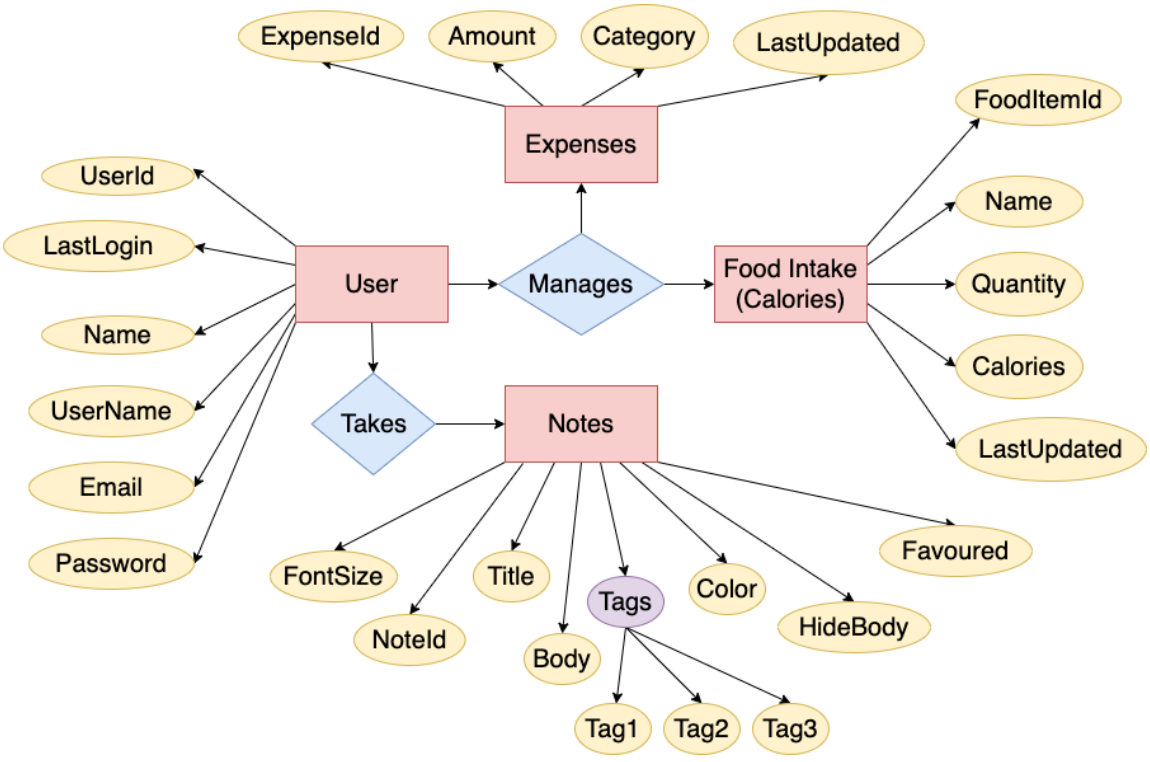


Figure 3.2: ER Diagram

3.3 Screen Layouts

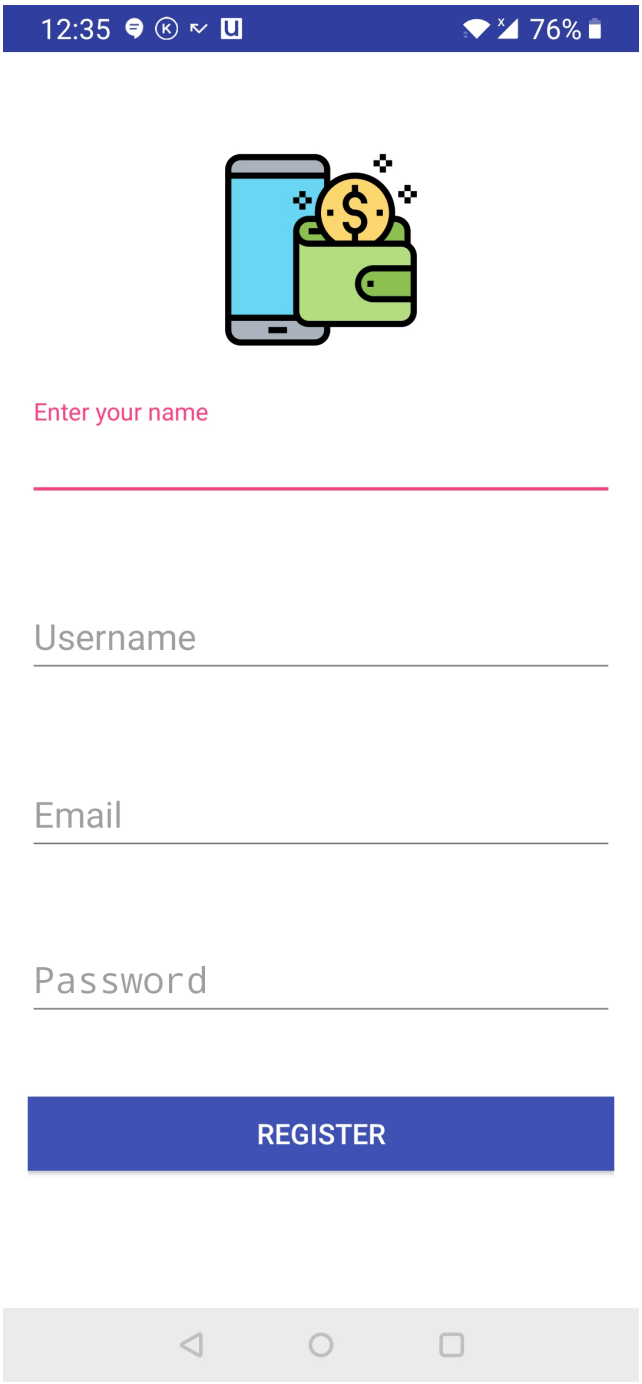


Figure 3.3: Sign-Up Page

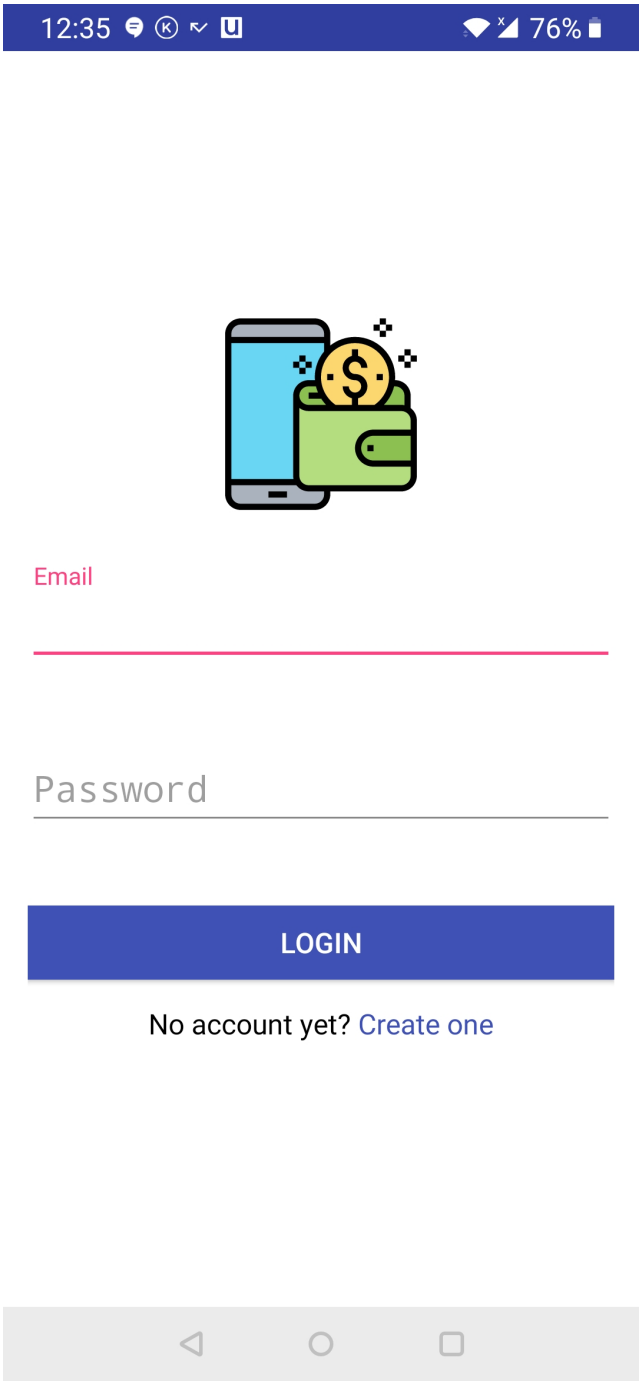
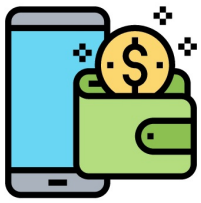


Figure 3.4: Login Page



Welcome User!

MANAGE YOUR EXPENSES

CHECK YOUR CALORIES INTAKE

TAKE NOTES

EXPORT MY DATA

LOGOUT

Figure 3.5: Home Page (Landing Page)

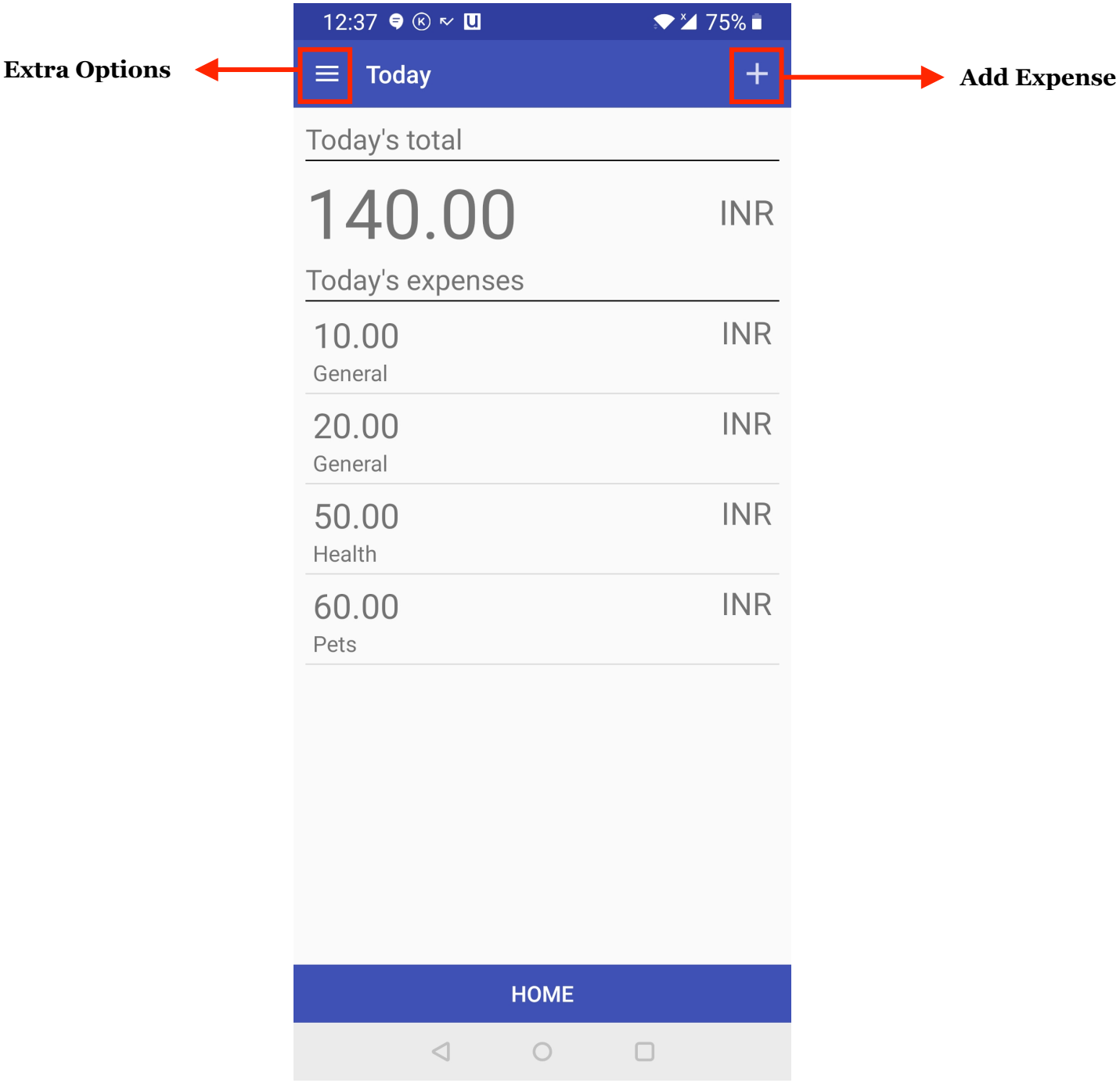


Figure 3.6: Expense Management page

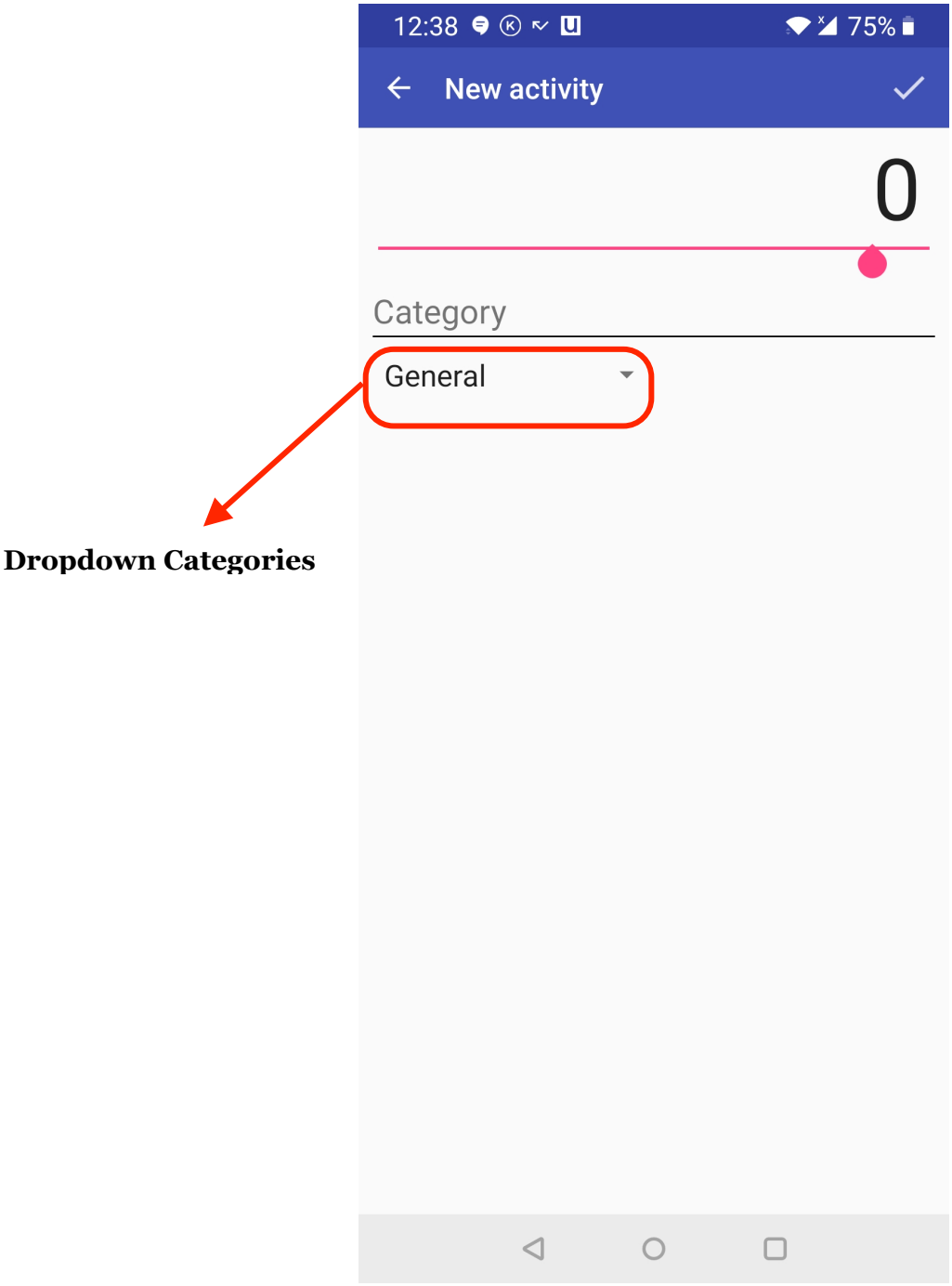


Figure 3.7: Add Expense page

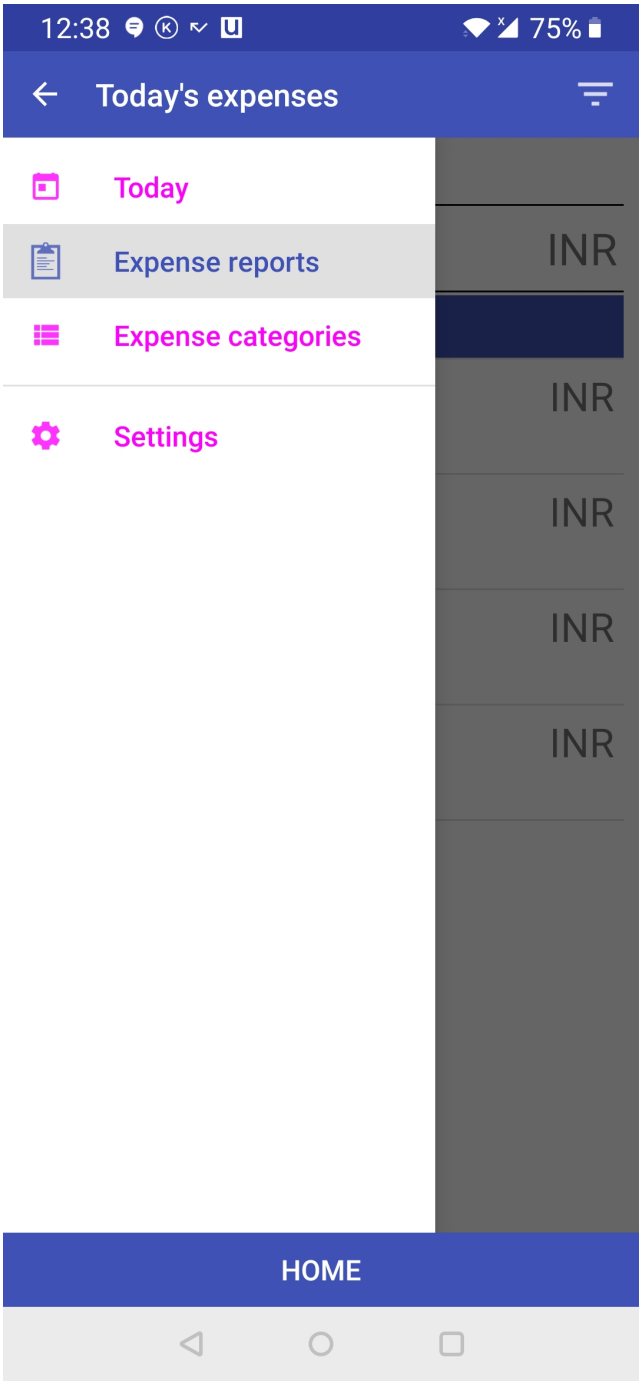


Figure 3.8: Additional options on Expenses Management page

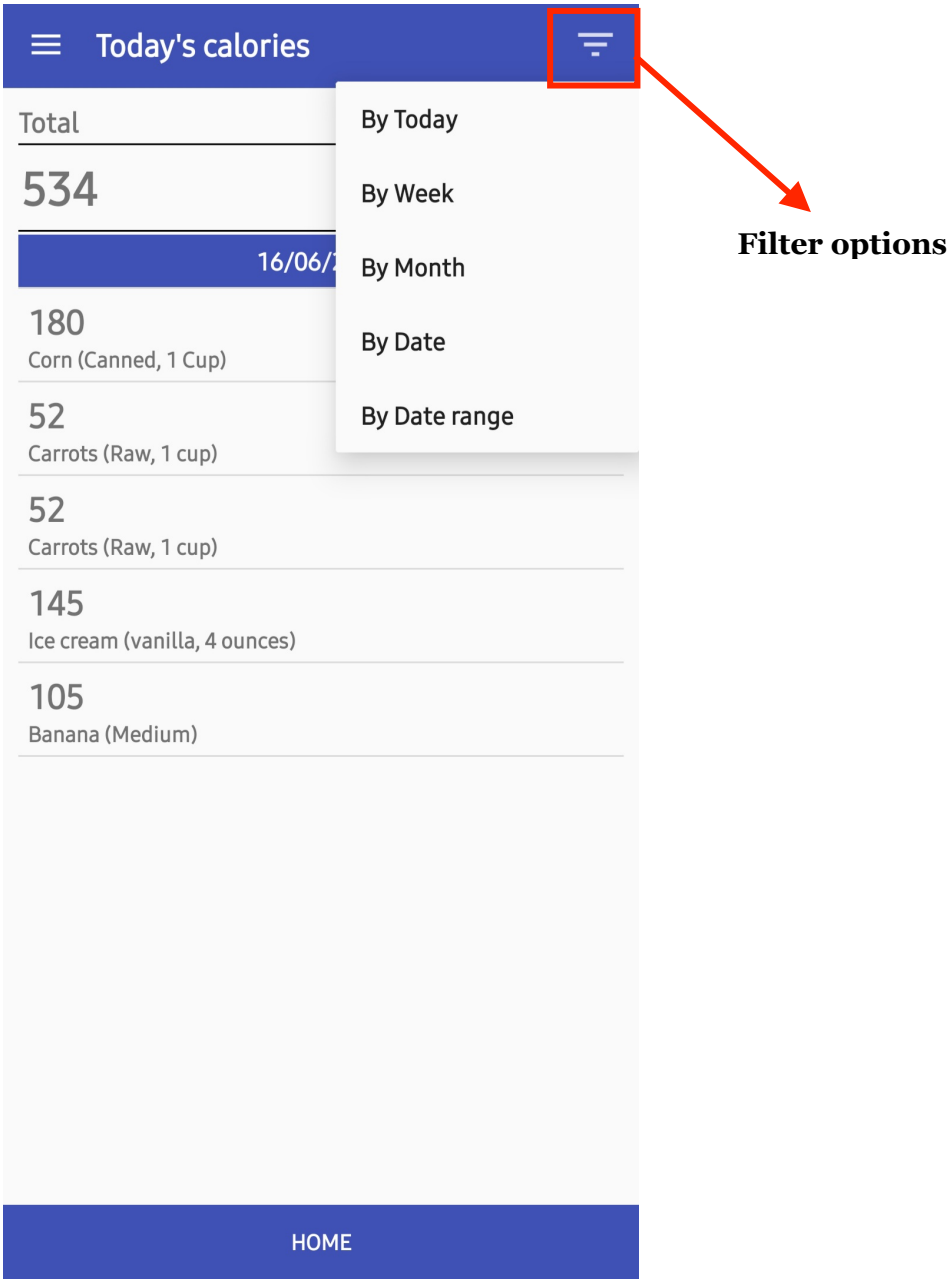


Figure 3.9: Calories Management page with filter option displayed

Figure 3.10: Add food entry page

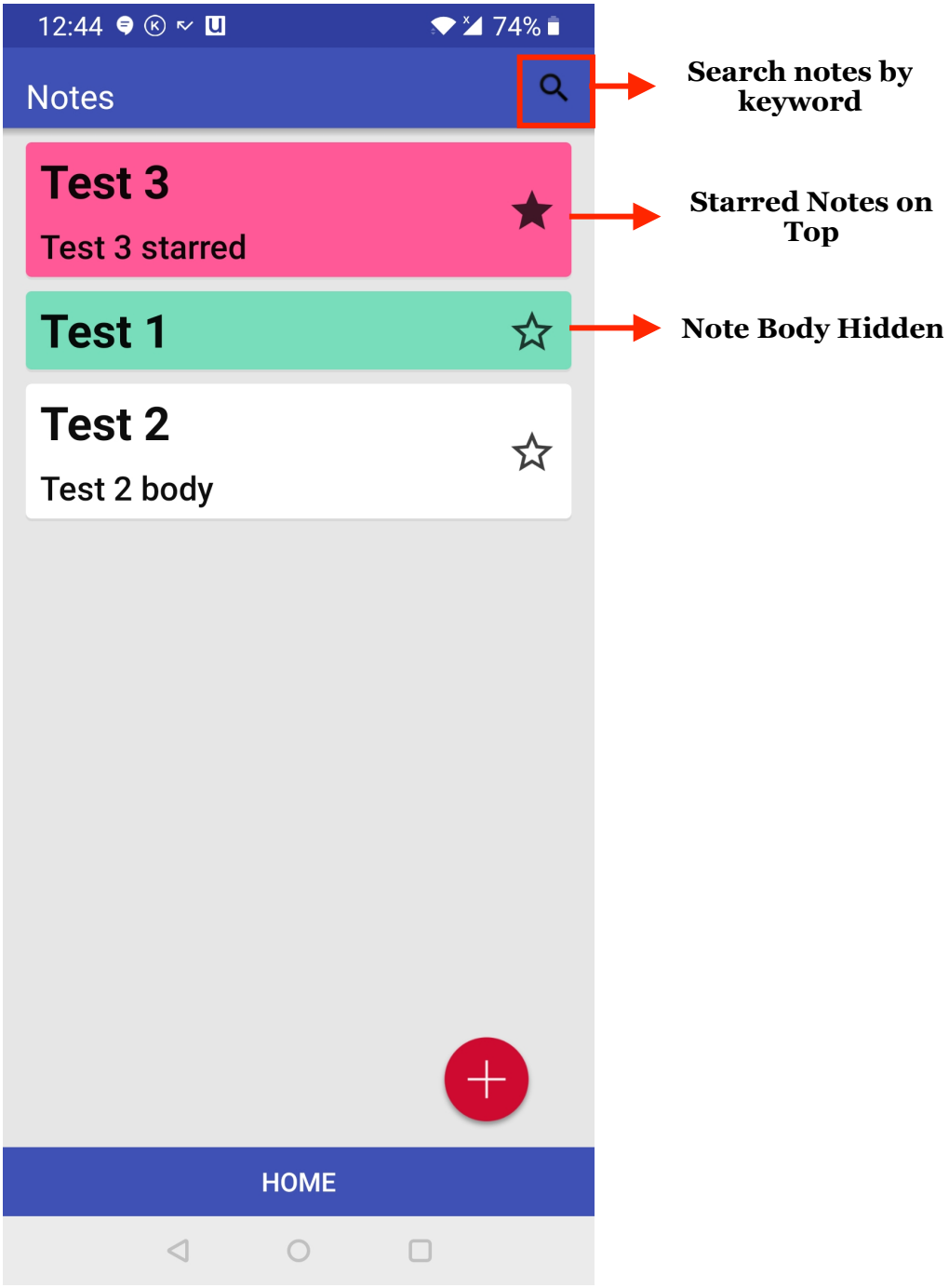


Figure 3.11: Notes Management page

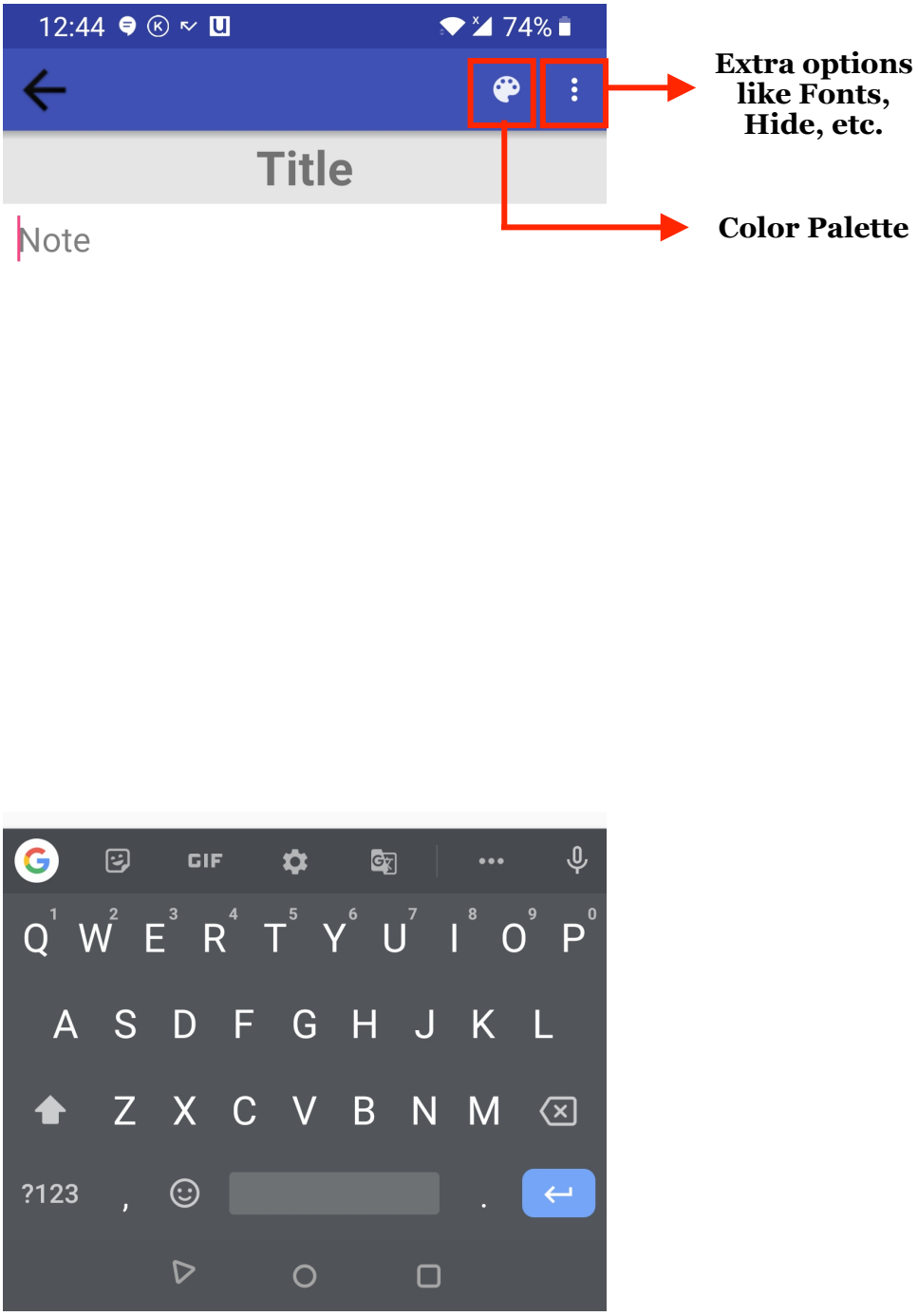


Figure 3.12: Add Note page

3.4 Use Cases

3.4.1 Use Case Scenario-1: Sign-in

Type	Description
Purpose	User signs in to the application and his records are fetched.
Input Data	Email id and password used to register for the application.
Output Data	Taken to the app home page and user-specific information fetched.
Pre-conditions	User is at the sign-in page, user has already registered using a valid email-id, and user inputs the correct password.
Post-conditions	Application home page displayed with username, and individual modules contain user data fetched from the database.
Basic Flow	Once the user inputs his email id and password, the sign-in API authenticates the user and fetches the previously saved user information and records from the database.
Alternative Flow	Invalid username/password, user not signed up, database is corrupted.
Business Rules	This will allow the user to go the application home page and his/her records - expenses, calorie intake and notes are fetched and displayed in respective tabs.

High Level Code (using module APIs)

Client Side:

Listing 3.1: Sign-in Code at Client Side

```
// User is already registered, enters his/her details and then presses Sign-In Button

signIn() {
    username, password <= getInputFromUser()
    hashedPassword <= computeHash(password)
    status <= loginUser(username, hashedPassword)
    if(status == Success) {
        userDetails <= fetchDetails(userId)
        beginLogin(userDetails)
    } else {
        handleErrorStatus(status)
    }
}
```

Server Side:

Listing 3.2: Sign-in Code at Server Side

```
// Receives the hash of the user entered password and the username

loginUser() {
    username, userHash <= getInputFromClient()
    correctHash <= fetchHash(username)
    status <= verifyCredentials(userHash, correctHash)
    sendStatus(status)
}
```

Sequence Diagram

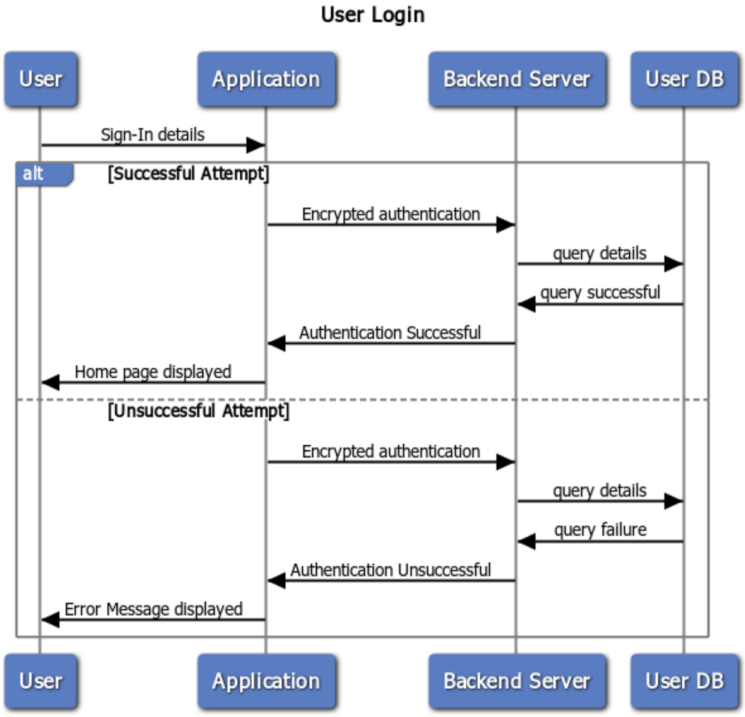


Figure 3.13: Sequence Diagram for Login use-case

3.4.2 Use Case Scenario-2: Expense Management

Add Expense Entry

Type	Description
Purpose	User adds an expense entry on the expenses homepage.
Input Data	Expense category, currency, amount.
Output Data	Added entry displayed in the list of expenses.
Pre-conditions	User is signed-in to the application, user is on the expenses page, currency category is selected from expense page settings.
Post-conditions	Expenses home page updated with the added entry displaying date, category, and expense amount, and total expense is also updated correctly.
Basic Flow	Once the user inputs the required fields for adding an expense and clicks ok, the entry is processed and added to the database, and then the database sends OK signal to the front-end to display the entry on the expenses list.
Alternative Flow	Invalid amount/category, entry corrupted while processing, entry not added to the database and not displayed on the page.
Business Rules	This will allow the user to add expenses and manage his/her expenses accordingly.

High Level Code (using module APIs)

Listing 3.3: Add Expenses

```
// User has logged-in successfully and has opened the Expenses Mangagement section of the app

addExpense() {
    // User selects the currency of his choice in Settings
    currency <= selectCurrency()

    // User selects the category of his expenses
    chosenCategory <= chooseCategory()

    // User enters the expense, adds a record in ExpensesDB and returns
    status <= addExpense(amount, category)

    return status
}
```

Edit Expense Entry

Type	Description
Purpose	User edits an expense entry on the expenses homepage.
Input Data	ExpenseId, new category, new amount.
Output Data	Edited entry displayed with the changes in the list of expenses.
Pre-conditions	User is signed-in to the application, user is on the expenses page, there is atleast one expense entry.
Post-conditions	Expenses home page updated with the edited entry displaying date, category, and expense amount, and total expense is also updated correctly.
Basic Flow	Once the user selects an expense entry and inputs the required fields for changing the entry and clicks ok, the entry is processed and updated in the database, and then the database sends OK signal to the front-end to display the changes corresponding to the entry on the expenses list.
Alternative Flow	Invalid amount/category, entry corrupted while processing, entry not updated in the database and no change on the front-end.
Business Rules	This will allow the user to edit expenses and manage his/her expenses accordingly.

High Level Code (using module APIs)

Listing 3.4: Edit Expenses

```
// User has logged-in successfully and has opened the Expenses Mangagement section of the app

editExpense() {
    // User taps on the expense to update
    hasClicked, expenseId <= detectClickGesture()
    status <= False
    if(hasClicked == Successs) {
        openUpdateActivity(expenseId)

        // User enters the new expense
        newExpenseEntry <= getExpenseInput()

        // expense entry of expenseId has been modified
        status <= modifyExpense(expenseId, newExpenseEntry)
    }
    return status
}
```

Delete Expense Entry

Type	Description
Purpose	User deletes an expense entry on the expenses homepage.
Input Data	ExpenseId.
Output Data	Deleted entry removed from the list of expenses, and total expenses updated accordingly.
Pre-conditions	User is signed-in to the application, user is on the expenses page, there is atleast one entry on the expenses page.
Post-conditions	Expenses home page updated with the deleted entry removed, and total expense updated correctly.
Basic Flow	Once the user selects an expense entry to delete and clicks ok, the entry is processed and deleted from the database, and then the database sends OK signal to the front-end to remove the entry from the expenses list.
Alternative Flow	Expense entry is corrupted in the database.
Business Rules	This will allow the user to delete expenses and manage his/her expenses accordingly.

High Level Code (using module APIs)

Listing 3.5: Delete Expenses

```
// User has logged-in successfully and has opened the Expenses Mangagement section of the app

deleteExpense() {
    // User presses and holds on the expense to delete
    hasClicked, expenseId <= detectLongClickGesture()
    status <= False
    if(hasClicked == Success) {
        showDeleteOption()

        //User taps on the delete button to confirm
        selection <= checkDeleteGesture()
        if(selection == Success) {
            status <= deleteExpense(expenseId)
        }
    }
    return status
}
```

Sequence Diagram

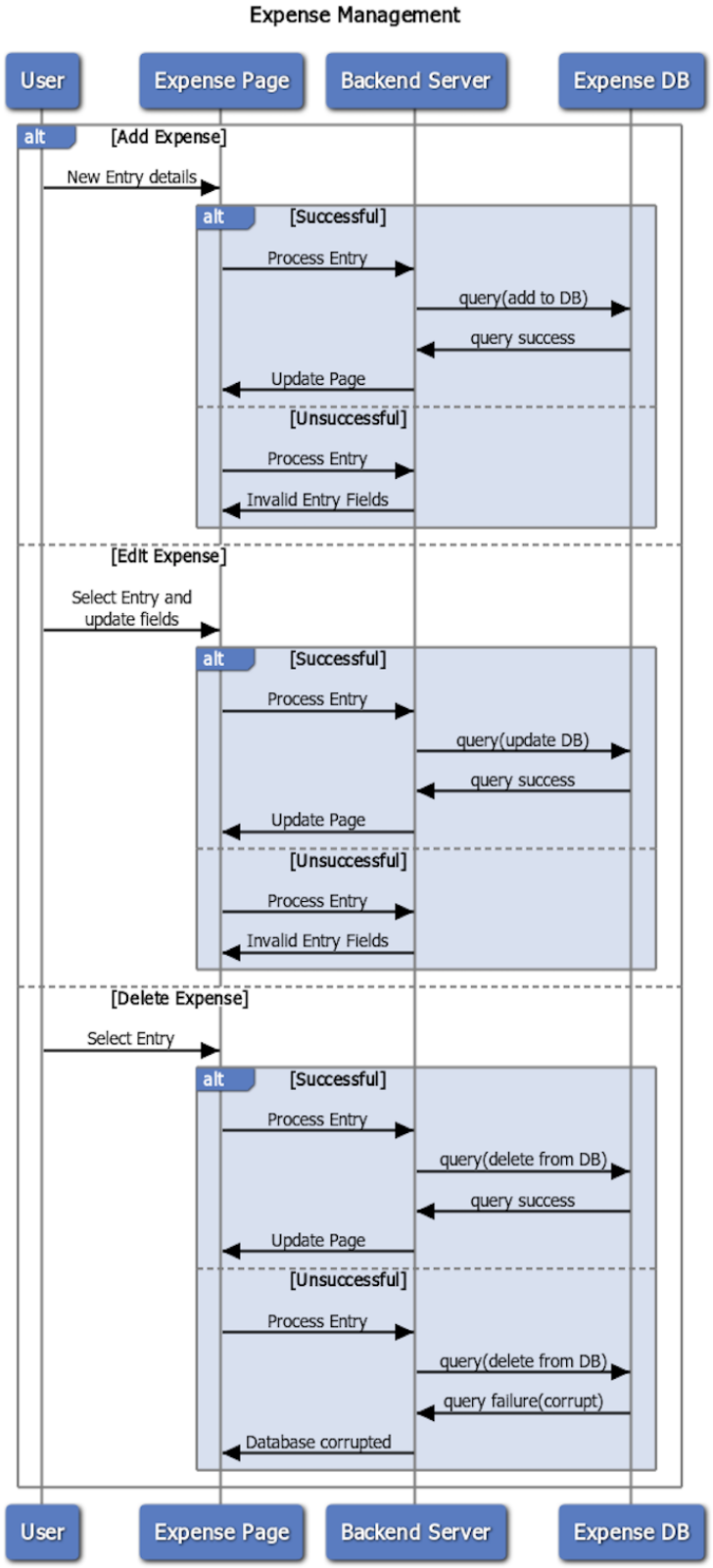


Figure 3.14: Expense Management Sequence Diagram

3.4.3 Use Case Scenario-3: Calories Management

For the sake of brevity of this document, we have not shown all the pre-conditions/post-conditions corresponding to this use-case since they were along the similar lines to those of Expenses Management (defined in the previous use-case 3.4.2). However, we will state the High Level Code using our Module APIs for more clarity.

High Level Code (using Module APIs)

Listing 3.6: Add/Edit/Delete Food Item

```
// User has logged-in successfully and has opened the Calories Mangagement section of the app

// ADD FOOD ITEM ENTRY
addCalories() {
    // User selects the category of his food item that he consumed
    chosenFoodItem <= chooseFoodCategory()

    // User selects the quantity of the food item (how many did he consume?)
    numFoodItems <= selectCount()

    // User enters the food items along with its quantity, adds a record in CaloriesDB and returns
    status <= addItem(chosenFoodItem, numFoodItems)

    return status
}

// EDIT FOOD ITEM ENTRY
editCalories() {
    // User taps on the food item to update
    hasClicked, foodItemId <= detectClickGesture()
    status <= False
    if(hasClicked == Successs) {
        openUpdateActivity(foodItemId)

        // User chooses the new food category
        newFoodEntry <= chooseFoodCategory()

        // food entry of foodItemId has been modified
        status <= modifyItem(foodItemId, newExpenseEntry)
    }
    return status
}

// DELETE FOOD ITEM ENTRY
deleteCalories() {
    // User presses and holds on the food item to delete
    hasClicked, foodItemId <= detectLongClickGesture()
    status <= False
    if(hasClicked == Success) {
        showDeleteOption()

        //User taps on the delete button to confirm
        selection <= checkDeleteGesture()
        if(selection == Success) {
            status <= deleteItem(foodItemId)
        }
    }
    return status
}
```

Sequence Diagram

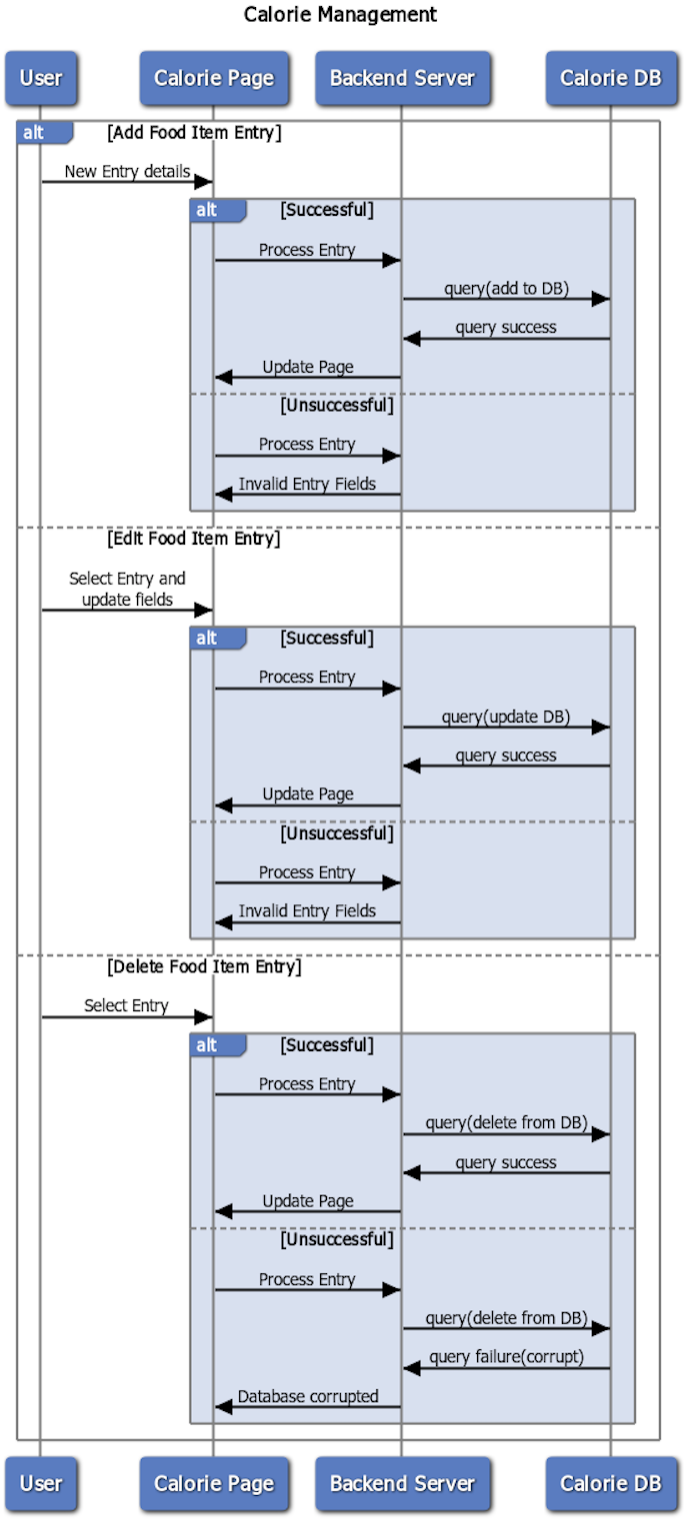


Figure 3.15: Calorie Management Sequence Diagram

3.4.4 Use Case Scenario-4: Notes Management

Add Note Entry

Type	Description
Purpose	User adds a note on the notes homepage.
Input Data	Note Title, Note Body, Note color, displayBody.
Output Data	Added note entry displayed in the list of notes.
Pre-conditions	User is signed-in to the application, user is on the notes page.
Post-conditions	Notes home page updated with the added entry displaying title, body (if displayBody == True), and the option to star the note.
Basic Flow	Once the user inputs the required fields for adding the note and clicks ok, the entry is processed and added to the database, and then the database sends OK signal to the front-end to display the entry on the notes list.
Alternative Flow	No Title entered, entry corrupted while processing, entry not added to the database and not displayed on the page.
Business Rules	This will allow the user to add notes and manage his/her notes page accordingly.

High Level Code (Using Module APIs)

Listing 3.7: Add Note

```
// User has logged-in successfully and has opened the Notes Mangagement section of the app

addNotes() {
    // User clicks on the new note (+) button on the bottom right of the screen
    noteId <= newNote()

    // User selects the font style and size of the note
    fontSize, fontStyle <= selectFontProperties()
    setFonts(noteId, (fontSize, fontStyle))

    // User selects the color theme from the color palette
    colorProps <= selectColorTheme()
    setColorTheme(noteId, colorProps)

    // If user wants to hide the text body of his/her note
    hideNoteBody(noteId)

    // User enters the title, text body of the note and clicks on 'Save Changes'
    status <= saveNote(noteId, title, textbody)

    return status
}
```

Edit Note Entry

Type	Description
Purpose	User edits a note entry on the notes homepage.
Input Data	NoteId, new title, new body, new color, displayBody.
Output Data	Edited entry displayed with the changes in the list of notes.
Pre-conditions	User is signed-in to the application, user is on the notes page, there is atleast one note entry.
Post-conditions	Notes home page updated with the edited entry displaying the changes correctly and star option retained.
Basic Flow	Once the user selects a note entry and inputs the required fields for changing the entry and clicks ok, the entry is processed and updated in the database, and then the database sends OK signal to the front-end to display the changes corresponding to the entry on the notes list.
Alternative Flow	Empty title, entry corrupted while processing, entry not updated in the database and no change on the front-end.
Business Rules	This will allow the user to edit notes and manage his/her notes page accordingly.

High Level Code (Using Module APIs)

Listing 3.8: Edit Note

```
// User has logged-in successfully and has opened the Notes Mangagement section of the app

editNotes() {
    // User taps on the note item to update
    hasClicked, noteId <= detectClickGesture()
    status <= False
    if(hasClicked == Successs) {
        updateNote(noteId)

        // User chooses the new food category
        newFoodEntry <= chooseFoodCategory()

        // User selects the font style and size of the note
        fontSize, fontStyle <= selectFontProperties()
        setFonts(noteId, (fontSize, fontStyle))

        // User selects the color theme from the color palette
        colorProps <= selectColorTheme()
        setColorTheme(noteId, colorProps)

        // If user wants to hide the text body of his/her note
        hideNoteBody(noteId)

        // User edits the title, text body of the note and clicks on 'Save Changes'
        status <= saveNote(noteId, title, textbody)
    }
    return status
}
```

Delete Note Entry

Type	Description
Purpose	User deletes a note entry on the notes homepage.
Input Data	NoteId.
Output Data	Deleted entry removed from the list of notes.
Pre-conditions	User is signed-in to the application, user is on the notes page, there is atleast one entry on the notes page.
Post-conditions	Notes home page updated with the deleted entry no longer visible.
Basic Flow	Once the user selects a note entry to delete and clicks ok, the entry is processed and deleted from the database, and then the database sends OK signal to the front-end to remove the entry from the notes list.
Alternative Flow	Note entry is corrupted in the database.
Business Rules	This will allow the user to delete notes and manage his/her notes accordingly.

High Level Code (Using Module APIs)

Listing 3.9: Delete Note

```
// User has logged-in successfully and has opened the Notes Mangagement section of the app

deleteNotes() {
    // User presses and holds on the food item to delete
    hasClicked, noteId <= detectLongClickGesture()
    status <= False
    if(hasClicked == Success) {
        showDeleteOption()

        //User taps on the delete button to confirm
        selection <= checkDeleteGesture()
        if(selection == Success) {
            status <= deleteNote(noteId)
        }
    }
    return status
}
```

Sequence Diagram

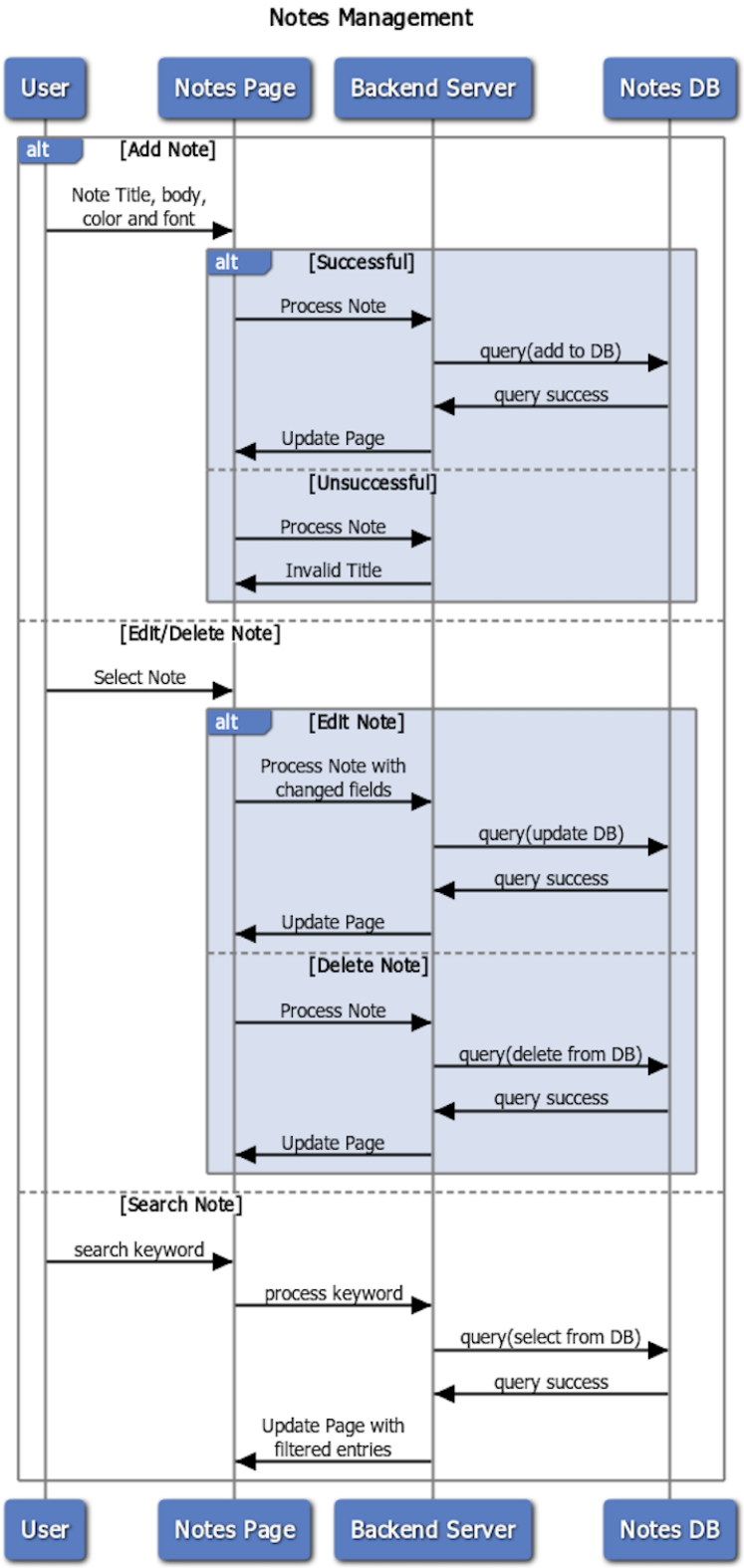


Figure 3.16: Notes Management Sequence Diagram

3.4.5 Use-Case Scenario-5: Tagging & Filtering Entries

Tagging Entries

Type	Description
Purpose	User tags the entries according to the type of the item added.
Input Data	Item Value, Category
Output Data	Item value added to the database with the input category.
Pre-conditions	User is signed-in to the application, user is on the relevant page - expenses/calories/notes.
Post-conditions	Relevant entry updated in the database with the category assigned to the item.
Basic Flow	Once the user selects the tags associated with the item and saves the changes, the corresponding entry gets marked in the database successfully.
Alternative Flow	Invalid category, incorrect query response due to database corruption.
Business Rules	This will allow the user to tag and analyze the records according to categories and manage his/her tasks accordingly.

High Level Code (Using Module APIs)

Listing 3.10: Tagging Entries

```
// User has logged-in successfully and has opened one of Expenses/Calories/Notes section of the app

// Add Tags
addTags() {
    // User can manually create his/her own desired tag
    tagText <= getUserInput()
    createTags(tagText)

    // User selects the tags/categories to assign
    [tagId]List <= selectTags()
    tagItemIntake(itemId, [tagId]List)

    // To show all the items with that tag
    showTagwiseItems(tagId)
}

// Delete Tags
deleteTags() {
    // User selects the tags/categories to delete
    [tagId]List <= selectTags()
    deleteTags([tagId]List)
}
```

Filtering Entries

Type	Description
Purpose	User filters the entries according to date range or categories.
Input Data	[ItemId]List, dateRange, byDate, category (optional argument)
Output Data	Only filtered entries displayed on the page according to date-range/categories.
Pre-conditions	User is signed-in to the application, user is on the relevant page - expenses/calories/notes.
Post-conditions	Relevant home page updated with the filtered entries, and total statistics updated accordingly.
Basic Flow	Once the user selects the date range and clicks ok, the request is sent to the Filter Module and it queries the database, and then the module sends the filtered items to the front-end to display on the page.
Alternative Flow	Invalid category/date-range, incorrect query response due to database corruption.
Business Rules	This will allow the user to filter and analyze the records according to date range/categories and manage his/her tasks accordingly.

High Level Code (Using Module APIs)

Listing 3.11: Filtering Entries

```
// User has logged-in successfully and has opened one of Expenses/Calories/Notes section of the app

// Filter Entries
filterEntries() {
    // User selects the category (optional argument)
    category <= chooseCategory()
    [itemId]List <= getCategoryItems(category)

    // User selects the data range. Other options include Daily, Weekly, Monthly, etc.
    fromDate <= getUserInput()
    toDate <= getUserInput()

    filteredItems <= filterItems([itemId]List, (fromDate, toDate))

    return filteredItems
}
```

Sequence Diagram

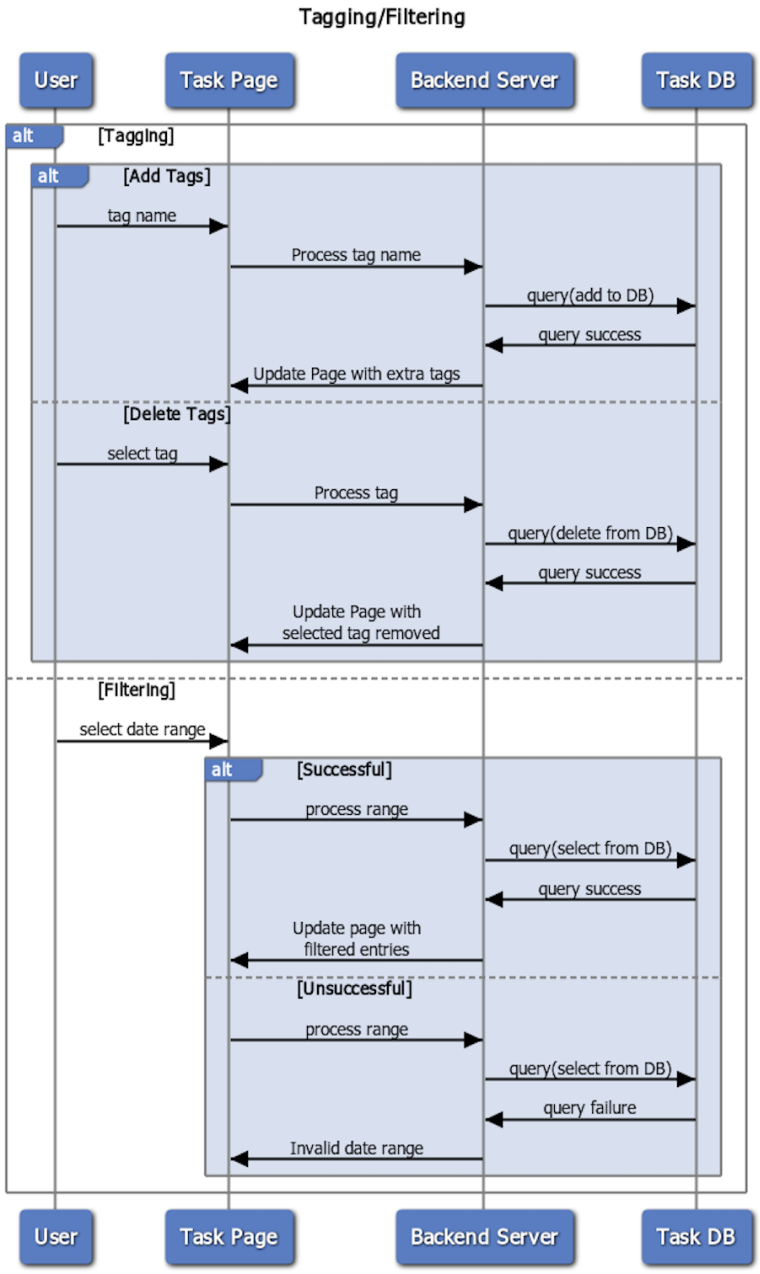


Figure 3.17: Sequence Diagram for Filtering/Tagging

Chapter 4

Deployment Design

We will deploy our front-end using an Android application (APP) and will use a Back-end server to make all the API calls from our application.

We are following a three-tier deployment strategy. Having this multi-tier strategy helps in building a scalable, robust, and secure service.

4.1 Three-Tier deployment

The three tiers in our deployment are the Client tier (frontend), Application Interface tier and a Database tier. The Client tier handles the interactions with the user in the form of a Graphical User Interface (GUI). The Application Interface tier handles the queries from the Client tier, processes the requests, and acts as a mediator between the Client tier and the Database tier. The Database tier stores all the information in the form of relational tables and acts as the knowledge base for all the queries.

4.1.1 Tier-1: Client

This tier is at the frontend. Client will have an Android based application that will be the face of this service. The frontend communicates with other tiers through application program interface (API) calls.

Deploying an Android application would require us to create an account on the app store, and publish the application there so users can access and download it easily.

4.1.2 Tier-2: Application Interface

It forms the major component of our application, and is distributed on several servers. These servers are hosted with the help of AWS that ensures high availability (24x7) and performance. AWS object stores have redundancy for high fault tolerance even if one of the servers is down. It's critical that it's available 24x7 to support and respond to the client API calls.

4.1.3 Tier-3: Database

This tier is hosted with some redundancy too, and is mainly responsible for keeping an account of all the information related to the user and the subsequent method calls. This allows the server to handle all the incoming data-lookup related queries.