

Personal Expense Management System (PEMS)

CIS 552 - Database Design Final Project

Group 1

Veda Sahaja Bandi (6)
Spoorthi Subramanya Bhat (8)
Sneha Holehonnur Sridhar (19)

Introduction

In today's fast-paced world, managing personal finances efficiently has become increasingly essential. With the multitude of financial transactions we engage in daily, it can be challenging to keep track of expenses, understand spending patterns, and maintain a healthy financial balance. The advent of digital technologies has paved the way for the development of Personal Expense Management Systems (PEMS) that cater to the diverse needs of users.

Part 1: Planned Database-driven Application

Requirements/Use-case(s)

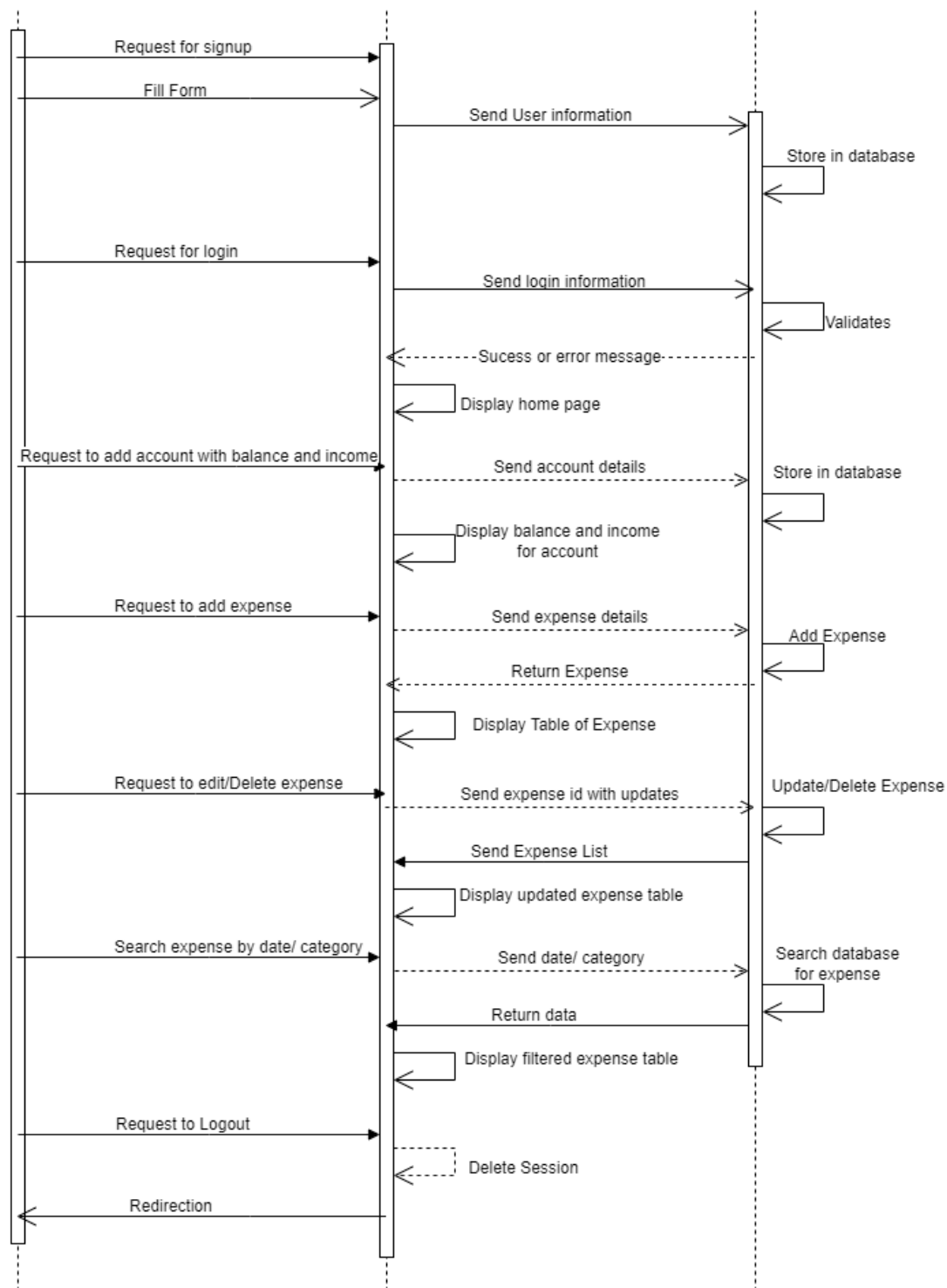
Our planned system, the Personal Expense Management System (PEMS), aims to provide users with a comprehensive platform to track, analyse, and manage their expenditures seamlessly using MongoDB, a NoSQL Database. Users can record individual transactions, update existing entries, and delete unwanted expenses. The system offers a powerful search functionality to quickly locate specific expenses based on date, or category. PEMS, hosted on our Local System, facilitates personal budgeting, helps identify spending patterns, and enhances financial literacy and awareness.

Softwares used: MongoDB Compass, Spring Boot, HTML, CSS and JS

System Features

- **User Authentication:** Users can register and log in securely to access their personal financial data.
- **Accounts Management:** Users can create accounts to manage income and their expenses.
- **Expense Management:**
 - Create expenses with details like item name, amount, date, category, and optional notes.
 - Update existing expense entries for accuracy or adjustments.
 - Delete unwanted expenses to maintain a clean system.
- **Search Functionality:** Users can efficiently search for specific expenses by date or category.
- **Financial Overview:** The system will display the total income, total expenses, loan amount and the resulting balance, providing a clear financial picture.
- **Logout:** Users can securely log out of the application.

Sequence Diagram

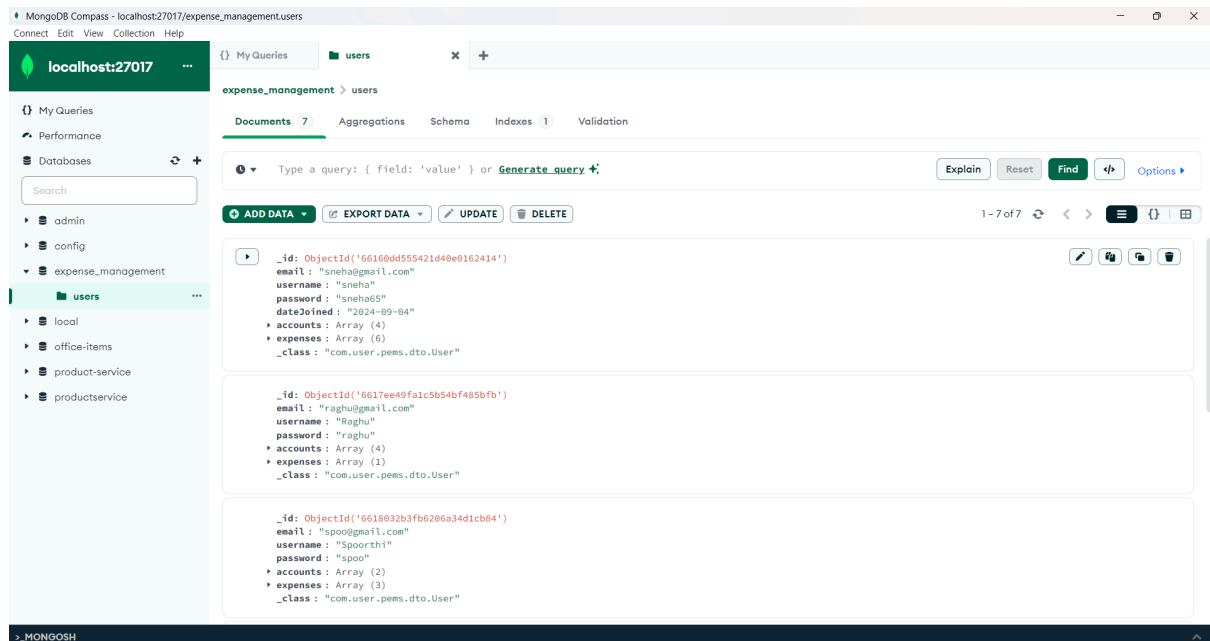


Part 2: MongoDB Implementation and Functional Demonstration

In our working implementation of the PEMS, we have incorporated MongoDB as the database engine to store and manage expense-related data. By implementing NoSQL MongoDB, we aim to enhance the scalability, flexibility, and performance of the Personal Expense Management System, providing users with a robust and efficient platform for managing their finances.

The following screenshots demonstrate the functionality of data creation, loading, updating, and querying in MongoDB through Web UI. Users can effortlessly create a user, add an account, add new expenses, update existing entries, delete expenses, and query expenses based on various parameters such as date, or category. The seamless integration of MongoDB enables efficient data management, enhancing the overall user experience of the application.

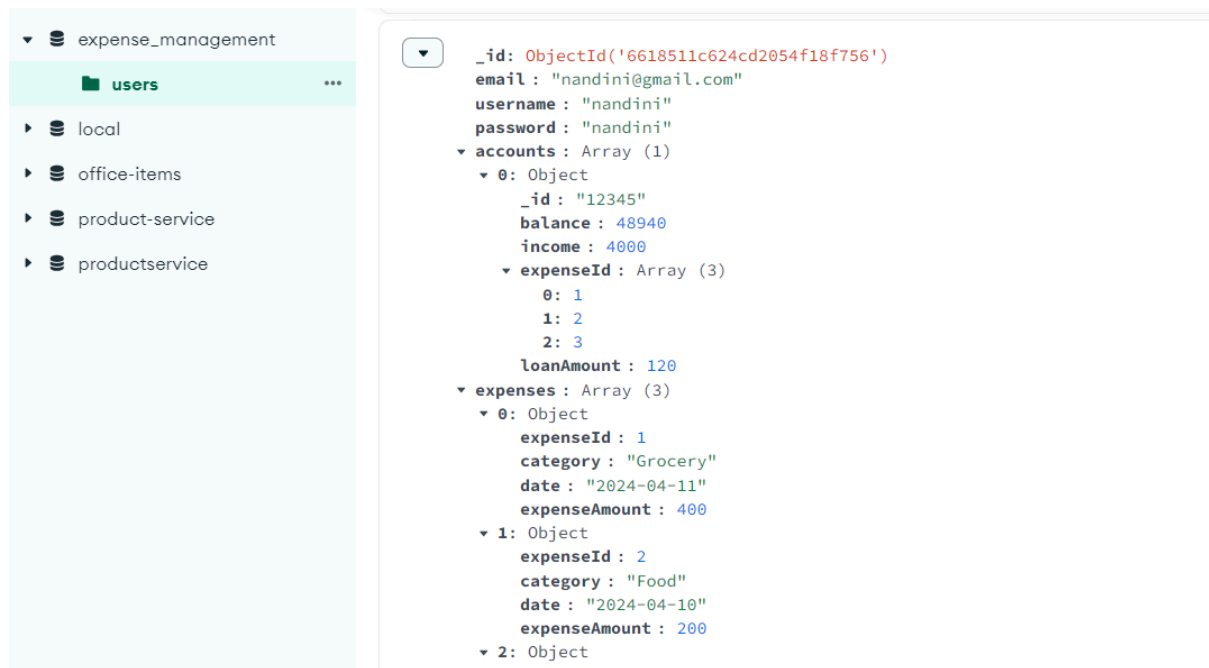
The following is the image of MongoDB Compass where our Database **expense_management** and Collection **Users** are created for our PEMS.



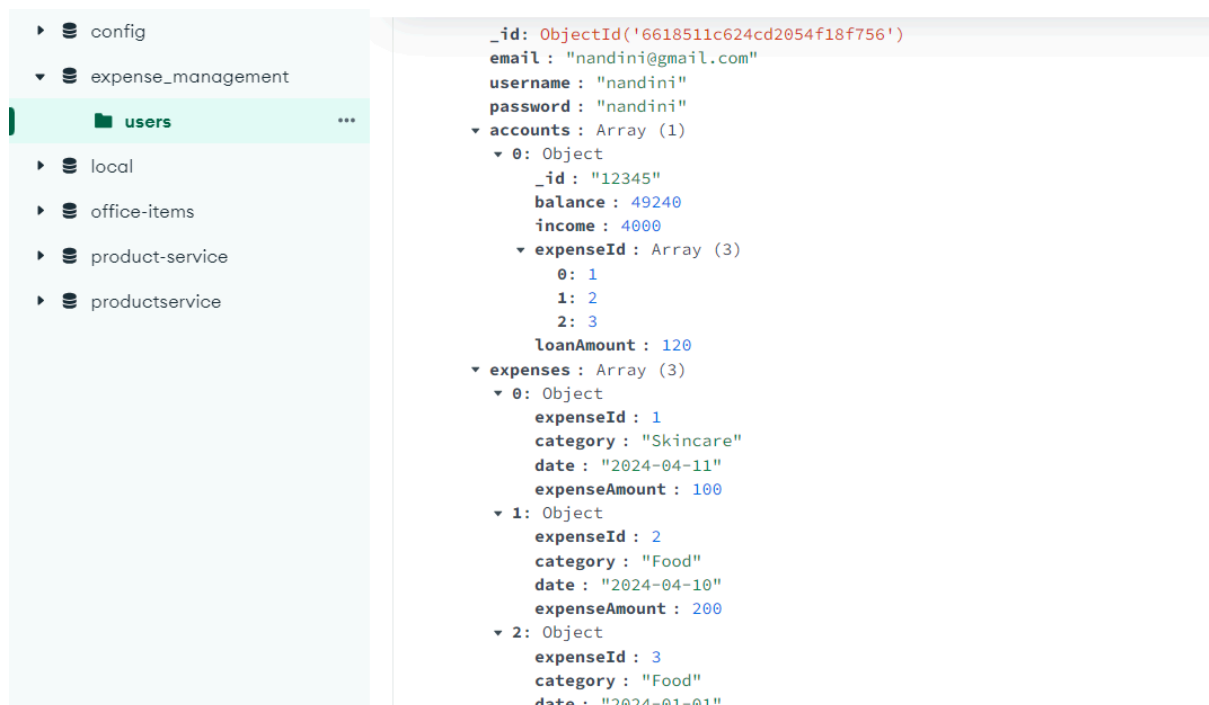
We've configured the MongoDB host, port, and database name within the applications.properties file in our Project Folder, ensuring that the data will be stored in MongoDB.

```
home.html index.html application... × UserControll... User.java
1 spring.application.name=pems
2 server.port = 8081
3
4 spring.data.mongodb.host=localhost
5 spring.data.mongodb.port=27017
6 spring.data.mongodb.database=expense_management
7 spring.jackson.default-property-inclusion = NON_NULL
8
9 spring.mvc.static-path-pattern=/**
10
```

When users add an account or expenses, the data is stored in MongoDB documents, as illustrated in the image below.



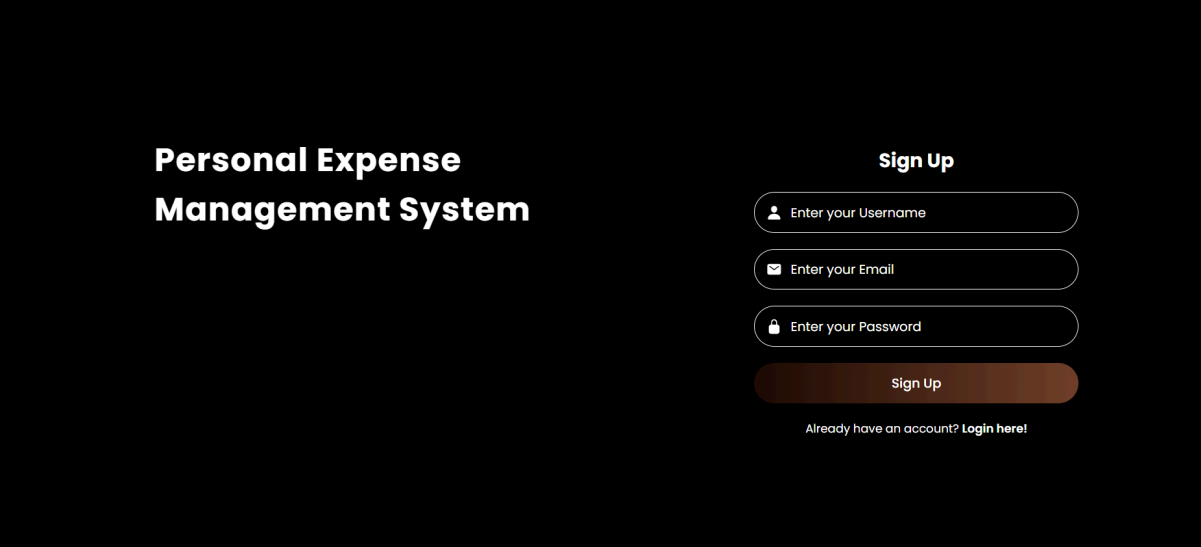
Users have the ability to edit and delete expenses, with corresponding updates made to the MongoDB documents. In the image below, the Grocery expense record has been updated to the Skincare category, demonstrating this functionality.



Part 3: Front-end Interface

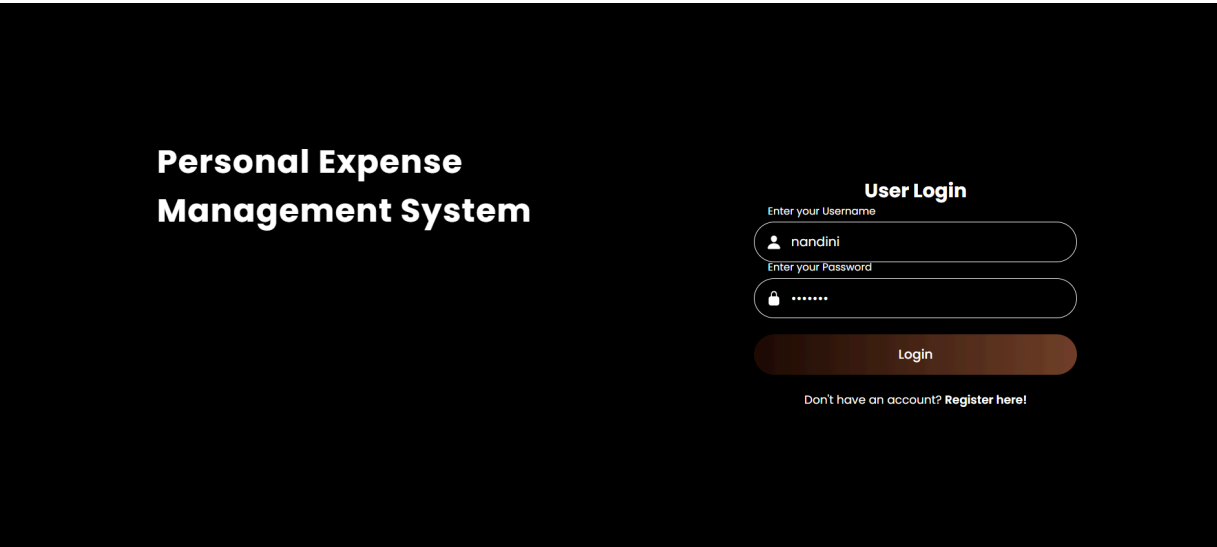
The webpage is locally hosted, utilizing HTML, CSS, and JavaScript to integrate system functionalities. These front-end technologies enable the implementation of user interaction and interface design for the personal expense management system.

Sign-Up: This is the sign-up page where users can register by giving their username, email and password.



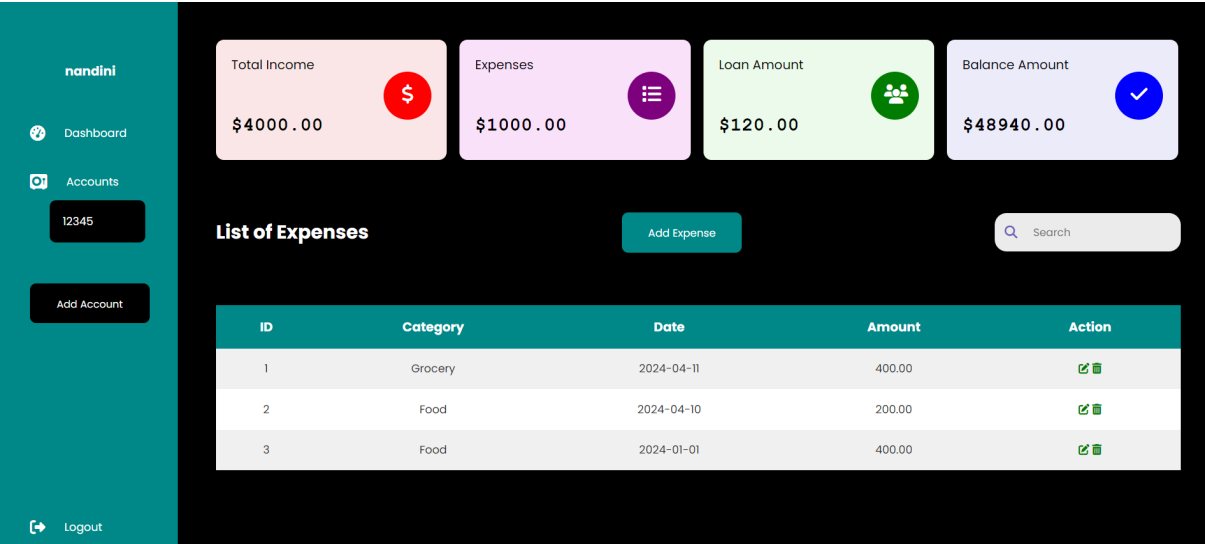
The image shows a dark-themed sign-up page for a 'Personal Expense Management System'. On the left, the title 'Personal Expense Management System' is displayed in white. On the right, under the heading 'Sign Up', there are three input fields: 'Enter your Username' with a person icon, 'Enter your Email' with an envelope icon, and 'Enter your Password' with a lock icon. Below these fields is a brown 'Sign Up' button. At the bottom, a link says 'Already have an account? Login here!'.

Login: Once the user registers, they can login using their username and password.

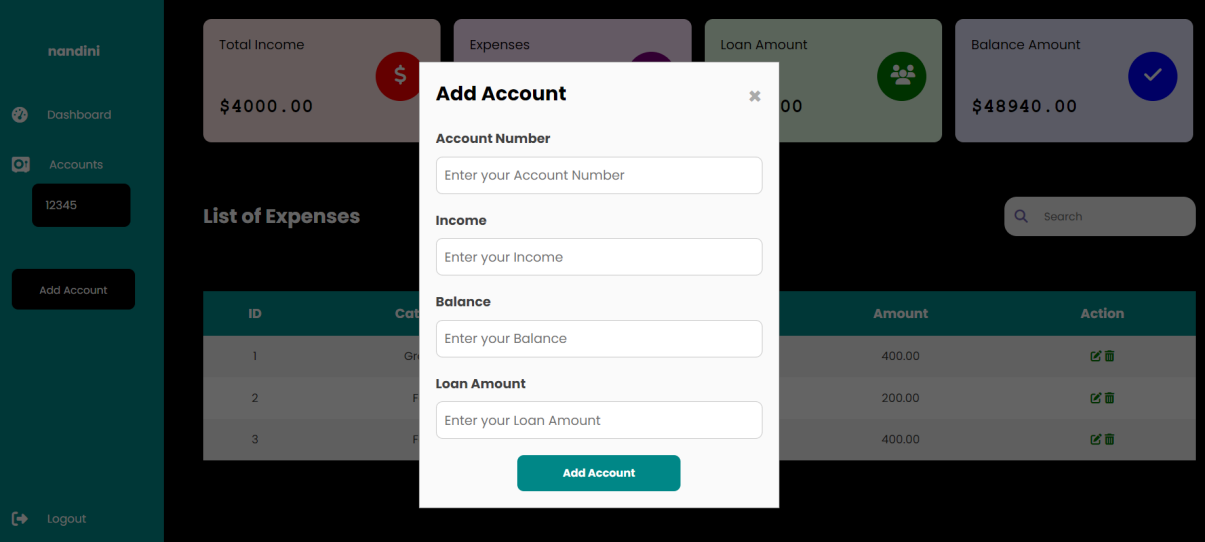


The image shows a dark-themed user login page for a 'Personal Expense Management System'. On the left, the title 'Personal Expense Management System' is displayed in white. On the right, under the heading 'User Login', there are two input fields: 'Enter your Username' with the text 'nandini' and a person icon, and 'Enter your Password' with masked characters '*****' and a lock icon. Below these fields is a brown 'Login' button. At the bottom, a link says 'Don't have an account? Register here!'.

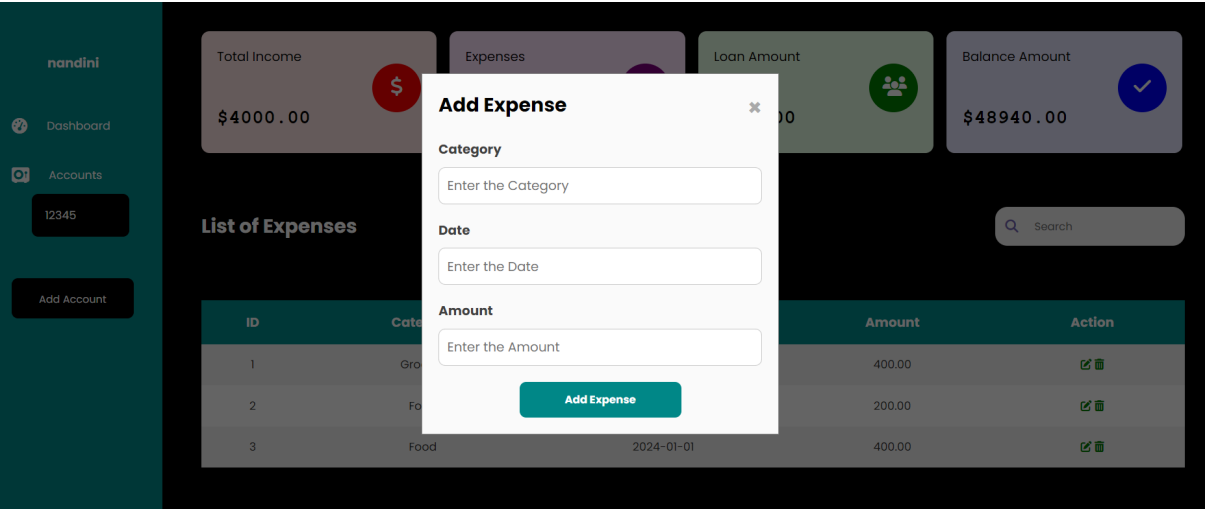
Home Screen: Once the user is logged in, this is the home screen that will be visible. The elements that are included here are Total Income, Expenses, Loan Amount and Balance Amount. The account number is to the left and the list of expenses for that account is displayed in the table.



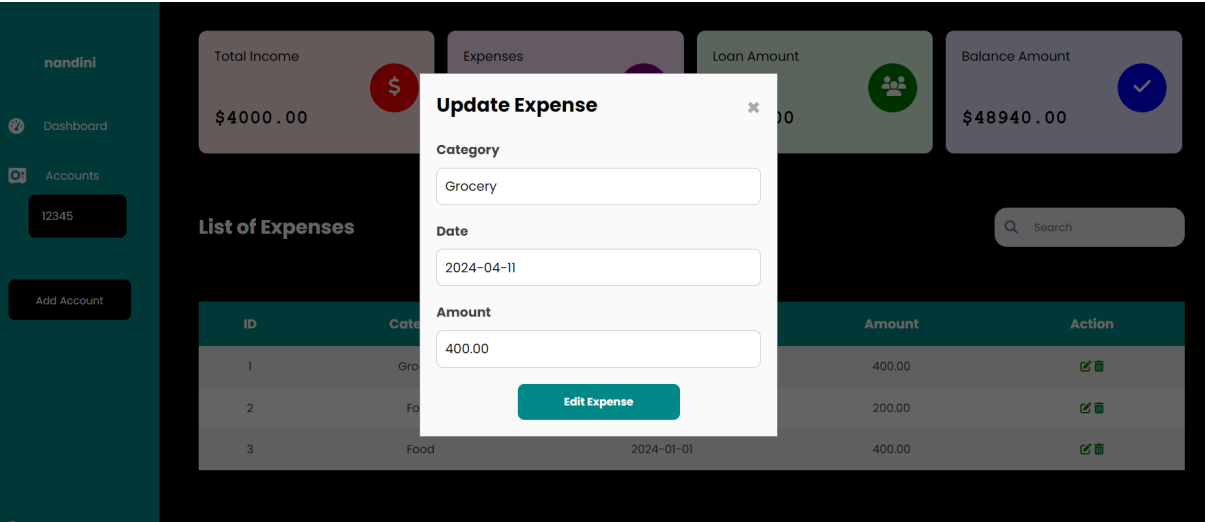
Add Account: The user can add the account details by giving the following information which is displayed here. If user manages multiple bank accounts, they can add it.



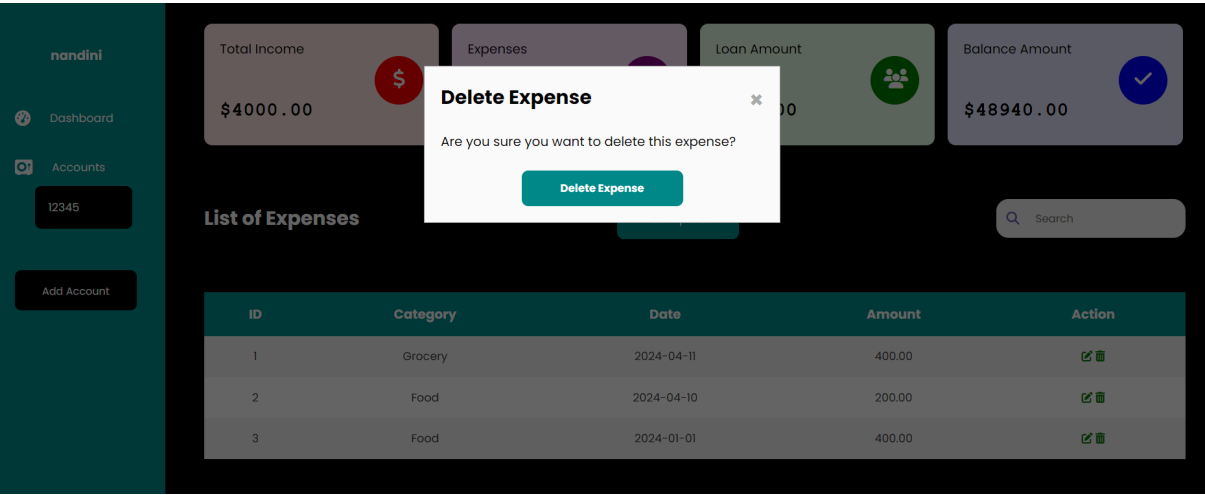
Add Expense: So for the account highlighted to the left, the user can add corresponding expenses by clicking the Add Expense Button.



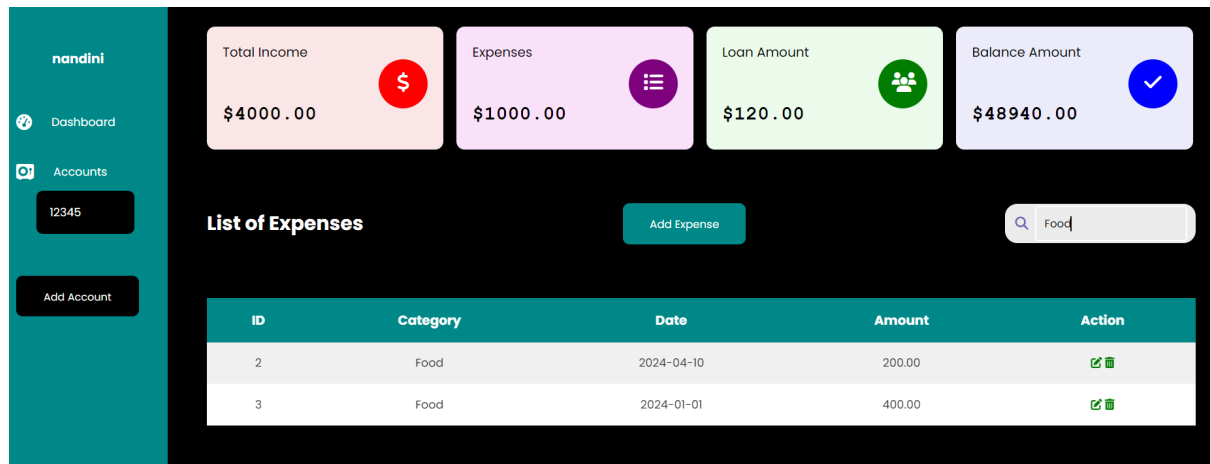
Update Expense: Each expense has the update button if the user wants to update any information regarding that.



Delete Expense: If the user thinks they no longer need that particular expense, they can click the delete button and confirm to delete that expense.



Search Operation: The user can search by Category or By Date and the matching results will be displayed in the dashboard.



Part 4: Source Code Screenshots

The following code snippets demonstrate how the interface is structured and styled using HTML, CSS, and JavaScript, as well as how the application communicates with the MongoDB database to perform CRUD operations for managing expenses effectively.

Data Transfer Objects:

In a Spring Boot application, DTOs are commonly used to encapsulate data transferred between the controller and service layers, or between the service layer and the persistence layer. This segregation helps in maintaining a clear separation of concerns and improves the overall modularity of the codebase. The following are the DTOs used:

User

```
package com.user.pems.dto;

import java.util.List;

@Document(collection = "users")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class User {
    @Id
    private String id;
    private String email;
    private String username;
    private String password;
    private String dateJoined;
    private List<Account> accounts;
    private List<Expense> expenses;
}
```

Represents the structure of User data in the database which includes the user id, email, username, password, dateJoined, accounts, expenses.

Account


```

package com.user.pems.dto;

import java.util.Collections;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Account {
    @Id
    private String accountNo;
    private double balance;
    private double income;
    private List<Integer> expenseId;
    private double loanAmount;
}

```

Represents the structure of Account data for every user including the account number, balance, income, expenseId, loanAmount.

Expense

```

package com.user.pems.dto;

import java.util.List;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Expense {
    private static int nextId;
    private int expenseId;
    private String category;
    private String date;
    private double expenseAmount;

    public static int getNextId(List<Expense> existingExpenses) {
        if (existingExpenses.isEmpty()) {
            nextId = 1;
        } else {
            nextId = existingExpenses.stream()
                .mapToInt(Expense::getExpenseId)
                .max()
                .orElse(0) + 1;
        }
        return nextId++;
    }
}

```

Represents the structure of Expense data for every user's account including Id, category, date and expenseAmount.

User Repository

```

1 package com.user.pems.repository;
2
3 import java.util.Optional;
4
5
6
7
8
9 public interface UserRepository extends MongoRepository<User, String>{
10     Optional<User> findByUsername(String username);
11 }
12

```

A user repository is used to manage data persistence and retrieval in a database. It provides an abstraction layer over the database and simplifies the process of interacting with it, allowing developers to focus on business logic rather than database operations.

Features

This section of screenshots contains the connection of features in our application.

User Login

This feature allows an already existing user to login to the system.

- Javascript connecting frontend and backend

```
function loginUser() {  
    var username = document.getElementById("loginUsername").value;  
    var password = document.getElementById("loginPassword").value;  
  
    fetch("http://localhost:8081/users/" + username)  
        .then(response => {  
            if (response.ok) {  
                return response.json();  
            } else {  
                alert("User does not exist. Please register first.");  
                throw new Error("User not found");  
            }  
        })  
        .then(user => {  
            if (user.password === password) {  
                sessionStorage.setItem("loggedInUsername", username);  
                window.location.href = "home.html";  
            } else {  
                alert("Incorrect password. Please try again.");  
            }  
        })  
        .catch(error => {  
            console.error("Error:", error);  
        });  
}
```

If the user is not in the database, "User does not exist. Please register first." alert appears and if there is an error in the password, "Incorrect password. Please try again" alert appears stopping the user from logging in to the website.

- Controller

```
@GetMapping("/{username}")  
public User getUserByUsername(@PathVariable String username) {  
    return userService.getUserByUsername(username);  
}
```

The controller defines a GET request with username as the parameter.

- Service

```
public User getUserByUsername(String username) {  
    Optional<User> userOptional = userRepository.findByUsername(username);  
    return userOptional.orElse(null);  
}
```

This function in the service layer calls the findByUsername method from the repository.

Register User:

Allows the user to register into the application

- Javascript connecting frontend and backend

```

function saveUser() {
    var form = document.getElementById("signupForm");
    var formData = new FormData(form);
    var user = {
        email: formData.get("email"),
        username: formData.get("username"),
        password: formData.get("password")
    };

    fetch("http://localhost:8081/users", {
        method: "POST",
        headers: {
            "Content-Type": "application/json"
        },
        body: JSON.stringify(user)
    })
    .then(response => {
        if (response.ok) {
            alert("User saved successfully");
        } else {
            response.text().then(errorMessage => {
                alert(errorMessage);
            });
        }
    })
    .catch(error => {
        console.error("Error:", error);
    });
}

```

The user object is passed to the API in order to register users in the database. There is a success as well as error message displayed accordingly.

- Controller

```

@PostMapping
public ResponseEntity<> addUser(@RequestBody User user) {
    User existingUser = userService.getUserByUsername(user.getUsername());
    if (existingUser != null) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Username already exists");
    } else {
        User savedUser = userService.addUser(user);
        return ResponseEntity.ok(savedUser);
    }
}

```

The controller defines a POST request to the database to add the user hoping to register to the system.

- Service

```

public User addUser(User user) {
    return userRepository.save(user);
}

```

The service addUser calls the save method of the user repository to register the new user data in the database.

Adding an account

Allows users to create accounts.

- Javascript code to connect frontend and backend

```
// Add an event listener to the form's submit event
document.getElementById("addAccountForm").addEventListener("submit", (event) => {
    const loggedInUsername = sessionStorage.getItem("loggedInUsername");
    event.preventDefault();

    const formData = new FormData(event.target);
    const newAccount = {
        accountNo: formData.get("accountNo"),
        balance: parseFloat(formData.get("balance")),
        income: parseFloat(formData.get("income")),
        loanAmount: formData.get("loanAmount"),
        expenseId: []
    };

    // Make a POST request to add the new account
    fetch("http://localhost:8081/users/" + loggedInUsername + "/accounts", {
        method: "POST",
        headers: {
            "Content-Type": "application/json"
        },
        body: JSON.stringify(newAccount)
    })
    .then(response => response.json())
    .then(user => {
        console.log("New account added:", user);
        location.reload();
    })
    .catch(error => console.error("Error adding account:", error));

    // Close the modal
    document.getElementById("addAccountModal").style.display = "none";
});
```

Records the account details and makes a request to add an account in the database.

- Controller

```
@PostMapping("/{username}/accounts")
public ResponseEntity<User> addAccount(@PathVariable String username, @RequestBody Account newAccount) {
    User updatedUser = userService.addAccount(username, newAccount);
    if (updatedUser != null) {
        return new ResponseEntity<>(updatedUser, HttpStatus.CREATED);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

The controller makes a POST request to the database to add the account for the username parameter.

- Service

```
public User addAccount(String username, Account newAccount) {
    User existingUser = userRepository.findByUsername(username).orElse(null);
    if (existingUser != null) {
        if (existingUser.getAccounts() == null) {
            existingUser.setAccounts(new ArrayList<>());
        }
        existingUser.getAccounts().add(newAccount);
        return userRepository.save(existingUser);
    }
    return null;
}
```

This service creates a new account list and adds the account under the user or appends the existing user list.

Adding an expense for the account

This feature allows the users to add expenses to particular accounts

- Javascript code connecting frontend and backend

```
// Add an event listener to the Add Expense button
document.getElementById("confirmAddBtn").addEventListener("click", () => {
    const loggedInUsername = sessionStorage.getItem("loggedInUsername");
    const accountNo = getCurrentAccountNo(); // Implement this function to get the current account number

    const category = document.getElementById("categoryInput").value;
    const date = document.getElementById("dateInput").value;
    const amount = parseFloat(document.getElementById("amountInput").value);

    // Create a new Expense object
    const newExpense = {
        category: category,
        date: date,
        expenseAmount: amount
    };

    // Make a POST request to add the new expense
    fetch("http://localhost:8081/users/" + loggedInUsername + "/expenses/" + accountNo, {
        method: "POST",
        headers: {
            "Content-Type": "application/json"
        },
        body: JSON.stringify(newExpense)
    })
    .then(response => response.json())
    .then(user => {
        console.log("New expense added:", user);
        location.reload(); // Refresh the page to update the expense table
    })
    .catch(error => console.error("Error adding expense:", error));
    document.getElementById("addExpenseModal").style.display = "none";
});
```

A new expense object is created with the user input and the username and accountNo are sent as parameters to the API.

- Controller

```
@PostMapping("/{username}/expenses/{accountNo}")
public User addExpense(@PathVariable String username, @PathVariable String accountNo, @RequestBody Expense newExpense) {
    return userService.addExpense(username, accountNo, newExpense);
}
```

The controller makes a POST request to the database to add expense to the particular user's particular account.

- Service

```
public User addExpense(String username, String accountNo, Expense newExpense) {
    User existingUser = getUserByUsername(username);
    if (existingUser != null) {
        // Initialize expenses list if null
        if (existingUser.getExpenses() == null) {
            existingUser.setExpenses(new ArrayList<>());
        }
        List<Expense> existingExpenses = existingUser.getExpenses();
        int expenseId = Expense.getNextId(existingExpenses);
        newExpense.setExpenseId(expenseId);

        // Add the new expense to the user's expenses
        existingExpenses.add(newExpense);

        List<Account> accounts = existingUser.getAccounts();
        for (Account account : accounts) {
            if (account.getAccountNo().equals(accountNo)) {
                if (account.getExpenseId() == null) {
                    account.setExpenseId(new ArrayList<>());
                }
                account.getExpenseId().add(expenseId);
                double currentBalance = account.getBalance();
                double expenseAmount = newExpense.getExpenseAmount();
                double updatedBalance = currentBalance - expenseAmount;
                account.setBalance(updatedBalance);
                break;
            }
        }
        return userRepository.save(existingUser);
    }
    return null;
}
```

A new list of expenses is created or the existing list is appended in every post of a new expense.

Updating an expense for the expenseld

This feature allows users to update any expense in their account with reference to an expense ID.

- Javascript code connecting frontend and backend

```
// Add an event listener to the update expense form's submit event
document.getElementById("updateExpenseForm").addEventListener("submit", (event) => {
  const loggedInUsername = sessionStorage.getItem("loggedInUsername");
  event.preventDefault();
  const category = document.getElementById("updateExpenseForm").elements["category"].value;
  const date = document.getElementById("updateExpenseForm").elements["date"].value;
  const amount = parseFloat(document.getElementById("updateExpenseForm").elements["amount"].value);
  const expenseId = parseInt(document.getElementById("expenseId").value);
  const updatedExpense = {
    expenseId: expenseId,
    category: category,
    date: date,
    expenseAmount: amount
  };
  // Make a PUT request to update the expense
  fetch(`http://localhost:8081/users/${loggedInUsername}/expenses/${expenseId}`, {
    method: "PUT",
    headers: {
      "Content-Type": "application/json"
    },
    body: JSON.stringify(updatedExpense)
  })
  .then(response => response.json())
  .then(user => {
    console.log("Expense updated:", user);
    location.reload(); // Refresh the page to update the expense table
  })
  .catch(error => console.error("Error updating expense:", error));
  document.getElementById("updateExpenseModal").style.display = "none";
});
```

The updated details are sent to the API and the expense table is refreshed to obtain the updated list.

- Controller

```
@PutMapping("/{username}/expenses/{expenseId}")
public User updateExpenseById(@PathVariable String username, @PathVariable int expenseId, @RequestBody Expense expense) {
    return userService.updateExpenseById(username, expenseId, expense);
}
```

The controller makes a PUT request to the database with parameters username and expenseld in order to update the expense.

- Service

```
public User updateExpenseById(String username, int expenseId, Expense updatedExpense) {
    User existingUser = getUserByUsername(username);
    if (existingUser != null) {
        List<Expense> expenses = existingUser.getExpenses();
        for (Expense expense : expenses) {
            if (expense.getExpenseId() == expenseId) {
                // Calculate the net expense difference
                double oldExpenseAmount = expense.getExpenseAmount();
                double newExpenseAmount = updatedExpense.getExpenseAmount();
                double expenseDifference = newExpenseAmount - oldExpenseAmount;
                // Update the existing expense
                expense.setCategory(updatedExpense.getCategory());
                expense.setDate(updatedExpense.getDate());
                expense.setExpenseAmount(newExpenseAmount);

                // Update the balanceAmount of the corresponding Account
                List<Account> accounts = existingUser.getAccounts();
                for (Account account : accounts) {
                    if (account.getExpenseId() != null && account.getExpenseId().contains(expenseId)) {
                        double currentBalance = account.getBalance();
                        double updatedBalance = currentBalance - expenseDifference;
                        account.setBalance(updatedBalance);
                        break; // Assuming expenseId is unique
                    }
                }
                return userRepository.save(existingUser);
            }
        }
    }
    return null;
}
```

The balanceAmount of the account is updated with the other details of the expense in the updateExpenseByExpenseId service.

Delete expense by expenseId

This feature allows the user to delete an expense with reference to the expense ID from the expense table.

- Javascript code to connect frontend and backend

```
//Add an event listener to the delete expense form's submit event
document.getElementById("deleteExpenseForm").addEventListener("submit", (event) => {
    const loggedInUsername = sessionStorage.getItem("loggedInUsername");
    event.preventDefault();
    const expenseId = parseInt(document.getElementById("expenseIdToDelete").value);

    // Make a DELETE request to delete the expense
    fetch(`http://localhost:8081/users/${loggedInUsername}/expenses/${expenseId}`, {
        method: "DELETE",
    })
    .then(response => {
        if (response.ok) {
            console.log("Expense deleted successfully.");
            location.reload(); // Refresh the page to update the expense table
        } else {
            throw new Error("Failed to delete expense.");
        }
    })
    .catch(error => console.error("Error deleting expense:", error));
    document.getElementById("deleteExpenseModal").style.display = "none";
});
```

On click of the delete button in the table, the respective expense is deleted and the expense table is refreshed to obtain the updated table.

- Controller

```
@DeleteMapping("/{username}/expenses/{expenseId}")
public String deleteExpenseByExpenseId(@PathVariable String username, @PathVariable int expenseId) {
    User updatedUser = userService.deleteExpenseByExpenseId(username, expenseId);
    if (updatedUser != null) {
        return "Expense with ID " + expenseId + " deleted successfully.";
    } else {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Expense not found.");
    }
}
```

The controller takes in the username and expenseId as parameters to perform the DELETE operation in the database for the respective expenseId.

- Service

```
public User deleteExpenseByExpenseId(String username, int expenseId) {
    User existingUser = getUserByUsername(username);
    if (existingUser != null) {
        List<Expense> expenses = existingUser.getExpenses();
        for (Expense expense : expenses) {
            if (expense.getExpenseId() == expenseId) {
                expenses.remove(expense);
                List<Account> accounts = existingUser.getAccounts();
                for (Account account : accounts) {
                    if (account.getExpenseId().contains(expenseId)) {
                        account.getExpenseId().remove(Integer.valueOf(expenseId));
                        break;
                    }
                }
            }
        }

        // Save the updated user
        return userRepository.save(existingUser);
    }
    return null;
}
```

The service checks if the expenseId is present in the list of expenses and deletes the expenseId from the expenseId list in the account as well the expense object.

Search expense by date or category of expense

This feature allows the users to search for expenses with respect to expense date and category.

- Javascript code to connect frontend with backend

```
const searchAndPopulateTable = (searchCriteria) => {
  const loggedInUsername = sessionStorage.getItem("loggedInUsername");
  const accountNo = getCurrentAccountNo();

  fetch(`http://localhost:8081/users/${loggedInUsername}/expenses/search/${accountNo}?searchCriteria=${searchCriteria}`)
    .then(response => response.json())
    .then(matchedExpenses => {
      const tbody = document.querySelector("table tbody");
      tbody.innerHTML = "";
      matchedExpenses.forEach(expense => {
        const tr = document.createElement("tr");
        tr.innerHTML = `
          <td>${expense.expenseId}</td>
          <td>${expense.category}</td>
          <td>${expense.date}</td>
          <td>${expense.expenseAmount.toFixed(2)}</td>
          <td>
            <button class="edit-expense-btn">
              <i class="fas fa-edit"></i>
            </button>
            <button class="delete-expense-btn">
              <i class="fas fa-trash-alt"></i>
            </button>
          </td>
        `;
        tbody.appendChild(tr);
      });
    })
    .catch(error => console.error("Error searching expenses:", error));
};
```

The code updates the expense table displayed according to the search criteria results.

- Controller

```
@GetMapping("/{username}/expenses/search/{accountNo}")
public ResponseEntity<List<Expense>> searchByDateOrCategory(@PathVariable String username, @PathVariable String accountNo,
    @RequestParam String searchCriteria) {
    List<Expense> matchedExpenses = userService.searchByDateOrCategory(username, accountNo, searchCriteria);
    if (matchedExpenses != null) {
        return ResponseEntity.ok(matchedExpenses);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

The controller takes in username, accountNo and searchCriteria as the parameters to make an API call to fetch the requested expenses.

- Service

```
public List<Expense> searchByDateOrCategory(String username, String accountNo, String searchCriteria) {
    User user = userRepository.findByUsername(username).orElse(null);
    if (user != null) {
        List<Expense> matchedExpenses = new ArrayList<>();
        // Find the account with the given accountNo
        Account account = user.getAccounts().stream()
            .filter(acc -> acc.getAccountNo().equals(accountNo))
            .findFirst()
            .orElse(null);

        if (account != null) {
            // Get the list of expenseIds associated with the account
            List<Integer> expenseIds = account.getExpenseId();
            // Filter expenses based on expenseId and search criteria
            for (Expense expense : user.getExpenses()) {
                if (expenseIds.contains(expense.getExpenseId()) &&
                    (expense.getDate().equals(searchCriteria) || expense.getCategory().equalsIgnoreCase(searchCriteria))) {
                    matchedExpenses.add(expense);
                }
            }
        }
        return matchedExpenses;
    }
    return null;
}
```


The service gets the list of expenses associated with the account and filters expenses based on expense and search criteria.

Future Scope

- Implement functionality for generating weekly or monthly reports and conducting analytics to provide users with insights into their spending patterns and financial habits.
- Integrate features for managing EMI payments, allowing users to track and manage their installment payments conveniently within the application.
- Enhance security and privacy measures by implementing advanced encryption techniques, multi-factor authentication, and access control mechanisms to safeguard user data.
- Incorporate community and collaboration features such as discussion forums, shared expense tracking for groups, and collaborative budgeting tools to facilitate financial management among friends, family, or colleagues.
- Utilize machine learning and AI-driven algorithms to provide personalized financial recommendations, budget optimization suggestions, and predictive analytics to assist users in making informed financial decisions.

Conclusion

In conclusion, our PEMS goes beyond mere expense tracking. By offering insightful analysis, goal-oriented budgeting, and seamless reporting, it empowers users to take control of their finances and achieve their financial aspirations. We believe our user-centric design and comprehensive features position our PEMS as a valuable tool for individuals seeking financial clarity and control in today's dynamic world.