

TABLE OF CONTENT

CHAPTER NO.	TITLE
1	INTRODUCTION
1.1	Project Overview
1.2	Purpose
2	LITERATURE SURVEY
2.1	Existing System
2.2	References
2.3	Problem Statement Definition
3	IDEATION & PROPOSED SYSTEM
3.1	Empathy Map Canvas
3.2	Ideation & Brainstorming
3.3	Proposed Solution
3.4	Problem Solution fit
4	REQUIREMENT ANALYSIS
4.1	Functional requirement
4.2	Non-Functional requirements
5	PROJECT DESIGN
5.1	Data Flow Diagrams
5.2	Solution & Technical Architecture
5.3	User Stories
6	PROJECT PLANNING & SCHEDULING
6.1	Sprint Planning & Estimation
6.2	Sprint Delivery Schedule
7	CODING & SOLUTIONING
7.1	Feature 1
7.2	Feature 2

8 TESTING

8.1 Test Cases

8.2 User Acceptance Testing

9 RESULTS

9.1 Home Page

9.2 Sign in page

9.3 Budget

9.4 Report Page

10 ADVANTAGES & DISADVANTAGES

11 CONCLUSION

12 FUTURE SCOPE

GitHub Link

Demo Link

CHAPTER 1

INTRODUCTION

1.1 Project Overview

This project is an attempt to manage our daily expenses in a more efficient and manageable way. Sometime we can't remember where our money goes. And we can't handle our cash flow. For this problem, we need a solution that everyone can manage their expenses. So we decided to find an easier way to get rid of this problem. So, our application attempts to free the user with as much as possible the burden of manual calculation and to keep the track of the expenditure. Instead of keeping a diary or a log of the expenses, this application enables the user to not just keep the control on the expenses but also to generate and save reports. With the help of this application, the user can manage their expenses on a daily, weekly and monthly basis. Users can insert and delete transactions as well as can generate and save their reports. The graphical representation of the application is the main part of the system as it appeals to the user more and is easy to understand.

1.2 Purpose

An expense tracker is a software or application that helps to keep an accurate record of your money inflow and outflow. Many people in India live on a fixed income, and they find that towards the end of the month they don't have sufficient money to meet their needs. So, for keep tracking on their income and expense this app was developed.

CHAPTER 2

LITERATURE SURVEY

2.1 Existing problem

The problem faced by today's people are that they can't even remember that how much they spend in their day to day expense and also they can't make note of their all expenses. In this time there is no such perfect solution which helps a person to track their daily expenditure easily and efficiently and notify them about the money shortage they have. For making this they have maintained ledgers for the expense and income or computer logs is to maintain for such data and the calculation is done manually by the user, which may generate errors leading to the money loss. It is not complete tracking process.

2.2 Reference

1. <https://nevonprojects.com/daily-expense-tracker-system/>
2. <https://phpgurukul.com/daily-expense-tracker-using-php-and-mysql/>
3. <https://ijarsct.co.in/Paper391.pdf>
4. <https://kandi.openweaver.com/>

2.3 Problem Statement Definition

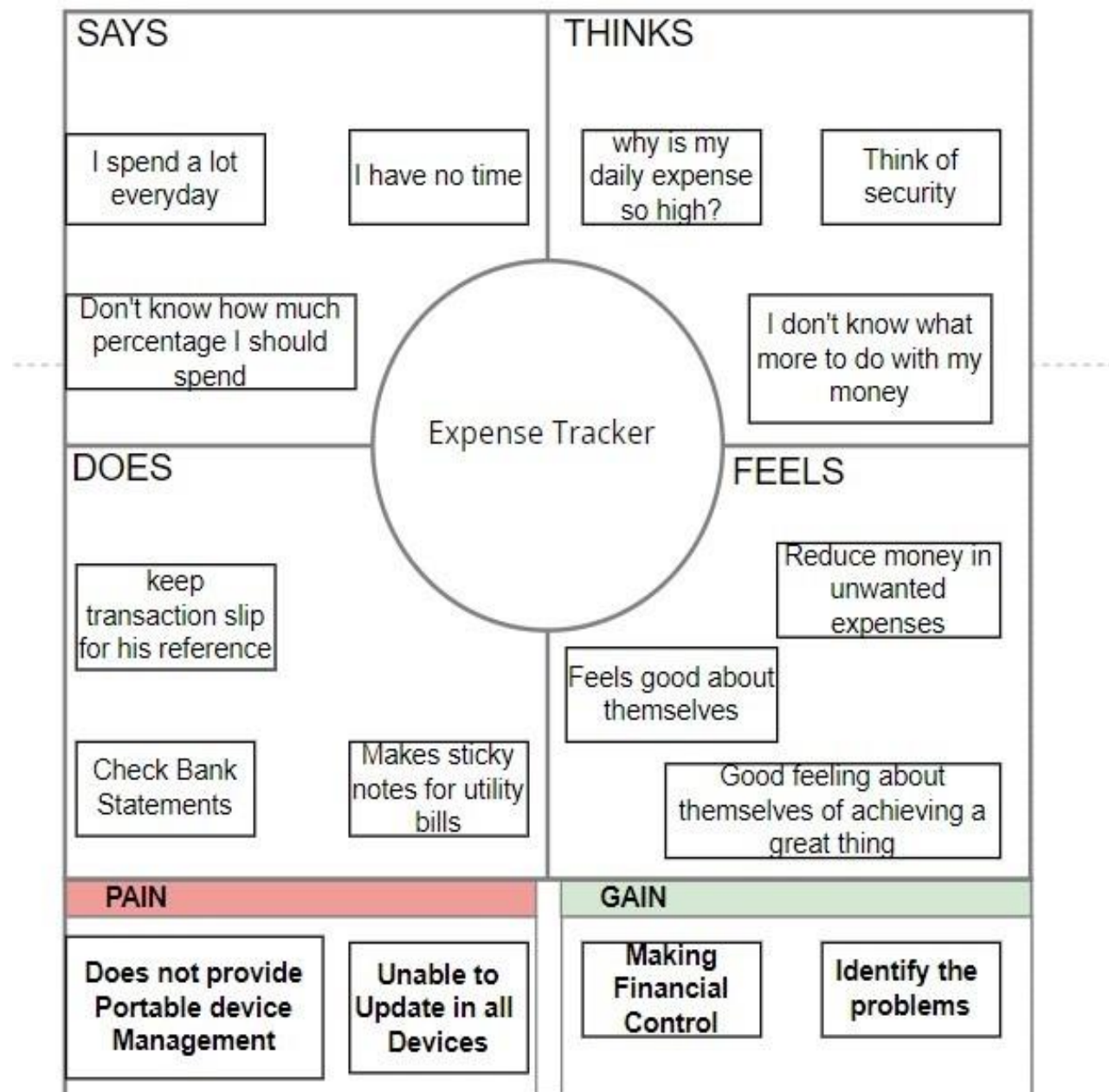
At the instant, there is no as such complete solution present easily or we should say free of cost which enables a person to keep a track of its daily expenditure easily. To do so a person has to keep a log in a diary or in a computer, also all the calculations needs to be done by the user which may sometimes results in errors leading to losses. Due to lack of a complete tracking system, there is a constant overload to rely on the daily entry of the expenditure and total estimation till the end

of the month. As the name itself suggests, this project is an attempt to manage our daily expenses in a more efficient and manageable way. The system attempts to free the user with as much as possible the burden of manual calculation and to keep the track of the expenditure. Instead of keeping a dairy or a log of the expenses on the smartphones or laptops, this system enables the user to calculate the expenses accurately without any bugs. One of the drawbacks is the on-going maintenance, a lot of budget software offer the simplicity of integrating with all users financial accounts and consolidating their activity into one dashboard. However though, some of this existing software mostly have complicated features that are not user friendly.

CHAPTER 3

IDEATION & PROPOSED SOLUTION

3.1 Empathy Map canvas



3.2 Ideation & Brainstorming

Our problem statement is about Expense Tracker, discussing this problem we team members came out with few solutions out of all the solutions we discussed we are here going to upload the top three ideas.

IDEA 1:

To reduce manual calculations, we propose an application. This application allows users to maintain a digital automated diary. Each user will be required to register on the system at registration time, the user will be provided id, which will be used to maintain the record of each unique user.

IDEA 2:

Expense Tracker application which will keep a track of Income-Expense of a user on a day to day basis. The best organizations have a way of tracking and handling these reimbursements. This ideal practice guarantees that the expenses tracked are accurately and in a timely manner.

IDEA 3:

From a company perspective, timely settlements of these expenses when tracked well will certainly boost employee's morale. Additional feature of Expense and income prediction helps to better budget management.

3.3 Proposed Solution

S.NO.	Parameter	Description
1.	Problem Statement	<p>In paper-based expense tracker system it is difficult to track our monthly expenses manually.</p> <p>In paper-based expense tracker system it is difficult to track our monthly expenses manually.</p> <p>The paper-based expense records may get lost in case of fire accidents, flood etc.</p>
2.	Scalability of the Solution	<p>This application can handle large number of users and data with high performance and security. This application can adapt for both large-scale and small-scale purposes. Easily available in all kinds of devices.</p>
3.	Idea / Solution description	<p>Daily expense management system which is specially designed for non-salaried and salaried personnel for keeping track of their daily expenditure with easy and effective way through computerized system which tends to eliminate manual paper works. Personal finance applications will ask users to add their expenses and based on their expenses wallet balance will be updated which will be visible to the user. They have an option to set a limit for the amount to be used for that particular month if the limit is exceeded the user will be notified with an email alert.</p>
4.	Novelty / Uniqueness	<p>The user gets notified when their expense exceeds the limit and also it reminds the user when they</p>

		forgot to make entry. Tracking expenses through SMS. Data analytics on expenses. Future expense prediction
5.	Social Impact / Customer Satisfaction	The application should be able to generate reports of their spending and notify users if they have exceeded their budget. It is designed to be dynamic to produce the prediction. It also provides users' personal information, their income as well as their expenses. This application can create awareness among common people about finance and stuffs. This application also helps user to be financially responsible. It Reduces time rather than entering details manually.
6.	Business Model (Revenue Model)	This Application is provided for free of cost. But It will have some advertisement. In premium version there is no advertisement and contains some additional features.

3.4 Proposed Solution Fit

The solution to this problem is, the user who spends more money can avoid unwanted expenses. The user can effectively spend their money for their essential needs.

Define CS, fit into CL	1. CUSTOMER SEGMENT(S) CS People who are struggling to track their expenses are our customers.They can use our app to maintain records about their income and expenses	6. CUSTOMER LIMITATIONS <small>EG. BUDGET, DEVICES</small> CL User have to entry every record manually.The category divided may be blunder or messy.person who is handling system must have some technical knowledge.	5. AVAILABLE SOLUTIONS <small>PLUSES & MINUSES</small> AS User can add their income and expenses.They have an option to set a limit for the amount to be used for that particular month if the limit is exceeded the user will be notified with an email alert.	Explore AS, differentiate
	2. PROBLEMS / PAINS <small>• ITS FREQUENCY</small> PR In paper-based expense tracker system it is difficult to track our monthly expenses manually.The paper-based expense records may get lost in case of fire accidents, flood etc.	9. PROBLEM ROOT / CAUSE RC When the digits could not be recognized correctly. When the transactions are not successful. When the elder people unable to understand the smaller handwritten digits.When the paper based expense tracker records are subjected to fire accident, flood, etc.	7. BEHAVIOR <small>• ITS INTENSITY</small> BE They may keep a temporary note on their mobile.He/She will tell the other persons to remember the expense they do while calculating the expenses they consider only on the expenses that are single time and huge and leave the rest	Focus on PR, tap into BE, understand RC
Identify strong TR & EM	3. TRIGGERS TO ACT TR This application can create awareness among common people about their income and expenses.It Reduces time rather than entering details manually.	10. YOUR SOLUTION SL The application should be able to generate reports of their spending and notify users if they have exceeded their budget. This application can create awareness among common people about finance and stuffs.This application also helps user to be financially responsible.	8. CHANNELS of BEHAVIOR CH ONLINE Download statements from bank and pay monthly installment	Extract online & offline CH of BE
	4. EMOTIONS <small>BEFORE / AFTER</small> EM Frustration, Confusion, Inadequate > Boost , Feeling smart , Be an example for others		 OFFLINE Using spreadsheets and notes for financial management	

CHAPTER 4

REQUIREMENT ANALYSIS

4.1 Functional requirement

Following are the functional requirements of the proposed solution.

FR No.	Functional Requirement (Epic)	Sub Requirement (Story / Sub-Task)
FR-1	User Registration	Form for collecting details
FR-2	Login	Enter username and password
FR-3	Calendar	Personal expense tracker application must allow user to add the data to their expenses.
FR-4	Expense Tracker	This application should graphically represent the expense in the form of report.
FR-5	Report Generation	Graphical representation of report must be generated.
FR-6	Category	This application shall allow users to add categories of their expenses.

4.2 Non-Functional requirement

Following are the non-functional requirements of the proposed solution.

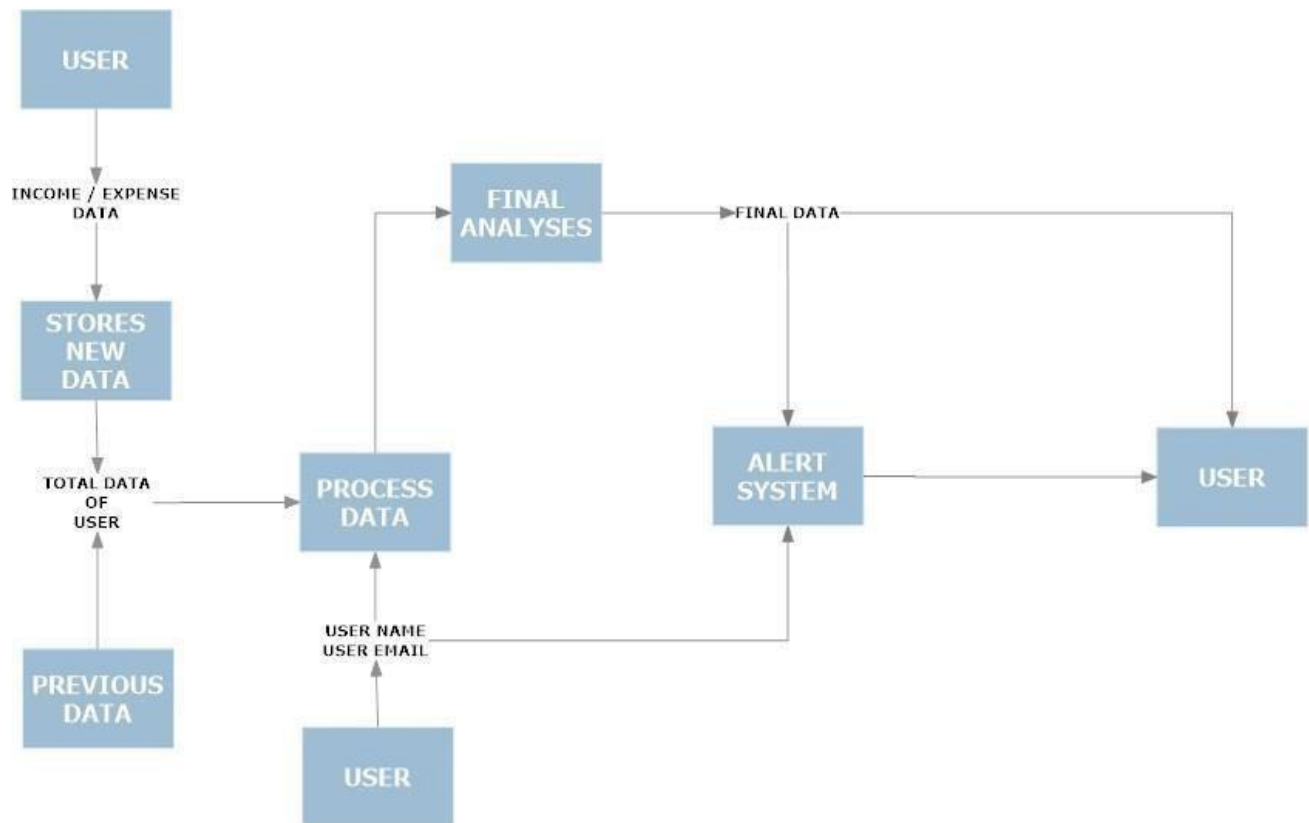
NFR No.	Non-Functional Requirement	Description
NFR-1	Usability	Helps to keep an accurate record of your income and expenses.
NFR-2	Security	Budget tracking apps are considered very safe from those who commit Cyber Crimes.
NFR-3	Reliability	Each data record is stored on a well built efficient database schema. There is no risk of data loss.
NFR-4	Performance	The types of expense are categories along with an option. Throughput of the system is increased due to light weight database support.
NFR-5	Availability	The application must have a 100% up-time.
NFR-6	Scalability	The ability to appropriately handle increasing demands.

CHAPTER 5

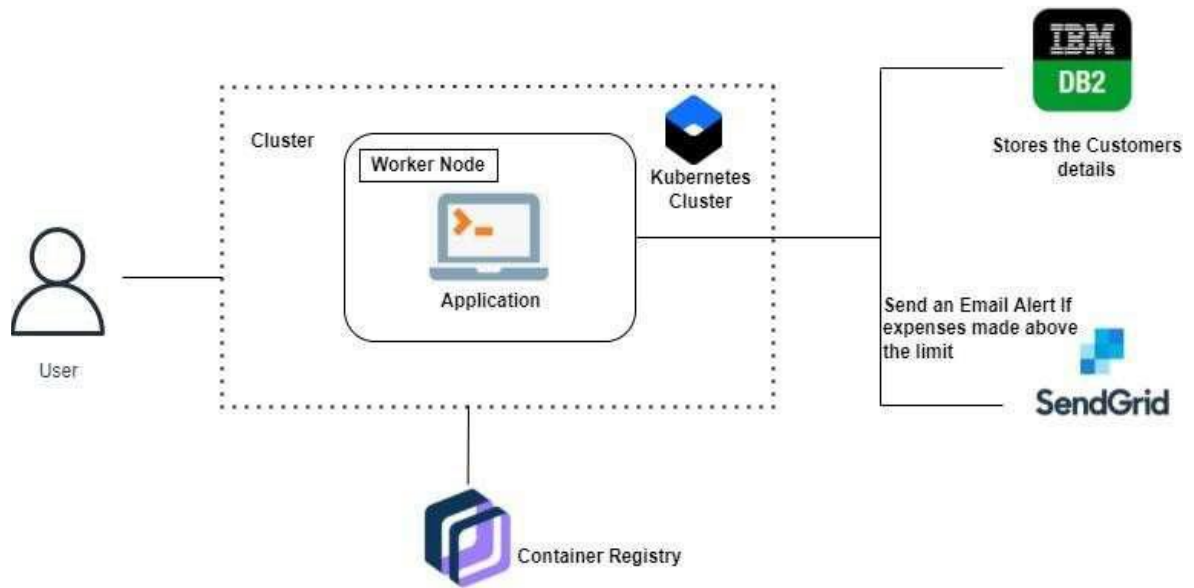
PROJECT DESIGN

5.1 Data Flow Diagrams

A Data Flow Diagram (DFD) is a traditional visual representation of the information flows within a system. A neat and clear DFD can depict the right amount of the system requirement graphically. It shows how data enters and leaves the system, what changes the information, and where data is stored.



5.2 Solution & Technical Architecture



5.3 User Stories

Use the below template to list all the user stories for the product.

User Type	Functional Requirement (Epic)	User Story Number	User Story / Task	Acceptance criteria	Priority
Customer (Mobile user & web user)	Registration	USN-1	As a user, I can register for the application by entering my email, password, and confirming my password.	I can access my account / dashboard	High
		USN-2	As a user, I will receive confirmation email once I have registered for the application	I can receive confirmation email & click confirm	High
		USN- 3	As a user, I can register for the	I can register & access the dashboard	Low

			application through Facebook	with Facebook Login	
	Login	USN - 4	As a user, I can log into the application by entering email & password	I can access the application	High
	Dashboard	USN - 5	As a user I can enter my income and expenditure details.	I can view my daily expenses	High
Customer Care Executive		USN - 6	As a customer care executive. I can solve the log in issues and other issues of the application.	I can provide support or solution at any time 24*7	Medium
Administrator	Application	USN - 7	As an administrator I can upgrade or update the application. customers and users of the application	I can fix the bug which arises for the customers and users of the application.	Medium

CHAPTER 6

PROJECT PLANNING & SCHEDULING

6.1 Sprint Planning & Estimation

Sprint	Functional Requirement (Epic)	User Story Number	User Story / Task	Story Points	Priority	Team Members
Sprint-1	Registration,Login	USN-1	As an User, I can register for the application by entering my email, password, and confirming my password and I can log into the application by entering email and password.	2	High	Shanmuga Parimalam R Rajesh V
Sprint-1	Dashboard	USN-2	As an user, I can Login to the Application by entering correct login credentials and I will be able to Access My dashboard.	2	High	Mukilan M Sasson Taffwin Moses S
Sprint-2	Work Space,Charts	USN-3	There will be a work space for the register user for personal expense tracking and creating various graphs and statistics of user data.	2	Medium	Mukilan M Sasson Taffwin Moses S
Sprint-2	Connecting to IBM DB2	USN-4	Linking the database with dashboard and making dashboard more interactive with the JavaScript	2	High	Shanmuga Parimalam R Rajesh V
Sprint-3	Watson Assistant, Send Grid	USN-5	Creating Chatbot for the expense tracking and for clarifying user's query and using sendgrid to send mail to the user about their expenses	2	High	Shanmuga Parimalam R Rajesh V
Sprint-4	Docker,Cloud Registry	USN-6	Creating image of website using docker and uploading the docker image to IBM Cloud Registry.	2	High	Mukilan M Sasson Taffwin Moses S

6.2 Sprint Delivery Schedule

Sprint	Total Story Points	Duration	Sprint Start Date	Sprint End Date (Planned)	Story Points Completed (as on Planned End Date)	Sprint Release Date (Actual)
Sprint-1	20	7 Days	24 Oct 2022	30 Oct 2022	20	30 Oct 2022
Sprint-2	20	7 Days	31 Oct 2022	06 Nov 2022	18	06 Nov 2022
Sprint-3	20	7 Days	07 Nov 2022	14 Nov 2022	15	14 Nov 2022
Sprint-4	20	7 Days	14 Nov 2022	21 Nov 2022	19	21 Nov 2022

CHAPTER 7

Coding And Solutioning

7.1. Features

Feature 1: Add Expense

Feature 2: Update expense

Feature 3: Delete Expense

Feature 4: Set Limit

Feature 5: Send Alert Emails to users

7.2. Other Features:

Track your expenses anywhere, anytime. Seamlessly manage your money and budget without any financial paperwork. Just click and submit your invoices and expenditures. Access, submit, and approve invoices irrespective of time and location. Avoid data loss by scanning your tickets and bills and saving in the app. Approval of bills and expenditures in real-time and get notified instantly. Quick settlement of claims and reduced human errors with an automated and streamlined billing process.

Codes:

```
import OS
import calendar
from flask import request, session
from flask session import Session
from sql alchemy import create engine
```

```
from sql alchemy.orm import scoped_session, sessionmaker
from datetime import datetime
from helpers import convertSQLToDict
```

```
# Create engine object to manage connections to DB, and scoped session to
separate user interactions with DB
engine = create_engine(os.getenv("DATABASE_URL"))
db = scoped_session(sessionmaker(bind=engine))
```

```
# Add expense(s) to the users expense records
# There are two entry points for this: 1) 'addexpenses' route and 2) 'index'
route. #1 allows many expenses whereas #2 only allows 1 expense per POST.
def addExpenses(formData, userID):
```

```
    expenses = []
    expense = {"description": None, "category": None,
               "date": None, "amount": None, "payer": None}
```

```
    # Check if the user is submitting via 'addexpenses' or 'index' route - this
determines if a user is adding 1 or potentially many expenses in a single
POST
```

```
    if "." not in formData[0][0]:
        for key, value in formData:
            # Add to dictionary
            expense[key] = value.strip()
```

```
    # Convert the amount from string to float for the DB
    expense["amount"] = float(expense["amount"])
```

```
    # Add dictionary to list (to comply with design/standard of
expensed.html)
    expenses.append(expense)
```

```
    # User is submitting via 'addexpenses' route
```

```

else:
    counter = 0
    for key, value in formData:
        # Keys are numbered by default in HTML form. Remove those
        # numbers so we can use the HTML element names as keys for the dictionary.
        cleanKey = key.split(".")

        # Add to dictionary
        expense[cleanKey[0]] = value.strip()

        # Every 5 loops add the expense to the list of expenses (because there
        # are 5 fields for an expense record)
        counter += 1
        if counter % 5 == 0:
            # Store the amount as a float
            expense["amount"] = float(expense["amount"])

            # Add dictionary to list
            expenses.append(expense.copy())

    # Insert expenses into DB
    for expense in expenses:
        now = datetime.now().strftime("%m/%d/%Y %H:%M:%S")
        db.execute("INSERT INTO expenses (description, category,
        expenseDate, amount, payer, submitTime, user_id) VALUES (:description,
        :category, :expenseDate, :amount, :payer, :submitTime, :usersID)",
        {"description": expense["description"], "category":
        expense["category"], "expenseDate": expense["date"], "amount":
        expense["amount"], "payer": expense["payer"], "submitTime": now,
        "usersID": userID})
        db.commit()

    return expenses

```

```
# Get and return the users lifetime expense history
def getHistory(userID):
    results = db.execute("SELECT description, category, expenseDate AS
date, payer, amount, submitTime FROM expenses WHERE user_id =
:usersID ORDER BY id ASC",
                        {"usersID": userID}).fetchall()
```

```
    history = convertSQLToDict(results)
```

```
    return history
```

```
# Get and return an existing expense record with ID from the DB
def getExpense(formData, userID):
    expense = {"description": None, "category": None,
               "date": None, "amount": None, "payer": None, "submitTime":
None, "id": None}
```

```
    expense["description"] = formData.get("oldDescription").strip()
```

```
    expense["category"] = formData.get("oldCategory").strip()
```

```
    expense["date"] = formData.get("oldDate").strip()
```

```
    expense["amount"] = formData.get("oldAmount").strip()
```

```
    expense["payer"] = formData.get("oldPayer").strip()
```

```
    expense["submitTime"] = formData.get("submitTime").strip()
```

```
# Remove dollar sign and comma from the old expense so we can convert
to float for the DB
```

```
    expense["amount"] = float(
```

```
        expense["amount"].replace("$", "").replace(",", ""))
```

```
# Query the DB for the expense unique identifier
```

```
    expenseID = db.execute("SELECT id FROM expenses WHERE user_id =
:usersID AND description = :oldDescription AND category = :oldCategory
```

```

AND expenseDate = :oldDate AND amount = :oldAmount AND payer =
:oldPayer AND submitTime = :oldSubmitTime",
        {"usersID": userID, "oldDescription":
expense["description"], "oldCategory": expense["category"], "oldDate":
expense["date"], "oldAmount": expense["amount"], "oldPayer":
expense["payer"], "oldSubmitTime": expense["submitTime"]}).fetchone()

```

```

# Make sure a record was found for the expense otherwise set as None
if expenseID:
    expense["id"] = expenseID[0]
else:
    expense["id"] = None

```

```

return expense

```

Delete an existing expense record for the user

```

def deleteExpense(expense, userID):
    result = db.execute("DELETE FROM expenses WHERE user_id =
:usersID AND id = :oldExpenseID",
        {"usersID": userID, "oldExpenseID": expense["id"]})
    db.commit()

    return result

```

Update an existing expense record for the user

```

def updateExpense(oldExpense, formData, userID):
    expense = {"description": None, "category": None,
        "date": None, "amount": None, "payer": None}
    expense["description"] = formData.get("description").strip()
    expense["category"] = formData.get("category").strip()
    expense["date"] = formData.get("date").strip()
    expense["amount"] = formData.get("amount").strip()

```

```

expense["payer"] = formData.get("payer").strip()

# Convert the amount from string to float for the DB
expense["amount"] = float(expense["amount"])

# Make sure the user actually is submitting changes and not saving the
existing expense again
hasChanges = False
for key, value in oldExpense.items():
    # Exit the loop when reaching submitTime since that is not something
the user provides in the form for a new expense
    if key == "submitTime":
        break
    else:
        if oldExpense[key] != expense[key]:
            hasChanges = True
            break
if hasChanges is False:
    return None

# Update the existing record
now = datetime.now().strftime("%m/%d/%Y %H:%M:%S")
result = db.execute("UPDATE expenses SET description =
:newDescription, category = :newCategory, expenseDate = :newDate,
amount = :newAmount, payer = :newPayer, submitTime = :newSubmitTime
WHERE id = :existingExpenseID AND user_id = :usersID",
    {"newDescription": expense["description"], "newCategory":
expense["category"], "newDate": expense["date"], "newAmount":
expense["amount"], "newPayer": expense["payer"], "newSubmitTime": now,
"existingExpenseID": oldExpense["id"], "usersID": userID}).rowcount
db.commit()

# Make sure result is not empty (indicating it could not update the expense)
if result:

```

```
    # Add dictionary to list (to comply with design/standard of
    expensed.html)
```

```
    expenses = []
    expenses.append(expense)
    return expenses
```

```
else:
```

```
    return None
```

```
import os
```

```
import calendar
```

```
import copy
```

```
import expenze_expenses
```

```
import expenze_dashboard
```

```
import expenze_categories
```

```
import expenze_budgets
```

```
from flask import request, session
```

```
from flask_session import Session
```

```
from sqlalchemy import create_engine
```

```
from sqlalchemy.orm import scoped_session, sessionmaker
```

```
from helpers import convertSQLToDict
```

```
from datetime import datetime
```

```
# Create engine object to manage connections to DB, and scoped session to
separate user interactions with DB
```

```
engine = create_engine(os.getenv("DATABASE_URL"))
```

```
db = scoped_session(sessionmaker(bind=engine))
```

```
# Generates data needed for the budget report by looping through each
budget and adding expense history where categories match between budgets
and expenses
```

```
# TODO: This data/reporting becomes less beneficial when users have
multiple budgets that have the same categories checked because 1 expense
```


with 'Category A' will be associated with for example 3 budgets that have 'Category A' checked

```
def generateBudgetsReport(userID, year=None):
    # Create data structure to hold users category spending data
    budgetsReport = []

    # Default to getting current years budgets
    if not year:
        year = datetime.now().year

    # Get every budgets spent/remaining for the user
    budgetsReport = expenze_dashboard.getBudgets(userID, year)

    # Loop through the budgets and add a new key/value pair to hold expense
    details per budget
    if budgetsReport:
        for record in budgetsReport:
            budgetID = expenze_budgets.getBudgetID(record["name"], userID)
            results = db.execute("SELECT expenses.description,
expenses.category, expenses.expenseDate, expenses.payer, expenses.amount
FROM expenses WHERE user_id = :usersID AND date_part('year',
date(expensedate)) = :year AND category IN (SELECT categories.name
FROM budgetcategories INNER JOIN categories on
budgetcategories.category_id = categories.id WHERE
budgetcategories.budgets_id = :budgetID)",
                                {"usersID": userID, "year": year, "budgetID":
budgetID}).fetchall()
            expenseDetails = convertSQLToDict(results)
            record["expenses"] = expenseDetails

    return budgetsReport

# Generates data needed for the monthly spending report
```

```

def generateMonthlyReport(userID, year=None):

    # Default to getting current years reports
    if not year:
        year = datetime.now().year

    # Create data structure to hold users monthly spending data for the chart
    (monthly summed data)
    spending_month_chart = expenze_dashboard.getMonthlySpending(userID,
year)

    # Get the spending data from DB for the table (individual expenses per
month)
    results = db.execute(
        "SELECT description, category, expensedate, amount, payer FROM
expenses WHERE user_id = :usersID AND date_part('year',
date(expensedate)) = :year ORDER BY id ASC", {"usersID": userID, "year":
year}).fetchall()
    spending_month_table = convertSQLToDict(results)

    # Combine both data points (chart and table) into a single data structure
    monthlyReport = {"chart": spending_month_chart,
        "table": spending_month_table}

    return monthlyReport


# Generates data needed for the spending trends report
def generateSpendingTrendsReport(userID, year=None):

    # Default to getting current years reports
    if not year:
        year = datetime.now().year

```

```
# Get chart data for spending trends
spending_trends_chart = expenze_dashboard.getSpendingTrends(userID,
year)
```

```
# Data structure for spending trends table
categories = []
category = {"name": None, "expenseMonth": 0,
            "expenseCount": 0, "amount": 0}
spending_trends_table = {
    "January": [],
    "February": [],
    "March": [],
    "April": [],
    "May": [],
    "June": [],
    "July": [],
    "August": [],
    "September": [],
    "October": [],
    "November": [],
    "December": []
}
```

```
# Get all of the users categories first (doesn't include old categories the
user deleted but are still tracked in Expenses)
```

```
categories_active = expenze_categories.getSpendCategories(userID)
```

```
# Get any categories that are in expenses but no longer exist as a selectable
category for the user (because they deleted the category)
```

```
categories_inactive =
expenze_categories.getSpendCategories_Inactive(userID)
```

```
# First fill using the users current categories, and then inactive categories
from Expenses
```

```
for activeCategory in categories_active:
    category["name"] = activeCategory["name"]
    categories.append(category.copy())
```

```
for inactiveCategory in categories_inactive:
    category["name"] = inactiveCategory["category"]
    categories.append(category.copy())
```

Place a deep copy of the categories into each month (need deep copy here because every category may have unique spend data month to month. TODO: optimize this for memory/performance later)

```
for month in spending_trends_table.keys():
    spending_trends_table[month] = copy.deepcopy(categories)
```

Get expense data for each category by month (retrieves the total amount of expenses per category by month, and the total count of expenses per category by month. Assumes there is at least 1 expense for the category)

```
results = db.execute(
    "SELECT date_part('month', date(expensedate)) AS
monthofcategoryexpense, category AS name, COUNT(category) AS count,
SUM(amount) AS amount FROM expenses WHERE user_id = :userID
AND date_part('year', date(expensedate)) = :year GROUP BY
date_part('month', date(expensedate)), category ORDER BY
COUNT(category) DESC",
    {"userID": userID, "year": year}).fetchall()
```

```
spending_trends_table_query = convertSQLToDict(results)
```

Loop thru each monthly category expense from above DB query and update the data structure that holds all monthly category expenses

```
for categoryExpense in spending_trends_table_query:
    # Get the key (month) for the data structure
    monthOfExpense = calendar.month_name[int(
        categoryExpense["monthofcategoryexpense"])]
```

Traverse the data structure: 1) go to the dict month based on the category expense date, 2) loop thru each dict category until a match in name occurs with the expense, 3) update the dict month/amount/count properties to match the DB record

```
for category in spending_trends_table[monthOfExpense]:
    if category["name"] == categoryExpense["name"]:
        category["expenseMonth"] =
categoryExpense["monthofcategoryexpense"]
        category["expenseCount"] = categoryExpense["count"]
        category["amount"] = categoryExpense["amount"]
        break
    else:
        continue
```

Calculates and stores the amount spent per category for the table (note: can't get this to work in jinja with the spending_trends_table dict because of how jinja scopes variables. TODO: rethink data-structure to combine these)

```
numberOfCategories = len(categories)
categoryTotal = 0
# Loops through every month per category and sums up the monthly
amounts
for i in range(numberOfCategories):
    for month in spending_trends_table.keys():
        categoryTotal += spending_trends_table[month][i]["amount"]
    categories[i]["amount"] = categoryTotal
    categoryTotal = 0
```

Combine both data points (chart, table, categories) into a single data structure

```
spendingTrendsReport = {"chart": spending_trends_chart,
                        "table": spending_trends_table, "categories": categories}
```

```
return spendingTrendsReport
```

```

# Generates data needed for the payers spending report
def generatePayersReport(userID, year=None):

    # Default to getting current years reports
    if not year:
        year = datetime.now().year

    # First get all of the payers from expenses table (this may include payers
    that don't exist anymore for the user (i.e. deleted the payer and didn't update
    expense records))
    results_payers = db.execute(
        "SELECT payer AS name, SUM(amount) AS amount FROM expenses
        WHERE user_id = :usersID AND date_part('year', date(expensedate)) = :year
        GROUP BY payer ORDER BY amount DESC", {"usersID": userID, "year":
        year}).fetchall()
    payers = convertSQLToDict(results_payers)

    # Now get any payers the user has in their account but haven't expensed
    anything
    results_nonExpensePayers = db.execute(
        "SELECT name FROM payers WHERE user_id = :usersID AND name
        NOT IN (SELECT payer FROM expenses WHERE expenses.user_id =
        :usersID AND date_part('year', date(expensedate)) = :year)", {"usersID":
        userID, "year": year}).fetchall()
    nonExpensePayers = convertSQLToDict(results_nonExpensePayers)

    # Add the non-expense payers to the payers data structure and set their
    amounts to 0
    for payer in nonExpensePayers:
        newPayer = {"name": payer["name"], "amount": 0}
        payers.append(newPayer)

    # Calculate the total paid for all payers combined

```

```

totalPaid = 0
for payer in payers:
    totalPaid = totalPaid + payer["amount"]

# Calculate the % paid per payer and add to the data structure
if totalPaid != 0:
    for payer in payers:
        payer["percentAmount"] = round((payer["amount"] / totalPaid) * 100)

    return payers
else:
    return None

from flask import request, session
from flask_session import Session
from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker
from helpers import convertSQLToDict

# Create engine object to manage connections to DB, and scoped session to
separate user interactions with DB
engine = create_engine(os.getenv("DATABASE_URL"))
db = scoped_session(sessionmaker(bind=engine))

# Gets and return the users spend categories
def getSpendCategories(userID):
    results = db.execute(
        "SELECT categories.name FROM usercategories INNER JOIN
categories ON usercategories.category_id = categories.id WHERE
usercategories.user_id = :usersID",
        {"usersID": userID}).fetchall()

    categories = convertSQLToDict(results)

```

```
return categories
```

```
# Gets and return the users *inactive* spend categories from their expenses  
(e.g. they deleted a category and didn't update their expense records that still  
use the old category name)
```

```
def getSpendCategories_Inactive(userID):
```

```
    results = db.execute(
```

```
        "SELECT category FROM expenses WHERE user_id = :usersID AND  
category NOT IN(SELECT categories.name FROM usercategories INNER  
JOIN categories ON categories.id = usercategories.category_id WHERE  
user_id = :usersID) GROUP BY category",
```

```
        {"usersID": userID}).fetchall()
```

```
    categories = convertSQLToDict(results)
```

```
    return categories
```

```
# Get and return all spend categories from the category library
```

```
def getSpendCategoryLibrary():
```

```
    results = db.execute("SELECT id, name FROM categories").fetchall()
```

```
    convertSQLToDict(results)
```

```
    return categories
```

```
# Get and return the name of a category from the library
```

```
def getSpendCategoryName(categoryID):
```

```
    name = db.execute(
```

```
        "SELECT name FROM categories WHERE id = :categoryID",  
        {"categoryID": categoryID}).fetchone()[0]
```



```
return name
```

```
# Gets and return the users budgets, and for each budget the categories  
they've selected
```

```
def getBudgetsSpendCategories(userID):
```

```
    results = db.execute("SELECT budgets.name AS budgetname,  
categories.id AS categoryid, categories.name AS categoryname FROM  
budgetcategories INNER JOIN budgets on budgetcategories.budgets_id =  
budgets.id INNER JOIN categories on budgetcategories.category_id =  
categories.id WHERE budgets.user_id = :usersID ORDER BY budgets.name,  
categories.name",  
        {"usersID": userID}).fetchall()
```

```
    budgetsWithCategories = convertSQLToDict(results)
```

```
    return budgetsWithCategories
```

```
# Gets and returns the users budgets for a specific category ID
```

```
def getBudgetsFromSpendCategory(categoryID, userID):
```

```
    results = db.execute("SELECT budgets.id AS budgetid, budgets.name AS  
budgetname, categories.id AS categoryid, categories.name AS categoryname  
FROM budgetcategories INNER JOIN budgets on  
budgetcategories.budgets_id = budgets.id INNER JOIN categories on  
budgetcategories.category_id = categories.id WHERE budgets.user_id =  
:usersID AND budgetcategories.category_id = :categoryID ORDER BY  
budgets.name, categories.name", {  
        "usersID": userID, "categoryID": categoryID}).fetchall()
```

```
    budgets = convertSQLToDict(results)
```

```
    return budgets
```

Updates budgets where an old category needs to be replaced with a new one (e.g. renaming a category)

```
def updateSpendCategoriesInBudgets(budgets, oldCategoryID,
newCategoryID):
    for budget in budgets:
        # Update existing budget record with the new category ID
        db.execute("UPDATE budgetcategories SET category_id = :newID
WHERE budgets_id = :budgetID AND category_id = :oldID",
            {"newID": newCategoryID, "budgetID": budget["budgetid"],
"oldID": oldCategoryID})
        db.commit()
```

Updates budgets where a category needs to be deleted

```
def deleteSpendCategoriesInBudgets(budgets, categoryID):
    for budget in budgets:
        # Delete existing budget record with the old category ID
        db.execute("DELETE FROM budgetcategories WHERE budgets_id =
:budgetID AND category_id = :categoryID",
            {"budgetID": budget["budgetid"], "categoryID": categoryID})

    db.commit()
```

Generates a dictionary containing all spend categories and the budgets associated with each category

```
def generateSpendCategoriesWithBudgets(categories, categoryBudgets):
    categoriesWithBudgets = []
```

```
    # Loop through every category
```

```
    for category in categories:
```

```
        # Build a dictionary to hold category ID + Name, and a list holding all
the budgets which have that category selected
```

```

categoryWithBudget = {"name": None, "budgets": []}
categoryWithBudget["name"] = category["name"]

# Insert the budget for the spend category if it exists
for budget in categoryBudgets:
    if category["name"] == budget["categoryname"]:
        categoryWithBudget["budgets"].append(budget["budgetname"])

# Add the completed dict to the list
categoriesWithBudgets.append(categoryWithBudget)

return categoriesWithBudgets

# Checks if the category name exists in the 'library' or 'registrar' (categories
table) - if so, return the ID for it so it can be passed to below add
def existsInLibrary(newName):
    # Query the library for a record that matches the name
    row = db.execute(
        "SELECT * FROM categories WHERE LOWER(name) = :name",
        {"name": newName.lower()}).fetchone()

    if row:
        return True
    else:
        return False

# Get category ID from DB
def getCategoryID(categoryName, userID=None):
    # If no userID is supplied, then it's searching the category library
    if userID is None:
        categoryID = db.execute(

```

```
        "SELECT id FROM categories WHERE LOWER(name) = :name",
        {"name": categoryName.lower()}).fetchone()
```

```
    if not categoryID:
```

```
        return None
```

```
    else:
```

```
        return categoryID["id"]
```

```
# Otherwise search the users selection of categories
```

```
else:
```

```
    categoryID = db.execute(
```

```
        "SELECT categories.id FROM usercategories INNER JOIN
categories ON usercategories.category_id = categories.id WHERE
usercategories.user_id = :userID AND LOWER(categories.name) = :name",
        {"userID": userID, "name": categoryName.lower()}).fetchone()
```

```
    if not categoryID:
```

```
        return None
```

```
    else:
```

```
        return categoryID["id"]
```

```
# Checks if the category name exists in the users selection of categories
(usercategories table) - if so, just return as False?
```

```
def existsForUser(newName, userID):
```

```
    # Query the library for a record that matches the name
```

```
    row = db.execute(
```

```
        "SELECT categories.id FROM usercategories INNER JOIN categories
ON usercategories.category_id = categories.id WHERE
usercategories.user_id = :userID AND LOWER(categories.name) = :name",
        {"userID": userID, "name": newName.lower()}).fetchone()
```

```
    if row:
```

```
        return True
```

```
else:  
    return False
```

```
# Adds a category to the database (but not to any specific users account)
```

```
def addCategory_DB(newName):
```

```
    # Create a new record in categories table
```

```
    categoryID = db.execute(  
        "INSERT INTO categories (name) VALUES (:name) RETURNING id",  
        {"name": newName}).fetchone()[0]
```

```
    db.commit()
```

```
    return categoryID
```

```
# Adds a category to the users account
```

```
def addCategory_User(categoryID, userID):
```

```
    db.execute("INSERT INTO usercategories (user_id, category_id)  
VALUES (:userID, :categoryID)",  
        {"userID": userID, "categoryID": categoryID})
```

```
    db.commit()
```

```
# Deletes a category from the users account
```

```
def deleteCategory_User(categoryID, userID):
```

```
    db.execute("DELETE FROM usercategories WHERE user_id = :userID  
AND category_id = :categoryID",  
        {"userID": userID, "categoryID": categoryID})
```

```
    db.commit()
```

```
# Update just the spend categories of expense records (used for category  
renaming)
```

```
def updateExpenseCategoryNames(oldCategoryName, newCategoryName,
userID):
    db.execute("UPDATE expenses SET category = :newName WHERE
user_id = :usersID AND category = :oldName",
        {"newName": newCategoryName, "usersID": userID, "oldName":
oldCategoryName})
    db.commit()
```

Rename a category

```
def renameCategory(oldCategoryID, newCategoryID, oldCategoryName,
newCategoryName, userID):
```

```
    # Add the renamed category to the users account
```

```
    addCategory_User(newCategoryID, userID)
```

```
    # Delete the old category from their account
```

```
    deleteCategory_User(oldCategoryID, userID)
```

```
    # Update users budgets (if any exist) that are using the old category to the
new one
```

```
    budgets = getBudgetsFromSpendCategory(oldCategoryID, userID)
```

```
    if budgets:
```

```
        updateSpendCategoriesInBudgets(budgets, oldCategoryID,
newCategoryID)
```

```
    # Update users expense records that are using the old category to the new
one
```

```
    updateExpenseCategoryNames(oldCategoryName, newCategoryName,
userID)
```

Delete a category

```
def deleteCategory(categoryID, userID):
```

```
# Get budgets that are currently using the category they want to delete
budgets = getBudgetsFromSpendCategory(categoryID, userID)
```

```
# Delete categories from the users budgets
```

```
if budgets:
```

```
    deleteSpendCategoriesInBudgets(budgets, categoryID)
```

```
# Delete the category from the users account
```

```
deleteCategory_User(categoryID, userID)
```

CHAPTER 8

TESTING

8.1. TESTING:

- Login Page (Functional)
- Login Page (UI)
- Add Expense Page (Functional)

8.2. User Acceptance Testing:

1. Purpose of Document

The purpose of this document is to briefly explain the test coverage and open issues of the [ProductName] project at the time of the release to User Acceptance Testing (UAT).

2. Defect Analysis

This report shows the number of resolved or closed bugs at each severity level, and how they were resolved

Resolution	Severity 1	Severity 2	Severity 3	Severity 4	Subtotal
By Design	10	4	2	8	15
Duplicate	1	0	3	0	4
External	2	3	0	1	6
Fixed	9	2	4	11	20
Not Reproduce	0	0	1	0	1

Skipped	0	0	1	1	2
Won't Fix	0	5	0	1	8
Totals	22	14	11	22	51

3. Test Case Analysis

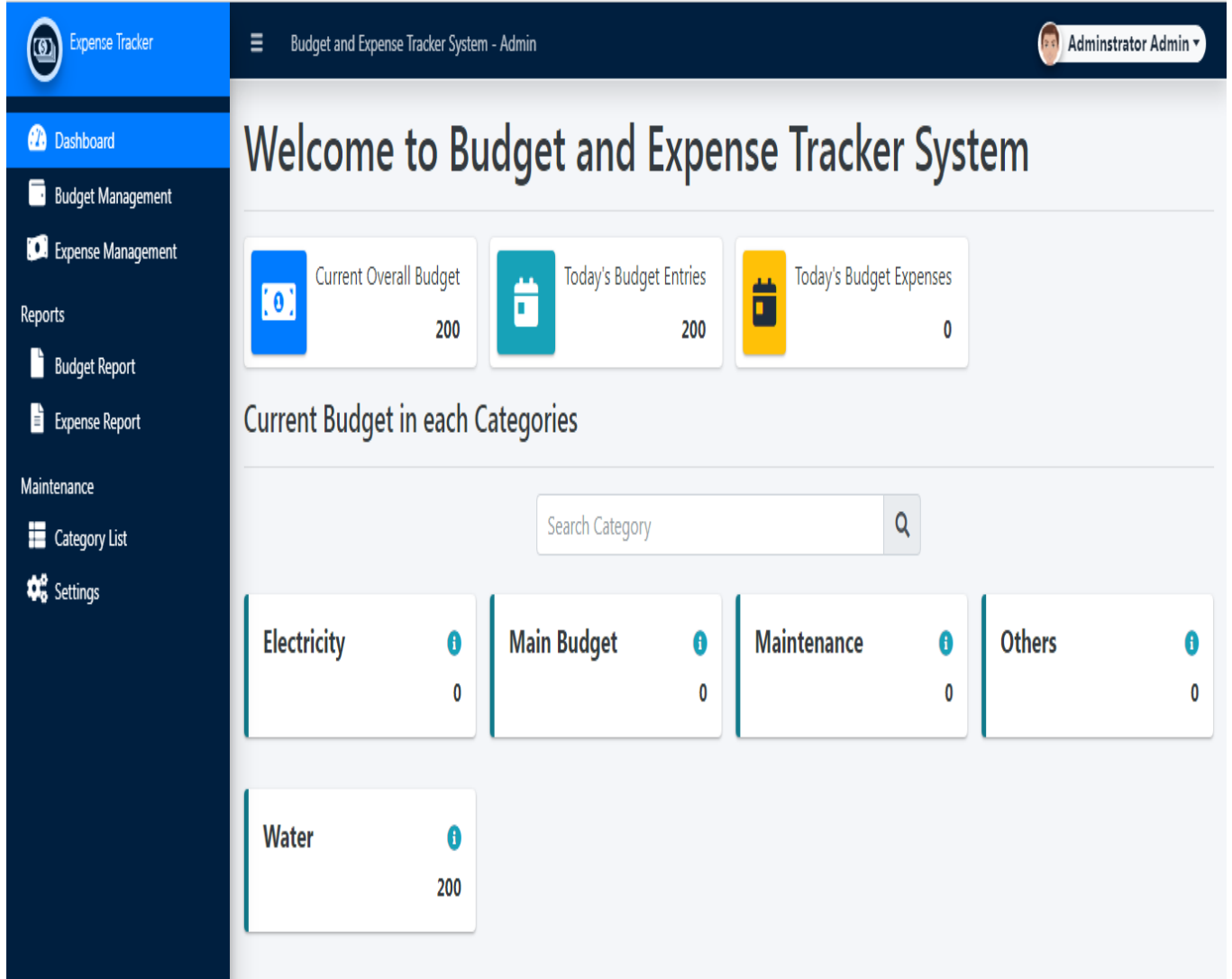
This report shows the number of test cases that have passed, failed, and untested

Section	Total Cases	Not Tested	Fail	Pass
Interface	7	0	0	7
Login	43	0	0	43
Logout	2	0	0	2

CHAPTER 9

RESULTS

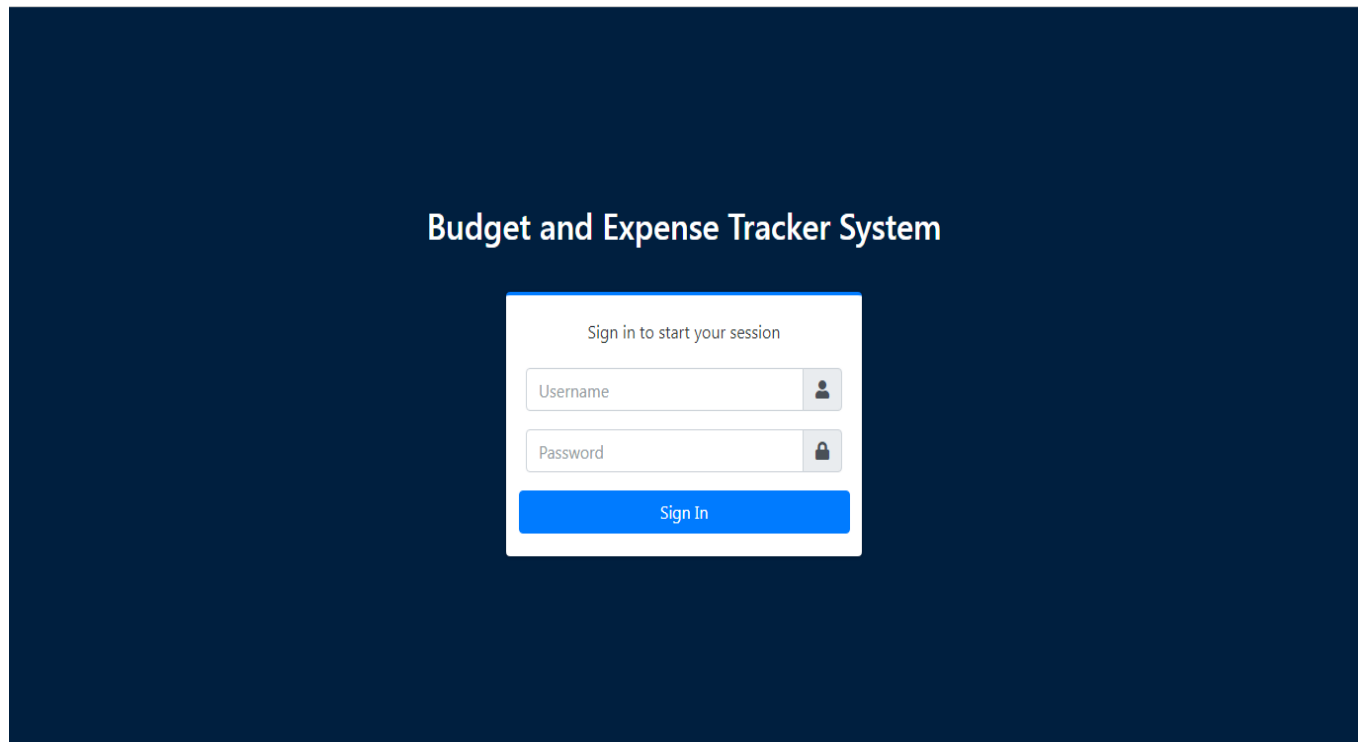
9.1 Home Page



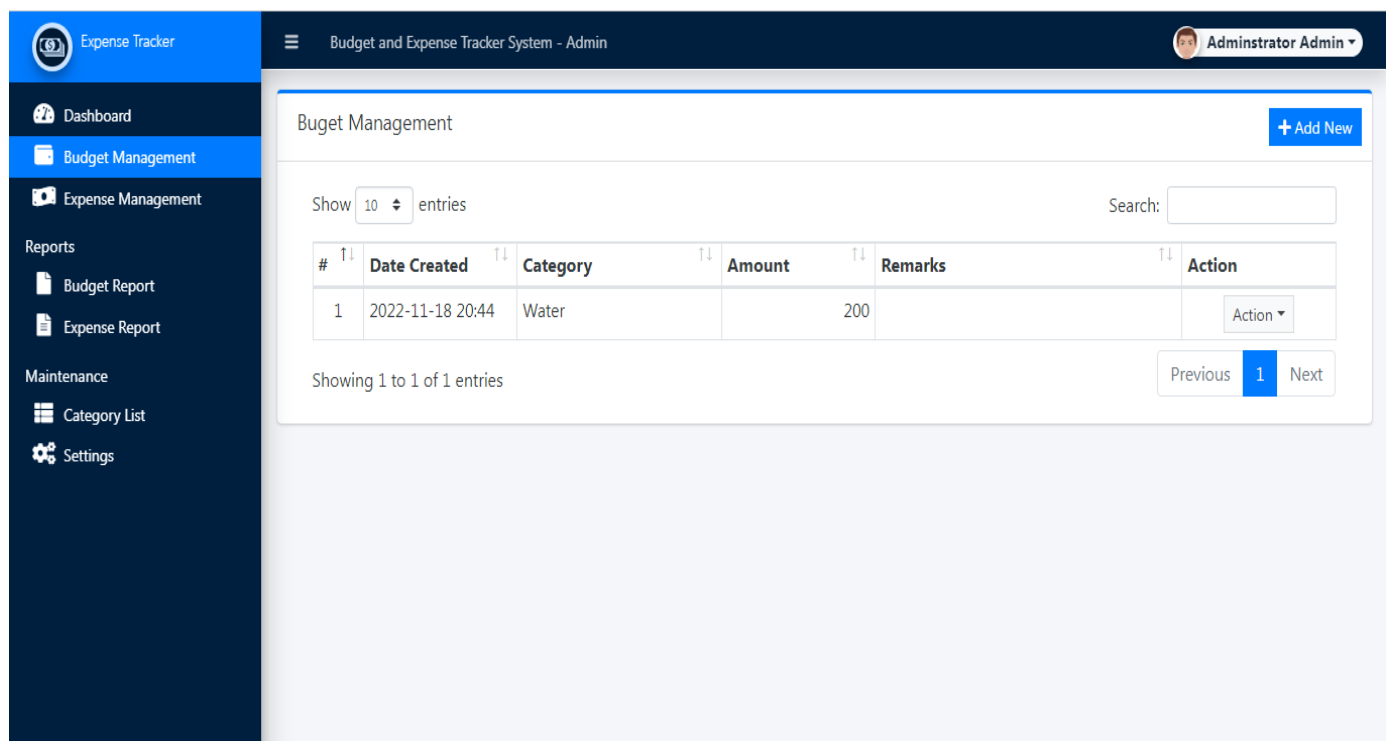
The screenshot displays the home page of the Budget and Expense Tracker System. The interface features a dark blue sidebar on the left with a menu containing 'Expense Tracker', 'Dashboard', 'Budget Management', 'Expense Management', 'Reports' (with sub-items 'Budget Report' and 'Expense Report'), 'Maintenance' (with sub-items 'Category List' and 'Settings'), and a user profile 'Administrator Admin' at the top right. The main content area has a header 'Welcome to Budget and Expense Tracker System' and three summary cards: 'Current Overall Budget' (200), 'Today's Budget Entries' (200), and 'Today's Budget Expenses' (0). Below these is a section titled 'Current Budget in each Categories' with a search bar and five category cards: 'Electricity' (0), 'Main Budget' (0), 'Maintenance' (0), 'Others' (0), and 'Water' (200). Each card includes an information icon.

Category	Budget Value
Current Overall Budget	200
Today's Budget Entries	200
Today's Budget Expenses	0
Electricity	0
Main Budget	0
Maintenance	0
Others	0
Water	200


9.2 Sign in Page



9.3 Budget



9.4 Report Page

Expense Tracker

Dashboard

Budget Management

Expense Management

Reports

Budget Report


Expense Report

Maintenance

Category List

Settings

Budget and Expense Tracker System - Admin

Administrator Admin

Budget Report

Date Start

Date End

11-11-2022

18-11-2022

Filter

Print

Budget and Expense Tracker System

Budget Report

Date Between Nov 11, 2022 and Nov 18, 2022

#	Entry DateTime	Category	Amount	Remarks
1	Nov 18, 2022	Water	200	
Total			200	

CHAPTER 10

ADVANTAGES AND DISADVANTAGES

10.1. ADVANTAGES:

As expending money is easier, we often fall prey to overspending. So, one of the immediate benefit of budgeting and expense tracking is that, it prevents overspending. Budgeting makes us aware of where we spend our money. Once we know this, it then helps us to fix a limit. Budgeting and expense tracking works like a control. It continuously gives us feedback about our spending patterns. This is what is called expense tracking. It makes us aware of how well we are performing on our different budgeted expense heads. Using the Expense Manager, you can easily make month on month comparisons of earning, expenses and spending in a more organized manner.

10.2. DISADVANTAGES:

A con with any system used to track spending is that one may start doing it then taper off until it's forgotten about all together. Yet, this is a risk for any new goal such as trying to lose weight or quit smoking. If a person first makes a budget plan, then places money in savings before spending any new pay period or month, the tracking goal can help. In this way, tracking spending and making sure all receipts are accounted for only needs to be done once or twice a month. Even with constant tracking of one's spending habits, there is no guarantee that financial goals will be met. Although this can be considered to be a con of tracking spending, it could be changed into a pro if one makes up his or her mind to keep trying to properly manage all finances.

CHAPTER 11

CONCLUSION

The new system has overcome most of the limitations of the existing system and works according to the design specification given. The project what we have developed is work more efficient than the other income and expense tracker. The project successfully avoids the manual calculation for avoiding calculating the income and expense per month. The modules are developed with efficient and also in an attractive manner. The developed systems dispense the problem and meet the needs of by providing reliable and comprehensive information. All the requirements projected by the user have been met by the system. The newly developed system consumes less processing time and all the details are updated and processed immediately. Since the screen provides online help messages and is very user friendly, any user will get familiarized with its usage. Module s are designed to be highly flexible so that any failure requirements can be easily added to the modules without facing many problems. The best organizations have a way of tracking and handling these reimbursements. This ideal practice guarantees that the expenses tracked are accurately and in a timely manner.

CHAPTER 12

FUTURE SCOPE

- It will have various options to keep record (for example Food, Travelling Fuel, Salary etc.).
- Automatically it will keep on sending notifications for our daily expenditure.
- In today's busy and expensive life, we are in a great rush to make moneys, but at the end of the month we broke off. As we are unknowingly spending money on title and unwanted things. So, we have come over with the plan to follow our profit.
- Here user can define their own categories for expense type like food, clothing, rent and bills where they have to enter the money that has been spend and likewise can add some data in extra data to indicate the expense.

GitHub Link:

<https://github.com/IBM-EPBL/IBM-Project-8031-1658906680>

Demo Link:

<https://youtu.be/CLFJL5McKS4>