

# Capstone Project Final Submission



CPSC 491-01

Expense Tracker

By

David Harboyan

## Proposal/Abstract

The idea is to create an application where users can visually see their monthly expenses. Users would be able to manually input their transactions to track any purchases they make and categorize these transactions so that they can see what types of things they are spending their money on.

# Table of Contents

<a href="#">Introduction</a>	<a href="#">5</a>
<a href="#">Project Plan</a>	<a href="#">6</a>
<a href="#">Metrics</a>	<a href="#">9</a>
<a href="#">Metric 1 - Velocity</a>	<a href="#">9</a>
<a href="#">Metric 2 - Burndown</a>	<a href="#">11</a>
<a href="#">Metric 3 - Defects</a>	<a href="#">19</a>
<a href="#">Operational and Support Plan</a>	<a href="#">21</a>
<a href="#">Maintenance Plan</a>	<a href="#">22</a>
<a href="#">Technical Sections</a>	<a href="#">23</a>
<a href="#">Requirements Engineering</a>	<a href="#">23</a>
<a href="#">Architecture and Design</a>	<a href="#">27</a>
<a href="#">UX Design</a>	<a href="#">32</a>
<a href="#">Prototyping</a>	<a href="#">38</a>
<a href="#">Implementation Methods</a>	<a href="#">41</a>
<a href="#">Link to the GitHub Code</a>	<a href="#">44</a>
<a href="#">Network Analysis and Map</a>	<a href="#">44</a>
<a href="#">Threat Model</a>	<a href="#">46</a>
<a href="#">Open Source / 3rd Party Components Used</a>	<a href="#">47</a>

<u>Test Plans and Results</u>	<u>48</u>
<u>Unit Tests</u>	<u>48</u>
<u>Performance Test</u>	<u>50</u>
<u>Security Tests</u>	<u>52</u>
<u>Bugs</u>	<u>56</u>
<u>Installation / Setup Guide</u>	<u>60</u>
<u>Run / Operations Books</u>	<u>62</u>
<u>SLIs/SLOs/SLAs</u>	<u>63</u>
<u>User Manuals</u>	<u>65</u>
<u>Landing Page</u>	<u>65</u>
<u>Signup Page</u>	<u>65</u>
<u>Login Page</u>	<u>66</u>
<u>Home Page</u>	<u>66</u>
<u>Conclusion</u>	<u>69</u>
<u>References</u>	<u>71</u>

## Introduction

Creating an application that allows users to track their expenses is incredibly important because of the high levels of inflation and uncertainty of the economy. It's crucial for people to be smart with their money by being aware of where their money is going, and to cut out any expenses that aren't too important. If people are able to follow these practices, they would mitigate the effects that inflation has on their finances.

By creating such an application, we would be able to help users know where their money is going and make it easier for them to plan out a budget. It would also make it easier for users to know when they exceed their budgets so they could adjust their financial goals. Such an application would also create convenience for users because it would have their expenses all on one, easy to access, page.

There are a few apps in this space, such as Intuit Mint and You Need A Budget. They have similar functionalities to each other and this project. Though, the main difference between them and this project, is that they have a feature to include bank accounts and credit card accounts, which is not reasonable for the scope of this project.

As for uniqueness, this project aims to be geared towards the individual user rather than a one-size-fits-all application, unlike most of the apps in this space. It would allow users to customize their user interface and create personalized budgets/financial goals, increasing overall user experience.

# Project Plan

## Plan during CPSC 490

- Gather user requirements and create requirement models
- Research and pick a software architecture plan
  - Design the structure of the application to explain its behavior
  - Design the relational database schema which will be used
- Research what type of user would use this application
  - Research and pick a UX framework
  - Sketch out a general design of each page
  - Create wireframes/mockups
- Create a prototype of the application
  - Goal is to create static pages and meet the design of the wireframes/mockups
  - Functionality won't be the focus, but will be worked on if time permits
- Research tools that could be used
  - Potential frameworks such as React or Tailwind CSS
  - Research coding practices for HTML, CSS, and JavaScript
    - Use any tools that may help with these practices
- Complete the rest of this document within CPSC 490 requirements

Plan during CPSC 491

- Sprint 1 (21 Story Points Committed)
  - Implement account creation (8 Story Points)
  - Implement account login (8 Story Points)
  - Setup and create database (5 Story Points)
- Sprint 2 (21 Story Points Committed)
  - Implement category creation (13 Story Points)
  - Implement category updating (8 Story Points)
- Sprint 3 (15 Story Points Committed)
  - Finish implementing category updating (7 Story Points)
  - Implement category deletion (8 Story Points)
- Sprint 4 (21 Story Points Committed)
  - Implement adding transactions to categories (13 Story Points)
  - Implement updating transactions for categories (8 Story Points)
- Sprint 5 (13 Story Points Committed)
  - Finish implementing transaction updating (5 Story Points)
  - Implement transaction deletion for categories (5 Story Points)
  - Add handling for numeric input (3 Story Points)
- Sprint 6 (13 Story Points Committed)
  - Swap global variables with session storage (5 Story Points)
  - Update user interface for homepage (3 Story Points)
  - Add messages to inform users of invalid input (5 Story Points)
- Sprint 7 (9 Story Points Committed)

- Create unit tests for endpoints (8 Story Points)
- Use linters to fix and clean up code (1 Story Point)
- Future Tasks
  - Implement darkmode
  - Implement responsiveness for more device types such as mobile phones
  - Display date and time of transactions on homepage



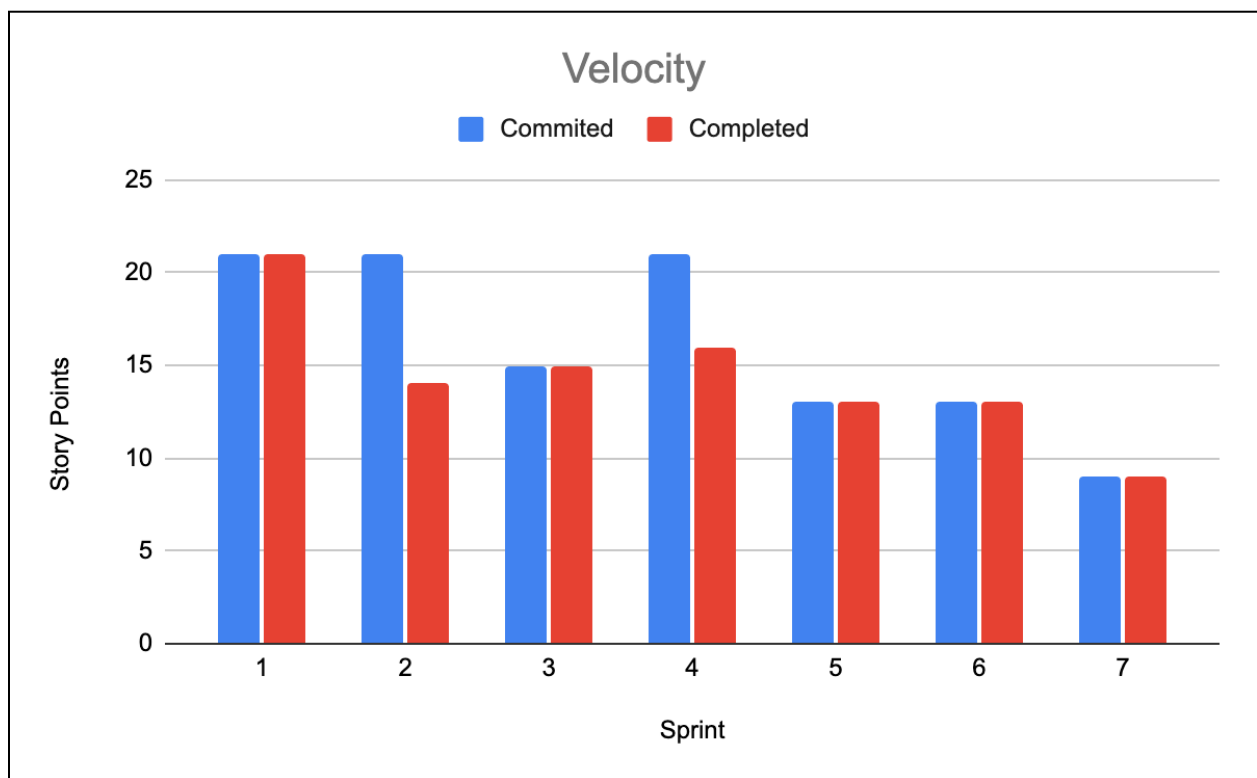
## Metrics

For this project, we will follow Agile principles, and define the length of our sprints as 2 weeks. Now that the length of our sprints have been determined, we can define and measure our metrics with simplicity.

### Metric 1 - Velocity

The first metric will be to track velocity by noting how many story points have been committed and how many story points have been completed during each sprint. We will record this data in a Google Sheets document and have three columns: Sprint, Committed, and Completed. The Sprint column identifies a specific sprint. The Committed column represents the number of story points that were said to be completed during the sprint. The Completed column represents the number of story points that were actually completed during the sprint. From this we will be able to create a double bar graph, where the x-axis represents the sprint, the y-axis represents the number of story points, and it will have two sets of bars: one for committed story points and one for completed story points. Below are the values gathered and the resulting graph for the Velocity metric.

Sprint	Committed	Completed
1	21	21
2	21	14
3	15	15
4	21	16
5	13	13
6	13	13
7	9	9

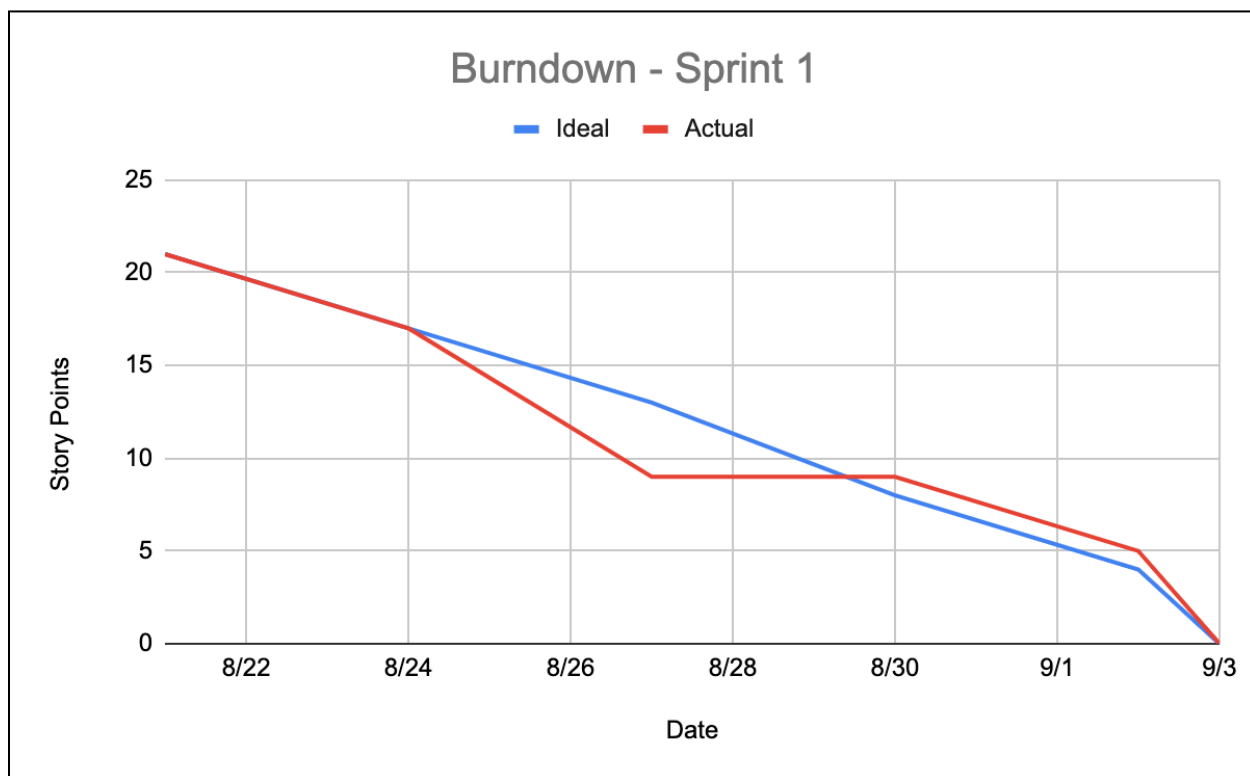


As can be seen with the visual, we completed less story points than what was committed for sprint 2 and 4. This meant we were behind schedule for those sprints. Despite this, we were able to get back on track for the following sprints and everything was completed by the last sprint.

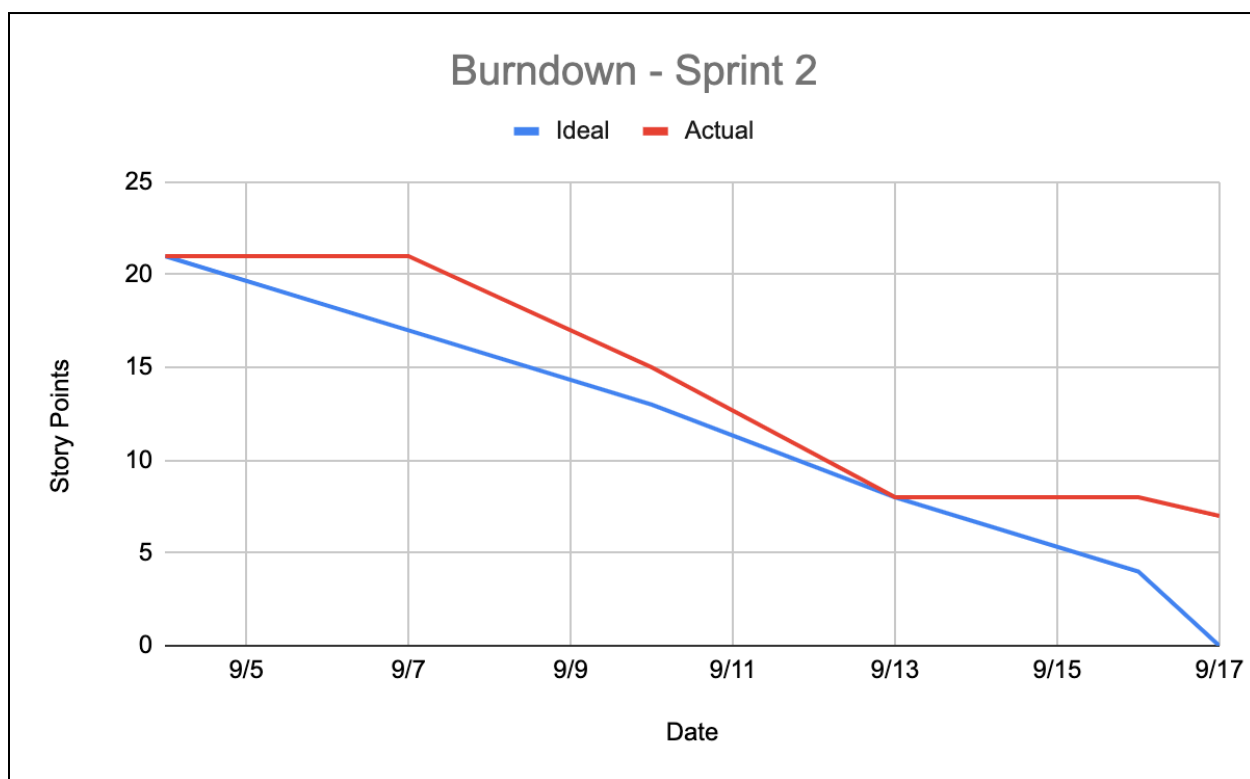
## Metric 2 - Burndown

The second metric will be to track the burndown of each individual sprint. In terms of measurements, we will create an ideal schedule of which story points will be completed and measure our actual progress with remaining story points. We will record these measurements on most days of the sprint, and each sprint will have its own burndown chart. We will store this data in a Google Sheets document and have three columns: Date, Ideal, and Actual. The Date column will specify the day of the sprint. The Ideal column will represent the desired number of remaining story points while the Actual column represents the real number of remaining story points. With these measurements, we will be able to create a double line graph, where the x-axis represents the date during the sprint, the y-axis represents the number of remaining story points, and it will have two sets of lines: one for the ideal number of remaining story points and one for the actual number of remaining story points. Below are the values gathered and the resulting graphs for the Burnout metric. Note that each sprint has its own values and graph.

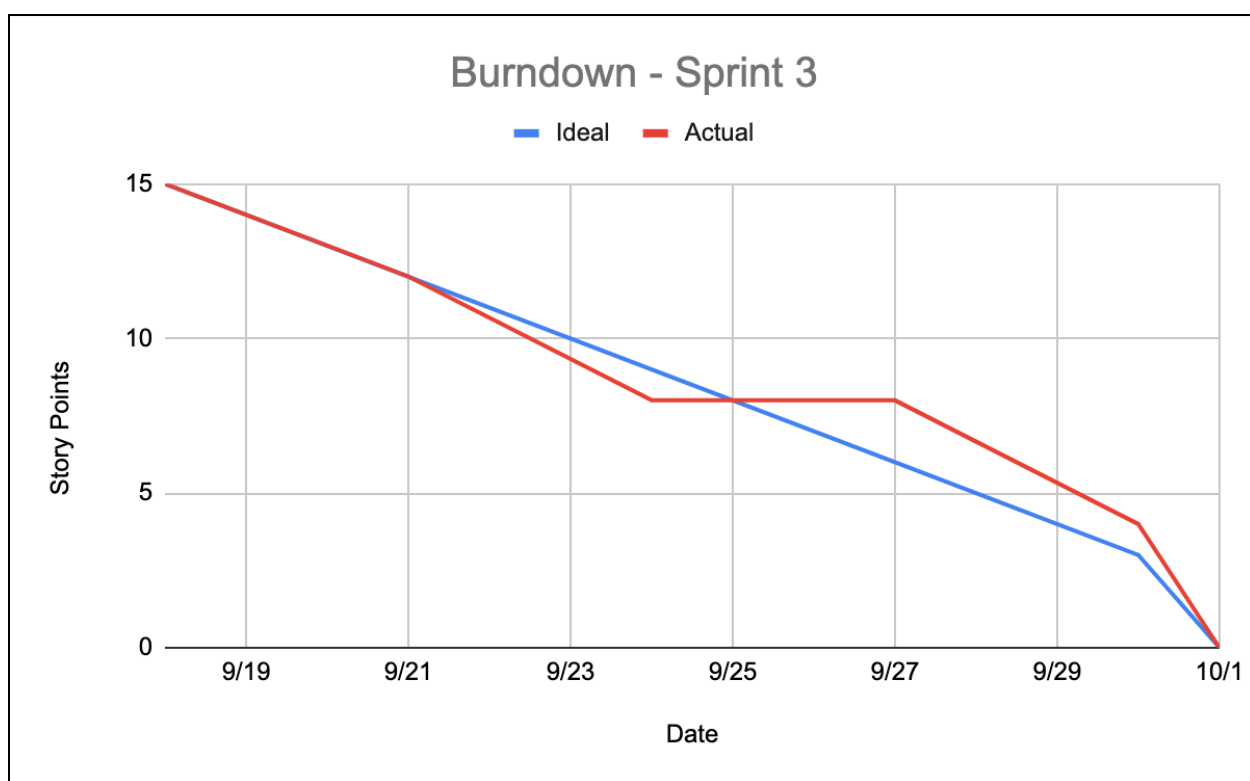
Date	Ideal	Actual
8/21	21	21
8/24	17	17
8/27	13	9
8/30	8	9
9/2	4	5
9/3	0	0



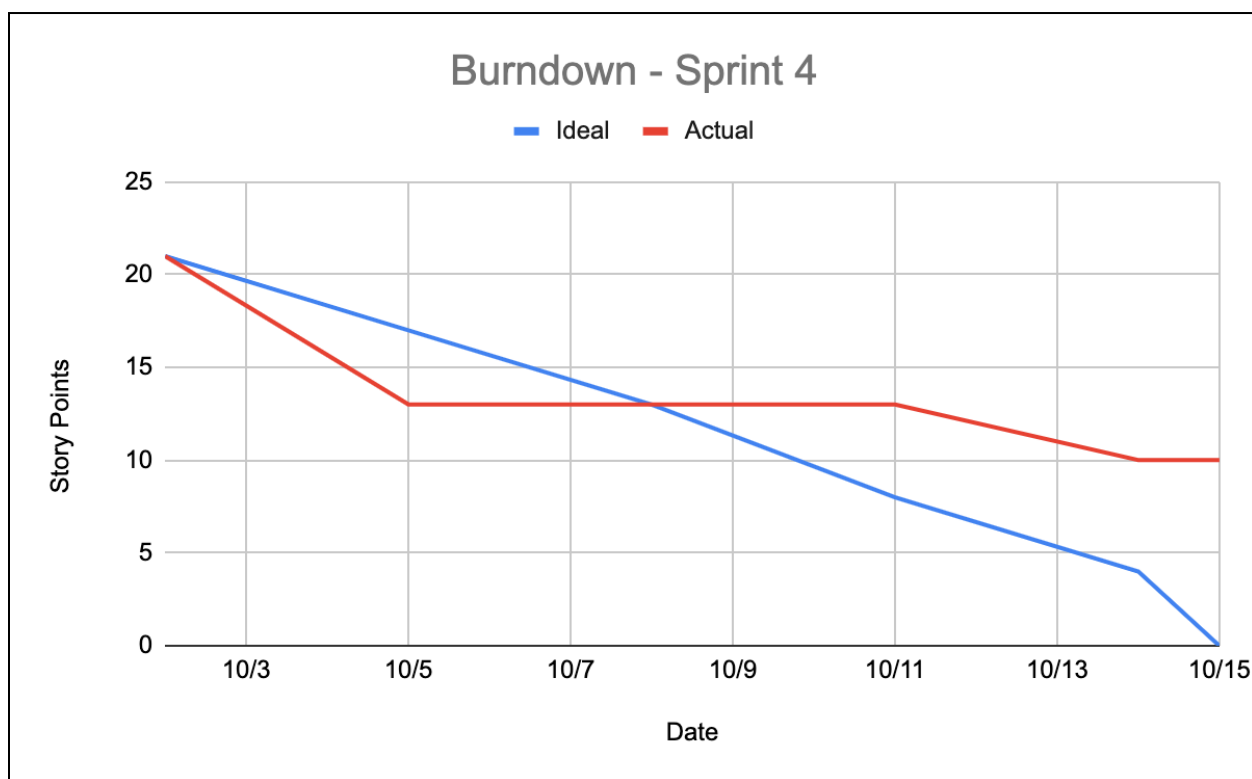
Date	Ideal	Actual
9/4	21	21
9/7	17	21
9/10	13	15
9/13	8	8
9/16	4	8
9/17	0	7



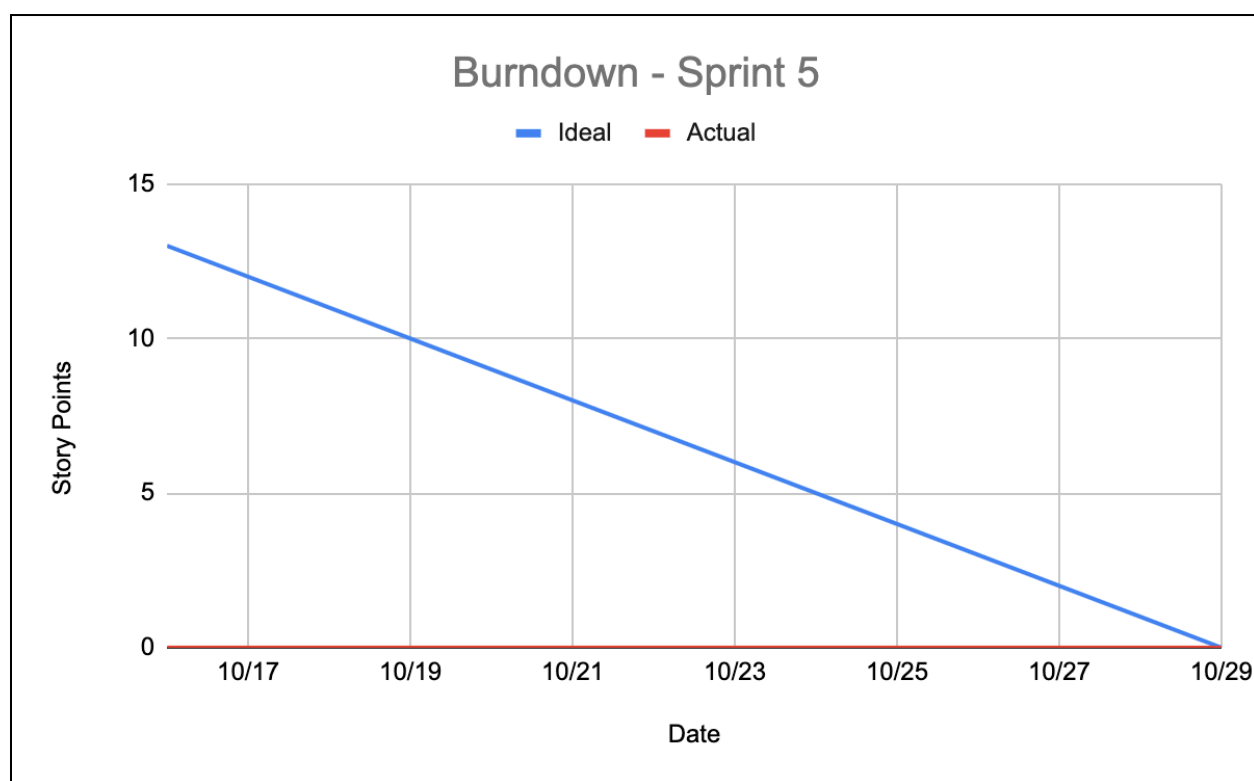
Date	Ideal	Actual
9/18	15	15
9/21	12	12
9/24	9	8
9/27	6	8
9/30	3	4
10/1	0	0



Date	Ideal	Actual
10/2	21	21
10/5	17	13
10/8	13	13
10/11	8	13
10/14	4	10
10/15	0	10

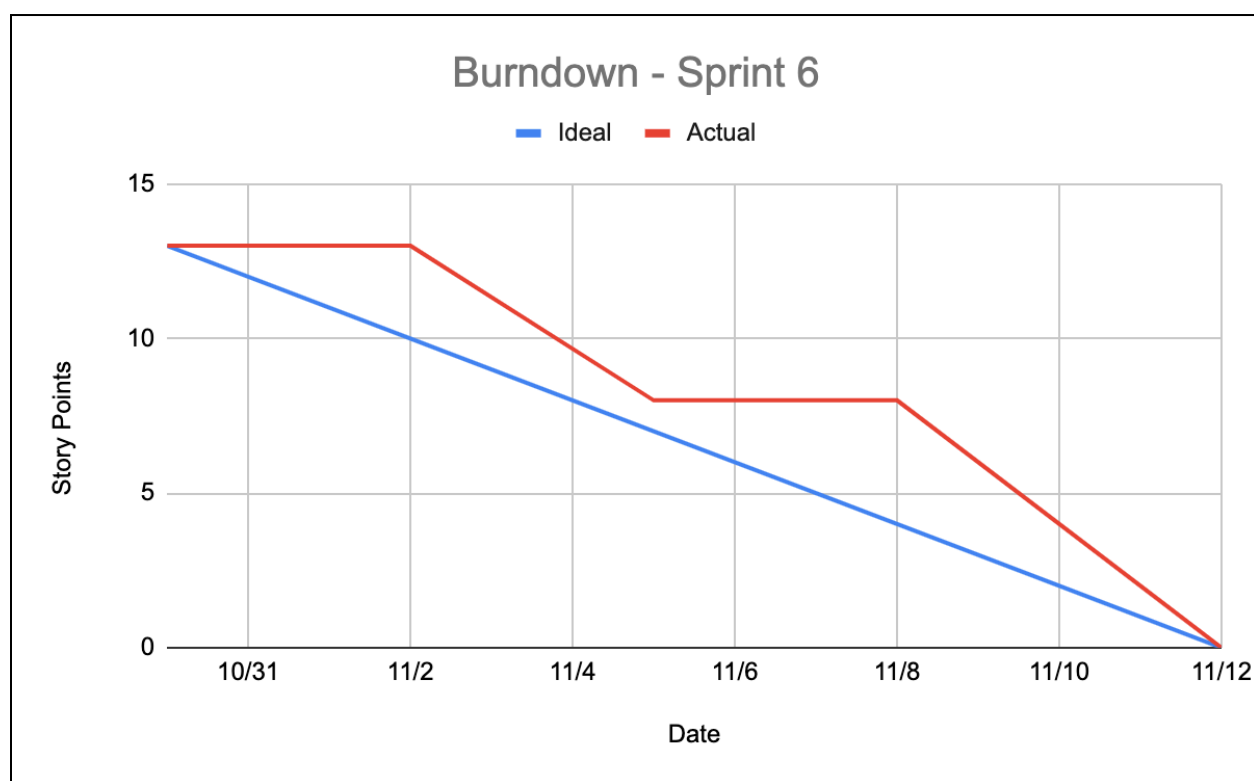


Date	Ideal	Actual
10/16	13	0
10/19	10	0
10/22	7	0
10/25	4	0
10/28	1	0
10/29	0	0

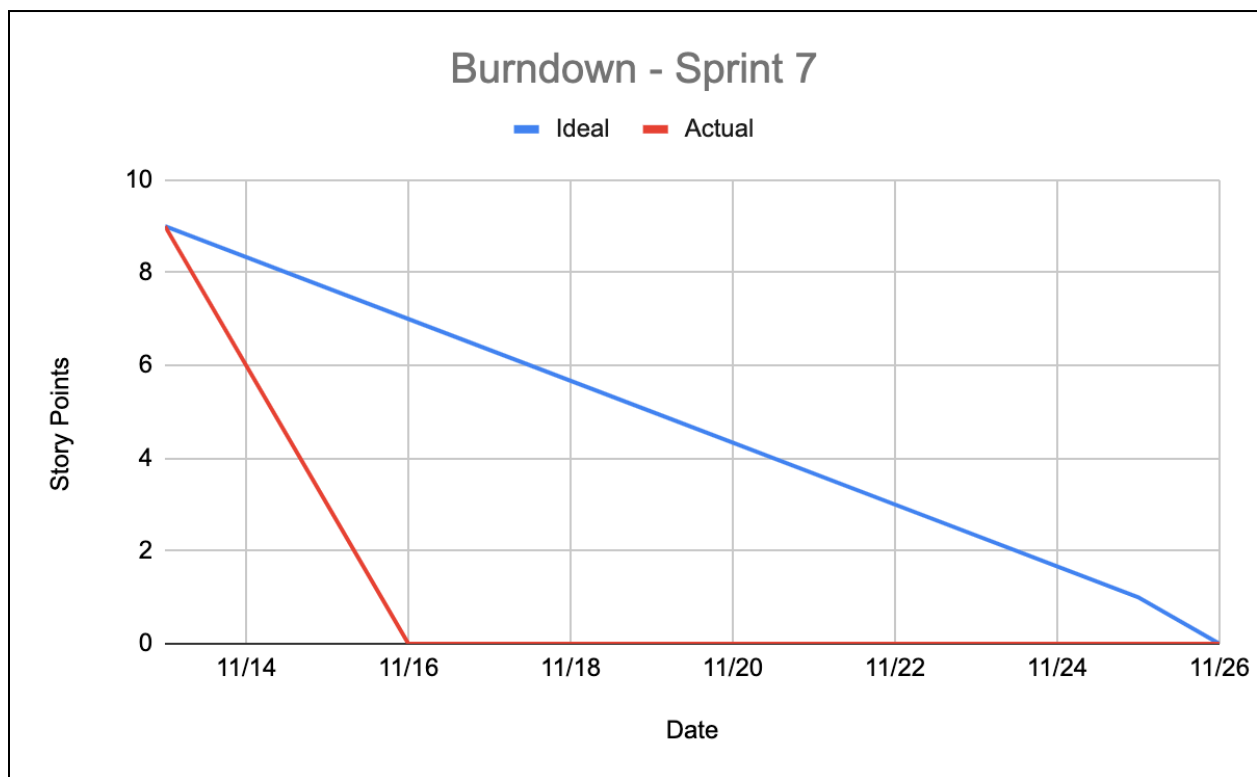




Date	Ideal	Actual
10/30	13	13
11/2	10	13
11/5	7	8
11/8	4	8
11/11	1	2
11/12	0	0



Date	Ideal	Actual
11/13	9	9
11/16	7	0
11/19	5	0
11/22	3	0
11/25	1	0
11/26	0	0

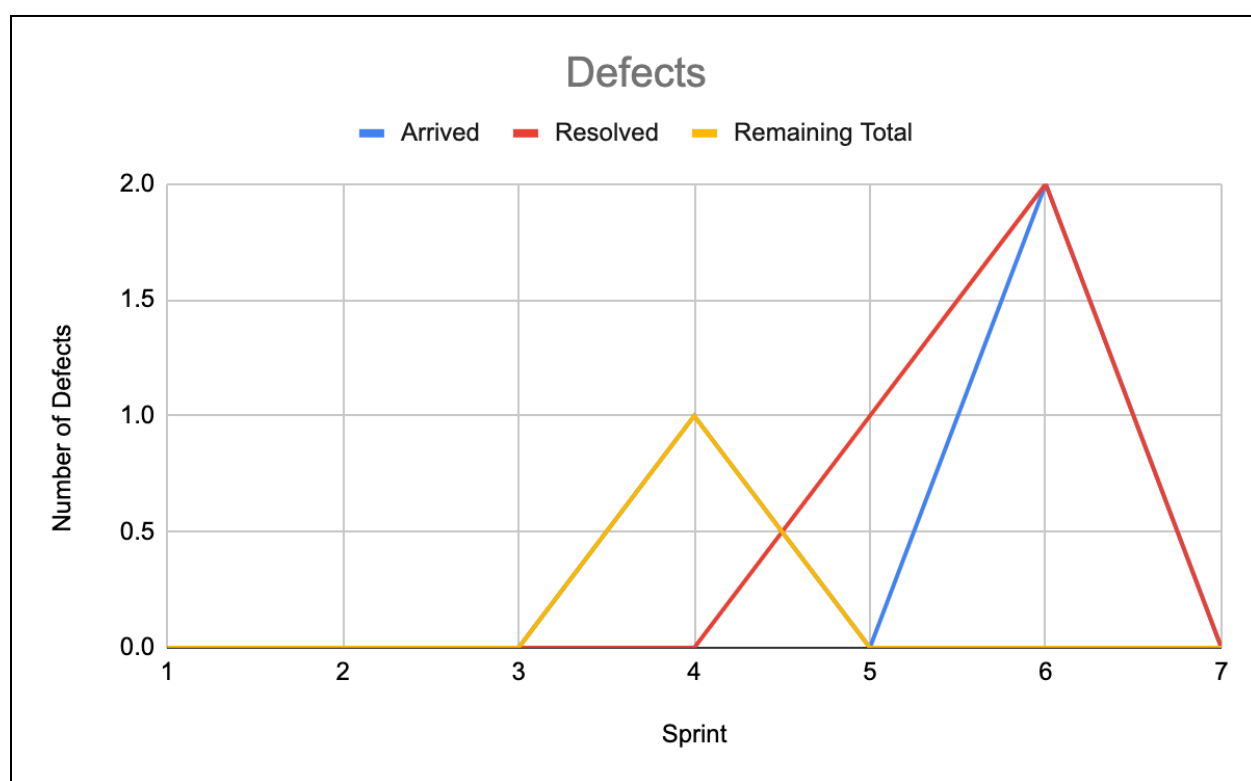


With the results of the Burndown metric in mind, we can see that we were typically following the ideal schedule of completing story points. Though it's worth noting that there were some sprints where we did not complete everything, and some sprints where we'd complete more story points faster than the ideal schedule which is good.

## Metric 3 - Defects

The third metric will be to track any defects that may arise. To do so, we will note the number of defects that arrive and the number of defects that have been resolved during a sprint, as well as the number of total defects that exist in the application. We will record this data in a Google Sheets document and have four columns: Sprint, Arrived, Resolved, and Total Remaining. The Sprint column identifies a specific sprint. The Arrived column represents the amount of defects that have appeared during a sprint while the Resolved column represents the amount of defects that have been solved during a sprint. The Total Remaining column signifies the amount of defects that remain throughout the application. From these measurements, we can construct a triple line graph, where the x-axis represents the sprint, the y-axis represents the number of defects, and it will have three sets of lines: one for the number of arrived defects, one for the number of resolved defects, and one for the number of total remaining defects. Below are the values gathered and the resulting graph for the Defects metric.

Sprint	Arrived	Resolved	Remaining Total
1	0	0	0
2	0	0	0
3	0	0	0
4	1	0	1
5	0	1	0
6	2	2	0
7	0	0	0



Based on the values and the resulting graph for this metric, we notice that there were 3 defects discovered during development. The first defect being found during sprint 4 and resolved during sprint 5, and the other two defects being found and resolved during sprint 6. Also, for the final sprint, the remaining total is 0 which means that there are no more defects.

## Operational and Support Plan

This project is planned to be used by end users and accessed over the Internet. The web application that users will interact with relies on a database. This database will contain information about the user, such as account details and their personalized categories and expenses. There will not necessarily be an operator to support the project, but if there was, it would be to maintain the database and this operator would be known as the database administrator. According to Oracle (2023), the best type of database administrator that meets these expectations is a system administrator, who is responsible for managing and maintaining a system. The system administrator would need to be knowledgeable about optimizing and providing secure access to the database, which may require training to be completed.

If an outage were to occur, it would most likely be from the database which is on the server-side. Once we are alerted of an outage, we will first identify what the issue is. Having an idea of our software's architecture structure in mind, we can look for the cause more efficiently by immediately going to the server. Potential causes of an outage related to this could be network outages, lack of power or storage for the server, and too many requests made to the server. After the cause of the application's outage is identified, appropriate fixes and accommodations will be made to ensure that the outage gets resolved. If only a temporary fix was made, we will reflect on how to support long-term changes to make sure that a similar outage is avoided in the future.

## Maintenance Plan

The plan for patching and continued maintenance is to acquire information about software errors and aspects of the software that can be improved upon. After identifying what needs to be patching and updated, we can begin implementing, testing, and deploying appropriate changes. It's also important to track how these patches are affecting users; whether they have made a positive or negative impact for users is necessary to see if an issue has been resolved.

If there were to be patchlines, they would consist of patches that address issues in the current version of the software. Changes in a patch will be defined in patchnotes (in the form of a README.md), which will note specific changes to functionalities and more general changes such as an updated user interface.

If a new version of the software is released, the new version will be documented similarly to patchnotes (in the form of a README.md). The contents of the new version documentation will clearly highlight new features and display how these features are used. The documentation will also identify existing features that have received an update, as well as features that have been removed entirely.

All changes that will be deployed will be captured in a branch on GitHub. The purpose of a branch is to help identify a specific patch or version of the project. The format that will be used for patches and versioning will be similar to "1.0.0", which is a common versioning scheme (Microsoft, 2023). The "1" represents the major version number, where a big update is made. The first "0" represents the minor update number, which is when a small update is made for a version (e.g. a feature or two have been affected). Lastly, the second "0" is the patch number for a minor update.

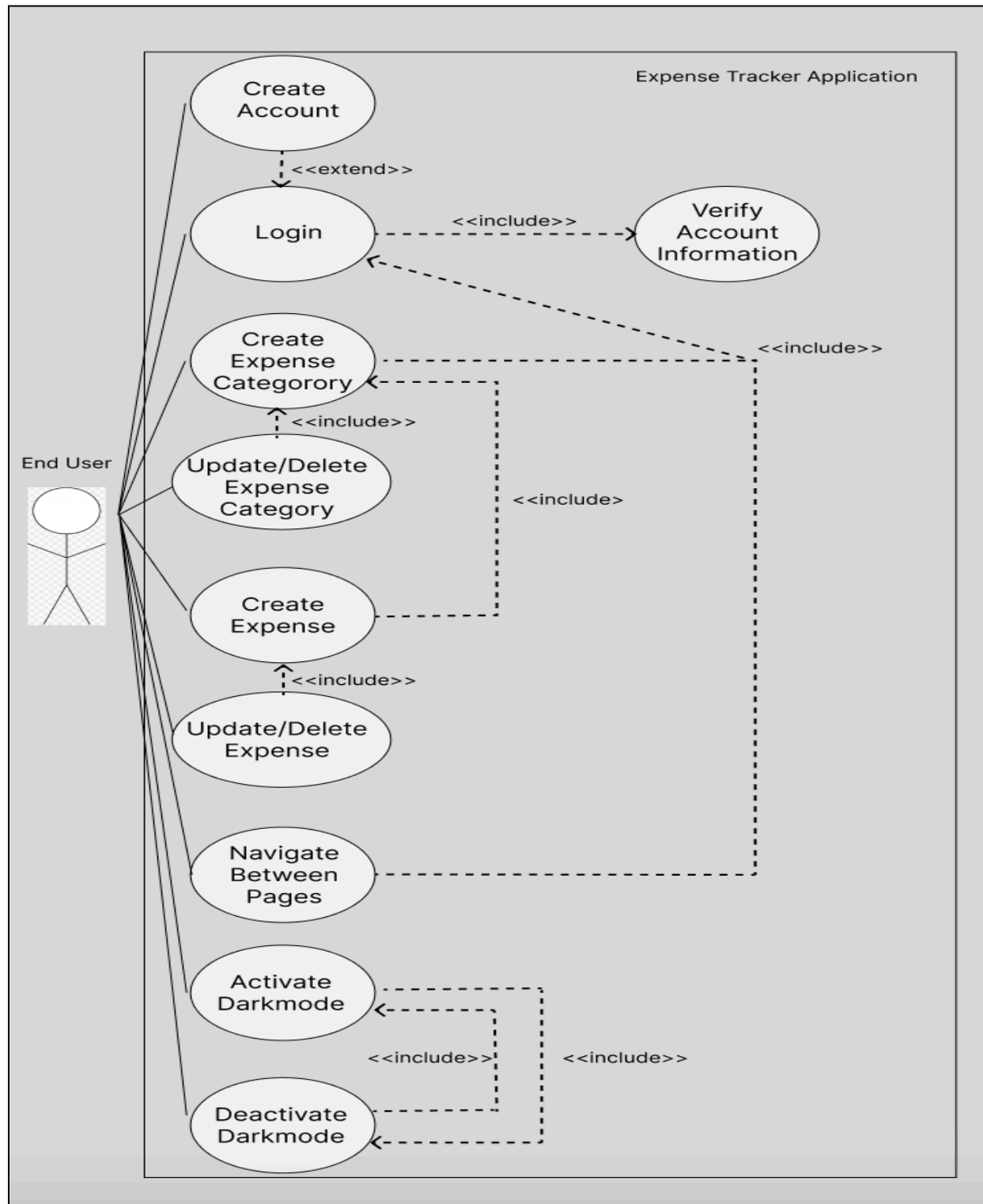
## Technical Sections

### Requirements Engineering

#### User Requirements

Requirement	Description
R01	A user can create an account using a username and password.
R02	A user can sign onto the application after they create an account.
R03	A user can sign out of their account.
R04	A user can create a custom expense category.
R05	A user can update a custom expense category.
R06	A user can delete a custom expense category.
R07	A user can manually input an expense under a specific category.
R08	A user can update an expense.
R09	A user can delete an expense.
R10	A user can view all of their activity on a separate page.
R11	A user can input their budgets for each category.
R12	A user can navigate between the homepage, expense page, and activity page.
R13	A user can press a button to turn “dark mode” on and off.

This table represents the user requirements of the application, ranging from user authentication to darkmode.

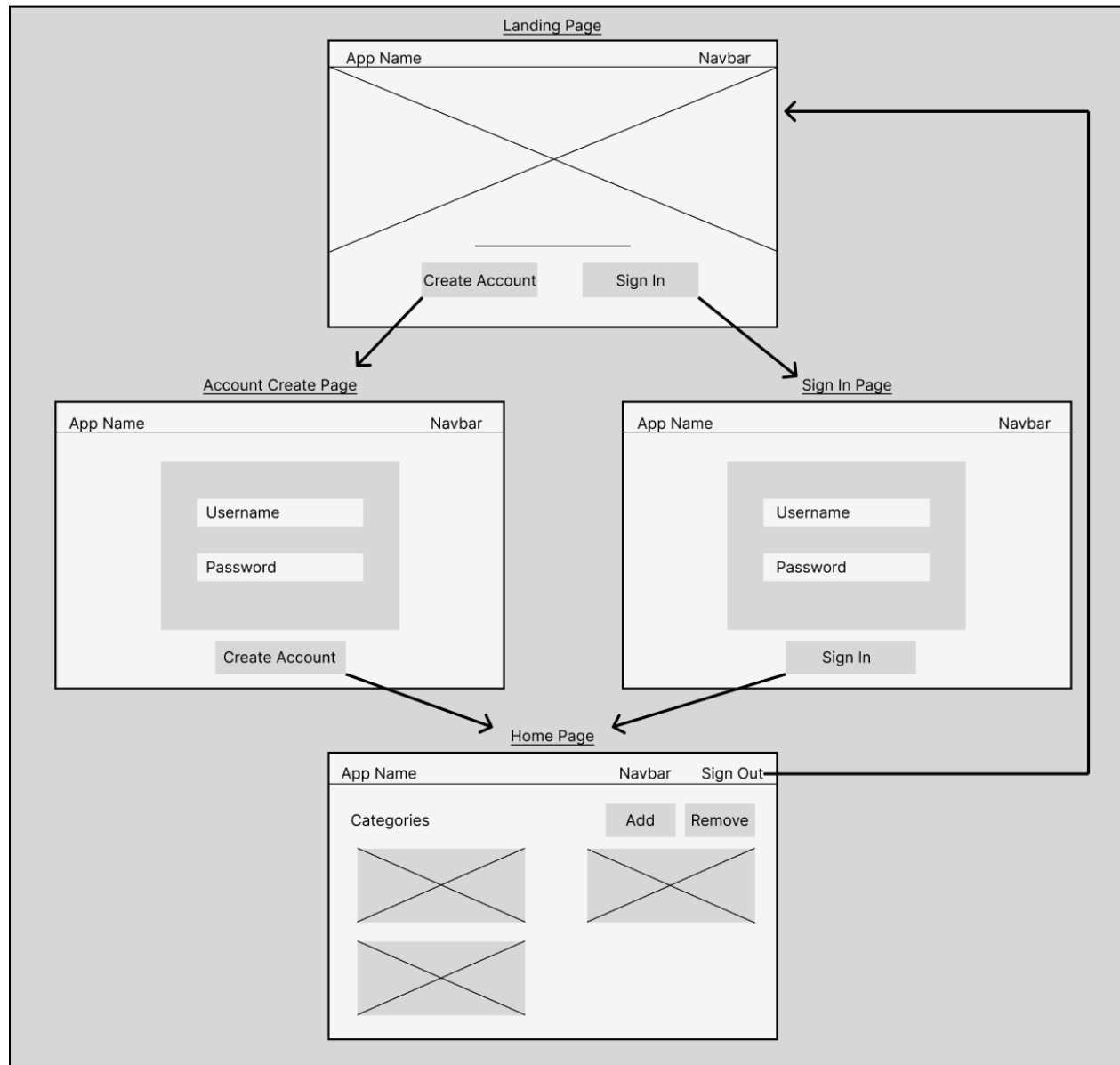
Use-Case Diagram



This use-case diagram illustrates and summarizes how a user may interact with the system by depicting the relationships between them (Lucidchart, 2023). For example, if a user logs in, the system will also verify their credentials to ensure they have an account. Another example would be if a user updates an expense. In this case, the system would have already handled logging in, verifying account information, creating an expense category, and creating an expense.

With this use-case diagram in mind, we can see that there are a few actions that rely on others. When developing the functionality of the application, we can use this information to prioritize the foundational features and work our way up. This will ensure that all prerequisites of a feature are met before we implement the feature itself.

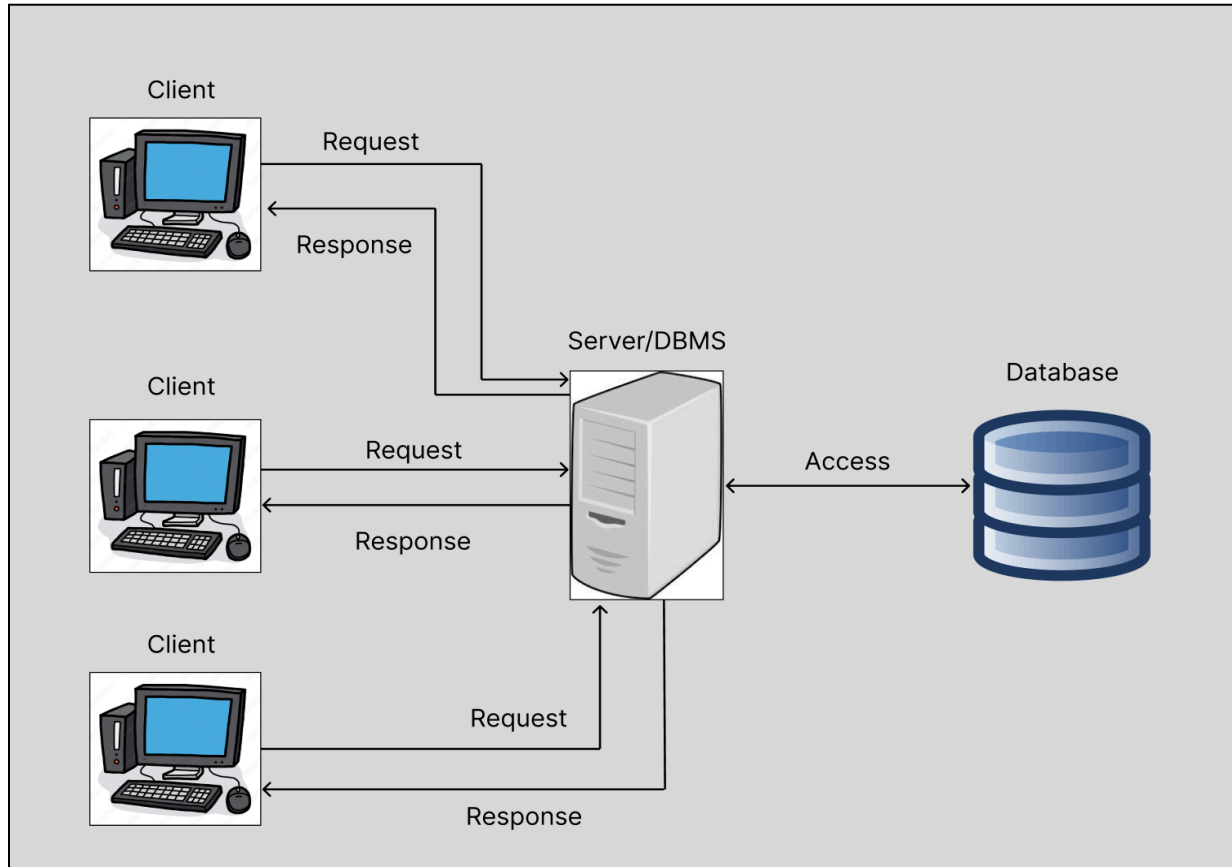
## Storyboard



This storyboard gives a high-level overview of how a user will travel throughout the application by depicting the results from a user's actions (Visual Paradigm, 2023). Typically, a user will start on the landing page. From there, they can either login or create an account, and they will get sent to the respective page. Once their information is entered, they will go to their personal home page. From their home page, they will get sent back to the landing page if they choose to sign out.

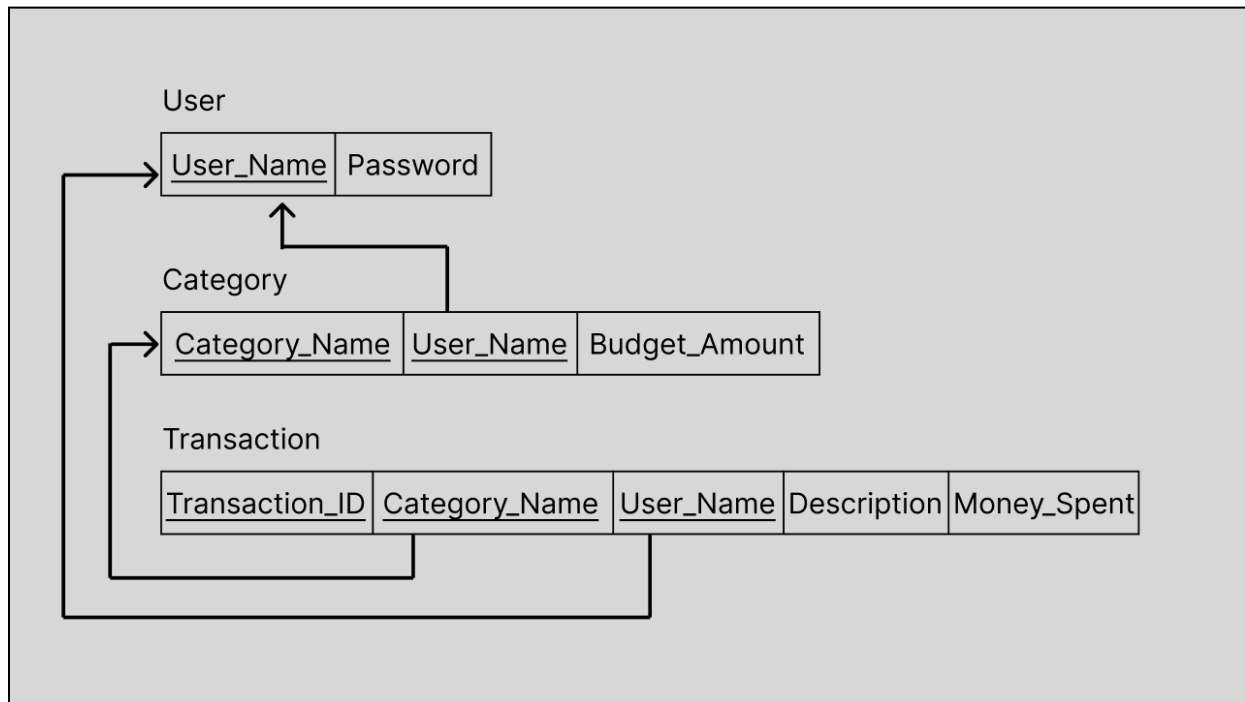
## Architecture and Design

### Client/Server Architecture



Of all the software architecture patterns researched, the client/server pattern seems to fit the needs of this project. Using this pattern, a client can send a request to the server and the database management system (DBMS) will be able to access the database to perform what is needed. Then the server will send a response back to the client based on the request (Butani, 2020). For example, in the context of this application, when a user creates a category, they will input some information about the category which will get sent over to the server. From there, the DBMS will access and try to store this information in the database, and then send a response representing success or failure.

### Relational Database Schema



The database will primarily be used for user authentication, and to store users' categories and transactions.

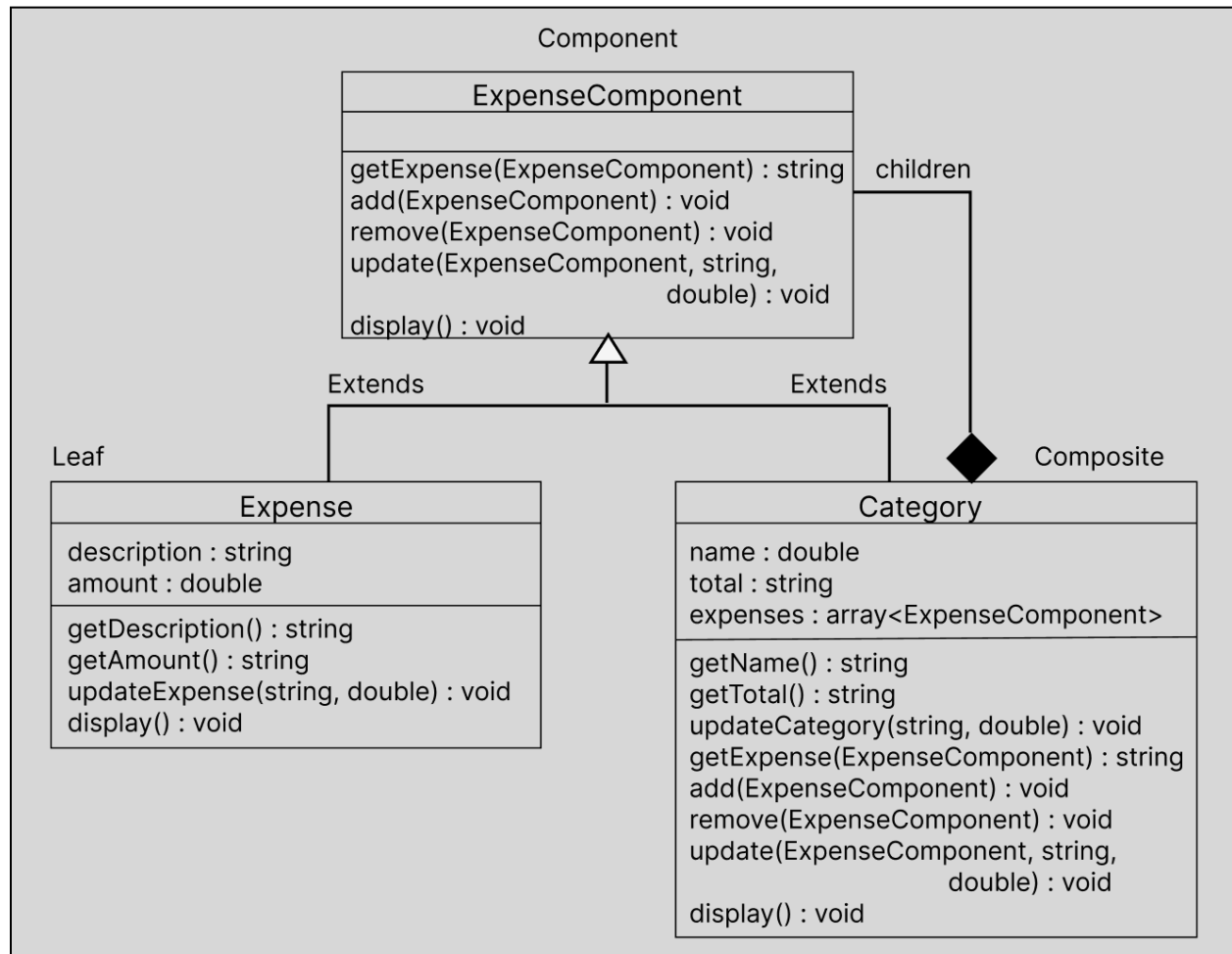
- A User needs a username and password. The primary key of a User is the username.
- A Category needs a name and the username that it is related to, as well as a budgeted amount for it. The primary key of a Category is the category name and a username, which is a foreign key.
- A Transaction needs a unique ID, and the category name and username they are related to. It also needs a description and the amount spent. The primary key of a Transaction is its ID, category name and username both of which are foreign keys.

An example of how this database can be used is if a user wants to create a category. The application will take the name of the category and the budgeted amount as input. Then it will check if there is already a category with this name, for this user. If not, the category will be added to the database. If so, it won't be added to the database since there shouldn't be duplicate categories.

Another example can be for account creation. When a user enters their username and password on the account creation page, the application will check if that username is already in the User relation of the database. If there is no matching username, the username and password will be added to the database. If that username already exists, the username and password won't be added, and the user will be prompted to enter a different username.

Course material learned from CPSC 332 was used when constructing this relational database schema.

## Composite Software Design



The composite software design pattern would be most appropriate for the purposes of this application. This structural design pattern allows for objects to be treated similarly (Pankaj, 2022). For example, a **Category** and an **Expense** both have some common relation with each other. In a sense, a **Category** is a composite, or a collection, of **Expenses** where a **Category** should be able to manage them. A **Category** should be able to display its information, and since a **Category** is a composite of **Expenses**, it should display their information as well. Operations such as adding, updating, and removing **Expenses** from a **Category** are crucial features of this application and this design pattern can be used to ensure that these objects are in an

organized structure and that they can be treated similarly. For this application and design pattern, the ExpenseComponent class will act as an abstract class where the display function is common with all objects. The Category class will be the composite class which holds Expense objects and has functions to manage them. The Expense class is the leaf class that implements ExpenseComponent.

## UX Design

In order to develop the user interface of the application, we must first look at who the users are going to be so that we can have a better idea of how to produce optimal user experience.

These users will most likely be people who are working or has a source of income, and fit in one or more of the following:


- Wants to save for a big purchase, such as a home
- Wants to be organized in terms of finances
- Has several bills
- Has loans, such as mortgage, student loans, car loans
- Has children, and wants to save for their education
- Near retirement

From this user research, we can create user personas to help us visualize who some of our users are.



## User Personas

Basic information



Rebecca Martinez

- 37
- High School Teacher
- Los Angeles
- Single
- 2 Children


Bio

Rebecca Martinez is a Spanish teacher for a high school in Los Angeles. Since she is a teacher supporting two kids, Rebecca wants to make sure that she does not spend more than what she makes.

Personality

Introvert	Extrovert
Analytical	Creative
Busy	Time rich
Messy	Organized
Independent	Team player
Passive	Active
Safe	Risky

Basic information



Tom Jackman

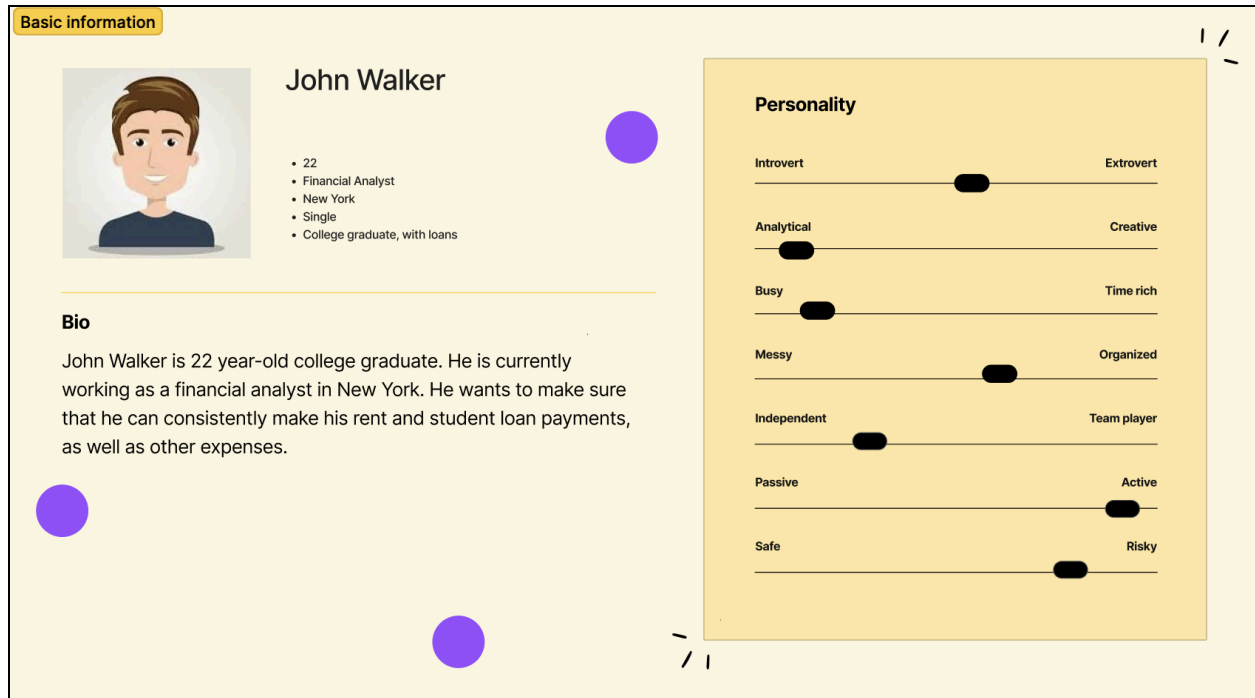
- 63
- Semi-retired engineering manager
- Palm Springs
- Father, Grandfather
- 5 years left on mortgage

Bio

Tom Jackman is an engineering manager who is going to retire in a few years. He is only working part time because he wants to enjoy his life. He has about 5 years left on his mortgage and wants to make sure that once he retires, he will still have enough money to pay it off.

Personality

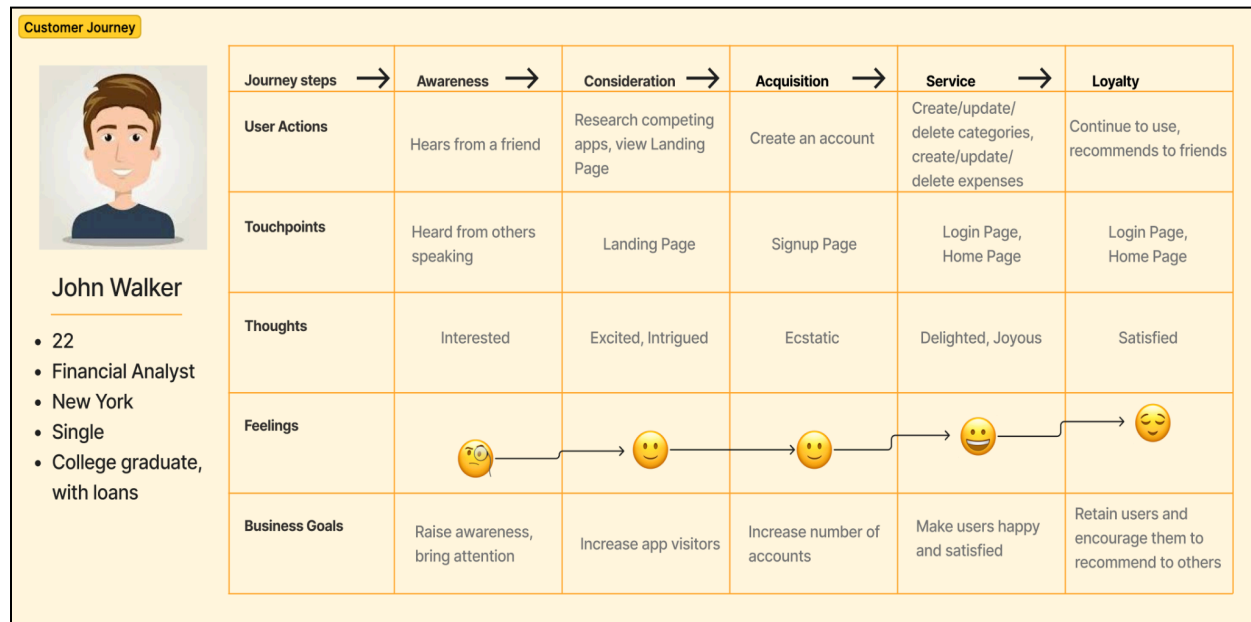
Introvert	Extrovert
Analytical	Creative
Busy	Time rich
Messy	Organized
Independent	Team player
Passive	Active
Safe	Risky



With the user personas in mind, we are able to produce ideas about the application's aesthetic. For example, users will vary in age so we want to make sure that the user interface is not too complicated. It should be easy and intuitive to navigate between pages, entering information and pressing buttons. We can ensure this by including a navigation bar, and buttons to redirect users to certain pages. Also, a page to sign in and another page to create an account are required, so we could include a link to the other page if the user went to the wrong one.

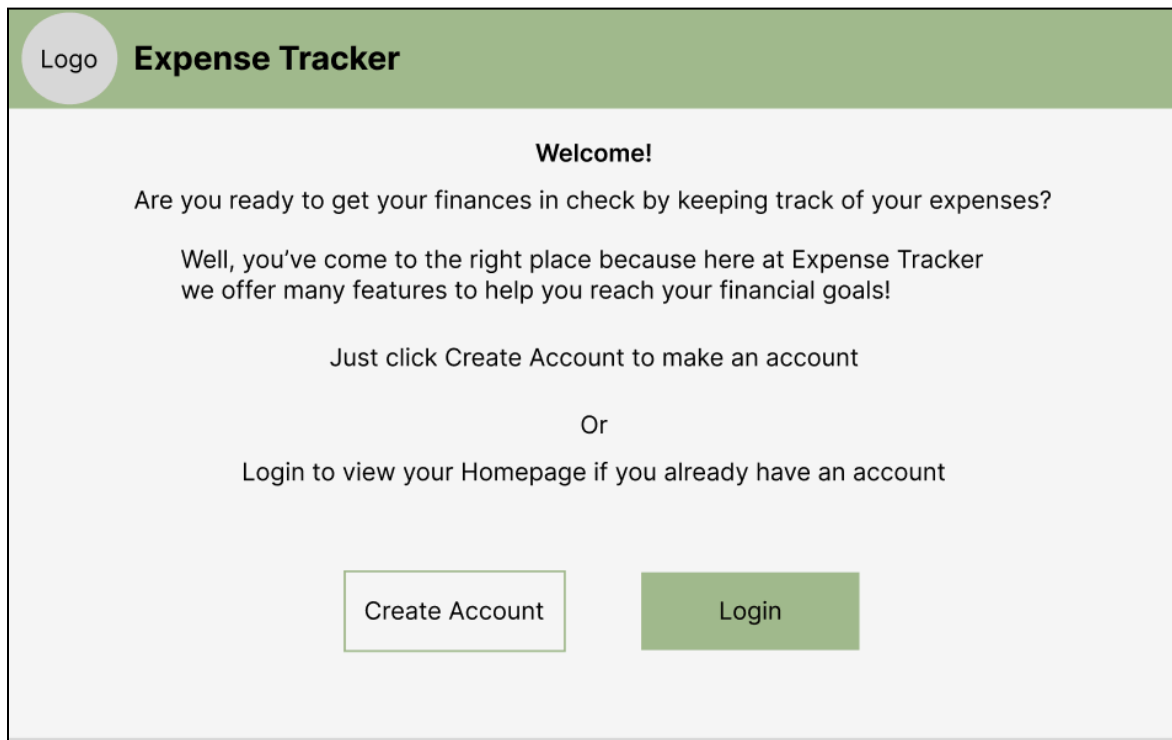
Another way to increase user experience is based on the color scheme of the application. Colors should stay consistent across pages, and it should be easy to distinguish one button's functionality from another. Also, picking an appropriate logo is important because it will be associated with the application, which can serve as a link to the landing page. Based on these ideas, we can create mockups of the application's user interface.

### Customer Journey for John Walker User Persona



This customer journey depicts the path of the John Walker user persona in regards to the application. Initially, he would hear about this application from a friend, and would then do some research about other applications in this domain and maybe even visit the landing page of this application. After he is convinced to use this application, he would create an account and become a user. Then he will start using the application and its features to meet his expectations. Once he is satisfied with the application's performance, he would be a returning user and could potentially spread awareness of this application to others. This customer journey also illustrates John Walker's thoughts and feelings, the different pages visited, and some business-related goals during each stage of his journey. By using this customer journey, in addition to the user personas, we can enhance our mockups.

## Mockups



The landing page features a green header with a circular logo containing the word "Logo" and the text "Expense Tracker". The main content area is light gray and contains a welcome message, a question about getting finances in check, a description of features, and instructions to either create an account or login. At the bottom, there are two buttons: "Create Account" (white with a green border) and "Login" (solid green).

**Expense Tracker**

**Welcome!**

Are you ready to get your finances in check by keeping track of your expenses?

Well, you've come to the right place because here at Expense Tracker we offer many features to help you reach your financial goals!

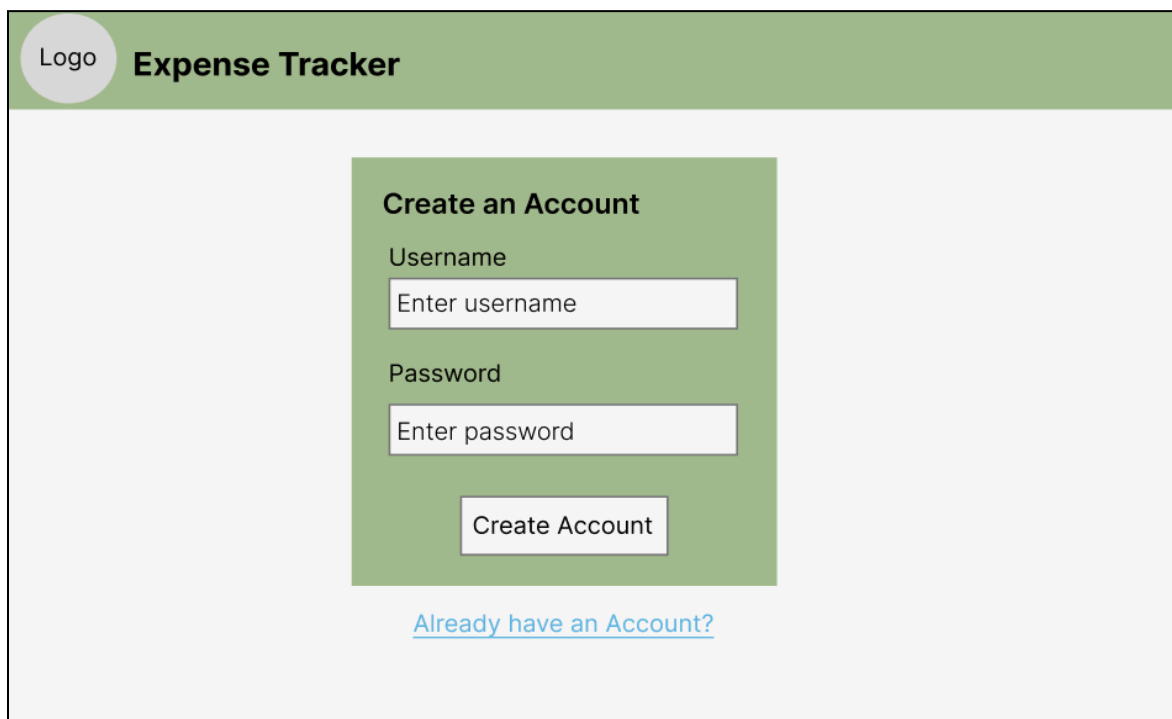
Just click Create Account to make an account

Or

Login to view your Homepage if you already have an account

Create Account Login

- This mockup is for the landing page



The signup page features a green header with a circular logo containing the word "Logo" and the text "Expense Tracker". The main content area is light gray and contains a green box with the title "Create an Account", input fields for "Username" and "Password", and a "Create Account" button. Below the box is a blue link that says "Already have an Account?".

**Expense Tracker**

**Create an Account**

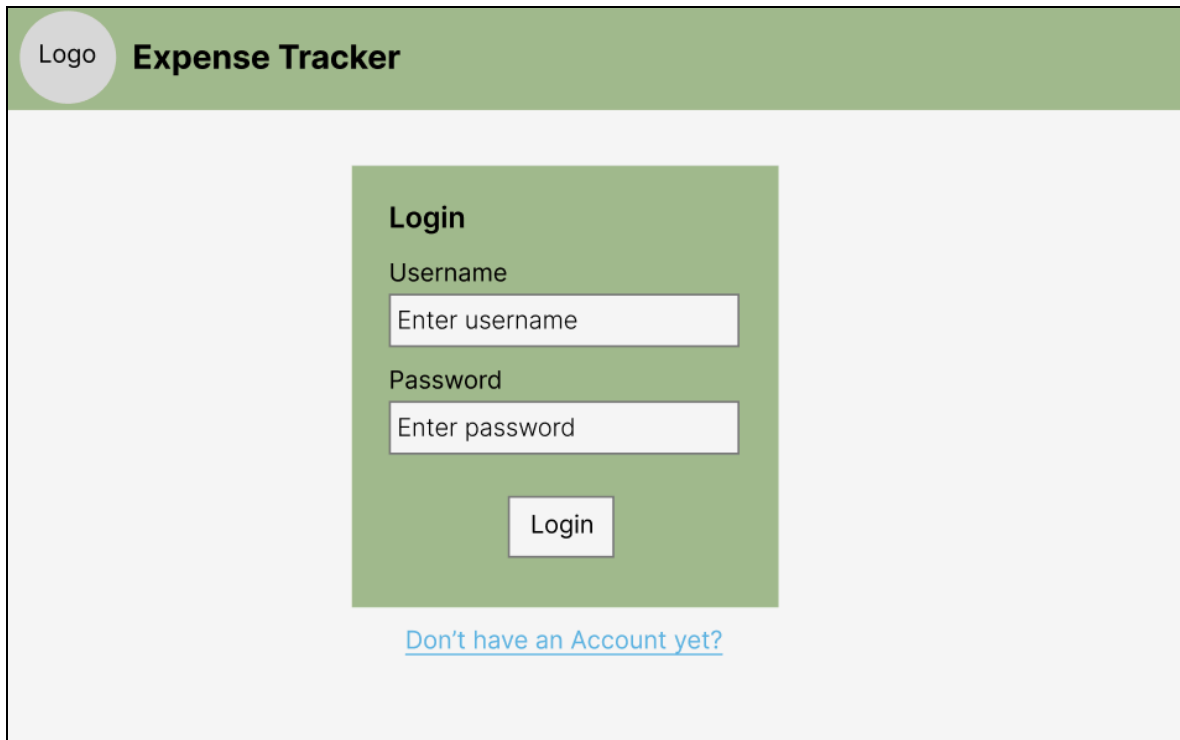
Username  
Enter username

Password  
Enter password

Create Account

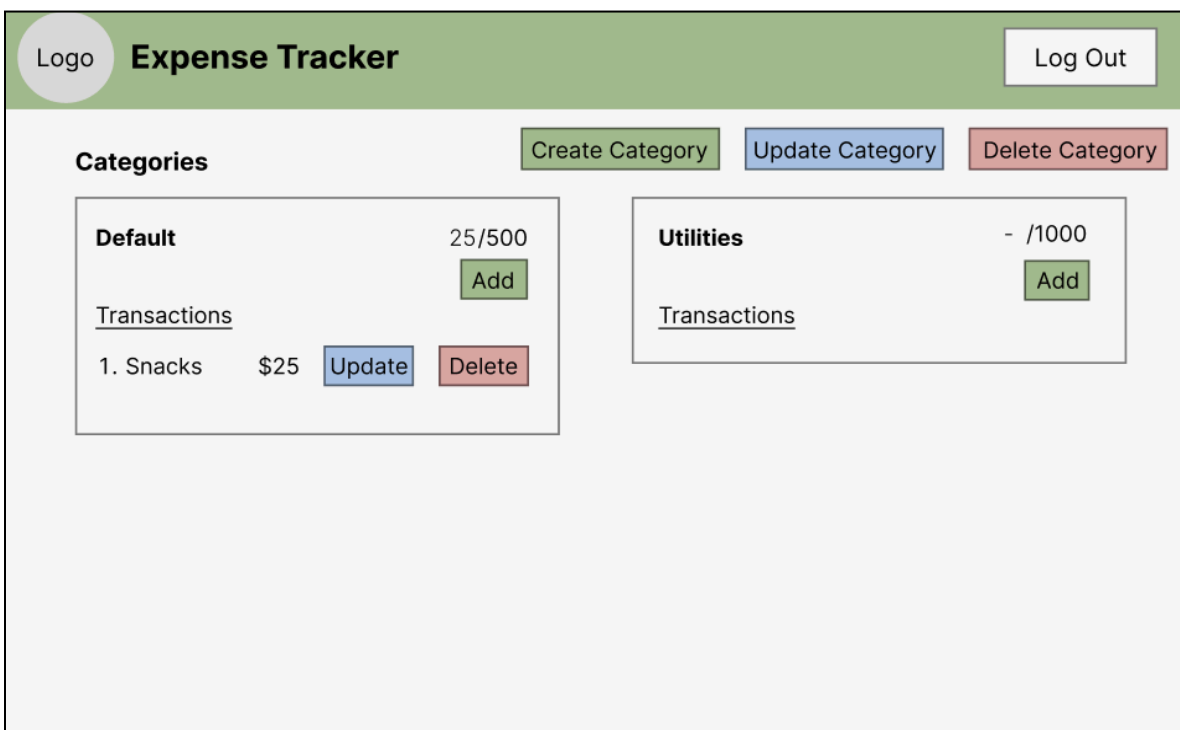
[Already have an Account?](#)

- This mockup is for the signup page



The mockup shows a login page for an 'Expense Tracker' application. At the top, there is a green header bar containing a circular 'Logo' placeholder and the text 'Expense Tracker'. The main content area is light gray and features a central green box with the title 'Login'. Inside this box, there are two input fields: 'Username' with the placeholder text 'Enter username' and 'Password' with the placeholder text 'Enter password'. Below these fields is a 'Login' button. Underneath the green box, there is a blue hyperlink that reads 'Don't have an Account yet?'.

- This mockup is for the login page



The mockup shows the home page of the 'Expense Tracker' application. The top green header bar includes the 'Logo' placeholder, the text 'Expense Tracker', and a 'Log Out' button on the right. Below the header, there are three buttons: 'Create Category' (green), 'Update Category' (blue), and 'Delete Category' (red). The main content area is divided into two columns. The left column is titled 'Categories' and contains a box for the 'Default' category, which shows a balance of '25/500' and an 'Add' button. Below this, there is a 'Transactions' section with a table listing one transaction: '1. Snacks' for '\$25', with 'Update' and 'Delete' buttons. The right column contains a box for the 'Utilities' category, showing a balance of '- /1000' and an 'Add' button. Below this, there is also a 'Transactions' section, which is currently empty.

- This mockup is for the home page, which will vary user-to-user

## Prototyping

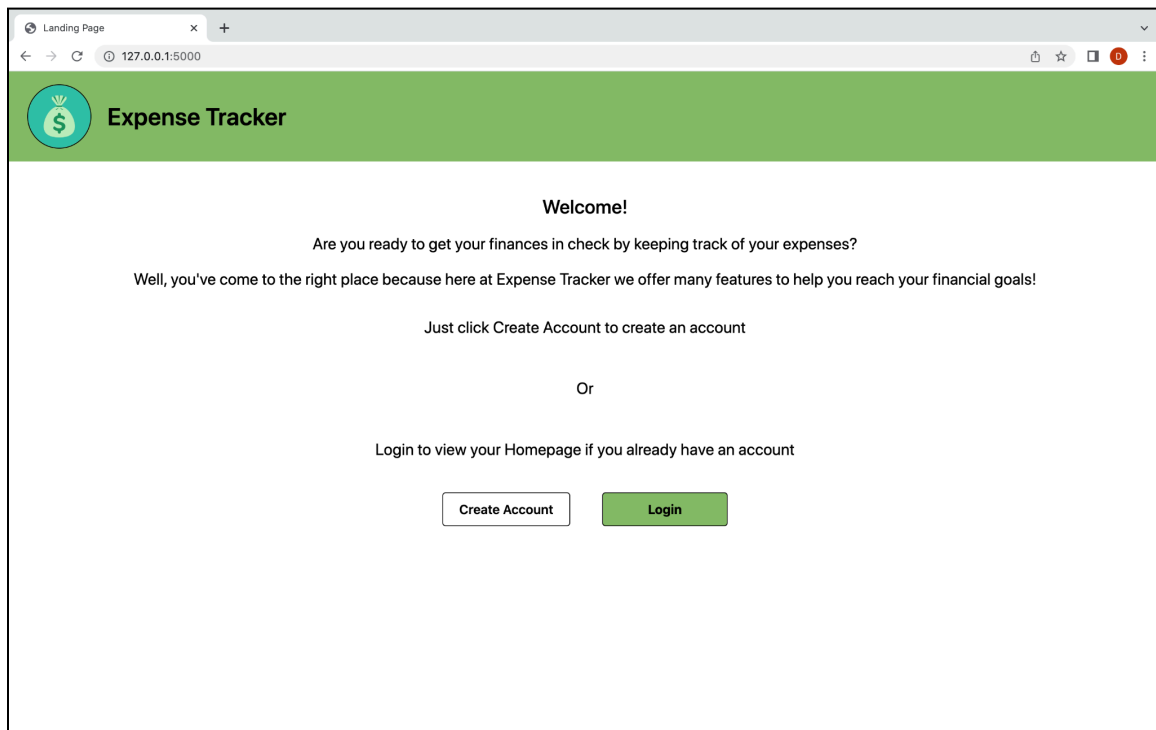
We were able to create a prototype of the application, which uses Flask, Tailwind CSS, Node.js, and npm. The goal of the prototype was to create a user interface that resembles the mockups that were included in the UX Design section.

Git was used as a version control tool, and the code was uploaded to a GitHub repository, which can be found at this [commit](#).

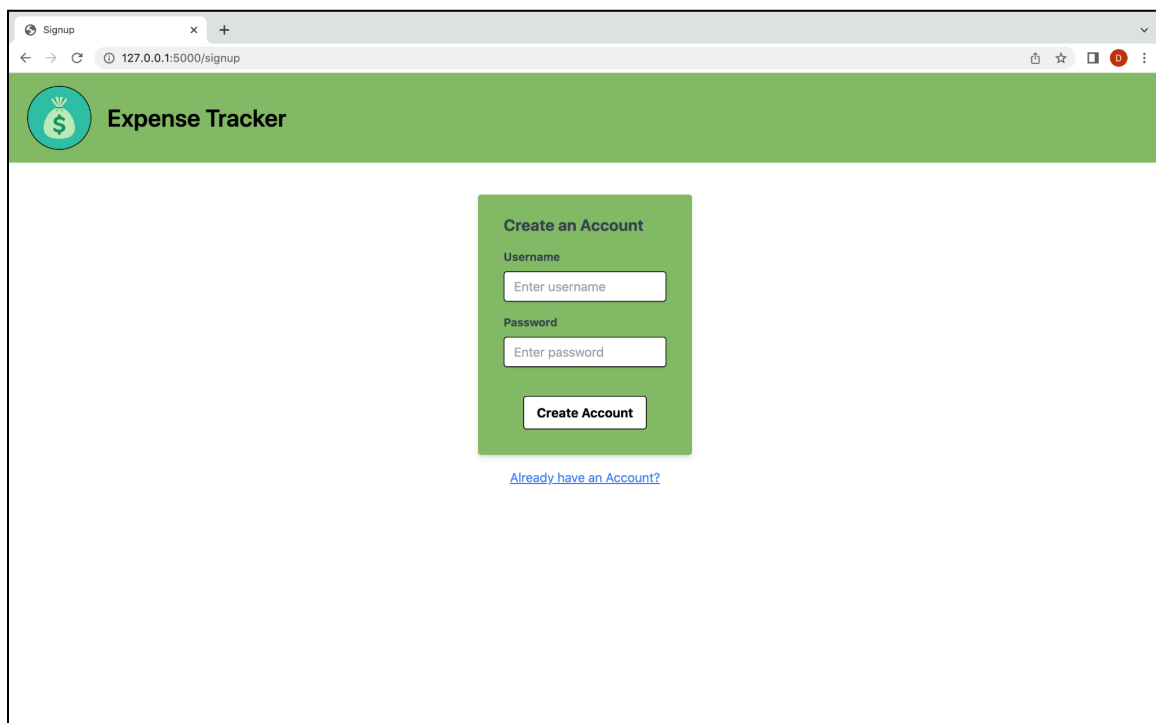
The prototype from CPSC 490 wasn't put onto a separate branch, but the above commit captures the prototype in the GitHub repository. To run the application locally, you must have all the tools mentioned installed. After that, running the npm script "npm run dev" will start the application.

This prototype had no functionality besides navigating through pages. During CPSC 491, this prototype was revised and all functionalities were incorporated.

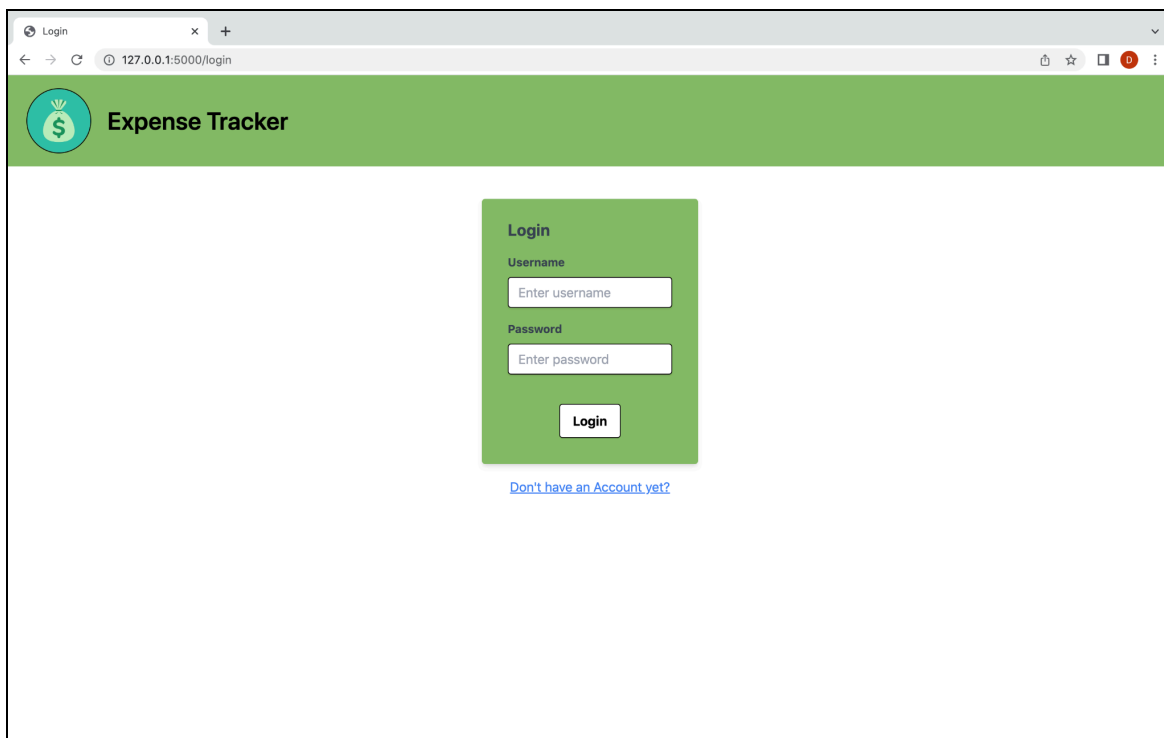
## Screenshots of the Prototype



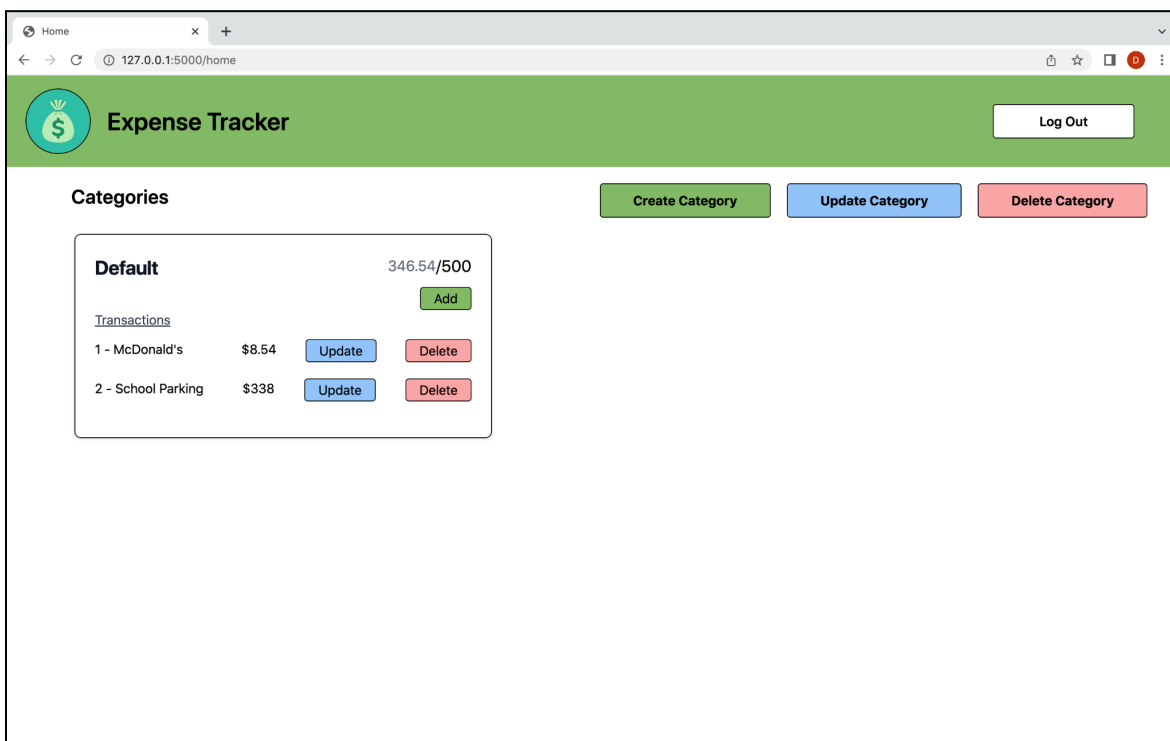
- This screenshot is for the landing page of the prototype



- This screenshot is for the signup page of the prototype



- This screenshot is for the login page of the prototype



- This screenshot is for the home page of the prototype, with some filler data



## Implementation Methods

This application makes use of HTML, TailwindCSS, JavaScript, and the Flask framework which uses Python. In order to ensure consistency, readability, and reusability, we will establish a few coding standards, tools for evaluation such as linters, and best practices.

### Coding Standards

#### HTML:

- Use lowercase letters for HTML tags/attributes
- Have an ending tag for each opening tag where applicable
- Use indentation to organize nested HTML tags
- Don't use inline styles
- Don't use inline scripts

#### Tailwind CSS:

- Create custom utilities
- Appropriately name custom utilities

#### Javascript:

- Use camelCase for variable and function names
- Meaningful variable and function names
- Avoid using 'var', and use either 'let' or 'const' instead for variables
- Compare using '===' rather than '=='
- Use indentation to organize code
- Leave comments for functions and complex bits of code

Python:

- Use snake\_case for variable and function names
- Meaningful variable and function names
- Leave comments for functions and complex bits of code
- Have appropriate indentation to prevent code from breaking
- Use functions to make code reusable

### Tools for Evaluation

- Prettier for HTML
- Tailwind CSS IntelliSense for Tailwind CSS, which can be used in Visual Studio Code
- Standard for JavaScript
- PyLint for Python

### Best Practices

- Write readable code by using comments, descriptive variable and function names, and consistent indentations
- Write reusable code by creating functions
- Use a version control system such as Git, and commit changes frequently
- Frequently test different parts of the code
- Whenever a new feature is implemented, also make sure that the older features are working as expected
- Prevent memory leaks by deallocating any memory that was created

List of Tools

- HTML
- Tailwind CSS
- JavaScript
- Python
- Flask framework
- SQLite
- Git
- GitHub
- Visual Studio Code
- Node.js
- npm
- pip
- Prettier
- Tailwind CSS IntelliSense
- Standard
- PyLint

## Link to the GitHub Code

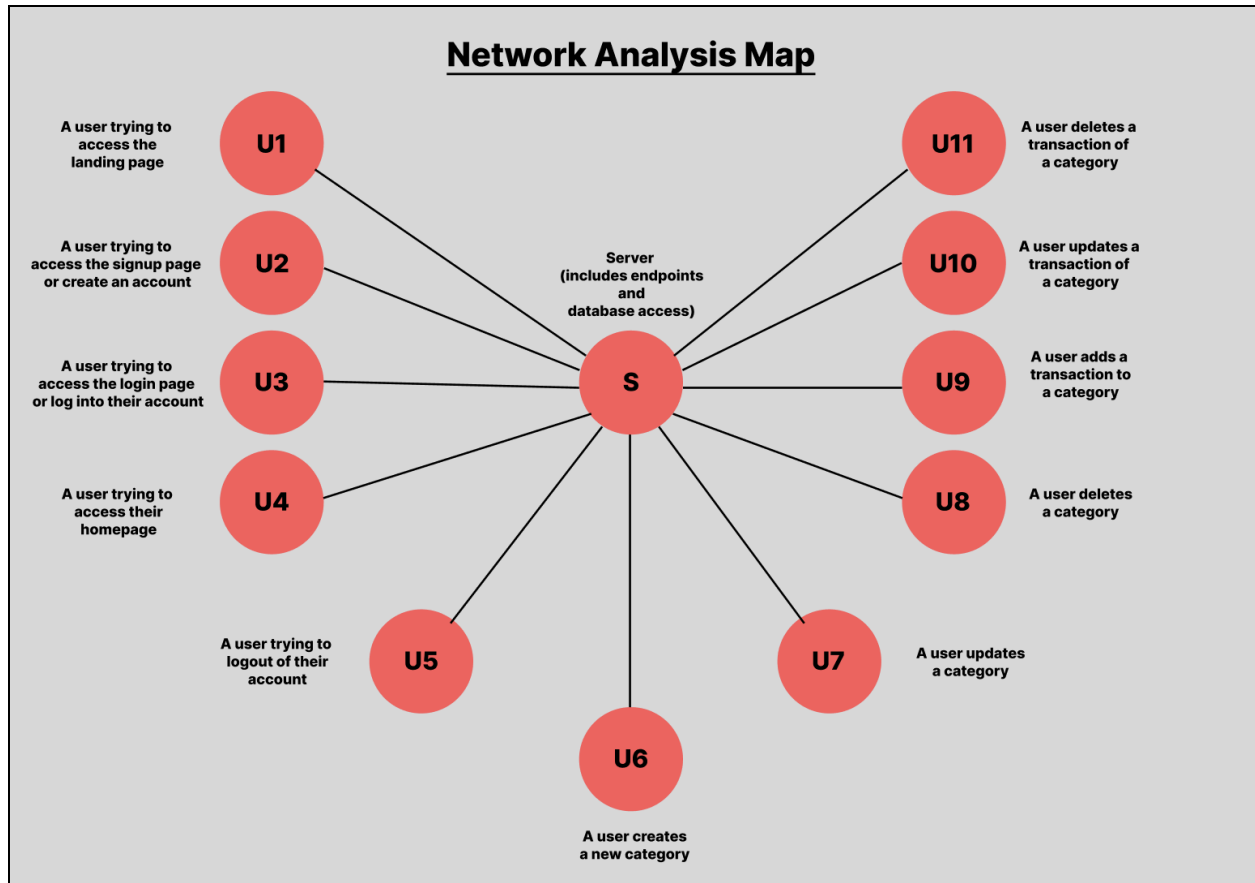
The GitHub repository can be found at <https://github.com/dharbo/490Project>.

Detailed instructions on how to install/setup and run the application locally can be found at [Installation / Setup Guide](#). Note that there are two branches: main and v1.0.0. They are currently identical, but v1.0.0 would be the branch used for deployment.

## Network Analysis and Map

In deployment, the application will be hosted on a server which is why it is important to perform network analysis and see how the application can be interacted with at the network level. According to Loem (2021), a network structure is comprised of nodes and edges, where nodes represent an object and edges represent the relationship between objects. In the scope of this application, our nodes would fall under two types: user and server. As for edges, the only type of relationship would be HTTP Requests, such as HTTP GET and HTTP POST.

Essentially, the application allows a user-type object to make HTTP Requests to a server-type object. A user-type object cannot interact with another user-type object, which means that there is no relationship between them. Below is a network analysis map that details the relationship between nodes for this software.



The map displays user-type objects performing different actions and a server-type object that would be hosting the application. Note that the server-type object would host the application, providing access directly to endpoints and indirectly to the database. As can be seen, a relationship is formed with the server-type object whenever a user-type object performs an action that triggers an HTTP Request. For example, node U6 is a user creating a category which sends an HTTP POST to node S; sending the data in the request to the corresponding endpoint for creating categories. Once here, the endpoint will make proper use of database queries and data processing, and redirect the user to an endpoint that returns an updated HTML page. It is also worth mentioning that the server-type object is considered as the most important node in this map because it has the highest degree and thus higher centrality.

## Threat Model

As explained by the Centers for Medicare and Medicaid Services (2023), the purpose of Threat Modeling is to find potential risks and vulnerabilities in an application that an attacker can use to cause harm. Then with this information, appropriate actions can be taken to minimize these risks and vulnerabilities before an attack occurs.

This application has a few risks and vulnerabilities that could be improved. The first is a denial of service (DoS) attack. If the software is being hosted on a single server, users would face issues if a DoS attack occurs because the server would be overwhelmed with false requests. In order to offset the results of a DoS attack, implementing load balancing with multiple servers could greatly reduce the amount of traffic a single server would receive.

Another risk is an attacker stealing a user's login credentials. This could happen in a number of different ways such as from using a keylogger. Currently, there is not a way for the software to notice when an attacker obtains a user's credentials or when an attacker logs in with another user's credentials. Despite this, a common tactic to mitigate any harm in this situation is to implement Two-Factor Authentication where a temporary passcode gets sent to a user's email or cell phone and needs to be entered to login. Another method would be to display the most recent login for a user's account, and the user may determine if their account has been compromised.

It's important to note that a good practice to reduce vulnerabilities from being exploited by an attacker is to keep the source code private. This might be a simple tactic to minimize threats, but makes it much more difficult for attackers to find potential vulnerabilities.

## Open Source / 3rd Party Components Used

There are a few open source components that were used to develop this web application. These components are Flask, Node.js, npm, and Tailwind CSS. Flask serves as our server that our application runs on and we are able to take advantage of many features such as templating and routing. Node.js and npm are needed to install Tailwind CSS as well as run our application in a development environment. They are more like “tools” and aren’t entirely necessary once the web application is deployed. As for Tailwind CSS, it is a Cascading Stylesheet framework to help style our HTML. It has utility classes which are predefined CSS classes that can be used directly in HTML, allowing developers the convenience of not needing to create custom CSS files. Another component used is SQLite, which is used for our database. Conveniently, a “sqlite3” module is bundled with Python, meaning SQLite can be used in our project since Flask is Python-based. Despite the “sqlite3” module not being entirely open source, it is worth noting why we can use it without a 3rd party component.

## Test Plans and Results

### Unit Tests

For unit tests, Python's "unittest framework" ([here](#)) will be used. The focus of these unit tests will be to test each endpoint, ranging from account creation to adding a transaction. By definition, a unit test should not rely on any other component and testing each endpoint individually would be effective. A Python script will be used to run these unit tests, where a test-oriented database will be generated and then deleted. This would also for a clean testing interface.

The unit tests are defined in the `unit_tests.py` file in the GitHub repository, [https://github.com/dharbo/490Project/blob/main/unit\\_tests.py](https://github.com/dharbo/490Project/blob/main/unit_tests.py). As can be seen, there is a unit test for each endpoint. If the request is an HTTP GET request, we primarily check to see if the resulting HTTP status code is 200, meaning that the request succeeded. To test HTTP POST requests, we will pass in "dummy" data to simulate a real request and then access the test-oriented database to ensure it was added appropriately. For these HTTP POST requests, we also test with invalid "dummy" data to make sure that the endpoint handles it correctly. By using Python's unittest framework, we are able to simply run the Python test file and it should yield successful results.

Below is the test file being run on the most recent version of the software. The messages that start with "Could not..." are print statements from some endpoints that are encountered when testing invalid input and can be ignored. From this output, it can be seen that 11 tests have been run and the "OK" means they were all successful. This is in line with our expected results.



```
dharbo@Davids-Laptop:~/Desktop/491Project$ python3 unit_tests.py
.Could not add transaction. Category does not exist and/or money spent input is invalid.
.Could not create category. Category already exists and/or budget input is invalid.
...Could not update category. Old category either does not exist, new category already exists and/or new budget input is invalid.
.Could not update transaction. Category does not exist, transaction id is invalid and/or new money spent input is invalid.
.....
-----
Ran 11 tests in 0.405s

OK
dharbo@Davids-Laptop:~/Desktop/491Project$
```

## Performance Tests

To test performance, emphasizing queries to the database would be best. This could be done by using SQLite's ".timer on" instruction for the queries used by the software. If any of them are unusually slow, then that would encourage investigation.

Below are the test cases and results for queries made by the endpoints for signing up and logging in.

```
dharbo@Davids-Laptop:~/Desktop/491Project$ sqlite3 database.db
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
sqlite>
sqlite> .timer on
sqlite>
sqlite> INSERT INTO USER VALUES ('DavidH', 'password');
Run Time: real 0.007 user 0.001002 sys 0.001002
sqlite>
sqlite> INSERT INTO CATEGORY VALUES ('Default', 'DavidH', 1000)
...> ;
Run Time: real 0.024 user 0.000842 sys 0.000421
sqlite>
sqlite>
sqlite> SELECT User_Name, Password FROM USER WHERE User_Name='DavidH' AND Password='password'
...> ;
DavidH|password
Run Time: real 0.001 user 0.000203 sys 0.000135
sqlite>
sqlite> █
```

Below are the test cases and results for queries made by the endpoints for creating, updating, and deleting categories.

```
sqlite> SELECT * FROM CATEGORY WHERE Category_Name='Car' AND User_Name='DavidH';
Run Time: real 0.001 user 0.000000 sys 0.000789
sqlite>
sqlite> INSERT INTO CATEGORY VALUES ('Car', 'DavidH', '400');
Run Time: real 0.023 user 0.000000 sys 0.001525
sqlite>
sqlite>
sqlite> UPDATE CATEGORY SET Category_Name='Car Expense', Budget_Amount='500' WHERE Category_Name='Car' AND User_Name='DavidH';
Run Time: real 0.022 user 0.000000 sys 0.001094
sqlite>
sqlite> UPDATE TRANSACTIONS SET Category_Name='Car Expense' WHERE Category_Name='Car' AND User_Name='DavidH';
Run Time: real 0.001 user 0.000306 sys 0.000000
sqlite>
sqlite>
sqlite> DELETE FROM CATEGORY WHERE Category_Name='Car Expenses' AND User_Name='DavidH';
Run Time: real 0.001 user 0.000237 sys 0.000000
sqlite>
sqlite> DELETE FROM TRANSACTIONS WHERE Category_Name='Car Expenses' AND User_Name='DavidH';
Run Time: real 0.000 user 0.000272 sys 0.000000
sqlite>
```

Below are the test cases and results for queries made by the endpoints for adding, updating, and deleting transactions.

```
sqlite> INSERT INTO TRANSACTIONS VALUES ('None', 'Default', 'DavidH', 'Tip', '5.00', '2023-11-18 19:29:29.971858');
Run Time: real 0.006 user 0.000967 sys 0.000470
sqlite>
sqlite>
sqlite> SELECT COUNT(*) FROM TRANSACTIONS WHERE Category_Name='Default' AND User_Name='DavidH';
1
Run Time: real 0.001 user 0.000446 sys 0.000000
sqlite>
sqlite> UPDATE TRANSACTIONS SET Transaction_ID='1' WHERE Category_Name='Default' AND User_Name='DavidH' AND Date='2023-11-18 19:29:29.971858';
Run Time: real 0.024 user 0.000000 sys 0.001652
sqlite>
sqlite>
sqlite> SELECT * FROM TRANSACTIONS WHERE Transaction_ID='1' AND Category_Name='Default' AND User_Name='DavidH';
1|Default|DavidH|Tip|5.00|2023-11-18 19:29:29.971858
Run Time: real 0.000 user 0.000301 sys 0.000145
sqlite>
sqlite> UPDATE TRANSACTIONS SET Description='Tip', Money_Spent='6.00' WHERE Transaction_ID='1' AND Category_Name='Default' AND User_Name='DavidH';
Run Time: real 0.006 user 0.000930 sys 0.000439
sqlite>
sqlite>
sqlite> DELETE FROM TRANSACTIONS WHERE Transaction_ID='1' AND Category_Name='Default' AND User_Name='DavidH';
Run Time: real 0.006 user 0.000000 sys 0.001399
sqlite>
sqlite> SELECT * FROM TRANSACTIONS WHERE Category_Name='Default' AND User_Name='DavidH' ORDER BY Date;
Run Time: real 0.000 user 0.000260 sys 0.000122
sqlite>
```

Below are the test cases and results for queries made by a helper function that retrieves category and transaction data from the database.

```
sqlite> SELECT * FROM CATEGORY WHERE User_Name='DavidH'
...> ;
Default|DavidH|1000
Run Time: real 0.000 user 0.000179 sys 0.000119
sqlite>
sqlite>
sqlite> SELECT * FROM TRANSACTIONS WHERE User Name='DavidH' AND Category_Name='Default';
Run Time: real 0.000 user 0.000075 sys 0.000050
sqlite> _
```

Based on the results from testing the speed of these queries, we can tell that they are all very fast. The slowest query took about 0.024 seconds which is very good for the size of the database used for testing. We may assume that these queries might slow down as the database grows, but it is more than enough for the scope of this project. If we need to scale the project upward in the future, we could potentially incorporate database indexing or even load balancing. Though, the current results meet our expectations in terms of performance.

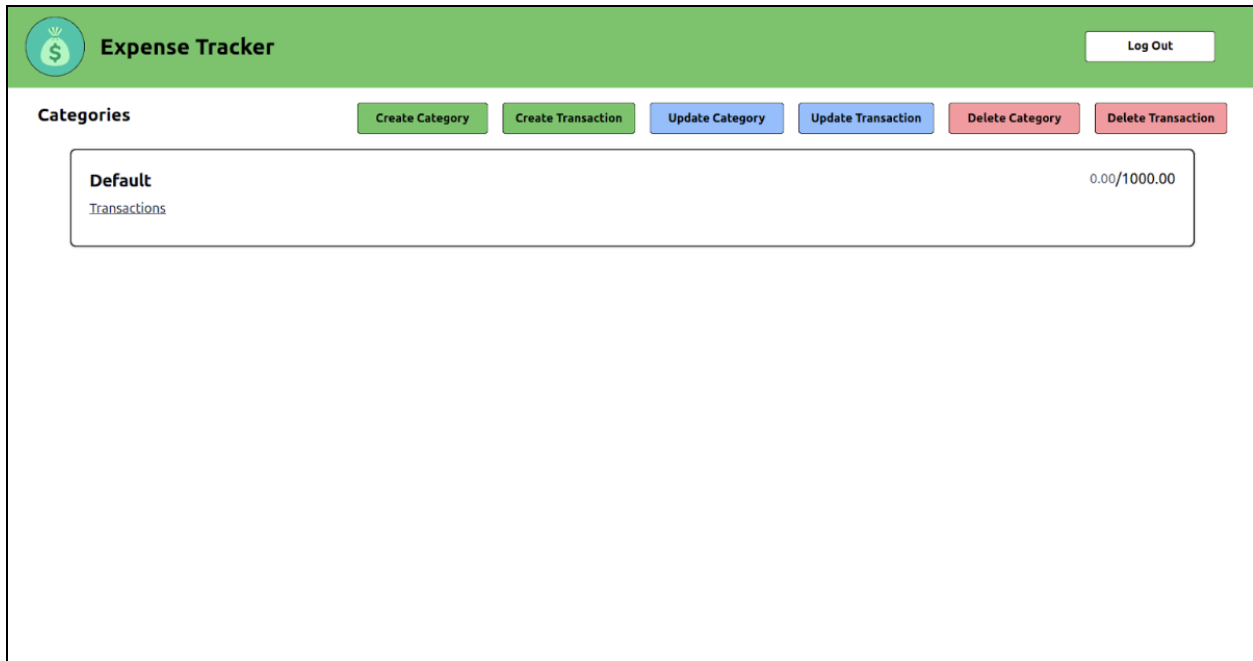
## Security Tests

As for security tests, the software should be able to prevent a user from accessing data that belongs to other users. This is primarily done through authentication, with the use of logins. So security tests should revolve around checking that a user can only access their data from the database.

Our test will be to create 2 separate accounts and see if changes in one user's homepage are reflected in the other. If the software is secure, the other user's homepage should not be affected.

We'll create accounts with usernames CSUF123 and CSUF789. For CSUF123, we add a few categories and transactions, and for CSUF789 we don't add anything. Below is the current state of the homepage for users CSUF123 and CSUF789 respectively.

Expense Tracker		Log Out
Categories	<a>Create Category</a> <a>Create Transaction</a> <a>Update Category</a> <a>Update Transaction</a> <a>Delete Category</a> <a>Delete Transaction</a>	
<b>Default</b> <a>Transactions</a>	0.00/1000.00	
<b>Food</b> <a>Transactions</a>	80.60/400.00	
1 - Groceries	\$74.00	
2 - Lunch	\$6.60	
<b>Car</b> <a>Transactions</a>	49.00/500.00	
1 - Gas	\$49.00	



The image shows a web application titled "Expense Tracker" with a green header bar. On the left of the header is a circular icon with a dollar sign and a crown. On the right is a "Log Out" button. Below the header, there is a "Categories" section with a list of categories. The first category is "Default" with a sub-label "Transactions" and a balance of "0.00/1000.00". Above the category list are six buttons: "Create Category" (green), "Create Transaction" (green), "Update Category" (blue), "Update Transaction" (blue), "Delete Category" (red), and "Delete Transaction" (red).

Now, we'll try to delete the category named "Car" while signed in as CSUF789. Since CSUF789 is performing the operation and doesn't have a category named "Car", a warning message should appear about that category not existing. Also, if we log back in as CSUF123, the homepage should remain the same since CSUF123 did not perform that action. Below is user CSUF789 performing that action, CSUF789's homepage, and CSUF123's homepage respectively.

 Expense Tracker Log Out

Categories

Create Category>Create TransactionUpdate CategoryUpdate TransactionDelete CategoryDelete Transaction

Default

Transactions


0.00/1000.00

Delete Category

Category Name

Car

Delete

 Expense Tracker Log Out

Categories

Create Category>Create TransactionUpdate CategoryUpdate TransactionDelete CategoryDelete Transaction


Warning!

Could not delete category. Category does not exist.

Default

Transactions

0.00/1000.00

 **Expense Tracker** Log Out

**Categories** Create Category Create Transaction Update Category Update Transaction Delete Category Delete Transaction

**Default** 0.00/1000.00  
[Transactions](#)

**Food** 80.60/400.00  
[Transactions](#)  
1 - Groceries \$74.00  
2 - Lunch \$6.60

**Car** 49.00/500.00  
[Transactions](#)  
1 - Gas \$49.00

As can be seen, the action done by a user only affects that user which promotes security of all users' data. Similar security tests were done for the other operations (creating, updating, and deleting categories and adding, updating transactions), all of which succeeded.

## Bugs

Throughout development, we stumbled upon three bugs. The first bug had to do with numeric inputs in HTML forms. By design, the type of input needs to be an “int” or a “float”, and the value gets converted either way when added to the database. To fix this, we added a helper function that checks if the provided value is an acceptable number which is either 0, 0.0, a positive “int”, or a positive “float”. Below is evidence of the bug and the helper function created as the solution. Here we try creating a category with “100a” as the budget and it results in an error when submitting the form.



The screenshot displays the 'Expense Tracker' application interface. A modal window titled 'Create Category' is open, allowing the user to add a new category. The modal contains two input fields: 'Category Name' with the value 'Food' and 'Budget' with the value '100a'. A red error message is visible below the budget field, indicating an invalid input. The background shows a list of categories and transactions, with buttons for 'Create Category', 'Create Transaction', 'Update Category', 'Update Transaction', 'Delete Category', and 'Delete Transaction'.



## ValueError

ValueError: could not convert string to float: '100a'

### Traceback (most recent call last)

```
File "home\harbo\local\lib\python3.8\site-packages\task\app.py", line 2551, in __call__
    return self.wsgi_app(environ, start_response)
File "home\harbo\local\lib\python3.8\site-packages\task\app.py", line 2531, in wsgi_app
    response = self.handle_exception(e)
File "home\harbo\local\lib\python3.8\site-packages\task\app.py", line 2528, in wsgi_app
    response = self.full_dispatch_request()
File "home\harbo\local\lib\python3.8\site-packages\task\app.py", line 1825, in full_dispatch_request
    rv = self.handle_user_exception(e)
File "home\harbo\local\lib\python3.8\site-packages\task\app.py", line 1823, in full_dispatch_request
    rv = self.dispatch_request()
File "home\harbo\local\lib\python3.8\site-packages\task\app.py", line 1799, in dispatch_request
    return self.ensure_sync(self.view_functions[rule.endpoint])(**view_args)
File "home\harbo\Desktop\491Project\app.py", line 164, in createCategory
    cursor.execute("INSERT INTO CATEGORY VALUES (?, ?, ?)", (category_name, session['username'], f'{{float(budget):.2f}}'))
```

ValueError: could not convert string to float: '100a'

The debugger caught an exception in your WSGI application. You can now look at the traceback which led to the error.

To switch between the interactive traceback and the plaintext one, you can click on the "Traceback" headline. From the text traceback you can also create a paste of it. For code execution mouse-over the frame you want to debug and click on the console icon on the right side.

You can execute arbitrary Python code in the stack frames and there are some extra helpers available for introspection:

- `dump()` shows all variables in the frame
- `dump(obj)` dumps all that's known about the object

Brought to you by **DON'T PANIC**, your friendly Werkzeug powered traceback interpreter.

```
# An acceptable number is any non-negative value
def is_acceptable_number(num):
    # Check if num is not an int or a float. Negative values get handled here.
    if num.isdigit() or num.replace(".", "", 1).isnumeric():
        return True
    return False
```

The second bug had to do with the user interface. Before, the user interface allowed two or more categories to be on the same row. Though, it became problematic when 1 category of a row would have more transactions than the others. This is because it would cause the other categories to also stretch downward and create a lot of whitespace isn't what we wanted. A reasonable workaround was to have 1 category per row and stretch the width of the categories. By doing this we not only resolved this bug, but improved the overall user interface by making it look cleaner. We can see the bug below, as well as the updated user interface that fixes the issue.

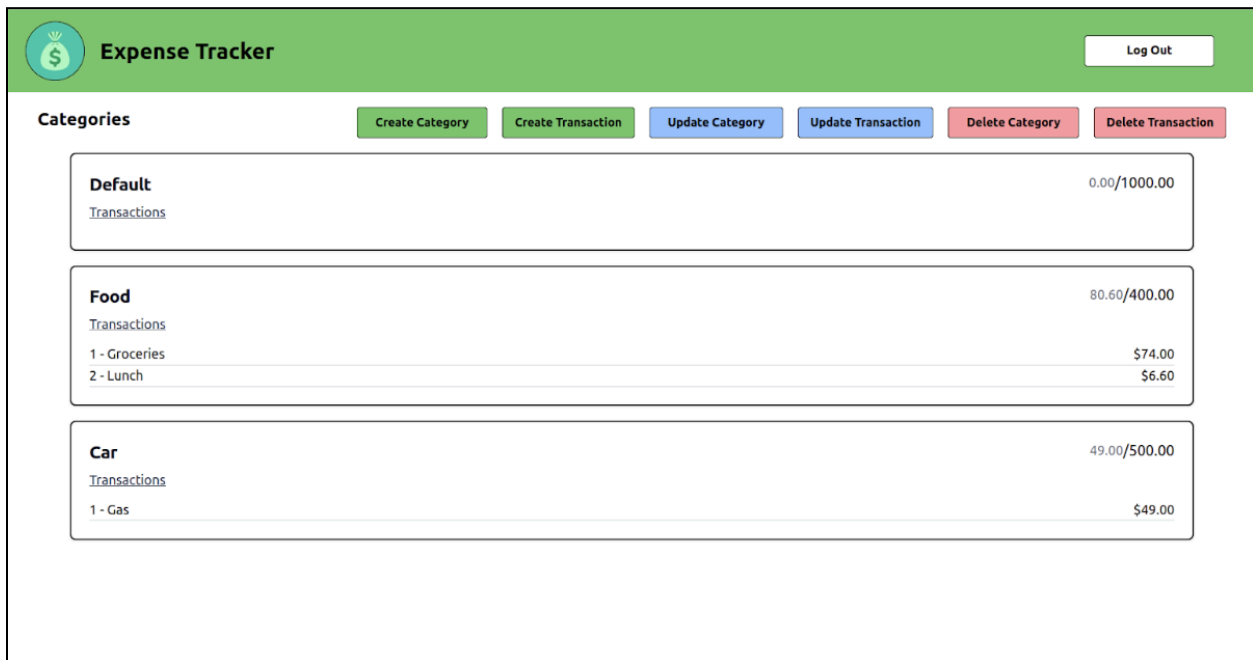


**Expense Tracker** Log Out

**Categories** Create Category Create Transaction Update Category Update Transaction Delete Category Delete Transaction

Category	Current Balance	Limit
<b>School</b> <a href="#">Transactions</a>	150.00	1000.00
1 - Books \$150.00		

Category	Current Balance	Limit
<b>Food</b> <a href="#">Transactions</a>	16.55	200.00
1 - McDonalds \$5.00		
2 - Lunch \$3.99		
3 - McDonalds \$7.56		



**Expense Tracker** Log Out

**Categories** Create Category Create Transaction Update Category Update Transaction Delete Category Delete Transaction

Category	Current Balance	Limit
<b>Default</b> <a href="#">Transactions</a>	0.00	1000.00

Category	Current Balance	Limit
<b>Food</b> <a href="#">Transactions</a>	80.60	400.00
1 - Groceries \$74.00		
2 - Lunch \$6.60		

Category	Current Balance	Limit
<b>Car</b> <a href="#">Transactions</a>	49.00	500.00
1 - Gas \$49.00		

The third bug revolved around how we'd check track of a user being logged in or not. Originally, we would have a global variable that would store the username and a boolean, but that became an issue when a user would not log out and access different

pages using URLs. Then if they try to go back to their homepage using the “/home” URL, they would be welcomed with an “Unauthorized” message. This felt like a bug because in this case, a user should be able to access their homepage because they haven’t technically logged out. In order to fix this, we replaced the global variable with Flask’s Session. Switching over to session storage allows data to persist for the current session of the application which is quite convenient because we were able to set variables in session storage when a user logs in and clear it when they log out. In addition to addressing the bug, we are also providing more security by utilizing session storage over global variables. Below we can see the “Authorized” message that was mentioned.

**Unauthorized**

The server could not verify that you are authorized to access the URL requested. You either supplied the wrong credentials (e.g. a bad password), or your browser doesn't understand how to supply the credentials required.

## Installation / Setup Guide

To install the software locally, first clone the git repository from <https://github.com/dharbo/490Project> using the command in a terminal:

```
git clone https://github.com/dharbo/490Project.git
```

After cloning from the git repository, change directories to the root of the project, and make sure that python and pip are installed onto the local machine used to run the software.

Now, we will install required dependencies for this software to run. The steps shown are targeted towards machines running Ubuntu, but it should be similar on other operating systems. You may want to use python virtual environments to avoid installing Flask on your machine globally. To create a python virtual environment, follow the steps at “Virtual Environments” ([here](#)). This is not required, but might be something to think about before installing these dependencies.

Flask is our main dependency which hosts our application. Official documentation, including installation can be found at “Flask” ([here](#)). To install it, run the following command:

```
pip install Flask
```

Next is Node.js and npm, which will be used to install another dependency and run the application. There are a few different ways to install these on Ubuntu, according to “DigitalOcean” ([here](#)), but the first way is the simplest. To install them, run the following commands:

```
sudo apt install nodejs
```

```
sudo apt install npm
```

After those dependencies are installed, we can now install Tailwind CSS by following the “official documentation” ([here](#)). Tailwind CSS has already been configured in the GitHub repository, so we simply need to install it by running the following command:

```
npm install -D tailwindcss
```

These are all the dependencies required to run the software locally. Once they are installed, open a terminal and make sure that the current directory is the project's root directory. In order to start the application, simply run the following command:

```
npm run dev
```

In the terminal where the above command was run, a log should specify where the application is being hosted. It is usually [localhost:5000](#), but it may be different based on how your machine is set up. To access the application, open a browser and enter the address that was logged. You will be directed to the landing page and use the application.

## Run / Operations Books

If an issue or failure arises with the software application and must be addressed immediately, then the operations personnel can simply stop the server that the application is running on to avoid further harm. This would affect users who are currently using the application, which is not ideal but necessary to ensure the software maintains its integrity. During this downtime, the problem should be found and addressed promptly. After an update has been made, the server can be started back up and users may resume with their use of the software.

The primary storage of data that the application uses is a database. The state of the database, along with its contents, should be backed-up every day or so. This is to prevent a major loss of data if a disaster were to occur and the database's contents are lost. Another thing that should be backed-up is the code for the software. Luckily, GitHub is able to handle that as well as keep track of previous changes.

It should not be too difficult to recover from a disaster-level issue if the database and code are backed up. This is because the application does not rely on too many different parts. Restoring the database with the most recent backup would be simple and would limit the implications of a disaster if it is related to the database. If the disaster might have been introduced with a new update, it could be possible to revert to a previous version while a patch is being worked on since GitHub would hold the different versions.

## SLIs/SLOs/SLAs

This software will consist of a few SLIs, SLOs, and SLAs. Before defining the SLIs, or Service-Level Indicators, it is important to establish what an SLI is. According to Carter and Judkowitz (2018), a SLI refers to a metric that can be measured based on a service's capabilities. The first SLI that this software will have is for response time when performing actions that involve a database query, which will help keep track of how efficient the database queries are or if a solution such as load balancing needs to be implemented. The second SLI is to track the error rate that users would encounter. Situations that are considered as errors are when the software fails or has a bug. This error rate SLI will help track how many software-related errors occur and let us know when resources need to be shifted to fix them. The third SLI is to track availability, or how often the software can be used. These three SLIs will help us focus on what needs to be measured to ensure that the software is performing well and is healthy.

Moving on to Service-Level Objectives, a SLO can be defined as the target value for a given SLI (IBM, 2023). With this in mind, the SLOs should be reasonable goals so that not every resource is focusing on meeting these SLOs, and can instead focus on improving the software in other ways. For the first SLO, which corresponds to the response time SLI, the target value is to have the software process and return data in less than 1 second. This might seem like high value, but happens to be reasonable due to resource constraints. The second SLO will align with the error rate SLI with the goal of errors occurring less than 1% of the time. As for the third SLO that relates to the availability SLI, the goal is to have the software ready for users at least 99% of the time.

These SLOs should be kept as a promise to users and to create accountability we establish expectations between us and users in SLAs, or Service-Level Agreements.

SLAs simply define what will be delivered to the user and expectations for performance (Atlassian, 2023). With this in mind, the product is meant to be an web application that users can create an account for, log on, and input and categorize their expenses. The software should also meet or exceed the goals defined in the SLOs. So, the software's response time SLI must be less than 1 second, the software's error rate SLI must be less than 1%, and the software availability SLI must be at least 99%. The users will be satisfied if these target values are met. Though, if these values are not met there needs to be a consequence to compensate the users. In order to do so, users will receive a partial refund when any of the SLIs do not meet the target values mentioned on a monthly basis.



# User Manuals

## Landing Page

When the user accesses the application, they will most likely find themselves on the landing page. Here, a user will have two options: create an account or login. If a user already has an account, they may open a new account but they will not be able to link information between multiple accounts. Information in one account belongs to that account only and the purpose of this is to ensure data privacy. Once the user has made a selection, they may click on the corresponding button displayed on this page.

## Signup Page

If a user clicks on the “Create Account” button from the landing page, they will get directed to the signup page. Here, a user may enter a username and a password, and click on the “Create Account” button to create an account. If successful, their account will have been created and the user will be directed to their home page. If unsuccessful, a message will pop up stating that the inputted username is already taken. Unfortunately, the only way to resolve this is to enter a different username that has not been used already. There is also a link with the text “Already have an Account?” which allows the user to easily navigate to the login page if they need to.

## Login Page

If a user clicks on the “Create Account” button from the landing page, they will get directed to the login page. Here, a user may enter a username and a password, and click on the “Login” button to log into their account. If successful, they will get directed to the home page. If unsuccessful, a message will pop up stating that the username or password inputted was incorrect. If an account has already been made, make sure to enter the correct username and password when trying to login again. Similar to the signup page, there is a link with the text “Don’t have an Account yet?” which allows the user to easily navigate to the signup page if they need to.

## Home Page

After creating an account or logging in, the user will arrive at their home page, which is where most of the application’s core functionalities are.

Once here, a user can create a new category by clicking on the “Create Category” button. A popup will appear asking for a category name and budget, and after entering the information, the “Create” button can be clicked to create the new category. If successful, a new category box will be generated with the inputted name and budget.

To update an existing category, a user can click on the “Update Category” button. A popup will appear asking for the old category name, new category name, and new budget. When the information is entered and the “Update” button is clicked, the old category will be updated with the new category name and budget if successful.

If a user would like to delete a category, they can click on the “Delete Category” button. A popup will appear asking for a category name. Once a category name is

inputted and “Delete” is clicked, the selected category as well as all transactions under that category will be deleted if successful.

When a user wants to create a new transaction, they can click on the “Create Transaction” button. A popup will be displayed asking for a category name to put the transaction under, description for the transaction, and the money spent for the transaction. After this information is inputted and the “Create” button is clicked, the transaction’s description and money spent will be added under the specified category if successful. Note that a unique ID for the transaction in the specified category will be generated.

If a user would like to update an existing transaction, they can click on the “Update Transaction” button and a popup will appear asking for a category name, old transaction ID, new description, and new money spent. After clicking on the “Update” button, the old transaction will be updated to have the new description and new money spent if successful. Note that the transaction ID for the transaction remains the same.

To delete a transaction, a user can click on the “Delete Transaction” button which will display a popup. This popup will ask for a category name and a transaction ID. After entering the required information, the user can click the “Delete” button. If successful, the transaction with the specified transaction ID under the specified category will be deleted.

Note that for the six functionalities mentioned above, if the action turns out to be unsuccessful, a message will appear at the top of the page. The message will contain information about possible reasons for the actions not being completed. Typical reasons

are non-existing category names or invalid numeric input because numeric inputs need to be 0 or positive values.

Additional features include the ability to log out by clicking the “Log Out” button which will log out the user from their account and direct them to the landing page. If a user does this, they would need to login again to access their home page. Another feature is that the sum of transactions in a category is taken and displayed next to the allotted budget. Also, this sum gets automatically updated if changes are made to the transactions.

The last feature, which is on all pages, is that a user can be directed to the landing page by clicking on the application’s logo or name.

## Conclusion

The proposal made for this project was to create a web application so that users can view their monthly expenses while allowing personal customization such that they can create their own expense categories. The idea for this application came to mind due to the state of the economy and to help people stay afloat financially during these times.

This project made use of many different technologies. Starting off with the languages used, Python was the main language used and there was also a little bit of JavaScript involved. There was also HTML for the application's web pages and CSS through the use of the Tailwind CSS framework. In addition to the Tailwind CSS framework, this project largely relies on the Flask framework for handling requests. SQLite was the database used to store data for users, their categories, and their transactions. These are the main technologies that the application uses, but there are also some that helped during development. The most important of these were Git and GitHub which were used for version control. Linters, such as PyLint, were used to improve code, and Visual Studio Code was used as the Integrated Development Environment. Node.js, npm, and pip were used to install packages and enhance development.

Moving on to methodologies, Agile had a big influence on development. This project was developed by following Agile principles and taking an iterative approach. Sprints were defined as 2 weeks and tasks were done on a sprint-by-sprint basis, which allowed a lot of flexibility to adapt if requirements changed. Also, a few metrics were tracked during development to see whether the project was on track and successful. These metrics were velocity, burndown, and defects. There were a few testing

methodologies used such as unit testing, performance testing, and security testing. For this project, unit testing primarily focused on individual functionalities to ensure that they worked as intended. Performance testing mainly revolved around timing database queries and making sure that they took a reasonable amount of time. Security testing was concerned with user authentication and data privacy in regards to the database. Several more methods were used on the technical side such as requirements engineering, software architecture and design, UX design, network analysis, and threat modeling.

To conclude, this project was a huge success. It has its main requirements met and it functions as expected. Users can create an account, login, logout, create categories, update categories, delete categories, create transactions, update transactions, and delete transactions. These were the primary goals of this project, and the current user interface is much cleaner than what was initially created in the prototype. As for testing, results were positive since unit, performance, and security tests passed and were optimistic. As for recorded metrics, development was mostly ontrack besides two sprints where all tasks were not completed. Despite this, the project was finished by the end of the last sprint, and based on the Defects metric, there were no known bugs by the end of the last sprint.

## References

- Atlassian. (2023). *What is an SLA? Learn best practices and how to write one*.  
<https://www.atlassian.com/itsm/service-request-management/slas>
- Butani, A. (2020, December 16). *5 essential patterns of software architecture*. Red Hat.  
<https://www.redhat.com/architect/5-essential-patterns-software-architecture>
- Carter, M., & Judkowitz, J. (2018, July 19). *SRE fundamentals: SLAs vs SLOs vs SLIs*. Google Cloud.  
<https://cloud.google.com/blog/products/devops-sre/sre-fundamentals-slis-slas-and-slos>
- Centers for Medicare and Medicaid Services. (2023, June 1). *CMS Threat Modeling Handbook*. <https://security.cms.gov/policy-guidance/threat-modeling-handbook>
- IBM. (2023, September 6). *Service level objectives (SLO)*.  
<https://www.ibm.com/docs/en/instana-observability/current?topic=instana-service-level-objectives-slo>
- Loem, M. (2021, March 23). *What is Network Analysis?*. Medium.  
<https://towardsdatascience.com/network-analysis-d734cd7270f8>
- Lucidchart. (2023). *UML Use Case Diagram Tutorial*.  
<https://www.lucidchart.com/pages/uml-use-case-diagram>
- Microsoft. (2023). *Understand versioning of artifacts*.  
<https://learn.microsoft.com/en-us/training/modules/implement-versioning-strategy/2-understand-versioning-of-artifacts>
- Oracle. (2023). *What is a database administrator (DBA)*.  
<https://www.oracle.com/database/what-is-a-dba/>

Pankaj. (2022, August 3). *Composite Design Pattern in Java*. DigitalOcean.

<https://www.digitalocean.com/community/tutorials/composite-design-pattern-in-java>

Visual Paradigm. (2023). *UX Design: Wireframe vs Storyboard vs Wireflow vs Mockup vs Prototyping*.

<https://www.visual-paradigm.com/guide/ux-design/wireframe-vs-storyboard-vs-wireflow-vs-mockup-vs-prototyping/>