1st: Purpose:

The purpose of a house budgeting website is to help individuals and families keep track of their income and expenses and manage their finances effectively. The website allows users to create and manage a budget, track their spending, and analyze their financial situation. This helps users to better understand where their money is going and make informed decisions about their finances. The website provides users with tools and resources to plan and save for their financial goals and helps them to avoid overspending and debt. Ultimately, the goal of a house budgeting website is to empower users to take control of their finances and achieve financial stability and security.

2nd: Requirements:

User requirements:

Functional Requirements:

1. Authentication for different Users Registration.
2. Stay logged in for a year.
3. Editing profile and budgeting settings.
4. Budget Tracking & Analysis
5. Expense Tracking.
6. Filtering purchasing.
7. Financial Limit & Goal Setting.
8. Financial Reporting.
9. Storing old budgeting details.

Non-Functional Requirements:

1. Security: protect user data.
2. Performance: fast and responsive.
3. Usability: easy to use and user friendly.
4. Scalability: able to handle large amounts of data and users as the app grows.
5. Compatibility: compatible with different devices and browsers.

System requirements:

Functional Requirements:

1.1 User authentication system and interfaces to allow users to create their accounts and log in securely.

1.2 User profile management system to allow users to manage their account information and preferences.

2. Session management system to keep the user logged in for a specified period.

3. User profile management system to allow users to update their personal information and budgeting settings.

4.1 Budget tracking system to allow users to create and manage their budgets.

4.2 Budget analysis system to provide users with detailed reports and visualizations of their budgets.

5. Expense tracking system to allow users to track their expenses and categorize them for better analysis.

6. Purchase filtering system to allow users to filter their purchases by date, category, or other criteria.

7. Goal & Limit setting system to allow users to set financial goals and track their progress towards them.

8. Reporting system to generate reports and visualizations to help users understand their financial situation and progress towards their goals.

9. Data storage system to store old budgeting details for future reference.

Non-Functional Requirements:

1.1 Secure authentication system to protect user data, including login credentials and financial information.

1.2 Secure data storage system to protect user data from unauthorized access.

2.1 Fast and responsive application design to minimize lag and downtime.

2.2 Efficient database design to handle large amounts of data.

3.1 Scalable database design to handle an increasing number of users and data.

4.1 User-friendly interface design to improve the user experience and ease of use.

4.2 Responsive design to support different devices and screen sizes.

5.1 Cross-browser compatibility to ensure the app works well on different browsers.

5.2 Cross-device compatibility to ensure the app works well on different devices.

3rd: Software Process:

1) Suggested type of software process:

Agile approach (incremental delivery & development).

2) Division of Phases:

1. Login and Registration pages.

2. User Interface.

    2.1. Side Nav Bar on all pages.

    2.2 Home Page
        2.2.1 Display statistics of current month budget
        2.2.2 add new purchase button.
            2.2.2.1 new purchase window

    2.3 Purchasing Page
        2.3.1 display table of purchases
        2.3.2 traverse through tables of previous months

    2.4 Past Month Data Page
        2.4.1 Display statistics of each month budget

2.5 Settings Page
    2.5.1 Edit Profile Details
    2.5.2 Edit budget limit.
    2.5.3 Edit Goal Budget

2.6 Logout

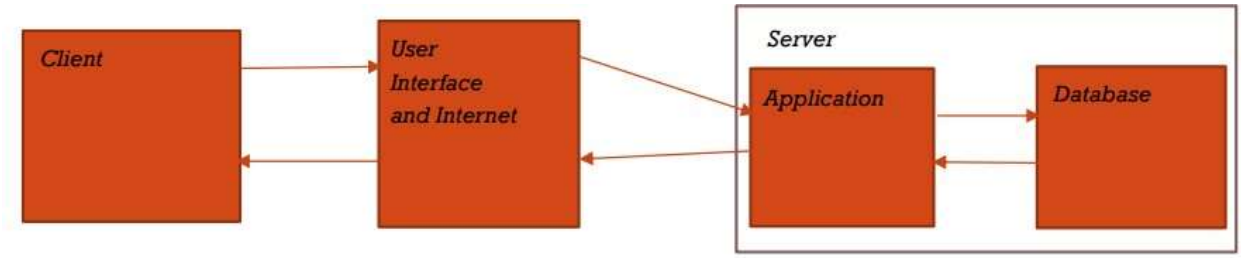
3. Database Building

3.1 Building Relations & defining their domains.
3.2 Building Relationships between different Attributes.
3.3 Apply restrictions.

4th: Architectural Design:

A) System Architecture:

Client-server Architecture:



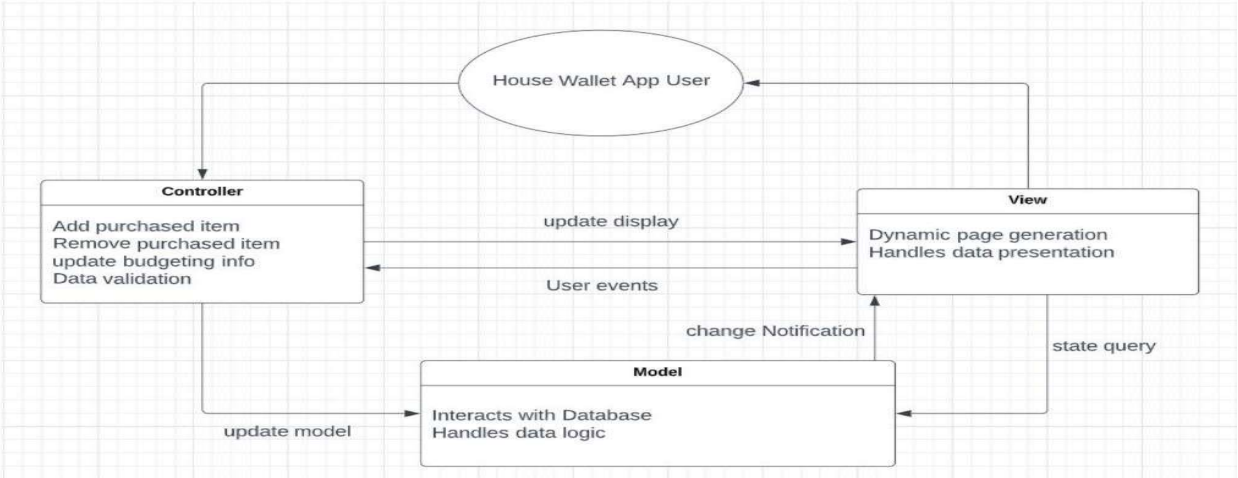
A client-server architecture is a common design pattern used for websites and web applications. This architecture consists of a client, which is typically a web browser, and a server, which provides the necessary resources to the client over the internet.

In the context of a website, the client-server architecture allows users to access the website from anywhere in the world, using any device that can connect to the internet. The client sends requests to the server, which responds with the necessary data and resources to display the website. This architecture also allows for the separation of concerns between the front-end (client-side) and back-end (server-side) of a website, making it easier to maintain and update.

Overall, a client-server architecture provides a scalable and flexible way to deliver web content to users while ensuring efficient management and security of resources on the server side.

B) Application Architecture:

MVC Architecture:



The Model-View-Controller (MVC) architecture is a popular choice for developing web applications because of its many advantages, including:

1. *Separation of concerns:* MVC separates the application's concerns into three main components: the model, the view, and the controller. This makes it easier to manage the application's complexity, as each component can be developed and maintained independently.
2. *Reusability*: Because the components are separated, they can be reused in other applications, making development faster and more efficient.
3. *Scalability*: With MVC, it's easier to scale the application, as each component can be optimized independently.
4. *Testability*: Each component can be tested independently, making it easier to identify and fix bugs.
5. *Flexibility*: With MVC, it's easier to change one component without affecting the others. For example, you can change the user interface without affecting the business logic.

For an in-house budgeting website, using MVC architecture can make development faster, more efficient, and easier to manage. It also makes the application more flexible and scalable, allowing for easier updates and changes as needed. In our case the MVC pattern helped us to separate logic from ui views in our website in the code portion.

5th: Project Backlog: ***Highest priority = 1*

User Story	Initial Estimation	Priority
As a user, I want to be able to authenticate and register different users.	2	1
As a user, I want to track and analyze my budget.	8	2
As a user, I want my data to be secure.	5	3
As a user, I want the app to be fast and responsive.	5	4
As a user, I want to be able to filter my expenses.	4	5
As a user, I want to be able to set financial limits and goals.	4	6
As a user, I want to be able to store my old budgeting details.	3	7
As a user, I want to be able to edit my profile and budgeting settings.	3	8
As a user, I want to be able to stay logged in for a year.	2	9
As a user, I want the app to be easy to use and user-friendly.	2	10
As a user, I want the app to be scalable and able to handle large amounts of data and users.	1	11

User Story	Initial Estimation	Priority
As a user, I want the app to be compatible with different devices and browsers.	1	12
As a user, I want to be able to generate financial reports.	1	13

6th: Design & Implementation:

1) Design Description: UML

1- **Class Diagram**

Class 'User' represents the users of the system and has attributes such as username, password, and email. Class 'Budget' represents the budget of each user and has attributes such as income, expenses, and financial goals. Class 'Purchase' represents the purchases made by users and has attributes such as item name, price, and date of purchase.

2- **Use Case Diagram**

Describes the high-level functions and scope of a system where:
Users can login, register, show budget information, add/edit purchases, view purchase history, and log out.

3- **Activity Diagrams**

The diagrams show the flow of several activities for a user, including logging in, viewing budget information, adding/editing purchases, and viewing purchase history.

4- **State Machine Diagram**

Describes the states that the user can attain as well as the transitions between those states. States in our project denote which page the user is on. States also include states of adding purchase or filtering lists. For each action, a transition from one state to another takes place depending on the user's current activity.

5- **Sequence Diagram**

Type of interaction diagram because it describes how and in what order a group of objects work together. We implemented several sequence diagrams that shows the sequence of actions that occurs when a user interacts with the system, including logging in, viewing budget information, adding/editing purchases, and viewing purchase history.

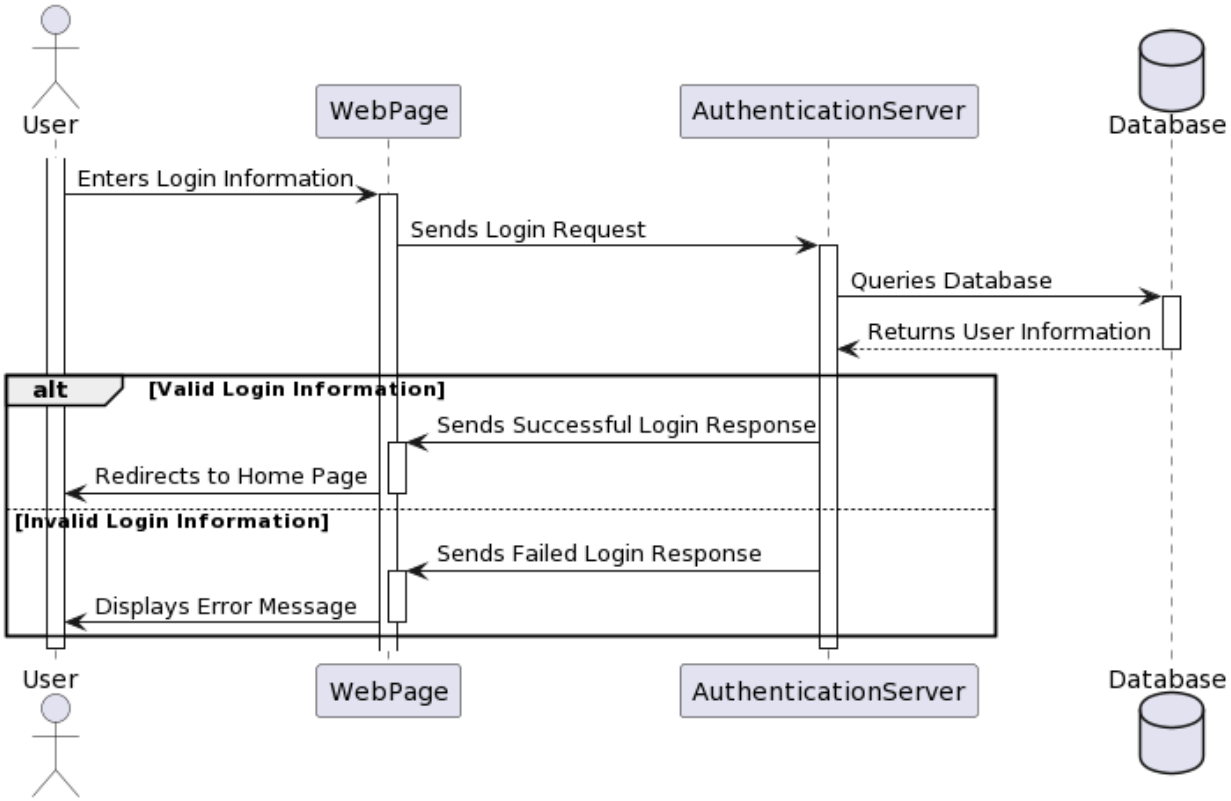
2) Implementation:

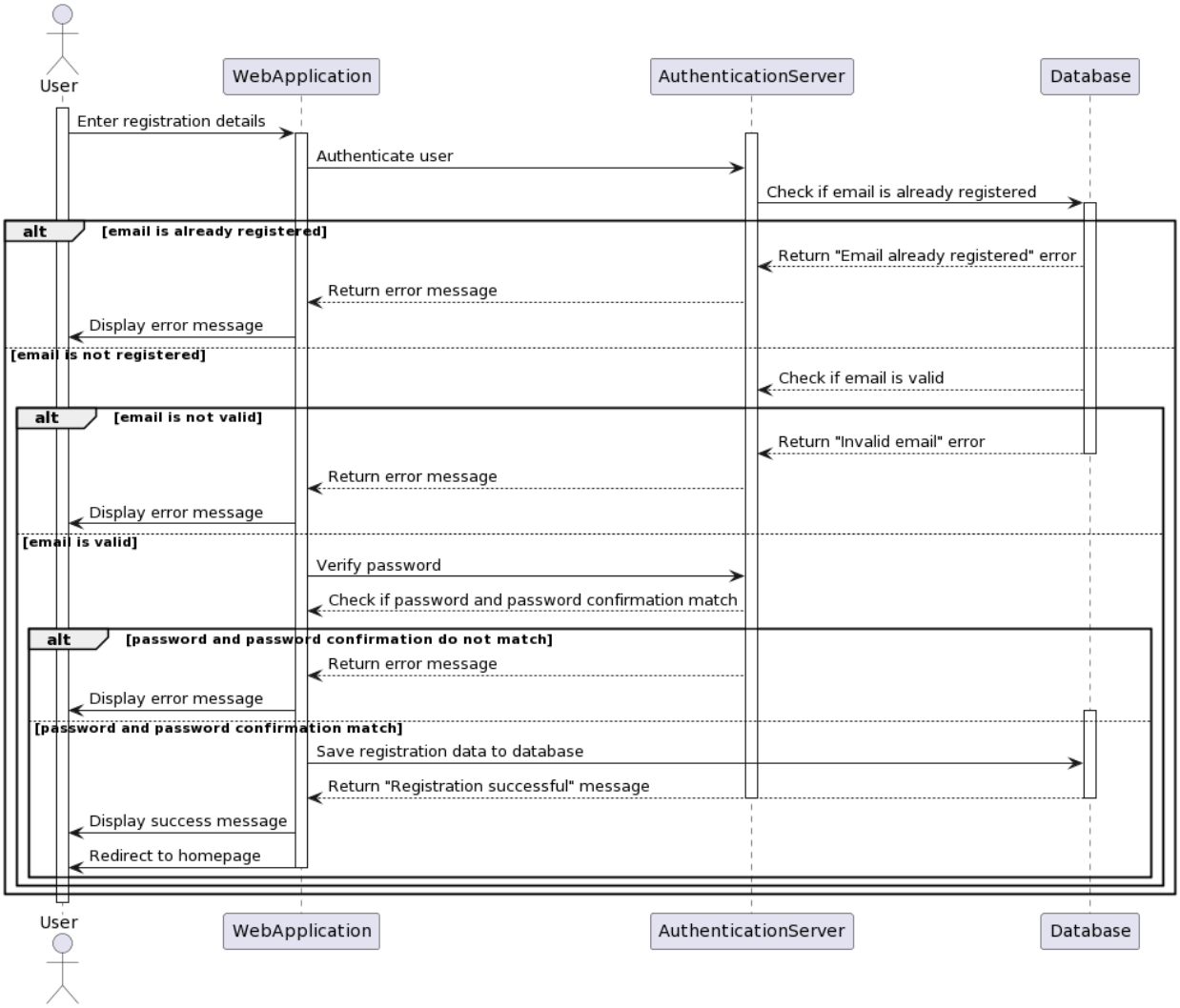
Development environment & Coding:

- HTML5, CSS3 & JavaScript.
- Bootstrap5.
- PHP (PDO paradigm).
- MySQL DBMS.

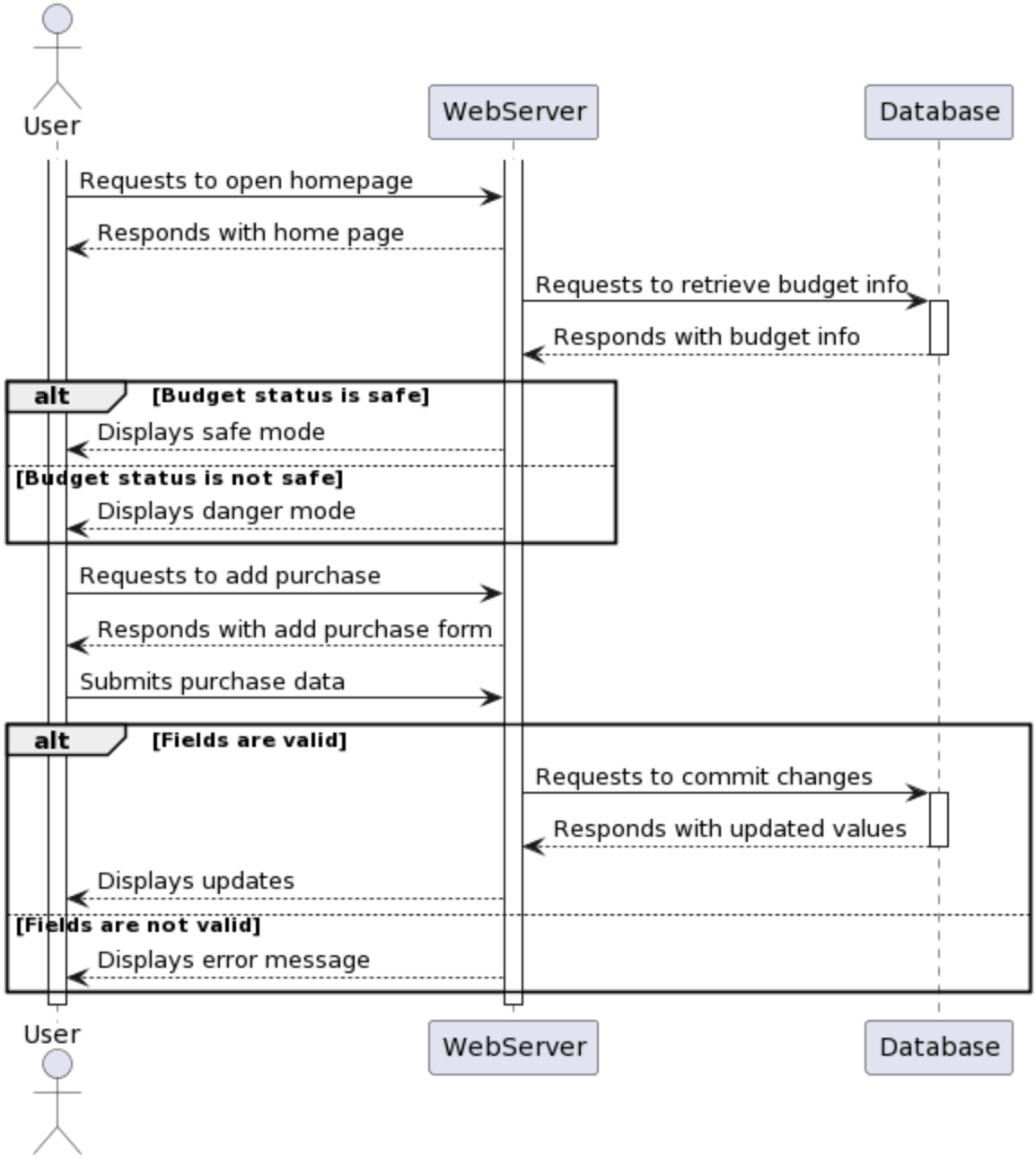
(UML diagrams in the next few pages)

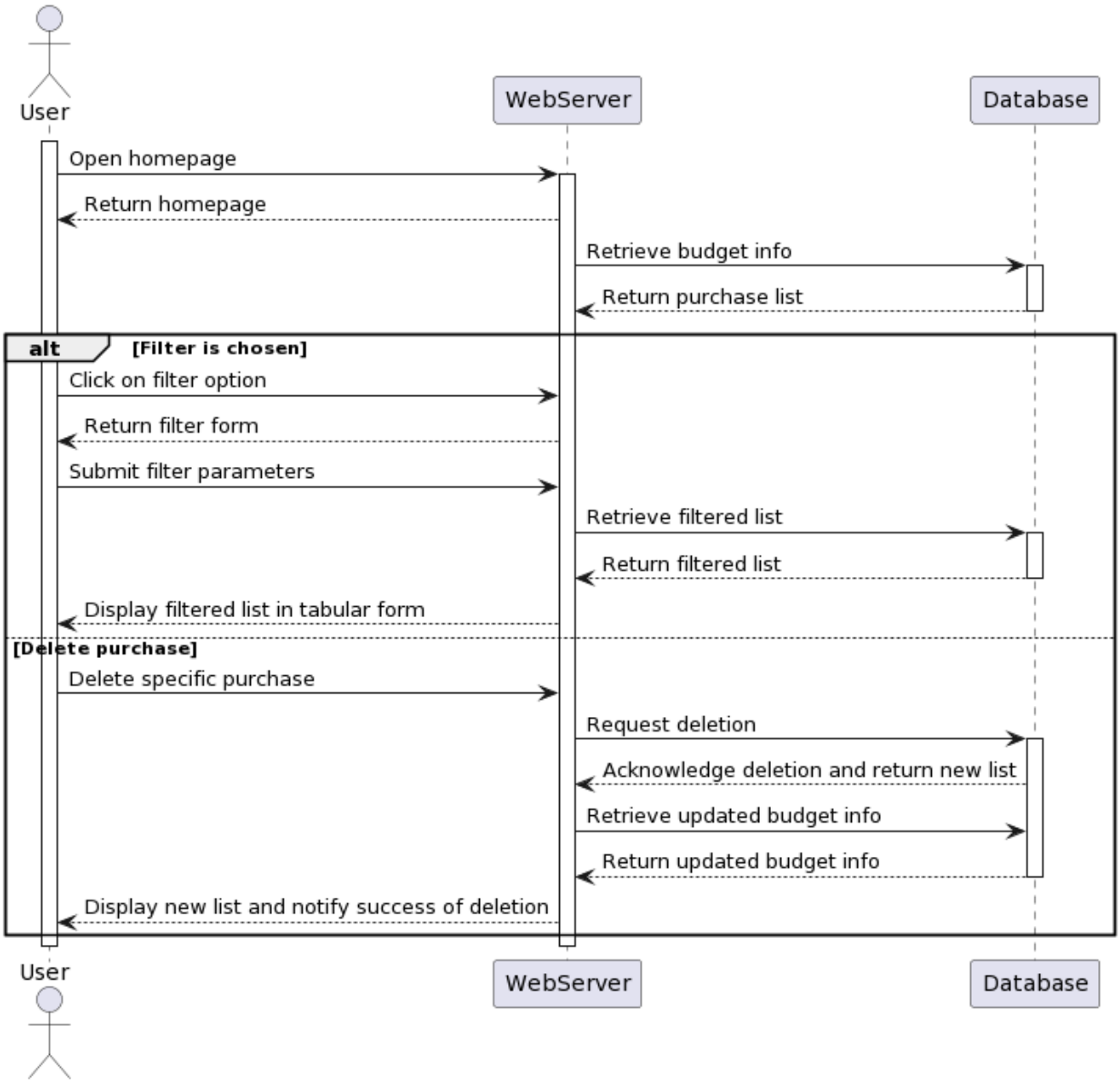
Web Page Login Sequence Diagram

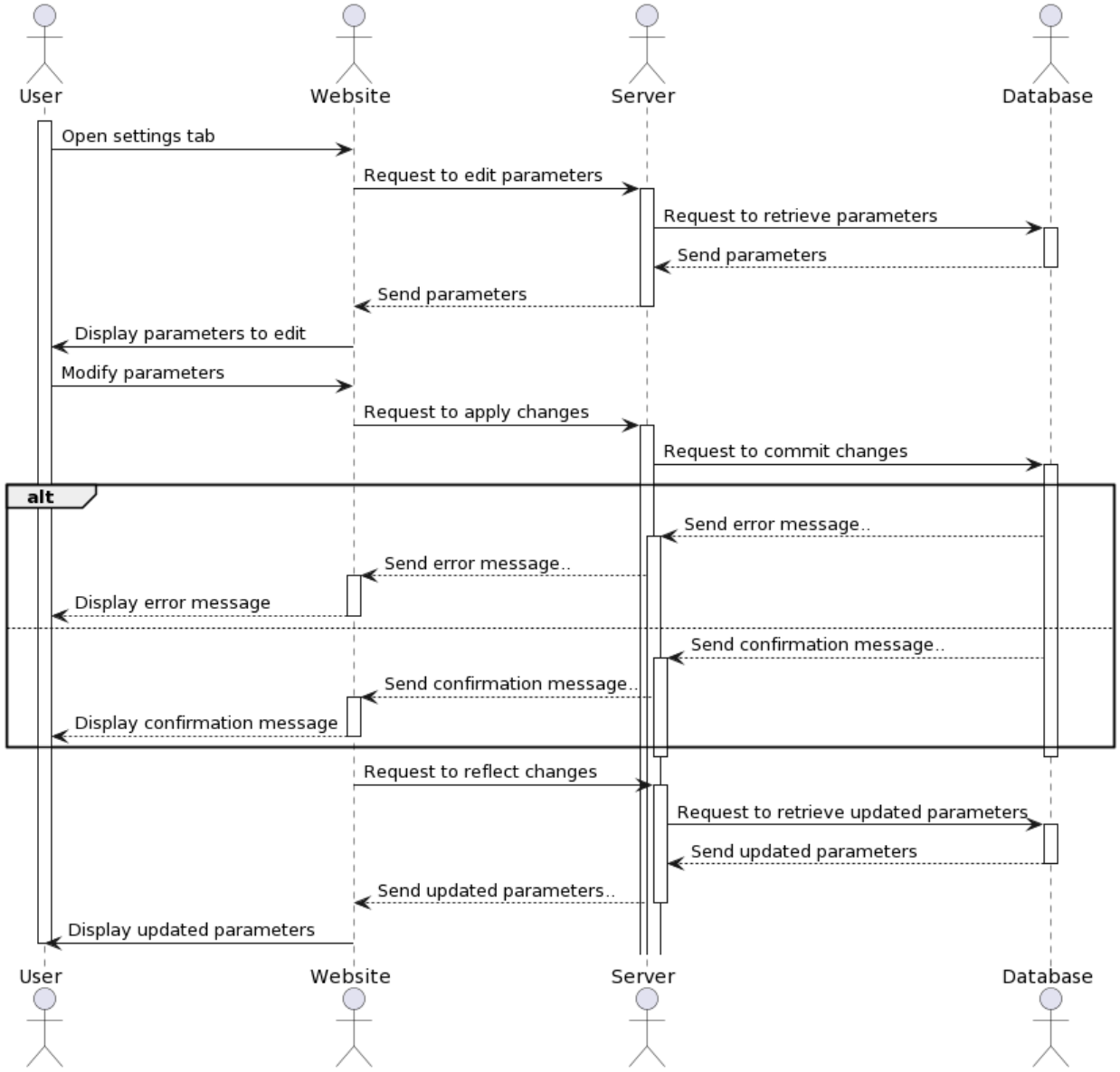



User
WebApplication
AuthenticationServer
Database
Enter registration details
Authenticate user
Check if email is already registered
alt
[email is already registered]
Return "Email already registered" error
Return error message
Display error message
[email is not registered]
Check if email is valid
alt
[email is not valid]
Return "Invalid email" error
Return error message
Display error message
[email is valid]
Verify password
Check if password and password confirmation match
alt
[password and password confirmation do not match]
Return error message
Display error message
[password and password confirmation match]
Save registration data to database
Return "Registration successful" message
Display success message
Redirect to homepage
User
WebApplication
AuthenticationServer
Database

Homepage and Add Purchase Sequence Diagram

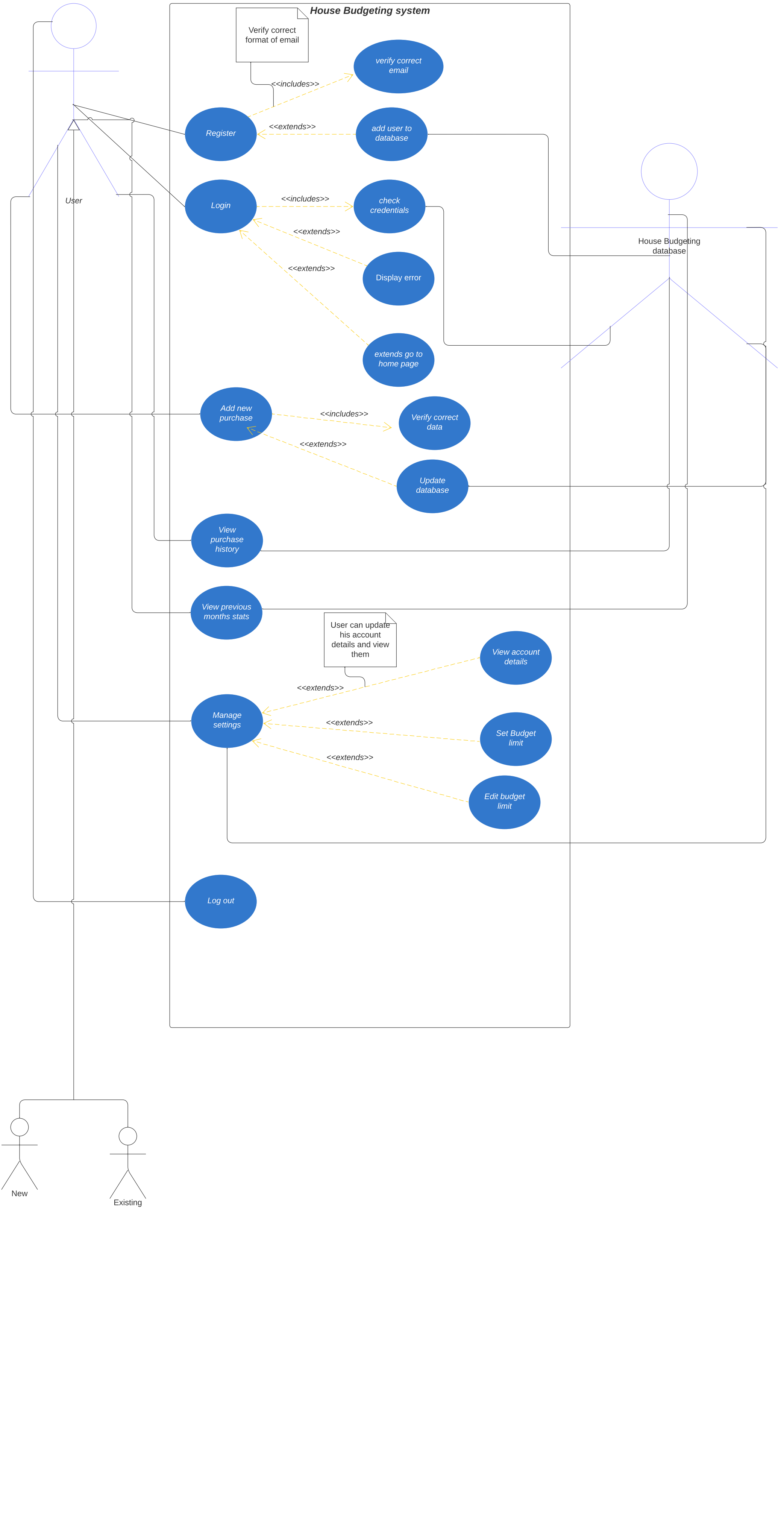


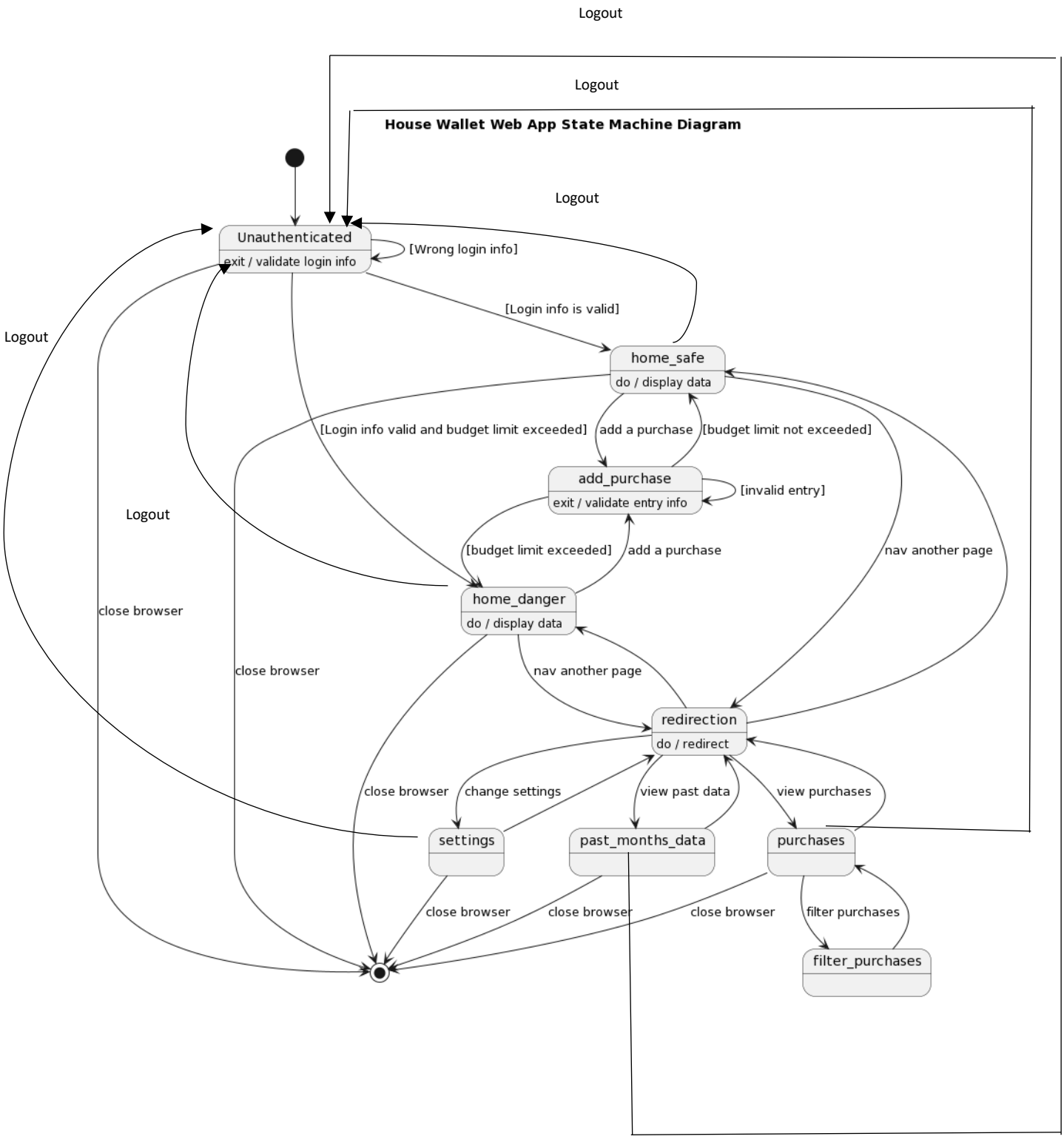

User
WebServer
Database
Open homepage
Return homepage
Retrieve budget info
Return purchase list
alt
[Filter is chosen]
Click on filter option
Return filter form
Submit filter parameters
Retrieve filtered list
Return filtered list
Display filtered list in tabular form
[Delete purchase]
Delete specific purchase
Request deletion
Acknowledge deletion and return new list
Retrieve updated budget info
Return updated budget info
Display new list and notify success of deletion
User
WebServer
Database

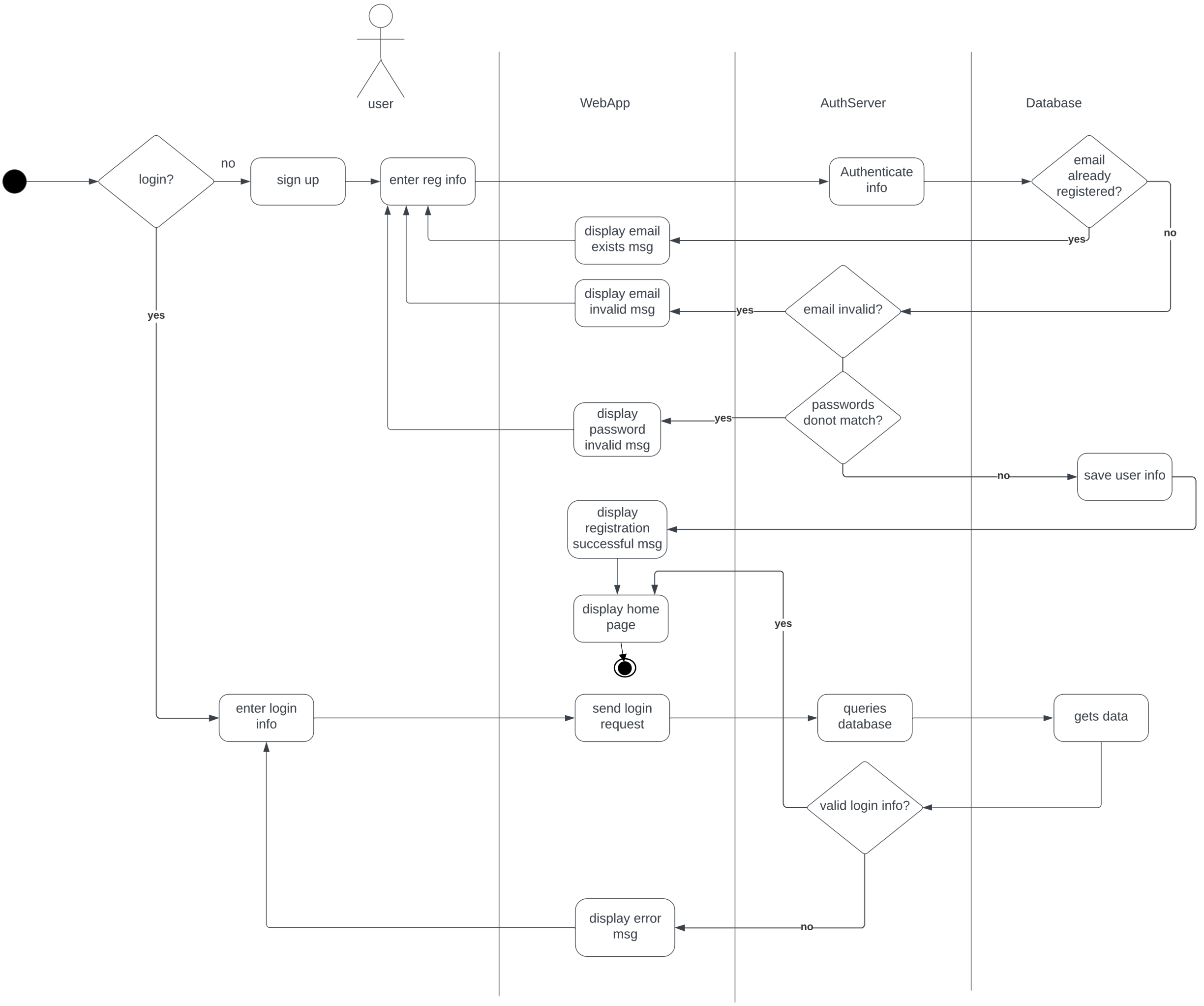

User
Website
Server
Database
Open settings tab
Request to edit parameters
Request to retrieve parameters
Send parameters
Send parameters
Display parameters to edit
Modify parameters
Request to apply changes
Request to commit changes
alt
Send error message..
Send error message..
Display error message
Send confirmation message..
Send confirmation message..
Display confirmation message
Request to reflect changes
Request to retrieve updated parameters
Send updated parameters
Send updated parameters..
Display updated parameters
User
Website
Server
Database

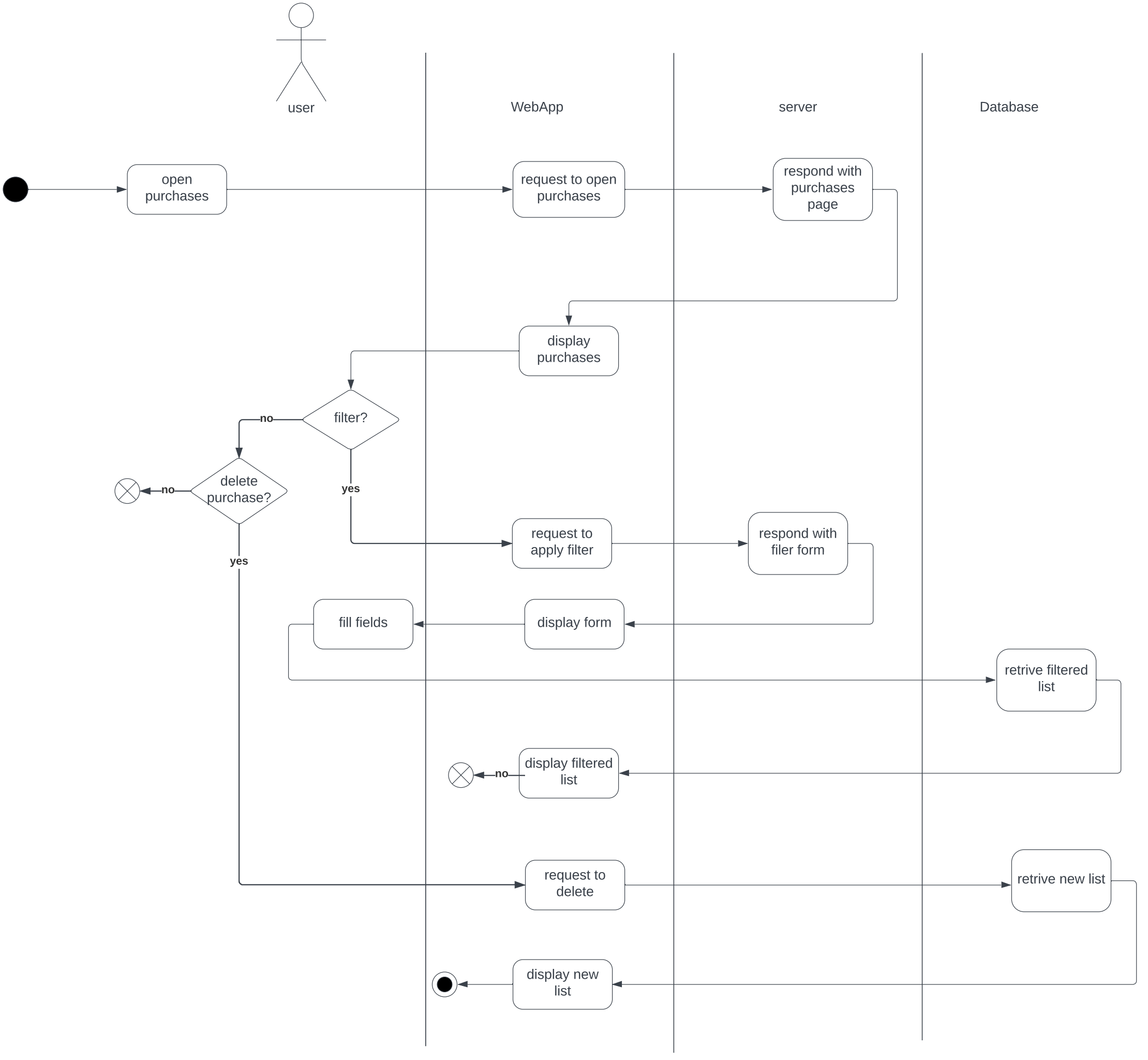
Use case diagram

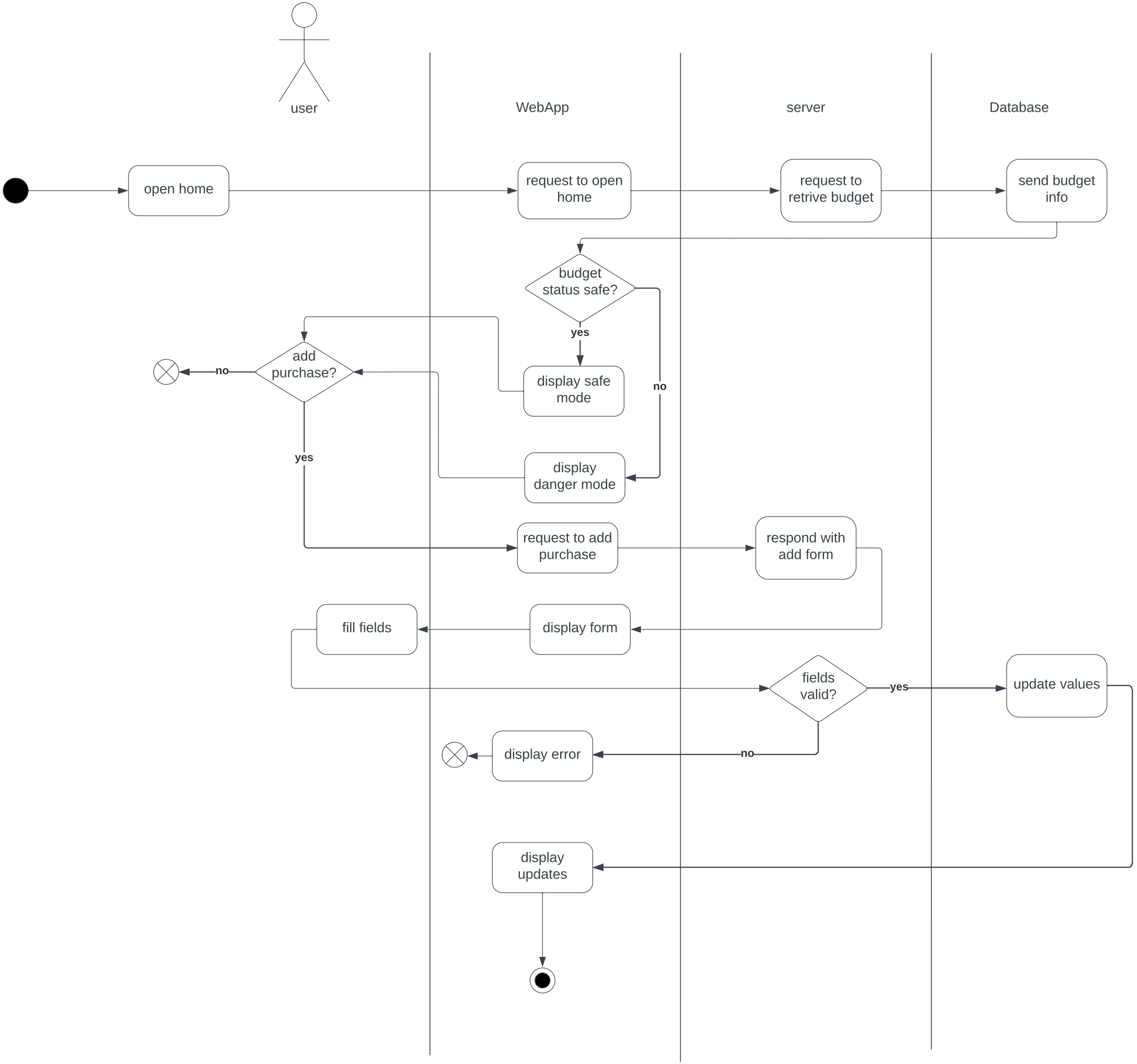
6907-Mostafa Ashraf Elmedany | May 6, 2023

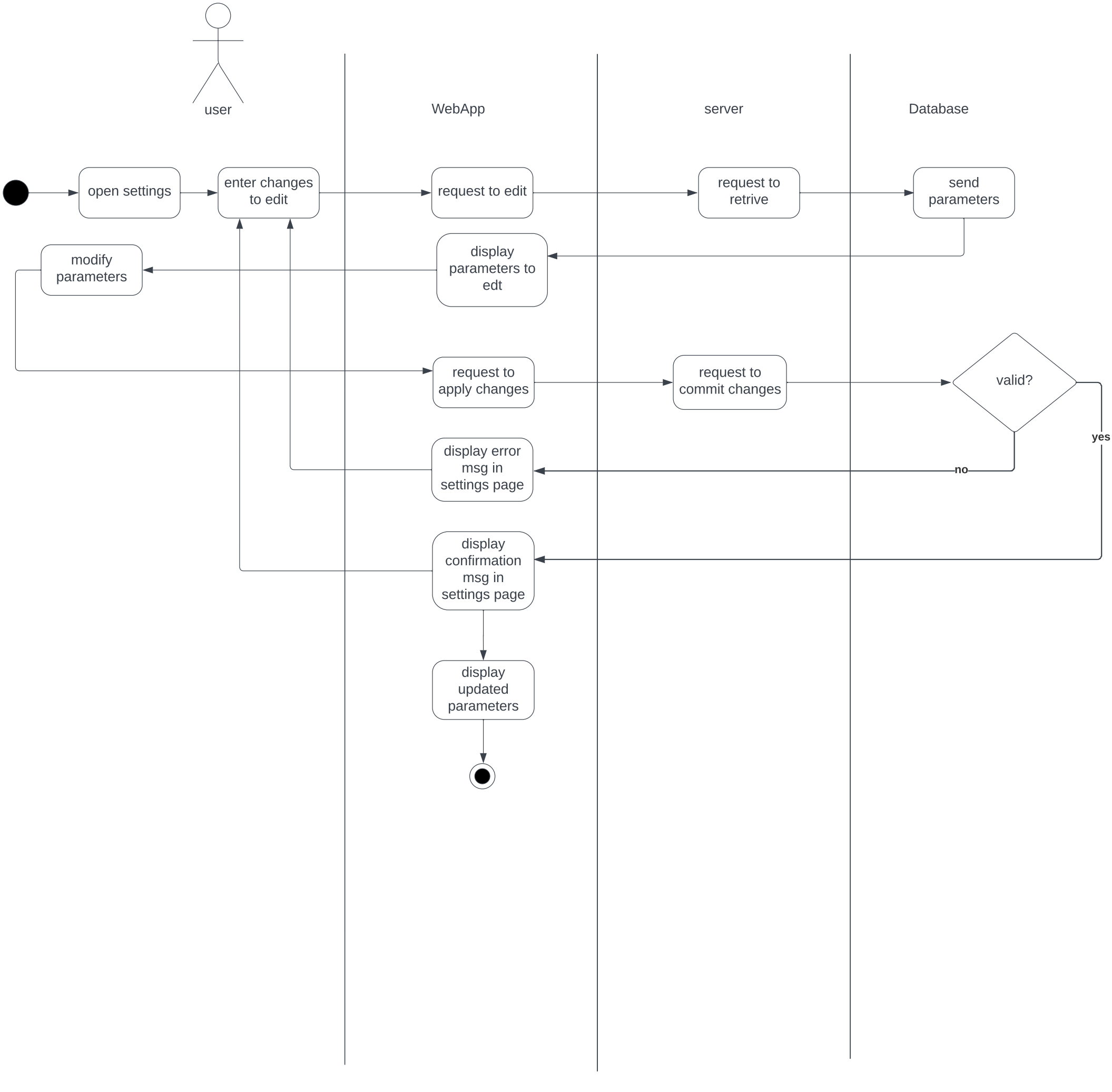


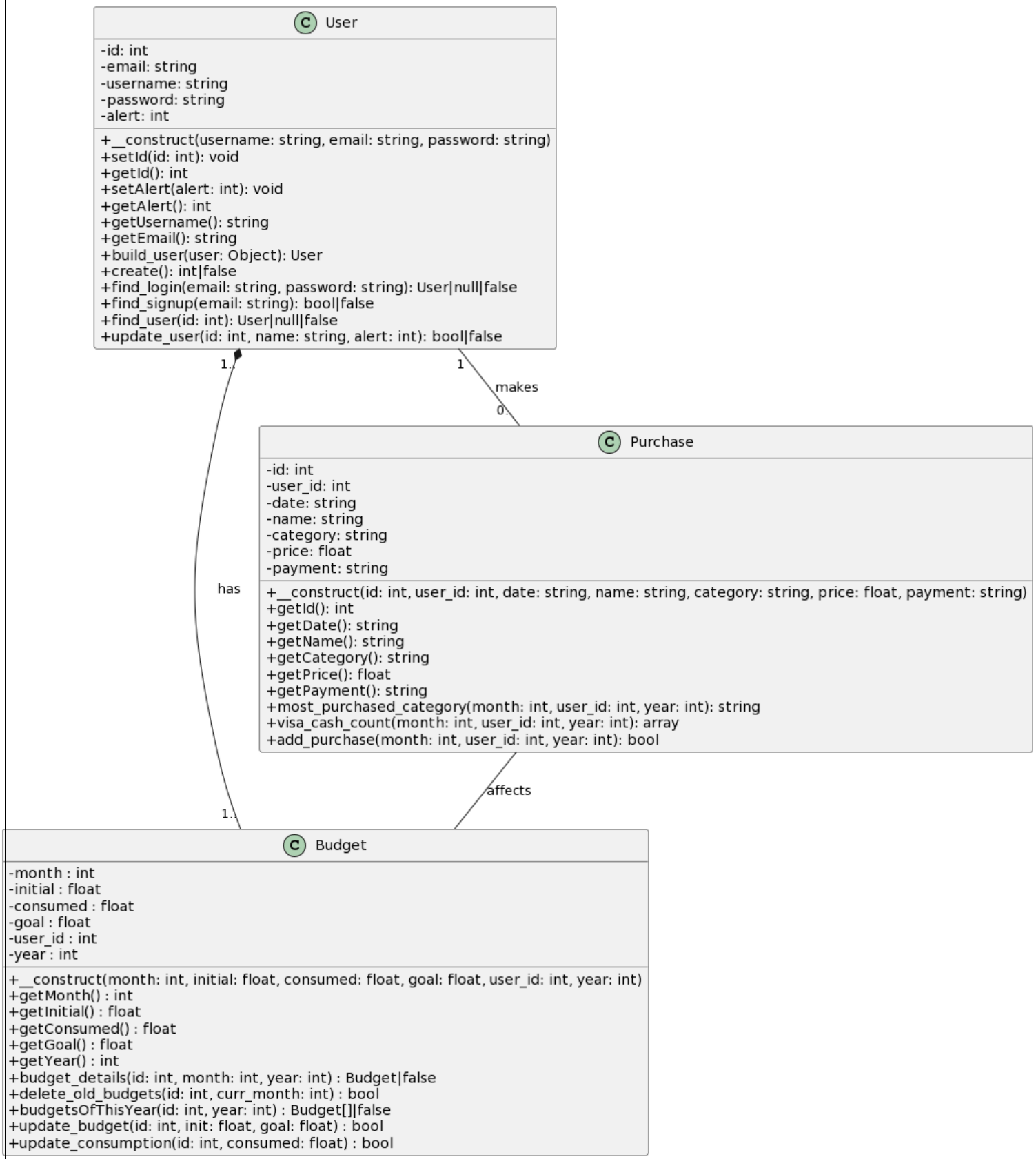

Logout
Logout
House Wallet Web App State Machine Diagram
Logout
Unauthenticated
exit / validate login info
[Wrong login info]
[Login info is valid]
Logout
home_safe
do / display data
[Login info valid and budget limit exceeded]
add a purchase
[budget limit not exceeded]
add_purchase
exit / validate entry info
[invalid entry]
Logout
[budget limit exceeded]
add a purchase
nav another page
home_danger
do / display data
close browser
close browser
nav another page
redirection
do / redirect
close browser
change settings
view past data
view purchases
settings
past_months_data
purchases
close browser
close browser
close browser
filter purchases
filter_purchases


user
WebApp
AuthServer
Database
login?
no
sign up
enter reg info
Authenticate info
email already registered?
yes
no
display email exists msg
display email invalid msg
email invalid?
yes
yes
display password invalid msg
passwords donot match?
no
save user info
display registration successful msg
display home page
yes
enter login info
send login request
queries database
gets data
valid login info?
display error msg
no


user
WebApp
server
Database
open purchases
request to open purchases
respond with purchases page
display purchases
filter?
no
yes
delete purchase?
no
yes
request to apply filter
respond with filer form
fill fields
display form
retrive filtered list
display filtered list
no
request to delete
retrive new list
display new list


user
WebApp
server
Database
open home
request to open home
request to retrive budget
send budget info
budget status safe?
yes
no
add purchase?
no
yes
display safe mode
display danger mode
request to add purchase
respond with add form
fill fields
display form
fields valid?
yes
update values
display error
no
display updates


user
WebApp
server
Database
open settings
enter changes to edit
request to edit
request to retrive
send parameters
modify parameters
display parameters to edt
request to apply changes
request to commit changes
valid?
yes
no
display error msg in settings page
display confirmation msg in settings page
display updated parameters


User
-id: int
-email: string
-username: string
-password: string
-alert: int
+__construct(username: string, email: string, password: string)
+setId(id: int): void
+getId(): int
+setAlert(alert: int): void
+getAlert(): int
+getUsername(): string
+getEmail(): string
+build_user(user: Object): User
+create(): int|false
+find_login(email: string, password: string): User|null|false
+find_signup(email: string): bool|false
+find_user(id: int): User|null|false
+update_user(id: int, name: string, alert: int): bool|false
1..
1
makes
0..
Purchase
-id: int
-user_id: int
-date: string
-name: string
-category: string
-price: float
-payment: string
+__construct(id: int, user_id: int, date: string, name: string, category: string, price: float, payment: string)
+getId(): int
+getDate(): string
+getName(): string
+getCategory(): string
+getPrice(): float
+getPayment(): string
+most_purchased_category(month: int, user_id: int, year: int): string
+visa_cash_count(month: int, user_id: int, year: int): array
+add_purchase(month: int, user_id: int, year: int): bool
has
affects
1..
Budget
-month : int
-initial : float
-consumed : float
-goal : float
-user_id : int
-year : int
+__construct(month: int, initial: float, consumed: float, goal: float, user_id: int, year: int)
+getMonth() : int
+getInitial() : float
+getConsumed() : float
+getGoal() : float
+getYear() : int
+budget_details(id: int, month: int, year: int) : Budget|false
+delete_old_budgets(id: int, curr_month: int) : bool
+budgetsOfThisYear(id: int, year: int) : Budget[]|false
+update_budget(id: int, init: float, goal: float) : bool
+update_consumption(id: int, consumed: float) : bool

7th: Testing:

A) Development Testing:

1) Unit testing:

Test Cases:

1. Login and Sign-Up page:

1.1 Login

1.1.1 Leave either of the fields empty or both:

- Verify that the form cannot be submitted if the email field is left empty.
- Verify that the form cannot be submitted if the password field is left empty.
- Verify that the form cannot be submitted if both the email and password fields are left empty.

1.1.2 Invalid emails - tested example (mostafa@mail):

- Verify that the form cannot be submitted if the email address is in an invalid format (e.g. missing the domain name or the @ symbol).
- Verify that the form cannot be submitted if the email address contains invalid characters (e.g. spaces or special characters).
- Verify that the form cannot be submitted if the email address is too long or too short.

1.1.3 Valid email but not registered user:

- Verify that the form can be submitted successfully with a valid email address and password.
- Verify that the user is redirected to a confirmation page after successfully submitting the form.
- Verify that the user's account is created in the database with the correct information.

1.1.4 Valid email but invalid password for registered user:

- Verify that the form cannot be submitted if the user enters an incorrect password for a registered email address.
- Verify that the user sees an appropriate error message indicating that the password is incorrect.
- Verify that the user is not logged in and remains on the login page.

1.1.5 Correct credentials entered:

- Verify that the form can be submitted successfully with a valid email address and password.
- Verify that the user is redirected to a confirmation page after successfully submitting the form.
- Verify that the user's account is created in the database with the correct information.
- Verify that the user is logged in and redirected to the appropriate page.

1.2 Sign Up

1.2.1 Test that all required form fields are present and properly labeled:

- Verify that the page displays fields for the user's name, email, password, and password confirmation.
- Verify that each field is labeled clearly and has a corresponding "required" indicator.

1.2.2. Test that form validation works correctly:

- Verify that the form cannot be submitted if any required fields are empty.
- Verify that the form cannot be submitted if the password and password confirmation fields do not match.
- Verify that the form cannot be submitted if the email address is not in a valid format.

1.2.3. Test that user input is sanitized and validated properly:

- Verify that any input from the user is properly sanitized to prevent SQL injection or other malicious attacks.
- Verify that the email address is unique and has not already been registered before.

1.2.4. Test that the user is properly registered:

- Verify that the user is redirected to home page after successfully submitting the form.
- Verify that the user's account is created in the database with the correct information.

1.2.5. Test that error messages are displayed correctly:

- Verify that the page displays appropriate error messages if any form fields are invalid or if there is a problem with user input.
- Verify that error messages are displayed in a clear and understandable format.

2. Side Nav Bar

2.1 Clicking on link navigates to correct page:

- Verify that clicking on a navigation link takes the user to the correct page.
- Verify that the URL in the browser's address bar reflects the correct page.
- Verify that the correct content is displayed on the page.

2.2 Highlight the correct active page:

- Verify that the active page is highlighted in the navigation bar.
- Verify that the active page is visually distinguishable from other pages.
- Verify that the active page remains highlighted when the user navigates to sub-pages or sub-menus.

2.3 Displaying correct username from cookies:

- Verify that the user's name is displayed correctly in the navbar if user is logged in.
- Verify that the user's name is not displayed if the user is not logged in.
- Verify that the user's name is displayed correctly even after navigating to other pages on the site.

2.4 Correct alignment:

- Verify that the navigation bar is aligned correctly with the rest of the page.
- Verify that the navigation bar does not overlap with other content on the page.
- Verify that the navigation bar is centered properly and does not look distorted on different screen sizes.

3. Home Page:

3.1. Correct username appearing in welcoming header:

- Verify that the user's name is displayed correctly in the welcoming header if the user is logged in.

3.2. Background of remaining budget color adaptation:

- Verify that the background color of the remaining budget changes to green when the consumed amount is less than the specified percentage.

- Verify that the background color of the remaining budget changes to red when the consumed amount exceeds the specified percentage.
- Verify that the correct percentage is used to determine the color of the remaining budget.

3.3. Display budgeting data and check they are retrieved correctly from database:

- Verify that the initial budget is displayed correctly on the home page.
- Verify that the consumed amount is calculated correctly and displayed on the home page.
- Verify that the goal to save is displayed correctly on the home page.
- Verify that the visa purchases are displayed correctly on the home page.
- Verify that the cash purchases are displayed correctly on the home page.
- Verify that the most purchased category is displayed correctly on the home page.
- Verify that the data is retrieved correctly from the database.

3.4. Add purchase component testing:

3.4.1. Add button opens a form for adding a new purchase:

- Verify that clicking on the add button opens a form for adding a new purchase.
- Verify that the form contains all the necessary fields for adding a purchase.

3.4.2. Test for empty fields:

- Verify that submitting the form with empty fields displays an error message.
- Verify that the purchase is not added to the database.

3.4.3. Test for numeric values in price field:

- Verify that submitting the form with a numeric value in the price field adds the purchase to the database.
- Verify that the correct purchase amount is displayed on the home page.

3.4.4. Test for string values in price field (incorrect):

- Verify that submitting the form with a string value in the price field displays an error message.
- Verify that the purchase is not added to the database.

3.4.5. Test for date value:

- Verify that the purchase date is set to the default date (today's date) if no date is selected.
- Verify that submitting the form with a date in the future or from another month is submitted as the default date which is today.
- Verify that the purchase is not added to the database.

3.5. User cannot access the home page if not logged in:

- Verify that the user is redirected to the login page if they try to access the home page without being logged in.
- Verify that the user is not able to access any data on the home page if not logged in.

3.6. Budgeting information updates when purchase is added successfully:

- Verify that when a purchase is successfully added, the consumed amount on the home page is updated.
- Verify that the remaining budget is updated based on the new consumed amount.
- Verify that the most purchased category is updated based on the new purchase.
- Verify that the visa purchases and cash purchases are updated based on the new purchase.
- Verify that the budgeting information is updated correctly in the database.

4. Purchases page:

4.1. Display purchases of current month by default:

- Verify that the purchases displayed on the page are for the current month by default.
- Verify that the purchases are displayed in descending order based on the date.

4.2. Deletion of a record updates budgeting information and display:

- Verify that when a purchase record is deleted, the consumed amount, visa and cash purchases, most purchased category, and remaining budget are updated accordingly.
- Verify that the deleted record is no longer displayed in the table.
- Verify that the deleted record is no longer displayed in database.

4.3. Filter by month and year pair:

- Verify that the user can filter purchases by a specific month and year pair.
- Verify that only the purchases for the selected month and year pair are displayed on the page.

4.4. Filter by other parameters:

- Verify that the user can filter purchases by category.
- Verify that only the purchases with the selected category are displayed on the page.
- Verify that the user can filter purchases by price range.
- Verify that only the purchases within the selected price range are displayed on the page.

4.5. User cannot access the purchases page if not logged in:

- Verify that the user is redirected to the login page if they try to access the purchases page without being logged in.
- Verify that the user is not able to access any data on the purchases page if not logged in.

5. Past Months data page

5.1. Display current month budgeting data by default:

- Verify that the budgeting data displayed on the page is for the current month by default.
- Verify that the budgeting data is retrieved correctly from the database.

5.2. Display all months of current year:

- Verify that all the months' budgeting data for the current year is displayed on the page.
- Verify that the budgeting data for each past month is displayed on the page.

5.3. Filter by year:

- Verify that the user can filter the past months' data by year.

5.4. User cannot access the past months page if not logged in:

- Verify that the user is redirected to the pastmonths page if they try to access the purchases page without being logged in.
- Verify that the user is not able to access any data on the pastmonths page if not logged in.

6. Settings page:

6.1. Retrieve correct values of fields from database:

- Verify that the values of the fields on the settings page are retrieved correctly from the database.

6.2. Email field cannot be modified:

- Verify that the email field is displayed on the settings page but cannot be modified.

6.3. No empty fields can be left:

- Verify that the user cannot submit the form with any empty fields.
- Verify that appropriate error messages are displayed for each empty field.

6.4. Initial budget, alert, and goal to save must be numeric values:

- Verify that the user cannot enter non-numeric values for the initial budget, alert, and goal to save fields.
- Verify that appropriate error messages are displayed for non-numeric values.

6.5. Alert field cannot exceed 100% or be negative:

- Verify that the user cannot enter a value greater than 100% for the alert field.
- Verify that appropriate error messages are displayed for values greater than 100%.
- Verify that appropriate error messages are displayed for negative values.

6.6. Goal to save cannot be greater than initial budget:

- Verify that the user cannot enter a value greater than the initial budget for the goal to save field.
- Verify that appropriate error messages are displayed for values greater than the initial budget.

6.7. Changes accepted and reflected in the database:

- Verify that when the user enters correct parameters in the form and submits it, the changes are accepted and reflected in the database.

6.8. User cannot access the Settings page if not logged in:

- Verify that the user is redirected to the Settings page if they try to access the purchases page without being logged in.
- Verify that the user is not able to access any data on the Settings page if not logged in.

7.LogOut

7.1: Logout successfully

- This test case verifies that when the user clicks on the logout button, they are logged out of the system and redirected to the login page. This can be done by simulating a click on the logout button and then checking that the user is indeed logged out and redirected to the login page.

7.2: User cannot access any page because cookies will be deleted

- This test case verifies that when the user logs out, their session cookies are deleted, and they can no longer access any pages. This can be done by attempting to access a page after logging out and verifying that access is denied.

8. Responsive Test:

8.1 Verify that the website displays correctly on different screen sizes - This can be done by testing the website on different devices or by using a tool like Google Chrome DevTools to simulate different screen sizes.

8.2 Verify that the website's content is still accessible and readable on small screens - This can be done by testing the website on a small mobile device or by using the DevTools to simulate a small screen size.

8.3 Verify that all buttons, links, and other interactive elements are easily clickable and tappable on small screens - This can be done by testing the website on a small mobile device or by using the DevTools to simulate a touch screen.

8.4 Verify that the website's images and media are optimized for different screen sizes - This can be done by testing the website on different devices and checking that images and videos load quickly and don't distort or appear pixelated.

8.5 Verify that the website's fonts and text are readable on different screen sizes - This can be done by testing the website on different devices and checking that the text is legible and doesn't appear too small or too large.

8.6 Verify that the website's forms and input fields are easy to use on small screens - This can be done by testing the website on a small mobile device and checking that input fields are visible and easy to fill out.

B) Release Testing:

We have deployed beta versions for the use of some people and upon their feedback it helped us take care of some unhandled bugs (Validating price range in filter form and negative alert percentage value) and to test the effectiveness of our website.

Note: To achieve automated testing , we can integrate component to system and run all past test cases on the system and ensure that they are passed correctly , this can also be done using junnit software but due to unavailability of junnit software we followed this approach manually as when we integrate a new component to the system we make sure all the test cases that we already passed for previous components are still passed.