Group 13
Penny Perfect
Garrett Hamilton, Praval Pandu, Gregory Roberts, Winnie Kubuanu, Joshua Ogundele

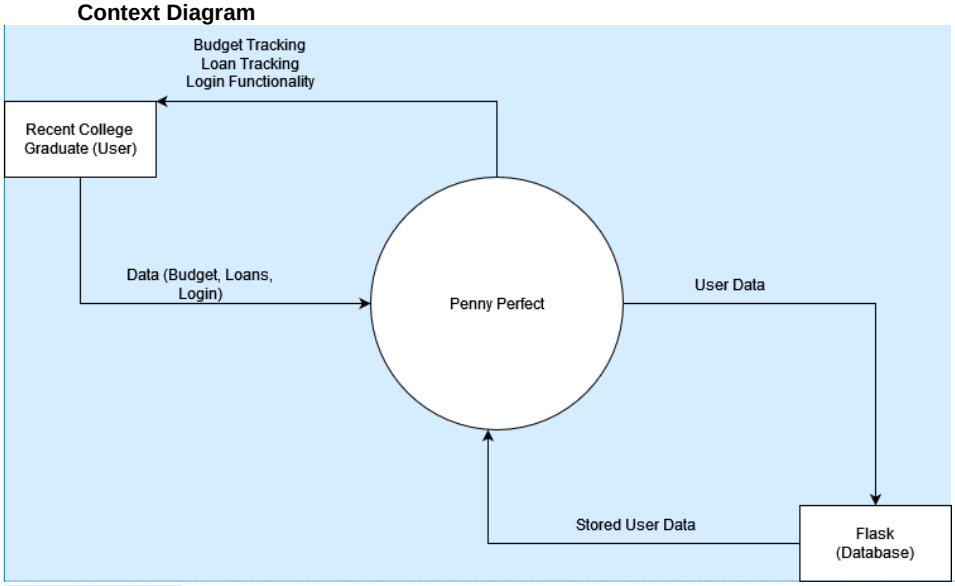
Template: https://docs.google.com/document/d/1WP7ozf2kiCg-gPAqMlzjRF-WDigcq6k386Qz8pgEaHs/edit

1. Project Overview

User Stories:

- As a recent college graduate who wants to budget better, I need an app that can help me assess my financial situation and goals so that I can effectively manage my finances and work towards financial stability and independence.
- As a traveler who is not near a computer all the time, I want to manage my finances conveniently whether I'm using my smartphone, tablet, or computer, so that I can access my financial information and tools wherever I go.
- As a customer I want to be able to upload financial information so that I can accurately track them so that I can conveniently input and organize my financial data, ensuring accurate tracking and comprehensive management of my finances within the app.

Context Diagram



Commented [1]: design document update

Product Backlog

As a recent college graduate who wants to budget better, I need an app that can help me	• **Login functionality**	**Praval Pandu**

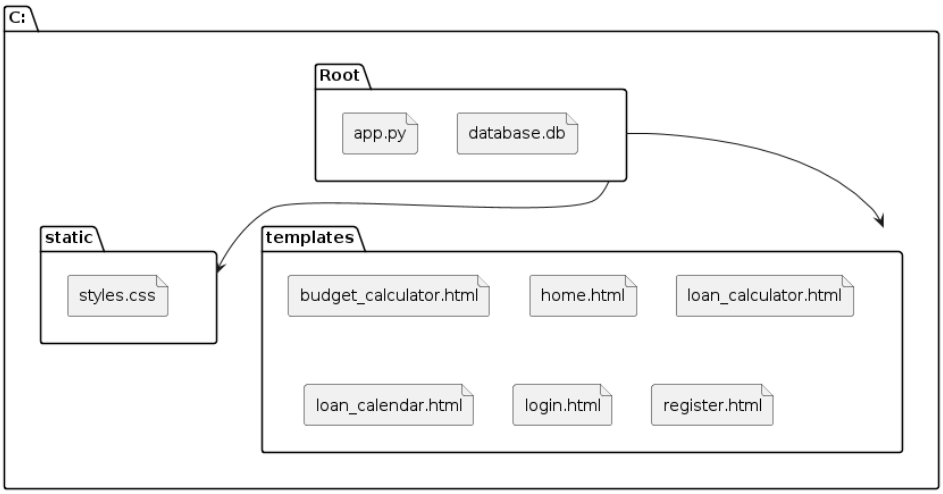
assess my financial situation and goals so that I can effectively manage my finances and work towards financial stability and independence.	• **Budget Chart Maker**	**Garrett Hamilton**
	• **Loan Calendar**	**Gregory Roberts**
As a traveler who is not near a computer all the time, I want to manage my finances conveniently whether I'm using my smartphone, tablet, or computer, so that I can access my financial information and tools wherever I go.	• **App should detect user device and change layout accordingly**	**Winnie Kubuanu**

2. Architectural Overview

We began our project discussing the possible programs and software we would use to complete our financial app. Languages such as Javascript, Java, Python and C were considered as well as software such as React JS, Visual Studio and VS code for a simple HTML format. We chose HTML for simplicity as opposed to using Java or C which would have required our group to learn various new concepts in a short amount of time. For our architectural style, we went with a design that closely resembles component based architecture. We are building each component of each page to be its own functioning piece of software. For instance, the calendar and loan calculator are both independently functioning pieces of software that are integrated together into our overall project to make a cohesive program that achieves our design goals.

Commented [2]: design document update

2.1 Subsystem Architecture



The project's architectural overview has a systematic layout, with packages arranged into levels according to designated roles. The "Root" package, which forms the basis of the application, is at the center of the system. Included in this package are files that are necessary for the operation of the system, such as "database.db" for data storage and "app.py" for the primary application logic.

The system stores "styles.css" and other static files in the "static" package. By keeping styling and presentation files separate from other components, this division complies with the separation of concerns approach.

HTML templates like "budget_calculator.html," "home.html," "loan_calculator.html," "loan_calendar.html," "login.html," and "register.html" are contained in the "templates" package. These templates manage user interface logic and make up the application's presentation layer. Although it isn't stated clearly, the MVC (Model-View-Controller) architecture paradigm is clearly used here. This structure divides the program into three different layers: the View (represented by the HTML templates), the Controller (presumably found in "app.py"), and the Model (managed by "database.db"). By isolating issues and enhancing code organization, this separation improves maintainability.

The relationships between these packages are shown in the subsystem dependency diagram. Because it incorporates components from both "static" and "templates," the "Root" package is dependent on both in order to build the finished application. Stylesheets and other "static" assets do not directly depend on "templates," and vice versa, hence the terms "static" and "templates" are mutually exclusive.

This architecture's design decisions, such as the MVC pattern and layered architecture, provide several benefits. By dividing components into horizontal layers, such as presentation, application logic, and data layers, layered architecture encourages a modular system. As the system expands, this division makes maintenance easier and facilitates better scalability. By encouraging code reuse and segregating concerns, the MVC pattern further improves maintainability.

Overall, this architectural strategy meets industry best practices for software architecture and offers a strong basis for a web application that is flexible, scalable, and maintainable.

2.2 Deployment Architecture

This software will run on a single processor and only needs to communicate with the database as discussed in section 2.3.

2.3 Data Model

We need to store several pieces of information, including user login information, and any information the user inputs. These user inputs, such as budget and loan tracking information, will be stored using Flask and PostgreSQL. PostgreSQL is a relational database. The database schema will consist of a user with a loginID and password. When the user signs up, this will be updated accordingly with the password encrypted. Budget information will be stored as a series of numerical inputs that consists of a budget amount as well as however many expenses the user wishes to enter and track. Loan information will be stored similarly. It will consist of 3 values: loan amount, loan interest rate, and loan duration.

2.4 Global Control Flow

The system follows an event-driven control flow model. The frontend components, consisting of HTML files and ReactJS, are responsible for handling user interactions and generating events. These events are connected to the backend Flask application through HTTP requests.

The Flask application of app.py, views.py, and models.py, waits in a loop for incoming HTTP requests from the frontend. When it gets a request, the view function in views.py is executed, which may interact with the database and respond with the necessary data.

The system does not have any time-dependent or real-time constraints. It operates in an event response type, where it responds to user-generated events without any concern for real-time execution. The system does not employ multiple threads or concurrency within the provided architecture. The Flask application handles requests in a single-threaded manner, and the ReactJS frontend components operate within the context of the user's web browser.

3. Detailed System Design

The architectural overview in Section 2 outlined the high-level structure and key components of the Penny Perfect financial management application, following a client-server architecture with separate frontend and backend packages. This section delves into the detailed design of the

major subsystems and components identified previously. Consistent with the architectural styles and package decomposition, the following subsections elaborate on the static class structure and dynamic behavior within each main part of the application. The frontend package, built with HTML and ReactJS, handles the user interface and client-side logic, while the backend package, implemented with Python and Flask, manages the application logic and data processing. The database package, utilizing SQLite, provides persistent storage for user data and financial information. For each of these subsystems, UML class diagrams model the static class relationships, attributes, operations, and multiplicities, while UML sequence diagrams capture the step-by-step interactions between objects for critical use cases and functionality. The detailed design presented in the following subsections aligns with the architectural decisions made and maintains traceability to the high-level architectural views.

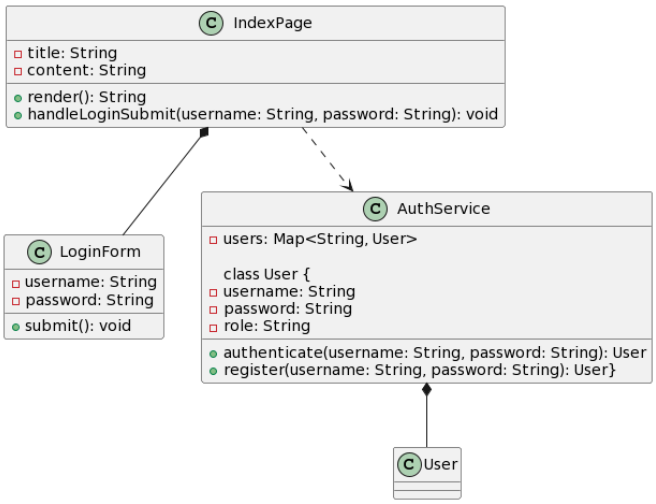
3.1 Static View

The IndexPage class represents the HTML structure and content of the login page. It contains attributes to store the page title and content, as well as a render() method to generate the HTML output. The handleLoginSubmit() method is responsible for handling the submission of the login form, taking the entered username and password as parameters.

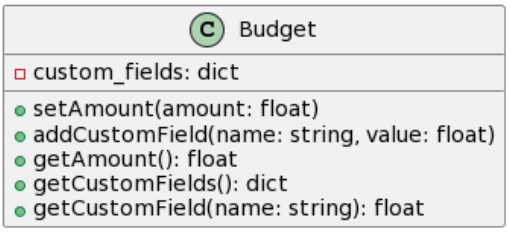
The LoginForm class models the login form itself, with attributes to store the user's input for username and password, and a submit() method to trigger the form submission.

The AuthService class encapsulates the authentication and registration logic. It maintains a map of registered users and provides methods to authenticate a user by verifying their credentials, as well as register a new user with a given username and password.

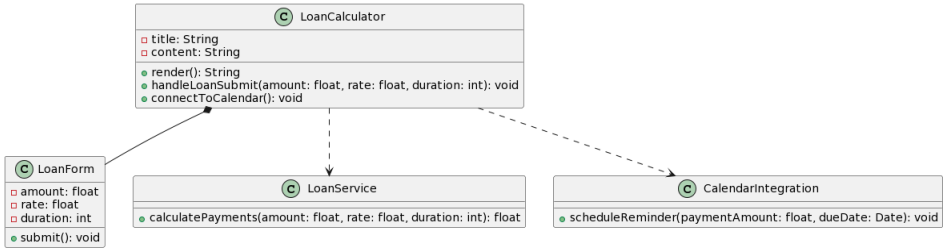
The User class represents a user object, containing properties such as username, password, and role (e.g., regular user, admin).



The budgetCalculator.html file represents the user interface for managing the budget within the Penny Perfect application. This interface allows users to input their budget details, add custom expense categories, and view their budget summary. The UML class diagram below illustrates the static view of the budget calculator interface:

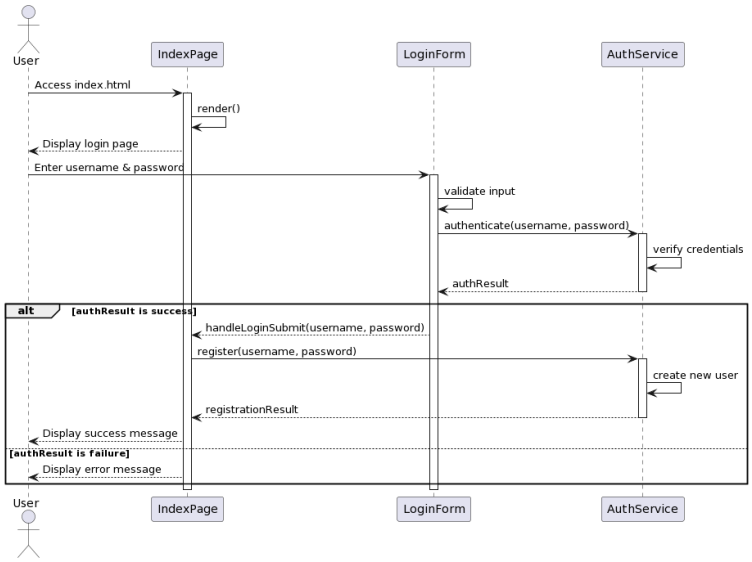


The LoanCalculator class represents the HTML structure and content of the loan calculator page. It contains attributes to store the page title and content, as well as methods to render the HTML output (render()), handle the submission of the loan form (handleLoanSubmit()), and connect to the Google Calendar integration (connectToCalendar()).
The LoanForm class models the loan form itself, with attributes to store the user's input for loan amount, interest rate, and duration, and a submit() method to trigger the form submission.
The LoanService class encapsulates the logic for calculating loan payments based on the provided loan details (amount, rate, duration). It provides a calculatePayments() method to perform this calculation.

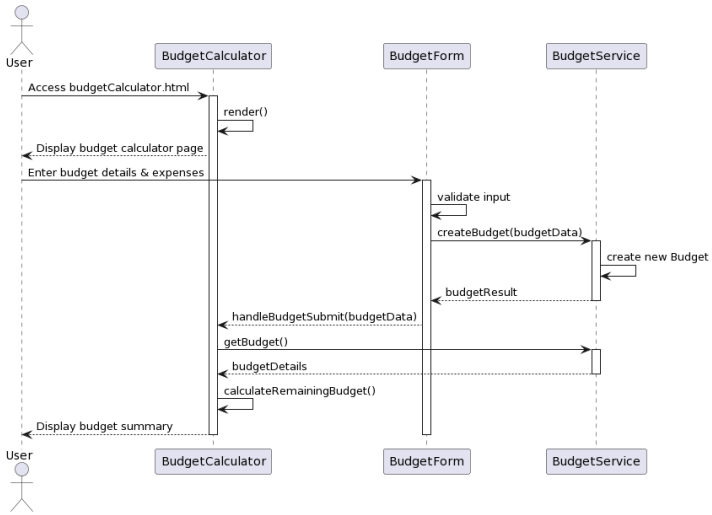


3.2 Dynamic View

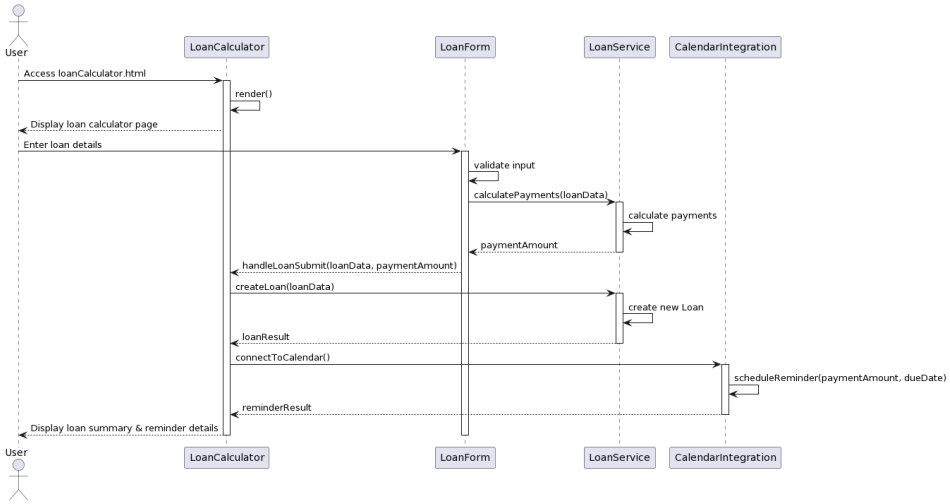
Index.html



BudgetCalculator.html



LoanCalculator.html



These sequence diagrams show how the different parts of the program work together for each webpage. They help explain the step-by-step process that happens behind the scenes when the user interacts with each webpage. They show which parts of the program are involved and what they do:

1. index.html (Login): The user goes to the login page and types in their username and password. The AuthService checks if the username and password are correct. If they

are right, the program registers a new user, and a message says "Success!" If the username or password is wrong, a different message says "Error!"

2. BudgetCalculator: The user goes to the budget calculator page. They type in their budget amount and any expenses they need to pay. The BudgetService checks that the information entered is valid. It then creates a new Budget object that stores the budget details. The program calculates how much money is left after paying the expenses. It shows this remaining budget amount to the user.

3. LoanCalculator: The user goes to the loan calculator page. They enter details about their loan, like the total amount, interest rate, and how long the loan lasts. The LoanService uses this information to calculate how much the user needs to pay each time. It creates a new Loan object that stores the loan details. The program shows the user the payment amount and loan information. It also connects to Google Calendar to set up a reminder for when the next loan payment is due.