COSC402

Smart Ledger - Shared Spending: Detailed Design Report

May 17, 2022

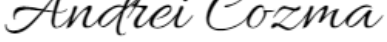
Authors:

Tyler Beichler, Emanuel Chavez, Blake Childress,

Lillian Coar, Andrei Cozma, Jacob Leonard, Hunter Price

Sponsor:

Dr. Bradley Vanderzanden



EXECUTIVE SUMMARY

Smart Ledger is an application that supports shared spending within groups. The application facilitates the recording of spending between multiple parties and prevents the loss of money through dropped or forgotten transactions. It provides support for families, friends, roommates, couples, and other groups who want to audit their spendings, settle balances, and pay other members outside of the application. By keeping track of all payments and purchases in one place, users can be sure that they are settling their balances and receiving what they are owed. Furthermore, the platform was made to be entirely free to use, with premium features available for paying members.

Smart Ledger is built with React and Node on the front-end and Flask and MongoDB on the back-end. It is built around five major engineering characteristics, which were determined by analyzing the project goals and requirements. These five requirements are: accessibility, reliability, response time, scalability, and security. Accessibility and scalability are constraints which measure how usable our application is and the rate at which we can expand the infrastructure to support more users. Furthermore, response time and reliability are variables that can be measured during the application loop and uptime. Finally, security is an important variable which can be analyzed to determine the posture of Smart Ledger and its integrated features/libraries.

There were several features that the team wanted to add but ran out of time/resources to complete. Some of these include optical character recognition (OCR), spending tracking/analysis, and full integration/compliance with banks and other payments services (e.g., Paypal, Venmo, etc.). Furthermore, testing that could not be completed, such as in-depth penetration testing, is always something to consider in the future. If another team were to pick up where the current team has left off, they would have a solid base for both the front and back-end, with ample room to expand functionality and create a fully fledged web application. Finally, the Smart Ledger team could not have got where they did without the help of Dr. Bradley Vanderzanden.

CONTENTS

1. PROBLEM DEFINITION & BACKGROUND

Smart Ledger aims to solve the problem of “no good solutions to support shared spending within multiple groups of individuals”. People frequently share expenses with their friends, family, and other associates, often in groups of two or more. However, since one person is in many different groups that may require transactions, keeping track of everything can be difficult.. One of the biggest problems is that there may not be enough transparency or organization while spending to settle balances quickly and efficiently.

The few products that were found that allow these features are not multiplatform and/or have limitations and missing features. There are some applications that are good at one aspect of expense sharing, such as restaurant bill splitting. A few of the current competitors are SplitWise, Spend Together, FinSplit, and MonShare. Although they all aim to solve a similar problem, each has its own set of shortcomings. Some solutions are preferred, but may be exclusive to iOS or Android, require monthly subscriptions for essential features, or do not have easy and intuitive interfaces.

Furthermore, Venmo and CashApp are two apps that can perform similar functions when used together with applications like Microsoft Excel or Google Sheets. The disadvantage is that users must keep track of the purchases and request/pay the other parties on their own. While this is an option, it is up to one of the parties involved to pay or request the money. Instead, Smart Ledger believes the application should do the legwork for the user, making their experience more streamlined.

Smart Ledger is a free and comprehensive solution, which is intended to be available on all major platforms. One of the basic functions of Smart Ledger is the ability to create groups of people where expenses are shared. Within these groups, purchases can be split up among the members. Another function is the manual insertion of purchases for when receipt scanning is not an option. The main priority is to ensure these functions work as they are the core of what users will be doing and to make a simple and effective user interface. The group decided to develop a web application to ensure access on any device with access to a browser.

Later, the group plans on adding receipt scanning to speed up the process of adding multiple purchases. Other planned features include automation, such as sending direct Venmo and PayPal requests to allow non-users to use the app directly. Advanced features may be primarily available to premium users, while maintaining the core functionality for free users. These advanced features can include but are not limited to automatic purchase classification and spending analysis. The background needed to complete this project includes knowledge of databases and managing them, web apps, interpreting financial statements, and knowledge of expense tracking.

Smart Ledger will be able to be utilized by both individuals and businesses alike. Individuals can use it for splitting expenses such as groceries, rent, and restaurants. Businesses will be able to use the app to keep track of costs in certain departments and how much is being spent monthly or yearly. Both groups can use the recorded data to plan for their future expenses and change where their funds are being utilized.

2. REQUIREMENT SPECIFICATIONS

This section will bring together the app's purpose, specifics, features, and information about how the product will serve the users' needs. The team has discussed the technical aspects of the project, user flow, and key features to provide a detailed description of the existing problems and the solutions.

Customer Requirements

The web application provides a simple, appealing, and intuitive user interface for users to easily navigate the application and perform tasks. The user interface must also function in a logical manner for the target audience, and must work seamlessly on all major platforms and devices. The application must be responsive and resize correctly, especially for mobile devices. Furthermore, it should function/display properly in the various browsers/versions, including Google Chrome, Mozilla Firefox, Microsoft Edge, and Safari. The web application will be thoroughly tested on different platforms, web-browsers, and screen orientations. Finally, it must allow a user to navigate back and forth within the application without having to use the back button or external navigation functionality.

Customer Requirements
1. Free to download and use
2. Multi-platform support
3. Support multiple currencies
4. Support multiple user-defined groups
5.Calculate totals for the group
6. Split expenses by percent or an exact amount
7. Payment deadlines & reminders
8. Activity feed & dashboard
9. Secure Network Protocols
10. Data Encryption
11. Receipt Scanning
12. Activity Evaluation

Table 1. Customer Requirements

Functional and Technical Requirements

Smart Ledger is expected to adhere to industry best practices. Furthermore, several requirements regarding appearance are described in this section.

Firstly, there is proper exception handling in server-side code. Code exceptions must be handled in a user-friendly manner by displaying a custom error page that does not display information such as database object names or source code.

Form fields must be validated to ensure required fields are completed, numeric fields have numeric data, and data input is properly formatted (e.g., date, email address).

User data can be protected through HTTPS during transmission and then encrypted when it reaches the database server to ensure integrity and confidentiality. Administrators can monitor all points of access and implement firewalls to prevent unauthorized users from entering the server.

Reports of the user activity will be needed to account for the best experience the application can deliver. This includes reporting bugs and other flaws that can be fixed as quickly as possible.

The web application must use styles that are consistent throughout the application and within the associated website, including:

- The use of capitalization (e.g., title case vs. sentence case).
- The use of punctuation (e.g., consistent use/no use of colons on labels).
- Error messages must appear in a consistent location and style.
- Consistent use of any web document notations (e.g., PDF, DOC, etc.).
- Layout/spacing (e.g., the space between a field label and input control).
- Descriptive metadata titles and descriptions.
- Form controls that are not available must be hidden--no use of inactive controls.

Engineering Characteristic	Units/Options	Significance
Accessibility	N/A	Constraint: Must be available on all major platforms and have an intuitive user experience
Availability	Uptime/Downtime	Variable: Determined by the dependability of the project and servers

Table 2. Engineering Requirements I

Tools and Technologies

The cloud service will need to integrate the full stack efficiently with documentation to work with a variety of web servers, databases, and other services together.

The backend server will provide API endpoints for performing queries to a database that stores user and group information as well as their respective ledger transactions.

As for the features that were not implemented such as OCR and AI analysis, the team theorized what needed to be done to fully integrate them in the future. For example, OCR may utilize image conversion of a picture to machine-encoded text for receipts and other related documents. Further, the back-end will be able to use Artificial Intelligence (AI) tools to help users optimize savings based on the history of

their spendings and budgets. These tools will focus on machine learning models to categorize spendings as well.

Availability, Performance, and Scalability Requirements

Availability is a key feature, as the group aims to cover most devices and platforms. For example, establishing an integrated native application on iOS and Android. However, it was advised the group should focus on one or the other, although neither have been shipped natively. The design is available on desktop and mobile devices which will be accessible to the web. If the user does not have an Internet connection, the group aims to provide offline capabilities so that the user can still record information whenever they need it. Performance is a key feature, as an unresponsive application will likely make the customer frustrated. The application web page should load as quickly as possible and any actions taken within the application should be fast.

Engineering Characteristic	Units/Options	Significance
Response Time	ms	Variable: Influenced by the efficiency of the application and quality of the servers/services
Scalability	N/A	Constraint: Relies on the product architecture and services chosen
Security	N/A	Variable: Stems from technology we choose and secure coding practices

Table 3. Engineering Requirements II

3. TECHNICAL APPROACH

Smart Ledger seeks to create a fully operational, streamlined, cross-platform web application. This application should support all users in any device and browser used. Furthermore, the application should be easy for the user to log purchases without the product getting in the way of itself..

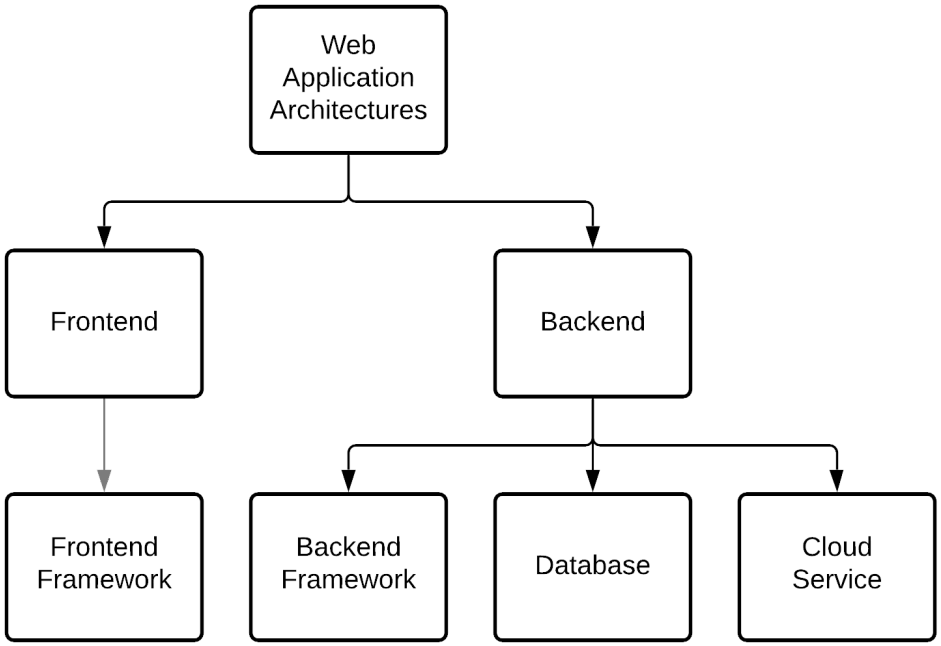


Figure 1. Design Decisions

The web application will consist of 2 major sections: the frontend and the backend. The first decision that was made is the overall architecture of the application, of which there were 3 main architectures considered. The first is the legacy HTML web application structure. In this architecture, the backend consists of a server that serves the client complete HTML pages handled by business logic in the server. The user must fully reload the page to see any changes made by the server.

The second architecture to consider is a widget based web application structure. In this, the creation of the HTML page structure is handled by a webservice. Requests are made from the client, and the server sends back data to populate the page with. This structure typically takes more time to develop full applications with.

The last architecture to consider is the single-page web application architecture (SPA). In an SPA, the client must only receive the preliminary web page structure once upon entry then each of the components are updated without needing to refresh the page. This architecture is considered by the industry to be the most agile and popular type. Therefore, the team only considered frontend and backend options that supported development of a single-page web application. The following design decisions are outlined in Figure 4.

There is one major decision that took place for the frontend. This is the frontend framework. Here the team considers three options: Angular, React, and Vue. For these options, the team must consider the engineering characteristics: accessibility, reliability, response time, and scalability for each framework. To do this, a weighted decision matrix was created. The weights for each factor to consider is a rating from 1 to 3 of each framework, where each weight in a row is summed to give a final score.. In table 5, the team found that React would be the best decision with the engineering characteristics.

Framework	Accessibility	Reliability	Response Time	Scalability	Score
Angular	2	2	1	3	9
React	3	3	2	2	10
Vue	1	1	3	1	7

Table 4. Front End Framework Weighted Decision Matrix

In the accessibility section, the team found that React had the best support on all modern browsers. This was closely followed by Angular. Vue had the worst support as it does not fully support some older browser clients. React has the best reliability as the library has the most support and therefore bugs and issues in the framework are fixed faster. Response time was dominated by Vue which is known to be a very lightweight framework. React then Angular followed with slower response times. The team found Angular has the best scalability in the sense that the framework gives the developer a very defined structure that he or she must follow. When following that structure, scaling the application is better streamlined. React was closely second, followed by Vue which lacked full scalability ability.

The backend portion of the web application is comprised of 3 sections: the backend framework, the database, and the cloud hosting service. The backend framework was guided by the engineering characteristics with an additional attribute for machine learning support. Furthermore, our backend framework weighted decision matrix, Table 6, follows the same format as the previous table.

Framework	Accessibility	Reliability	Response Time	Scalability	ML Support	Score
PHP	1	1	1	2	1	5
Node.js	3	3	3	3	2	14
Flask	2	2	2	1	3	11

Table 5. Backend Framework Weighted Decision Matrix

The team found Node.js to be the most accessible of all of the frameworks as Javascript is supported on all browser clients and servers. Flask requires an additional Python installation. For a similar reason, Node.js has superior reliability. Node.js is also scalable as it is the industry standard for a backend framework. PHP is scalable as well; however, it requires more code to be written. Flask is known to have problems with scalability. Lastly, Python is known to be superior for fast machine learning development and support with its numerous libraries, so Flask is a necessity. The score of the resulting weighted decision matrix has Node.js as the best decision for a backend; however, the team needs fast machine learning development as well. This means that the team will be using both Node.js and Flask as the backend framework.

The engineering characteristics guided the choice of database. Our database weighted decision matrix, Table 7, follows the same format as the previous table.

Database	**Accessibility**	**Reliability**	**Response Time**	**Scalability**	**Score**
MongoDB	2	3	1	3	9
CouchDB	3	1	3	1	8
Couchbase Server	1	2	2	2	7

Table 6. Database Weighted Decision Matrix

Cloud Service	**Accessibility**	**Reliability**	**Response Time**	**Scalability**	**Score**
AWS	2	3	1	3	9
GCP	3	1	3	1	8
Azure	1	2	2	2	7

Table 7. Cloud Services Weighted Decision Matrix

The engineering characteristics guided the choice of cloud service. Our weighted decision matrix, Table 8, follows the same format as the previous table.

The team ultimately chose a single-page web application using React as the frontend framework, Node.js and Flask as the backend framework, MongoDB as the Database, and Amazon Web Services as the cloud service.

From a design point of view, the application could consist of multiple tools that allow the user to gain more insights from their spending habits as well as to be able to more easily use the application interface. First, the application has an algorithm that acts as a ledger that keeps track of all purchases, the Sharing Algorithm™. It calculates how much each user has contributed to the ledger and how much each person is owed when a payout is due.

Given more development, the application would have a machine learning classifier that will predict the type of purchase based on the description of an item. This is especially useful to a user as they can go back through their history and get a summary of their travel expenses or food expenses. Third, the team will have a machine learning model coupled with statistical models that show and predict the trends of the spending habits of the groups.

Finally, the user interface will be designed to streamline the input of all users spending input, and the team will show the user suggestions of who should make the next purchase in a group depending on who has not put an equivalent amount of money into the ledger. Since these features were not necessary for a minimum viable product, the team decided to omit them for now.

4. DESIGN CONCEPTS, EVALUATION, AND SELECTION

Component	Tools/Libraries
Frontend	React.js
Backend	Node.js & Flask
Hosting Service	Amazon Web Services (AWS)
Database	MongoDB
Optical Character Recognition (OCR)	Tesseract.js
Artificial Intelligence (AI)	Tensorflow.js

Table 8. Database Weighted Decision Matrix

Concept Evaluation

Design 1: Frontend - React.js

Establishing a frontend framework is critical for usability for the customer and the developers. The JavaScript library, React.js, is an open-source framework that is best used for user interface and interactive components, which are nicely organized for the developers. Also, the framework saves time for re-utilizing components which prevents bugs in the application due to repeating code. The customer will be able to easily interact with the application and facilitate the necessary services.

Design 2: Backend - Node.js and Flask

Criteria	Pricing	Performance	Security	Total
Node.js	3	3	2	8
Flask	3	2	2	7

Table 9. Backend Frameworks Score

The biggest factor for choosing a backend framework was how well it is able to handle data through real-time applications. Node.js offers low response time and high output capability to serve multiple requests which saves resources for both the developers and customers. Most of the requests are handled with this framework such as the database, but the machine learning and artificial intelligence tools will be handled through Flask. Flask is a python-based framework that will handle code on the server when dealing with tasks such as receipt scanning or user reports. While utilizing both frameworks at the same

time may be difficult, Python is the primary backend framework as it will be easier to reuse most of the code. The customer will hopefully be satisfied with its many benefits for being able to client requests at a reasonable time.

Design 3: Backend - Amazon Web Services (AWS)

Amazon Web Services (AWS) offers reliable and cost-efficient cloud computing services that can provide a platform to utilize the full stack. A free tier is offered to all developers to host a webserver and database with easy-to-follow documentation. The documentation includes basic configuration for network protocols which is essential for hosting and connecting the servers. By enabling configuration for network and security protocols, the application will also provide better security that can be managed by the team as needed for the customers.

Design 4: Backend - MongoDB

A database is essential for the application as customers need accounts to store their personal information, transactions, and group affiliation. There is plenty of sensitive data that the application could be dealing with, so how the data is stored and accessed will be highly considered. In addition to the database prioritizing consistency, MongoDB is adamant on performance, simplicity, and flexibility, which make for a great choice for both the developers and customers. The environment was quick to set up and provides a dynamic schematic architecture that works with data and storage. The customer can ensure that the flexible database model can adapt to any security changes as businesses and technologies evolve. Additionally, because MongoDB stresses consistency in its design, the users will always have up to date data on their device. This allows users to have confidence in the data, which is important as they will be tracking their spending through the application.

Design 5: Backend - AI/ML Tools

The Artificial Intelligence and Machine Learning tools for the application will be written in Python. It is common to use Python for heavy numerical computation as it has numerous libraries that allow for extremely fast development. The flexible libraries will allow us to train models more effectively which in turn will end with the application getting more functionality. This is beneficial to users as they will have better insights on their spending habits and an easier to use graphical user interface with built in intelligent systems quicker with python as the tool.

5. EMBODIMENT DESIGN

5.1. Product Architecture:

The modules present in the application include the production host, the database, back-end, and front-end. The project is mainly split in half, with the database and back-end making up one half (henceforth referred to as the "back-end") while the front-end makes up the other half. The production host, despite being a model, does not actually make up the architecture of the application.

The front-end's outputs include making requests to the back-end server and requests with a Google OAuth 2.0 provider. The front-end can make various requests to the back-end, always including an OAuth token for the purpose of authenticating with the back-end. These requests are for the purpose of either manipulating data in the database or requesting information from the database to display. Requests to the Google OAuth 2.0 provider are used to validate the authenticity of an OAuth token. The front-end's inputs include responses from the back-end and responses from the Google OAuth 2.0 provider. All requests made to the back-end result in a response, even if it is just a status code. In some cases, responses can include information from the back-end which will in turn be displayed by the front-end. There are both outputs and inputs to and from the front-end module and the Google OAuth 2.0 provider. When a user registers an account using their Google login, the front-end will receive a token from the Google OAuth 2.0 provider. This token is then used for accessing protected routes in the front-end and for authentication when sending requests to the back-end.

The back-end's inputs include requests from the front-end and responses from the Google OAuth 2.0 provider. When a user logs in using their Google account, the Google OAuth 2.0 provider will send a token to the back-end. This token is used to authenticate any request from the user. For this reason, the request from the front-end must include the proper OAuth Token for the back-end to act upon any request. This token is the same one sent to the front-end server. The back-end's outputs include requests to the Google OAuth 2.0 provider and responses to the front-end. Requests are made to the Google OAuth 2.0 provider to validate the authenticity of an OAuth token. Responses to the front-end depend on the request it is a response to. If a request does not include an OAuth token, includes the incorrect OAuth token, is malformed, or is made for resources the current user does not have access to, the back-end will respond with an error status. If a request is made with the proper OAuth token, is correctly formed, and is to modify or create data present in the database which the user has access to, the data will be modified accordingly, and the back-end will respond with status code 200. If a request is made with the proper OAuth token, is correctly formed, and is to retrieve resources present in the database which the user has access to, the back-end will respond with status code 200 and the data which was requested.

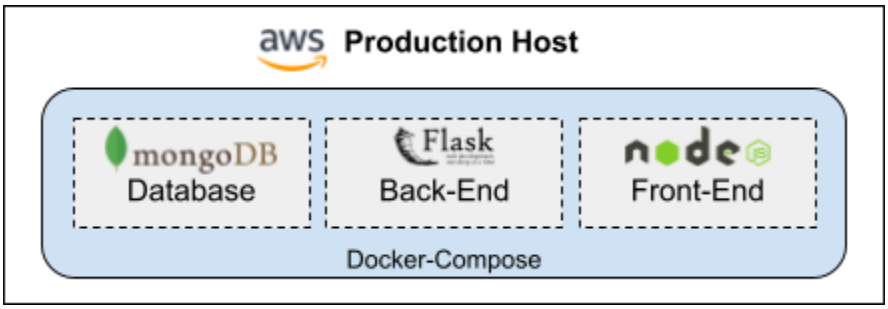


Figure 2. Overall Major Components Overview

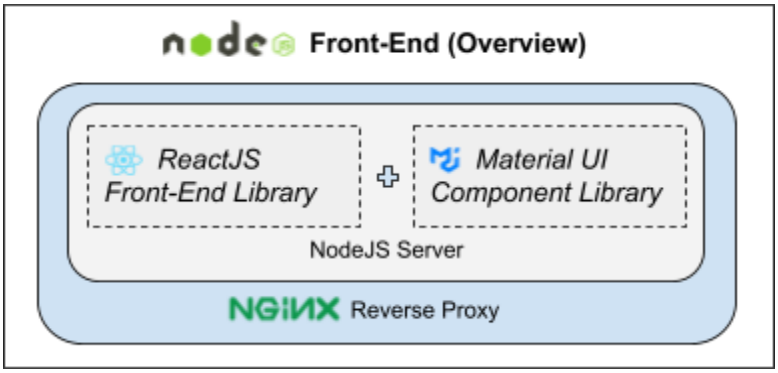


Figure 3. Front-End Components Overview

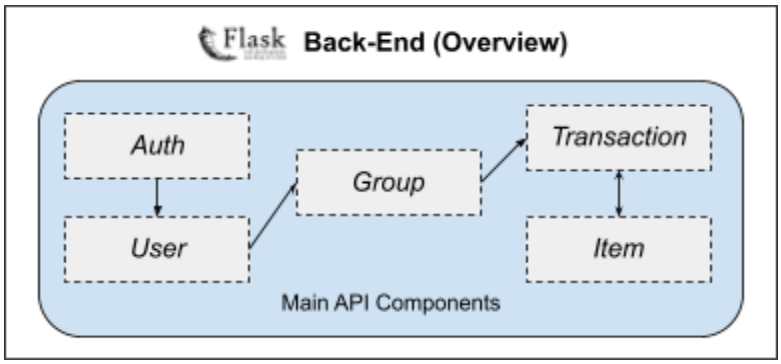


Figure 4. Back-End Components Overview

5.2. Configuration Design:

When first registering, the user is prompted to create an account by connecting an existing Google account with our application. This process requires the application to have authorization credentials for Google. This is done by going to “console.cloud.google.com” where an OAuth client ID is generated. This is also where settings to allow access to user profile information is set so that names, email, and profile pictures can be retrieved. When successfully registering an account, a Google OAuth 2.0 token is returned and is used to verify the authenticity of the user. The tokens have limited lifespans in order to maintain security, so when a token expires a new one is requested, which is then generated by

Google. Interaction between front-end and back-end is done by the front-end sending a JSON object with the auth token attached to the back-end. Every time an API call is called from the front-end, the back-end verifies the Google token before it performs operations on the database. Furthermore, before any operations are done on the database using data retrieved from the front-end, data sanitization is done in order to prevent any malicious attacks. Text data is also cleansed to prevent injection of malicious Javascript or HTML code. Depending on the API, data is retrieved or generated from the database and converted into a JSON object that is returned to the front-end.

The sharing algorithm on the back-end uses shared ledger concepts by using optimized calculations, which allows for continuous additions and updates. Furthermore, the sharing algorithm takes account of who used certain items and who purchased them in order to correctly split up the price of the items among the users. The algorithm also ensures that all of the items in a transaction and their quantities add up to the total amount to ensure correctness. Specifically, the sharing algorithm utilizes deltas for the optimization to maximize efficiency.

In terms of the best-practices and standards considered, user-centered design should improve users' perceptions of the product and any connected services while remaining unique and meeting a specific need. Furthermore, by making our web application more accessible, we can create a consistent experience for all users. Security and privacy are also essential components that influence the design process. Without security and privacy guarantees, customers can not ensure their data won’t be used maliciously or treated carelessly. Following current security best practices ensures that proper steps are taken in the advent of a hack or compromise while keeping as much of the organization safe as possible. This allows users to feel safe while using Smart Leder, and ensures the team is respecting all compliances that are required for the integrations.

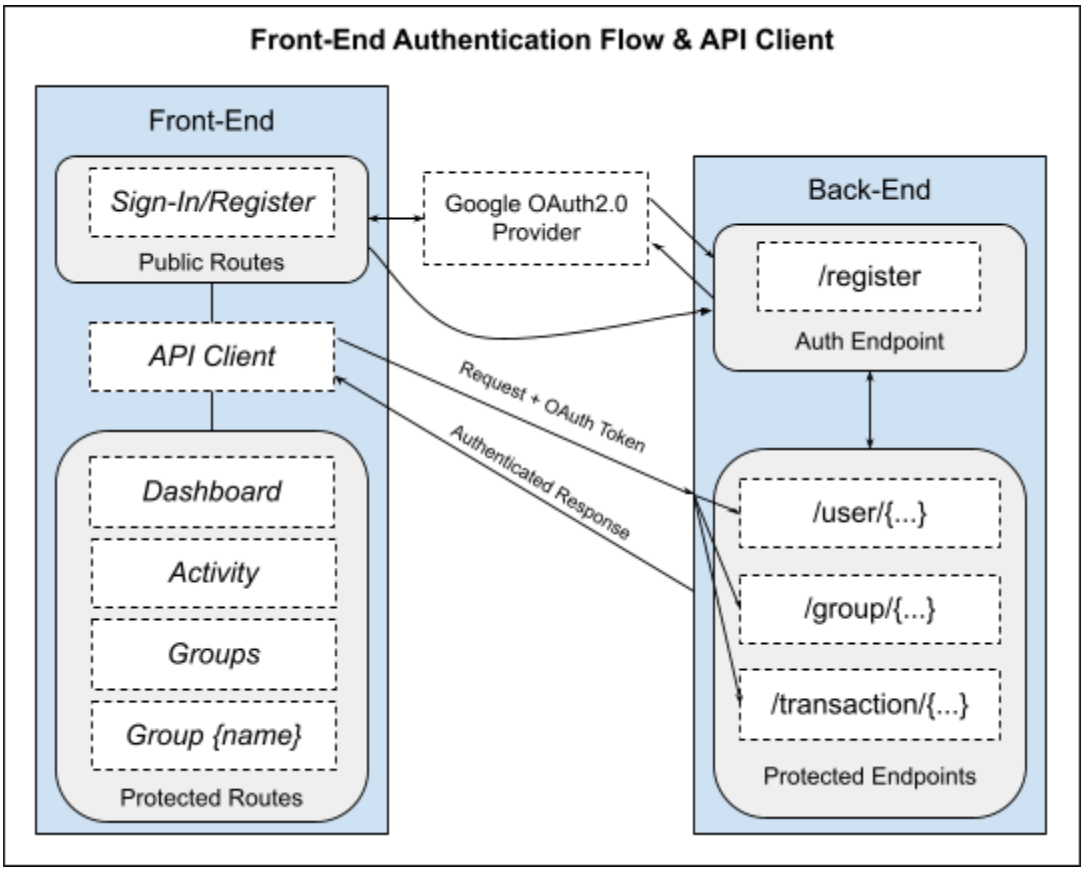


Figure 5. Front-End Authentication Flow & API Client

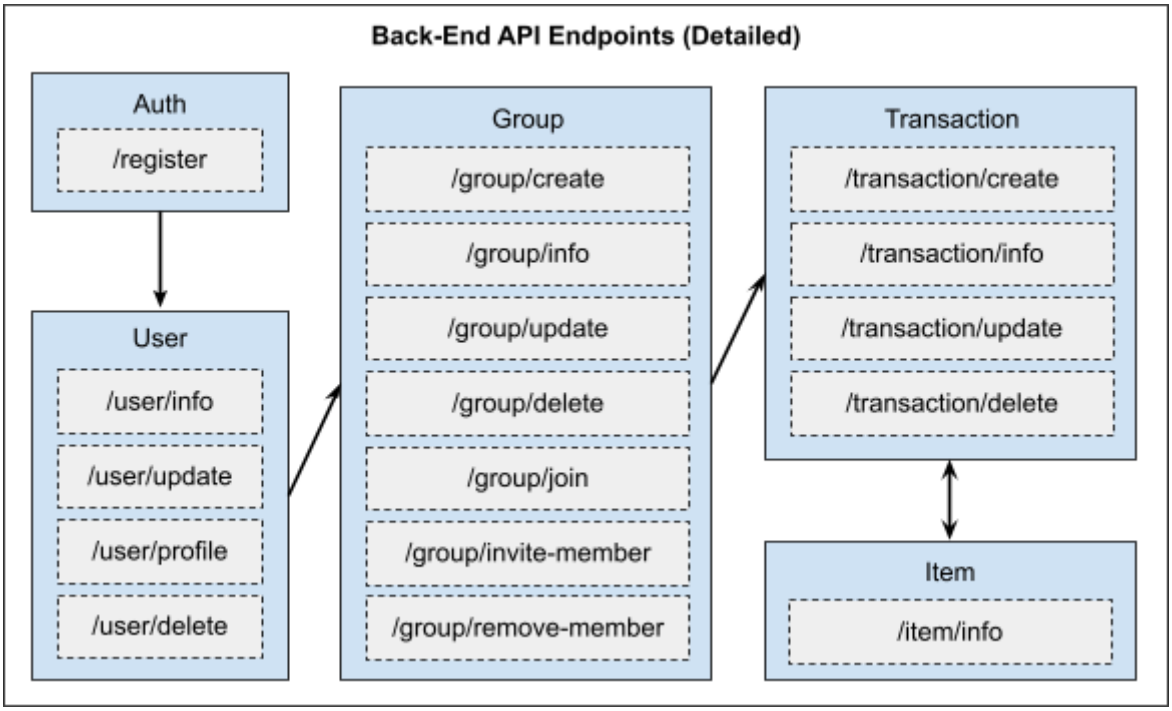


Figure 6. Back-End API Endpoints

6. TEST PLAN

Testing and evaluating performance is an essential part of ensuring the product is durable and efficient. The engineering characteristics to be considered are accessibility, availability, response time, scalability, and security. These characteristics can be determined by a variety of methods such as unit tests, analysis, and more. Unit tests in particular were created primarily with the intention of troubleshooting front end issues, but eventually became a way to ensure accessibility, response time, and security. Each of the unit tests covers a specific behavior that an API should exhibit. A couple of the unit tests are designed in such a way that the API call should fail since bad data is intentionally passed. Others were designed to ensure that the API was returning correct data and correct status codes. These tests were vital in maintaining both ends of the team.

6.1. Accessibility

Our application ensures that it is accessible through multiple devices and platforms. The most commonly used browsers such as Firefox, Google Chrome, and Safari are ones we aim to deem compatible. All functionality works as intended throughout these browsers as we tested user authentication as well as the creation, deletion, and modification of groups. As for mobile devices, we aim to make the application accessible on both Android and iOS devices.

6.2. Availability

The availability of the application is essential for the user experience as usability plays an important role. Amazon Web Services provides hosting for our stack which has shown to be reliable at all times. This service hosted our frontend, backend, and database which query API calls from the client. We restricted the amount of requests at a time to prevent an overload that would cause a denial of service.

6.3. Response Time

The frameworks and database were both selected with speed in mind. While our team would have had zero issues with nearly every framework or database we could have used, we wanted to ensure that any users attempting to access our site and application would be able to do so, regardless of the browser or system specifications. In order to test weak connection, we implemented a wait time to ensure the loading animation worked as intended before displaying the pages.

6.4. Scalability

Scalability relies on the database's ability to store users with their associated information which was tested through unit tests. Creation, deletion, and modification of this data went through a multitude of unit tests to ensure the functionality would still work with no errors.

6.5. Security

Application security was handled with various implementations primarily throughout the back end, as well as the front end. Users only have access to front end options which access API calls, so

database values cannot be directly changed. Any input on the front end or back end is also sanitized, so injections are less likely to occur.

7. PROJECT DELIVERABLES

In this section, the team will define the goals towards which the team places the focus for this project. First, the team will explain the project objectives, listed in order from most important to least important, as well as any customer requirements that require it. Finally, the team will explain how these requirements relate to customer needs and how the team arrived at the conclusions.

Principle Component Deliverable	**Notes**
Front-End	Convenient and appealing web application.
Back-End	Secure protocols for transmitting data/requests.
Database	Encrypted storage of user data.

Table 10. Main Deliverables for Working Prototype

Front-End Design Deliverable	**Notes**
Design Guidelines	Define how the application's front-end should look and behave with user interaction.
Wireframes/Renders	Generated based on design guidelines.
Graphics	Any images or icons used on the front-end

Table 11. Front-End Design Deliverables

Proof-of-Concept Item Deliverable	**Notes**
Artificial Intelligence	AI tools to evaluate user activity and transactions.
Optical Character Recognition Tools	ML tools to recognize text in an image (i.e. receipt).

Table 12. Proof of Concept Deliverables

Miscellaneous Deliverables	**Notes**
Documentation	Clear and concise instructions/information regarding the application.

Table 13. Miscellaneous Deliverables

The customer will be delivered a web application that offers a solution to shared ledgers. It is also accessible and functional on all browsers. The frontend implements an appealing design as well as minimal pages for usability purposes. The backend offers secure protocols to ensure the data being handled is protected from unauthorized users from accessing or modifying it. For future work, the Artificial Intelligence and Machine Learning tools will offer services such as transaction evaluation and

receipt scanning for convenience. Lastly, the documentation provides an outline of usage and how the team implemented the application.

These deliverables have been developed and confirmed with the team members, team sponsor (Dr. Bradley Vanderzanden), and third-party experts in software development. The importance of each deliverable is listed in the “Notes and Images” section, but a good rule of thumb is that these deliverables are listed by order of priority, descending from greatest to least.

8. PROJECT MANAGEMENT

During the development process, each key component of the application was allocated to a member of the team who monitored its progress and development as well as provided regular updates. To ensure that all requirements, deliverables, and milestones are met, the project manager will supervise the whole front-end and back-end development teams.

While working together, the group collaborated closely throughout the entire process to ensure that everyone was on the same page and that everything went according to plan. Every week on Friday from 11:45 a.m. to 12:35 p.m. the group convened for standup meetings to discuss the progress of each team and any roadblocks. This also provided an opportunity for us to discuss any concerns or challenges that any of the group members may have been experiencing, as well as to ensure that the group is on schedule to complete the project on time.

It was the purpose of the Fall semester of 2021 to acquire a head start on planning, research, and discovery, as well as the creation of a very basic architecture and prototype for the proposed application. It consisted of a front-end and a back-end design, which will help us to acquire a better understanding of the overall evolution of the application by building a more concrete prototype. With the Spring semester of 2022 completed, the goal for the team to continue to develop all of the application's basic components was mostly completed. For future work, adding additional features in order to broaden the application's reach in accordance with the project deliverables and objectives would be great.

As a recommended timeline for the production of the proposed application for the Spring 2022 semester, the following schedule has been established and is available for review. It will be heavily influenced by the effectiveness of research and development initiatives, as well as the availability of resources and participation from important players. Furthermore, under each sprint were a myriad of smaller issues which need to be addressed to reach the main goal.

Sprint	Goal
1	Front/back-end design and architecture
2	General front/bank-end application development
3	Further front-end development + back-end testing
4	Front/back-end release and demo

Table 14: Project Schedule

9. BUDGET

Item	Estimated Cost
AWS Hosting Services - Free Tier	$20
Jira - Student	$0
Discord	$0
GitHub Student Developer Pack	$0
Domain*	$0
Total	$20

Table 15: Actual Spending

The development costs for this project were minimal. The application was hosted by Amazon Web Services (AWS) during testing [1]. To host the website and database, it will especially make use of Amazon EC2 Compute, which is less expensive than purchasing dedicated hardware for the purpose (free for 12 months). For the duration of the month, the team will be provided with an additional 750 hours of Linux instance time from Amazon EC2, which will be more than enough for the needs.

Even in the worst-case situation, if the project were to take more than 750 hours per month, the budget will need to be amended to account for a payment of $0.051 per compute-hour utilized (which can be funded by the premium tier of Smart Ledger). Paid computation hours will be required to support the project's expanding user base; however, this is not covered in this course.

Jira is the primary project management platform [2]. It is fully free for students and it offers a free tier that allows up to ten users to utilize the software. Discord will serve as the primary means of communication for the team, and is a completely free service to utilize. Github will be the primary version control mechanism for the project. A free student developer bundle [3] is available on Github, and it contains all of the functionality required to build Smart Ledger.

Lastly, a domain name is required for the client audience to have external access to the application, and this typically costs roughly $15 per year to maintain. However, for Smart Ledger we simply used a team member’s already owned domain and made a subdomain to use for the public test server. Therefore, the total cost was ~$20 for the AWS hours accrued during testing, although more would be added given additional users and relevant costs.

10. REFERENCES

[1] Amazon EC2. (2021, 11 01). Amazon Web Services. https://aws.amazon.com/ec2/?did=ft_card&trk=ft_card&ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc

[2] Jira Pricing. (2021). Atlassian. https://www.atlassian.com/software/jira/pricing?&aceid=&adposition=&adgroup=89541890342&campaign=9124878120&creative=461842798799&device=c&keyword=jira%20pricing&matchtype=e&network=g&placement=&ds_kids=p51241495620&ds_e=GOOGLE&ds_eid=700000001558501&ds_e1

[3] GitHub. (2021). GitHub Education. https://education.github.com/pack

https://www.parvatiandsons.in/

11. APPENDICES

11.1. APPENDIX A:

<table>
<tr><td></td><td colspan="4">SMART LEDGER - BUSINESS MODEL CANVAS</td></tr>
<tr><td>Key Partners</td><td>Key Activities</td><td>Value Proposition</td><td>Customer Relationships</td><td>Customer Segments</td></tr>
<tr><td rowspan="3">The following partners can provide marketing and feedback on our product:
- Universities
- Students
- Administrators
- Financial Aid Offices
Main suppliers of services:
- AWS hosting services</td><td>- Product Developments & Management
- Research & Development.
- Marketing
- Sales</td><td rowspan="3">- Streamlined solution to tracking shared purchases.
- Helps users organize their finances.
- Quick and easy to use.
- No transaction fees for settling balances.
- Base features are completely free to users.</td><td>- Focus on keeping long term customers.
- Optional feedback emails.
- Customer service for hard to use features.
- Take customers input and fix features for streamlined process.</td><td rowspan="3">We hope to target groups of people that need to audit financial spendings between each other. The following are examples of customers:
- College students
- Young Adults
- Couples
- Homeowners</td></tr>
<tr><td>Key Resources</td><td>Channels</td></tr>
<tr><td>- Data: Database of users and purchases.
- User/customers base (students).
- Technology (Web App).</td><td>- Awareness: Advertised through social media campaigns and google ads
- Evaluation: Click and usage rates on features
- Delivery: Web App (Website), Mobile Apps (Apple App Store & Google Play Store).</td></tr>
<tr><td colspan="3">Cost Structure</td><td colspan="2">Revenue Streams</td></tr>
<tr><td colspan="3">- Server and hosting costs.
- Base features for app are free for users.
- Charge users for additional premium features, such as financial analytics.
- Research and Development costs
- Marketing costs (social media campaigns, marketing on campuses)</td><td colspan="2">- Charge users for certain additional premium features.
- Purchase additional analytics dashboards & customizations
- A.I. suggestions
- Google Ads for non-premium members
- Charge companies and organizations for premium features, such as HOA splitting fees among members.</td></tr>
</table>

Figure 6. Business Model Canvas

11.2. APPENDIX B:

Test Matrix for Back End Unit Tests		
Test Name	Description	Test Result
test_delete_and_register	Registers and deletes user	Successful
test_bad_payment	Assigns bad payment info to user in update call	Successful (intentionally fails)
test_user_info	Retrieves user info	Successful
test_group_no_desc	Creates a group without a description	Successful
test_bad_group	Creates a group with wrong data	Successful (intentionally fails)
test_create_group	Creates a group	Successful
test_add_receipt	Adds receipt to transaction	Successful
test_get_receipt	Retrieves receipt from transaction	Successful
test_create_receipt_image	Creates and attaches receipt image to transaction	Successful
test_invite	Invites user to group	Successful
test_group_refresh_id	Refreshes group id	Successful

Figure 7. Test Plan Matrix