GirlMath

Team 5 - CS 4485.0W1 Group Project

Gabrielle Kuruvilla,

Viet Vu,

Harper Wood,

Kacie Yee

Supervisor: Professor Sridhar Alagar

Course Coordinator: Thennannamalai Malligarjunan

Erik Jonsson School of Engineering and Computer Science

University of Texas at Dallas

11 May 2025

Table of Contents

8. References
9. Appendices

Chapter 1: Introduction

1.1 Background

Every member on our team struggled with finding a good budget app that was simple and had all the features that we wanted. This served as the motivation for our app, GirlMath. We wanted to create a budgeting app that we ourselves would want to use, and we're sure it will prove useful to many other students like us. This problem is relevant because most budgeting apps are tailored towards people who are working full-time jobs with a regular income, saving for retirement, and managing investments. Our app is focused on keeping things simple and easy for students who might not have a consistent salary to manage and aren't looking to save for retirement yet. GirlMath is simple, easy to use, and is flexible for any kind of financial situation.

1.2 Objectives, Scope, and Goals Achieved

- **Objectives & Scope:** The goal of the project was to design and build a web app that students or anyone trying to manage their finances for the first time can use. The app should allow users to track their spendings and log their expenses, as well as display all their past records in a visually appealing, easy-to-understand format. The app should make it as simple as possible for users to upload their expenses, to motivate them to do so consistently. The project should help students learn to manage their spendings, especially those who haven't done anything of the sort before.
- **Goals Achieved**: GirlMath achieved all of the goals that we set out in our initial objectives. Our app is easy-to-use, has a visually appealing UI, displays a user's past data in the form of color-coded line graphs and pie charts, and makes it easy for the user to tell if they are on track to maintaining their budgeting goals. With a receipt OCR feature and voice memo transcription feature, adding expenses is easier than ever for users.

1.3 Key Highlights

- **Feature 1: Data displays**
    - A user can see all their past data displayed in a pie chart and bar graph. The pie chart shows a breakdown of the percentages of each category of past expenses. The bar graph shows where your total budgeting goal is, and shows you where your total expenses are in relation to that line. By turning filters on and off, you can also customize which categories you'd like to see data for.
- **Feature 2: Receipt OCR**
    - By uploading a picture of a user's receipt to our web app, GirlMath will automatically use OCR to scan whats on the receipt and automatically use that information to fill out the

user's next expense. Users are free to edit the information as needed before adding the expense with the click of the button.

- **Feature 3: Voice Memo Transcription**
    - Similar to the Receipt OCR feature, the voice memo also automatically fills out the information needed to log the user's next expense. By clicking record and telling the mic what the user's last expense was, our app makes it so that users don't even need to type to log expenses. GirlMath transcribes the voice memo and extracts the data needed to log users' next expense automatically.
- **Feature 4: Sorting past expenses**
    - In our All Expenses page, the user can filter through our database of their past expenses by date, and specify exactly what start and end date they want to display on the screen.

1.4 Significance of the Project

For most of the younger generation, college is their first money management experience. As many live away from their parents for their first time, they are free to handle their own finances as they see fit. However, many students don't consider that their spending habits need to be adequately managed, even though there are many apps on the market for such purposes.

However, even those who would like to start managing their spending habits have a difficult time finding a good app for that purpose. Many current budgeting apps on the market use income to create a budget, which sometimes students might not have. Our app allows users to freely set budgets on their own, and is designed to maximize flexibility to allow users to track their expenses in whatever manner best fits their financial situation.

GirlMath allows users to track their expenses, sort them into categories, and observe trends in their spending habits, without forcing users to base anything on income, or overwhelm them with too many options like saving for retirement, managing stocks, or multiple savings accounts. By making expense tracking as simple and easy as possible, we make it so that students are motivated to use the app consistently, rather than trying it out for a month or two before abandoning it completely. With GirlMath, students will be able to hop into managing their own spendings without any intimidating vocabulary or messing with restrictions and requirements that don't pertain to them.

Chapter 2: High-Level Design

2.1 System Overview & Proposed Solution

Azure Document Intelligence API

For receipt processing, we leveraged an Azure Document Intelligence Form Recognizer using the prebuilt receipt model. Our application allows the user to upload a receipt, calls the Azure API to

perform all necessary processing, and automatically populates the expense form fields. The user is then able to edit what was returned by the API or fill in any missing data before submitting the expense.

The Azure Document Intelligence API uses a combination of Optical Character Recognition (OCR) and deep learning models to analyze and extract key information from printed or handwritten receipts. It extracts several data items, including the four which were relevant to our use case: merchant name, transaction total, transaction date, and receipt type. To determine the receipt type, the API utilizes a deep learning model to make a prediction from the extracted information, with possible results being meal, supplies, hotel, fuel & energy, transportation, communication, subscriptions, entertainment, training, and healthcare. It supports many different formats, including pdfs, images (jpeg/jpg, png, etc.), and documents (docx, xlsx, pptx, html).

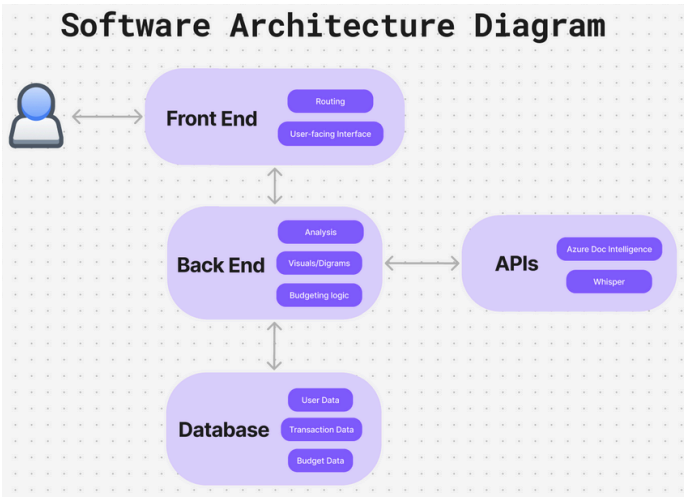
Our app performs two different API calls: one to send the document to the API for analysis, and one to retrieve the results. These two calls are linked by the apim-request-id, an ID generated for a job when a receipt is uploaded and used to fetch the results for that receipt. Because it takes a moment for the receipt to process, our app performs polling, and it does not make the API call to fetch the results until the status of the send document API call becomes “succeeded.” Once processing is complete, the API returns detailed, structured JSON data so that the attributes that are important to our use case are easy to retrieve.

OpenAI Whisper API

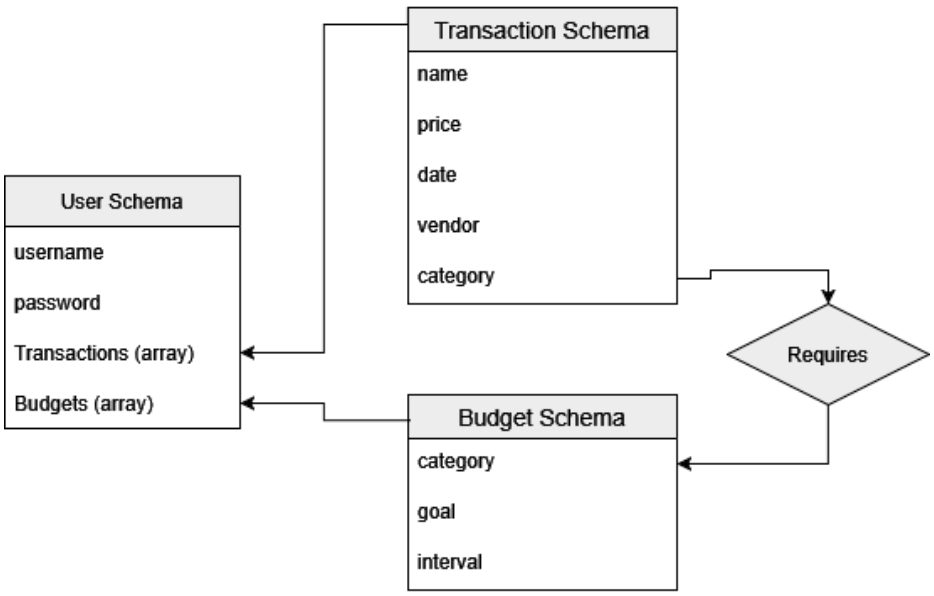
For the voice logging feature, our app leverages OpenAI’s Whisper model to convert the user’s speech to text so that key features can be extracted from it. The API supports mp3, mp4, wav, etc. for inputs and json, text, etc. for outputs. Our app sends the user’s speech file to the API as a wav file and retrieves the response in json format.

The result is processed in two ways: by regular expressions and by the Azure API (operating on the text result stored in a pdf). The regular expressions extract the price, date, and category if a category from the user’s list is present in the text output. The Azure API extracts all 4 fields as it would in a receipt. The form fields are then auto-populated, with the data extracted by the regular expressions taking precedence over the data from the Azure API.

2.2 Design and Architecture Diagrams



Our software architecture diagram shows how the app's elements react with one another upon user interaction. The user communicates with the front end website, inputting their data, viewing it, editing it, or anything that our app allows. Then, those interactions would route to the backend to give the user's interactions a function. This involves either sending the data to the API so that a receipt can get translated from either a voice memo or a receipt image to populate an expense's field, or to send or retrieve the data from the database, where all of the user's data would be stored.



The schema that we ended up making had simple relations; upon creation, a user would be stored directly into the database with its username, password, and two arrays for Transaction and Budget objects. The Transaction object schema contained a transaction's name, price, date, vendor, and category name, while Budgets contained a category name, price goal, and time interval. Since these two

objects both contained a category name but existed in different parts of the overall user schema, we had to ensure that all of the code that referred to our category names reflected changes in both schemas as needed. This included transactions creating a new budget if its category was previously nonexistent, budgets being unable to be deleted if a transaction held that category name, and general associations in order to properly calculate the data that we wished to use throughout our project.

2.3 Overview of Tools Used

- Visual Studio Code - The chosen IDE where our team programmed and collaborated together
- MongoDB Atlas - A cloud-based database hosting service that we used for our database
- MongoDB Compass - Software used to visualize and manage the data within our database
- GitHub - Our chosen collaborative tool to merge and manage our work
- Figma - A web tool used to map the layout and functionality of our project
- Azure Document Intelligence - The API used to process receipts and speech-to-text output
- OpenAI Whisper - The API used to convert user voice logs describing expenses into text

Chapter 3: Implementation Details

3.1 Technologies Used

Frontend

- Figma - A web tool used to map the layout and functionality of our project
- Next.js - A React framework for the web to help simplify development and organize our app
- React - A web development framework that allowed us to build a responsive interface

Backend

- Node.js - A JavaScript runtime environment used to support the backend of our application
- Azure Document Intelligence - The API used to process receipts and speech-to-text output
- OpenAI Whisper - The API used to convert user voice logs describing expenses into text

Database

- MongoDB Atlas - A cloud-based database hosting service that we used for our database
- MongoDB Compass - Software used to visualize and manage the data within our database

Development Tools

- Visual Studio Code - The chosen IDE where our team programmed and collaborated together
- Github - Our chosen collaborative tool to merge and manage our work

3.2 Code Documentation

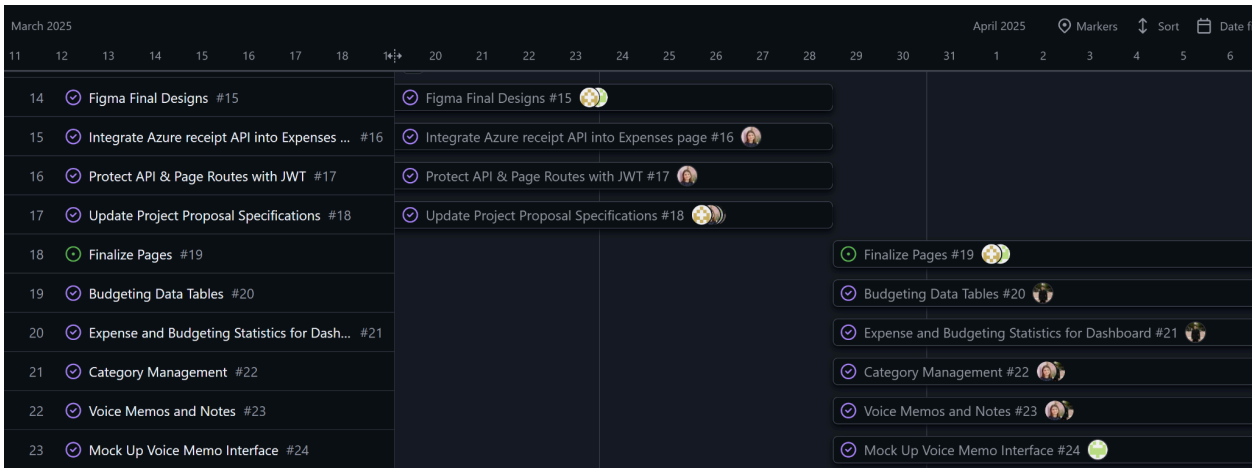
[GirlMath Github]

When a user signs up for our app and inputs their credentials, a user object is created in the database and stored for future logins. The user can then log in by providing a username and password combination that matches an object in the database. Upon successful login, a JSON Web Token is generated from their username and securely stored in the browser's cookies. This token is called upon throughout the app to authenticate the user and ensure their data is displayed.

When a user uploads a receipt, it is processed by the Azure API, and the expense tracking form fields are automatically populated with the information returned by the API. The user can then fill in any fields which may be missing or edit the data before submitting the expense. Similarly, when a user records a voice log describing an expense, the audio is transcribed by the Whisper API, then processed by custom regular expressions as well as the Azure API to automatically populate the form fields. The user is also able to manually input an expense by filling out the form, and when an expense is submitted, a new expense object is created in the database under that user. If the category listed with the expense is new, a budget object is also created under the user with that category name.

On the profile page, the user can manage their budgets which are retrieved from the database and displayed in a list. The user can enable editing and change the intervals and/or amounts for their budgets. They can also add a new budget category or delete an existing one. On the dashboard, when the user filters by category, the visible categories array is modified to include only the selected categories. The data displayed throughout the page in the recent expenses, line chart, and donut chart sections, is pulled from the database for only the categories in the visible categories array. All aspects of the page display data from the last six months dating back from the user's most recent transaction.

3.3 Development Process



A snapshot of our team's Github Projects board for sprint planning.

Our team used Github Projects to keep track of tasks, assign team members, and plan future development. We worked in two week sprints and shifted team leaders every three weeks. We communicated primarily through our team's Discord server, where we held weekly meetings, discussed bugs, and shared resources. Additionally, we maintained a shared Google Drive folder to store research, meeting minutes, and progress reports. We also uploaded these to Github at the end of each week. Every week, each team member worked on their own Git branch, and at the end of the week, we merged the branches and pushed the working version to main to ensure continuous integration.

Chapter 4: Performance

4.1 Testing and Validation

We used Postman to practice API calls directly and to test our app's API calls. When integrating the Azure API, we tested the authentication and connection through Postman, and we practiced the send document and get results calls to better understand them. Then, we tested our own app's API routes, both to independent endpoints like our login logic as well as the fully integrated send document and get results calls. Postman was central to the integration of the Azure API as well as testing all our app's API routes that we built with Next.js.

To assist with debugging, our team utilized a mixture of in-code console logs along with Google Chrome and Firefox's web debugging console tools. These console logs sent specified messages to the web debugging tools to help us discern between different console logs or to output a certain problematic variable that we would want to keep track of at certain points of our app's runtime. These tools helped us pinpoint where in our code any error may have occurred, which made debugging much more efficient. To reinforce this method, in much of our backend code we had code to throw errors everywhere we could. This sent error messages to our Visual Studio Code consoles instead of the web debugging one, which helped us discern between different types of issues we could encounter. This also allowed it so that our whole application would not crash on one error, as it would stay isolated which helped us access further console logs to once again pinpoint any issue.

Each team member contributed to manual testing both weekly as we continuously integrated our application and as we wrapped up development. This was integral to ensuring that new features worked as expected and old features were not broken by any new changes. It also helped us realize improvements that could be made to the app, as we were actively practicing using it.

4.2 Metrics Collected and Analysis

Page Compile and Initial Load Times

Page	Compile Time	Load Time
/home	4.3s	5635ms

/login	5.1s	5724ms
/signup	1587ms	2423ms
/middleware	841ms	N/A
/dashboard	2.2s	2554ms
/expenses	5.5s	5886ms
/profile	4.5s	4908ms

These metrics are as expected for the complexity of each page in our app. Signup compiled quickly due to the shared resources it has with login. The middleware compiled the quickest as expected, since this is a simple route for verifying users are authenticated. Load times ranged between about 2400ms to about 5900ms, which is within our expectations. Since we knew some of the pages take a long time to compile and load on first access, we added loading animations to each page so the user would know our app was working and remain patient.

Expense Logging Times

Logging Method	Processing Time	Time per Entry
Manual inputs	0.00s	20.13
Receipt upload	5.60s	15.12
Voice logging	5.81s	14.34

The average time taken for our app to process a receipt from submission to the results appearing in the form fields is **5.60 seconds**. For voice logging, this metric is slightly higher at **5.81 seconds**, which is logical considering the voice logging feature makes the same API call as receipt processing and incorporates additional logic. Manual entry does not involve any automated processing, so it takes **0.00 seconds**.

Although the user has to be patient while inputs are processed for receipts and voice logging, manual inputs take the longest overall time per entry at **20.13 seconds**. Receipt uploads, from file select to expense submission take an average of **15.12 seconds**. Voice logging, from the start of recording to expense submission takes an average of **14.34 seconds**. This makes voice logging and receipt upload comparable in terms of overall time spent.

Chapter 5: Lessons Learned

5.1 Insights Gained

As a team, working on this project gave us invaluable experience. Through ideation, user-focused design, and continuous feature implementation, we learned how to create an application from the ground up. One important lesson learned was in the value of user-centric thinking, because instead of adding features for complexity and wow factor, we were able to focus on improving our application to satisfy actual needs. Working as a relatively small team of 4 also gave us the opportunity to learn from one another and develop in ways that we might not have done separately.

5.2 Lessons from Challenges

One challenge that kept coming up for us was managing continuous integration. We decided early on in development that we would merge our separate GitHub branches weekly, which sometimes led to conflicts as our individual code didn't fit together perfectly. Even though this was frustrating at times, it helped us build better habits as developers and made our final product much cleaner. There were also a few instances of miscommunication within the team where two of us ended up working on the same element or feature without realizing it. To fix this, we started planning more clearly each week and clearly stating at the start of each sprint the explicit things we would be working on. Another tricky task was ensuring that our app would appeal to a wide range of people. While we did have a target audience in mind from the start, students, we began to worry that our pink theme and application name referring to a pop culture phrase might make some users feel left out. We overcame this by being intentional with our messaging and ensuring that the app would be useful for everyone, whether they were in our target audience or not.

5.3 Skills Developed and Improved

Technically, we learned a lot about working with React, especially when it came to managing states, building reusable components, and using Next.js to build an efficient frontend. We also sharpened our backend skills by utilizing MongoDB to set up secure routing and handle user authentication. Integrating external APIs like Azure Document Intelligence and OpenAI Whisper were both completely new to the team, but proved to be so helpful in implementing specific features in our application.

Aside from technical development, we also got better at managing, communicating with each other, and solving problems as they arise. Whether we were stuck debugging for hours or just coordinating everyone's schedules to find a time to meet, we became more effective at resolving conflicts and keeping our workflow on schedule.

Chapter 6: Future Work

6.1 Proposed Enhancements

The future of GirlMath is bright! There are several enhancements that could make the app even better. We would love to set up bank account integration for users through our application, so transactions could be tracked automatically rather than manually input. This would save time and also give user's a more accurate picture of their finances. Another big area of improvement would be to make the app more mobile-friendly. Whether that means creating a responsive web design or building a dedicated mobile app, this would make it much easier for students to keep track of their money on the go. We are also thinking about adding analysis features that would go over user spending patterns and offer personalized tips for improvement (ex: areas where a user might want to cut back, suggest ways to budget better, etc.). Finally, small changes to the UI would enhance the user experience. Adding pop-up windows for logging expenses could make the app more user-friendly, so instead of having a large portion of a page allotted to logs, we could minimize on-screen clutter.

6.2 Recommendations for Development

Looking ahead, it is important that we get our application into the world by hosting it on a public platform. This way, more people can try it out and we would be able to get feedback from real users! As the app grows, we hope developers keep the clean and simple design that makes GirlMath stand out from other, more complicated finance tools. While it's tempting to add new features, we want to make sure every update keeps the app easy and accessible for everyone. Finally, having clear instructions for both developers and users will make it easier for new team members to jump in and for new users to get started. Things like in-app guides or simple onboarding would help our users feel comfortable with the app right away. We feel that keeping these things in mind in the future will give our application the potential to keep growing.

Chapter 7: Conclusion

7.1 Summary of Key Accomplishments

Over the course of the past semester, we were able to design and build a fully functional, student-focused web app that achieved all of our initial objectives and more.

We built a clean and intuitive UI that allows users to easily manage their finances without prior experience in budgeting. The design and functionality of GirlMath reflect the real needs of students and other first-time budgeters.

We used technologies like the Azure Document Intelligence API and OpenAI Whisper to make the process of expense entry as quick and easy as possible for users. Users can upload pictures of

receipts or record voice memos to automatically populate the expense fields, allowing users to add expenses with the click of a button.

On the dashboard of GirlMath is where you'll find a dynamic pie chart and bar graph that allows uers to visually take in all of their spending data at a glance. Users can filter by categories, track their progress against budget goals, and see all of their data update live as they change filters.

We implemented secure user authentication using JSON Web Tokens and our MongoDB Atlas database, and ensured proper associations between transaction and budget data.

Lastly, our team completed the development process using tools like GitHub Projects and met twice a week, which ensured constant communication. We shared resources, research, and meeting minutes in a shared Google Drive, and integrated our Git branches together every week to ensure we were all on the same page.

7.2 Acknowledgements

Thank you to Professor Alagar, as well as our TA, Thenn, for giving us constructive criticism and pointing out potential areas for improvement every week as we were building our project. Your feedback was invaluable, helped us produce a well thought out application, and continues to inspire us to do more.

References

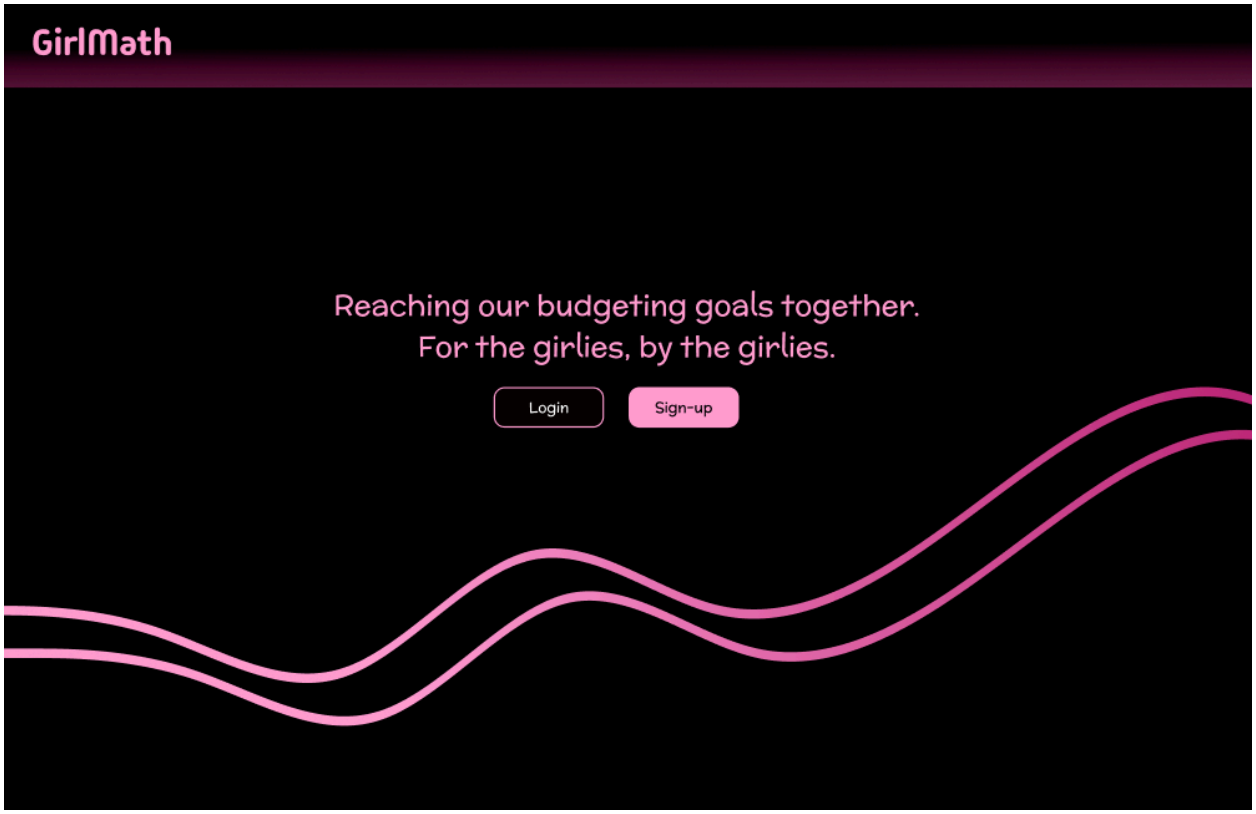
https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/prebuilt/receipt?view=doc-intel-4.0.0

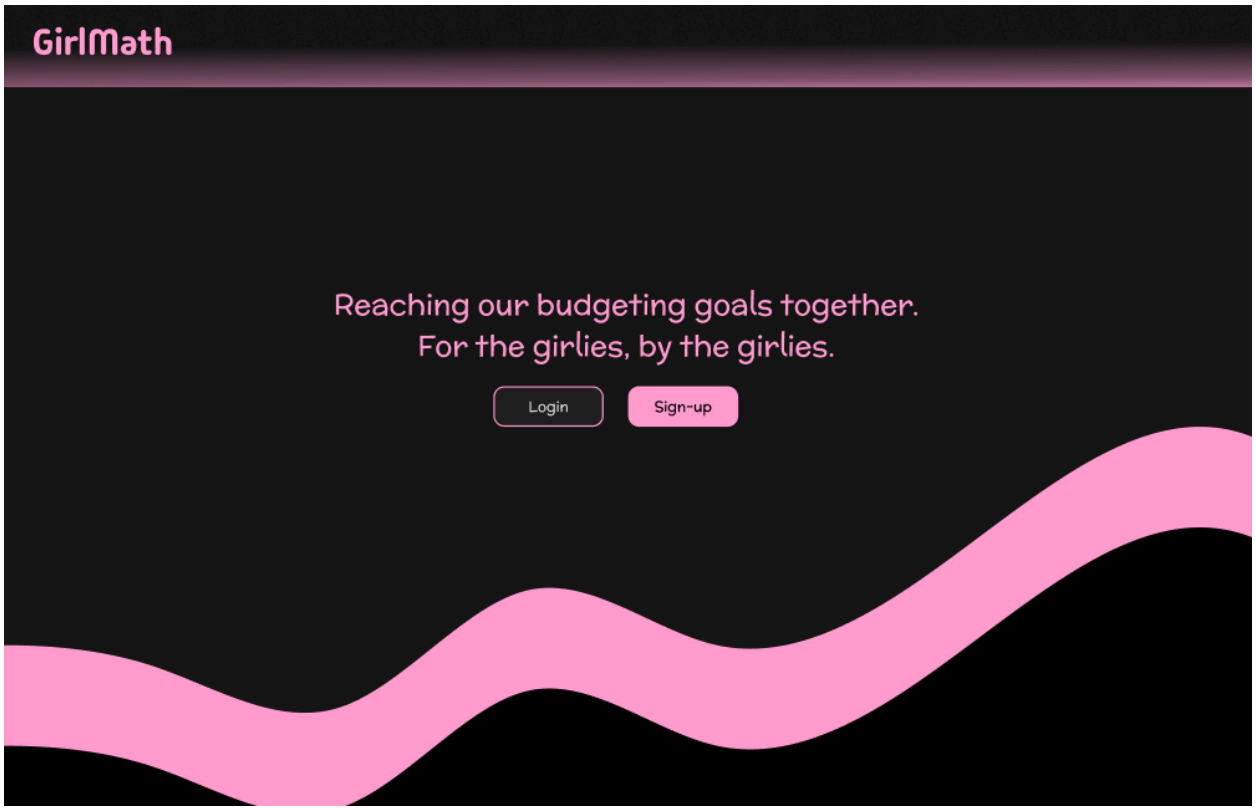
https://platform.openai.com/docs/guides/speech-to-text

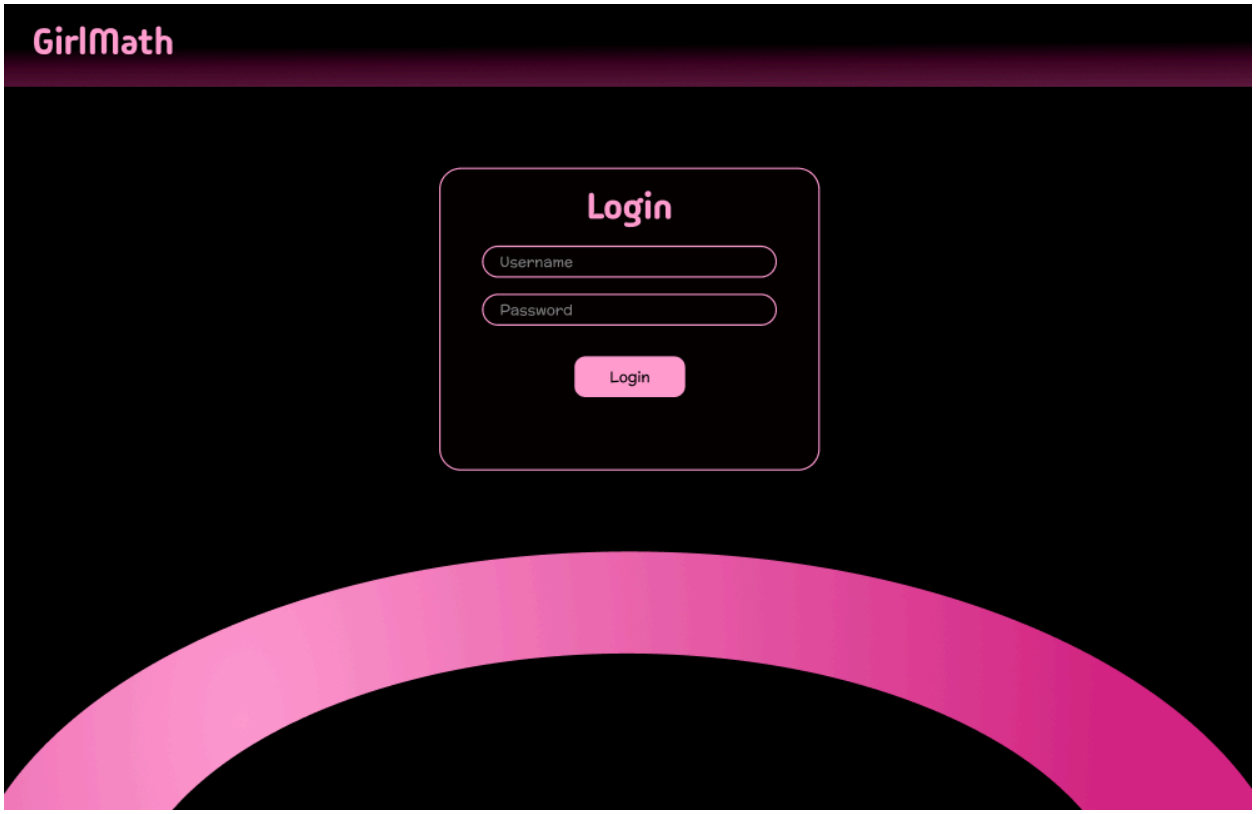
https://www.npmjs.com/package/react-bezier-curve-editor

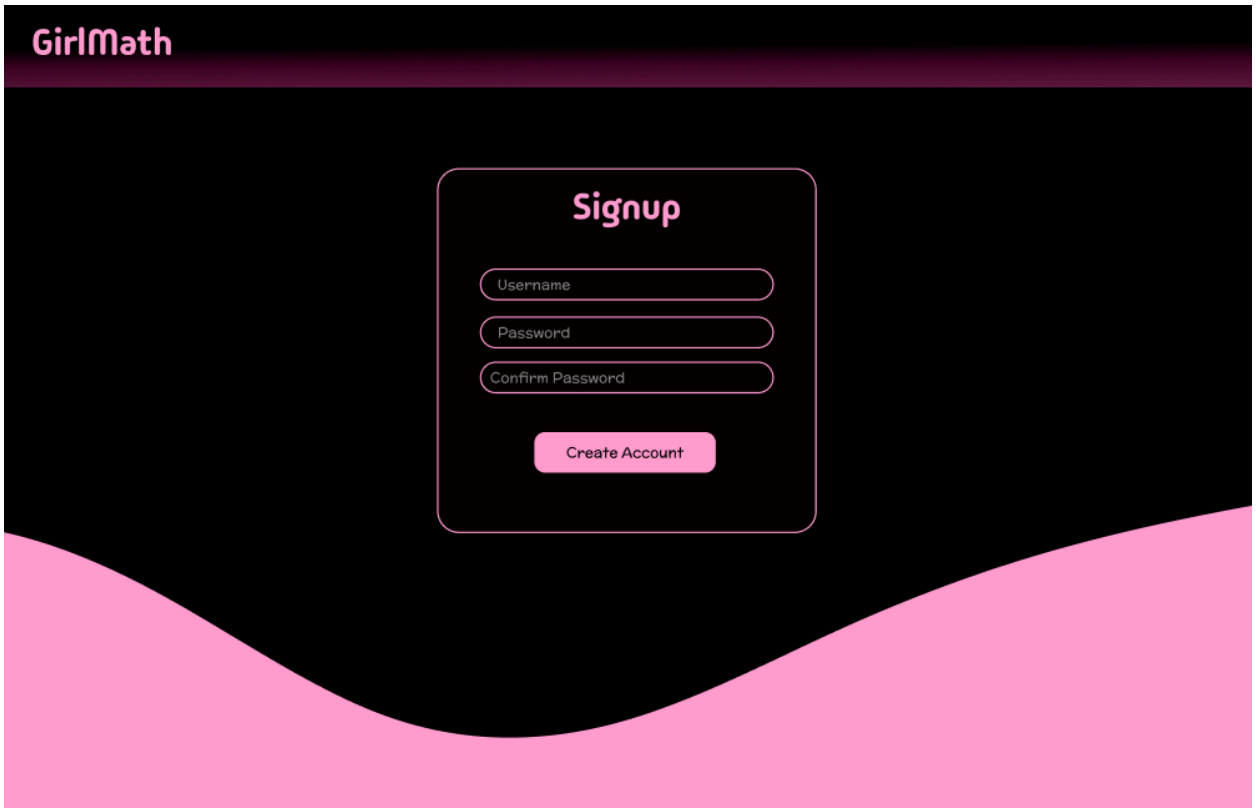
Appendices

Some of our design iterations for the app:


GirlMath
Reaching our budgeting goals together.
For the girlies, by the girlies.
Login
Sign-up


GirlMath
Reaching our budgeting goals together.
For the girlies, by the girlies.
Login
Sign-up


GirlMath
Login
Username
Password
Login


GirlMath
Signup
Username
Password
Confirm Password
Create Account

Some of the code in our OpenAI Whisper API call:

```
export const runtime = "nodejs";



import { NextResponse } from "next/server";

import axios from "axios";

import FormDataNode from "form-data";



export async function POST(req) {

  try {

    const formData = await req.formData();

    const audioBlob = formData.get("audio");



    if (!audioBlob || typeof audioBlob.arrayBuffer !== "function") {

      return NextResponse.json({ error: "No audio file given" }, { status:
400 });

    }



    const buffer = Buffer.from(await audioBlob.arrayBuffer());



    const openaiForm = new FormDataNode();

    openaiForm.append("file", buffer, {

      filename: "audio.ogg",

      contentType: audioBlob.type || "audio/ogg",
```

```
    });

    openaiForm.append("model", "whisper-1");



    const apiKey = process.env.OPENAI_API_KEY;



    const response = await
axios.post("https://api.openai.com/v1/audio/transcriptions", openaiForm, {

      headers: {

        Authorization: `Bearer ${apiKey}`,

        ...openaiForm.getHeaders(),

      },

    });



    return NextResponse.json({ text: response.data.text });

  } catch (error) {

    console.error("Whisper API error:", error.response?.data ||
error.message);

    return NextResponse.json({ error: "Failed to transcribe audio" }, {
status: 500 });

  }

}
```