CSE 546 — Hunger hurts: Food Recommendation System

Shivangi Singh
Yash Bhokare
Sayali Papat

1. Introduction

In our hectic daily lives, we frequently forget to buy groceries. It is not uncommon for us to decide to make a recipe only to discover that we lack all of the necessary ingredients, forcing us to either abandon the plan or pay a higher price for the ingredients. To tackle this issue in general, we present a website that analyzes users' weekly meal plan and assists him in following it in a most efficient way.

Aim: To create a web application for maintaining and analyzing a user's weekly meal schedule, as well as assisting him with it by suggesting the recipe, listing the ingredients needed, and providing their current prices, as well as interactively assisting him in preparing the dish by reading out the recipe to him.

2. Background

We all have a foodie inside us, and with this nature comes cravings for various dishes. But, due to our hectic daily schedules, whenever we decide to cook a meal, we find ourselves in a situation where we lack all of the ingredients needed to prepare the dish, and we end up canceling the plan. But what if we had a website that would take our weekly schedule and suggest a recipe with a list of all the ingredients needed? This would relieve us of a lot of work and allow us to enjoy our favorite dishes.

We have developed a website that allows users to maintain a weekly meal schedule, and the website will check available recipes for preparing the dish and will list down the necessary ingredients. It will then fetch the prices of the items on the list across a shopping site and display them so that the users can add them to the cart.

We are further using a LDA model that will make recipe recommendations based on the user's preferences. We intend to analyze different recipes' datasets based on various parameters such as the type of ingredients, the number of steps in the recipe, the number of ingredients, user reviews and ratings, and provide personalized recommendations. We are doing topic modelling to provide personalized recommendations.

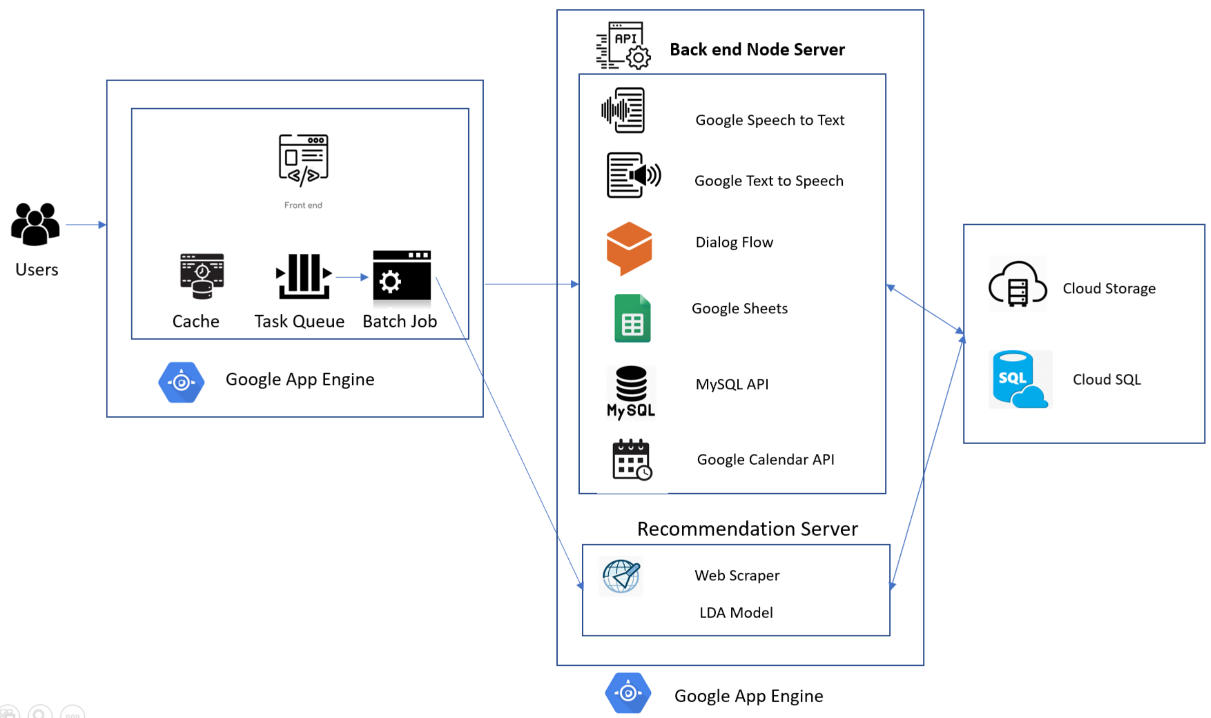
To make it more interactive, we added a feature in which we developed a personal assistant bot that would interact with users in real-time and read out the recipes to them. This would assist the user to follow any recipe, without the need of him actually reading it.

Current Solutions:.

Food.com is a pre-existing food-recommendation service. It organizes recipes into categories such as popular, breakfast & brunch, and so on. Such pre-existing food/recipe recommender systems, meanwhile, make suggestions based on the tastes of consumers as well as their dietary requirements. They do not, however, offer a weekly meal plan based on the user's food preferences. Further, they do not provide the current price of each ingredient.

3. Design and Implementation

1. Application Architecture:



Major Components:

1. Front-end App

On the front-end we use an angular app to build the functionality. The design is inspired from food.com. The journey is divided into 4 steps:

Step 1:

User logs into the application through a Login Page that consists of username and password. It then redirects the user to the portal. For the first time user, he needs to enter his details on the User Registration page which consists of a questionnaire related to preferences on food items. After registration, the user is directed to the portal.

Step 2:

On the home page we list down a bunch of recipes for users to select from or search it from the search tab. The user chooses a dish that he wants to eat and adds it to the calendar to schedule his weekly meal plan. We used Google Calendar to sync the user's schedule to his Google account, and we also provided an in-app calendar for viewing the plan.

Step 3:

In the following step, we list the recipe of the dish selected by the user and display its content, such as the dish description, recipe, and the ingredients needed. The user can then choose the ingredients he needs and check their current prices on a shopping portal.

Step 4:

When the user decides to make the dish, we provide a personal assistant who will interact with him in real time and read out the recipe to him. Voice commands are used by the personal assistant to interact with the user.

2. **Back-end App**

   NodeJS is used to create the back-end app. It is hosted on a separate Google App Engine instance. The goal is to make the application components loosely coupled. This ensures individual scalability.

   In this following major components are hosted:

   1. ***Personal Assistant:*** This component provides a way to interact with the user on a real time basis. It is developed using Google cloud’s Dialogflow and google’s speech to text and text to speech API.

      When a user needs to make a dish, he uses voice commands to interact with the personal assistant. This allows for a more interactive interaction with the users. In the Dialog Flow, we used training phrases such as next step, go to next step, previous step, and proceed, which the user uses as voice commands to interact with the bot. When the user says next or uses any other phrase, it is converted from speech to text using google API and then the next step is fetched from the database by the node server. This is then converted into speech and delivered to the user via the Google API. As a result, the user does not have to read the recipe himself; instead, the bot does so for him. The mp3 files generated during text-to-speech conversion or when the user uses voice commands are saved in the cloud storage.

2. ***Recommendation*** : It provides personal recommendation to the user. It interacts with the ML models and the Google SQL database.
3. ***Login and Registration System***: It takes personal information including from the user and stores it in the database. It allows the user to log into the application by entering username and password.

3. **ML model and Web Scraper**

ML model:

We begin by assuming that users prefer recipes that are close to recipes they have previously enjoyed. To put it another way, if a consumer enjoys chocolate cake, it's a safe bet that he or she would enjoy chocolate tart as well (consider them similar).

We use LDA to extract features from recipe ingredients: each recipe is transformed into a matrix represented by a bag of words model, and then the LDA model processes and generates a linear combination of possible words that contribute to the topic. The recommendation model calculates the "similarity" or distance between the recipes in the database and then "remembers" it (memory).

We use this similarity between the recipes to give personalized recommendations based on the topics that emerge. We are using a database from Kaggle to perform the analysis.

We examine the dataset's recipe data and try to uncover popular topics among the recipe data. Topic modeling assists us in identifying hidden topic blocks in data. The user will select his meal plan for a week. We based on which food dishes the user adds to his meal plan, recommend him the similar recipes falling under the similar topics.

Web Scraper:

We are using web scraping for fetching the current prices of the items from amazon website. It fetches recipe details from Google Sheets and web scaps the current price of the items from a shopping portal. It then provides the user the best price of the ingredients. This is added to the user’s cart and the total price of the food item is displayed. Google Sheets API was used to collect ingredients details of the recipe. Libraries like BeautifulSoup and Requests were used to scrap the shopping portals. The price and product name was then returned to the user into the google sheets which was updated into the web portal.

4. **Database**

We are using Google Cloud MYSQL 8.0 as the database in this application. It stores the user's personal details including username and password and food preferences.

This database is further used to store the list of dishes and their recipes, with the other information like, list of ingredients, number of ingredients, number of steps and the description of the dish. This is the dataset on which the ML model is being trained.

5. **External APIs**

   Google Calendar API is used to add the users meal plan to his google account. Google text to speech and Google speech to text API are used to provide a personalized bot and it interacts with the user using voice commands. Google Sheet API is used for fetching and storing ingredients information which is used in web scraping the price details from Amazon.

2. Cloud Services used:

1. **Google App Engine**

   GAE is used to separate the deployment of front-end and back-end applications. The front-end app engine sends HTTP requests to the back-end application, which is hosted on a different GAE. Our ML model and web scraper are deployed on another GAE.

2. **Google Cloud SQL**

   The application's MYSQL database is hosted by Google Cloud SQL. For data querying, the GAE hosting the back-end and the ML model plus the websraper communicates with Cloud SQL instance.

3. **Google Cloud Storage**

   Front-end applications' static data/files are stored in the cloud. It is also used to store app-related files for the application that is deployed in the back-end. The speech mp3 files are also stored in cloud storage.

4. **Google Dialoflow**

   The Chatbot developed for providing personal assistance to the user uses Google Dialogflow, which is a natural language processing (NLP) platform used to build conversational applications.

3. Role of Google App Engine:

The front end angular app, back-end node app and the ML model plus the web scraper are hosted on different GAE instances. We hosted them on different instances to have a decoupling between the application components. The reasons behind using google app engine for hosting application are:

1. Deploying applications is very easy, since no deployment configuration is required as it is a PaaS offering. We just need to specify the runtime to be used.

2. It has many built-in APIs which helps to build robust applications.
3. Highly scalable and autoscaling is managed internally by GAE
4. Systematic application error logging mechanism.
5. Pay as per usage: you have to pay as per the usage of resources as the application grows.

4. Autoscaling

Autoscaling refers to an application's ability to scale in/out automatically based on the application load in order to satisfy all requests in a seamless manner.

The definition of load varies depending on the application. In the context of this application, load can be specified as the number of requests the website is receiving at any given time. In addition, the time it takes to satisfy each request can vary depending on the user input. As a result, the essence of the input parameters may be used to determine if a request is lightweight or strong.

Using the above two parameters, GAE manages to scale out/in the resources to meet the new requirements based on the current load whenever the load on the website increases/decreases. GAE includes this type of auto scaling by default. As a result, the application developer is relieved of the responsibility of implementing autoscaling for GAE-deployed applications. One of the most compelling reasons to choose GAE is because of this.

5. How does the application solve the problem? How is it different from others?

Our solution provides a weekly meal schedule based on the user's food preferences. It provides a personalized recommendation system to the user. It checks the ingredients required for preparing the dish and fetches the prices of the items on the list across a shopping site. It displays them to the users and adds them to the shopping cart. Further, the application has a personalised bot which reads out the recipe to the user and assists them in cooking.

4. Testing and evaluation:

As a part of stress testing to witness autoscaling we used Apache Bench, a tool for benchmarking https web servers. We ran the below command.

abs -n 10000 -c 200<sample request url for the website>

This runs 10000 requests to perform the benchmarking session and does 200 requests concurrently. The results of the tests are shown below

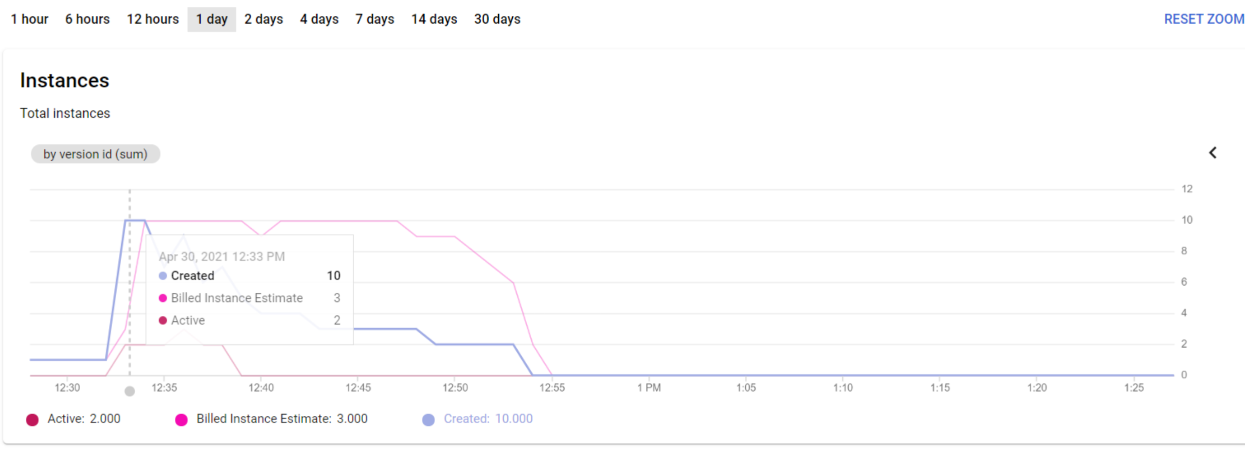


Figure 2 : Instance during stress testing

For ML model testing we tested, the response of the model with different user recipe choices and compared it with the topics generated.

We even tested the code by changing the number of topics extracted and other model parameters to check the accuracy of the model. By following these steps we tried to fit the model with best feasible parameters.

We have tested the application against different constraints for all options available to the user. We performed testing for multiple users and checked the response of the application as well as scaling up and scaling down of the instances.

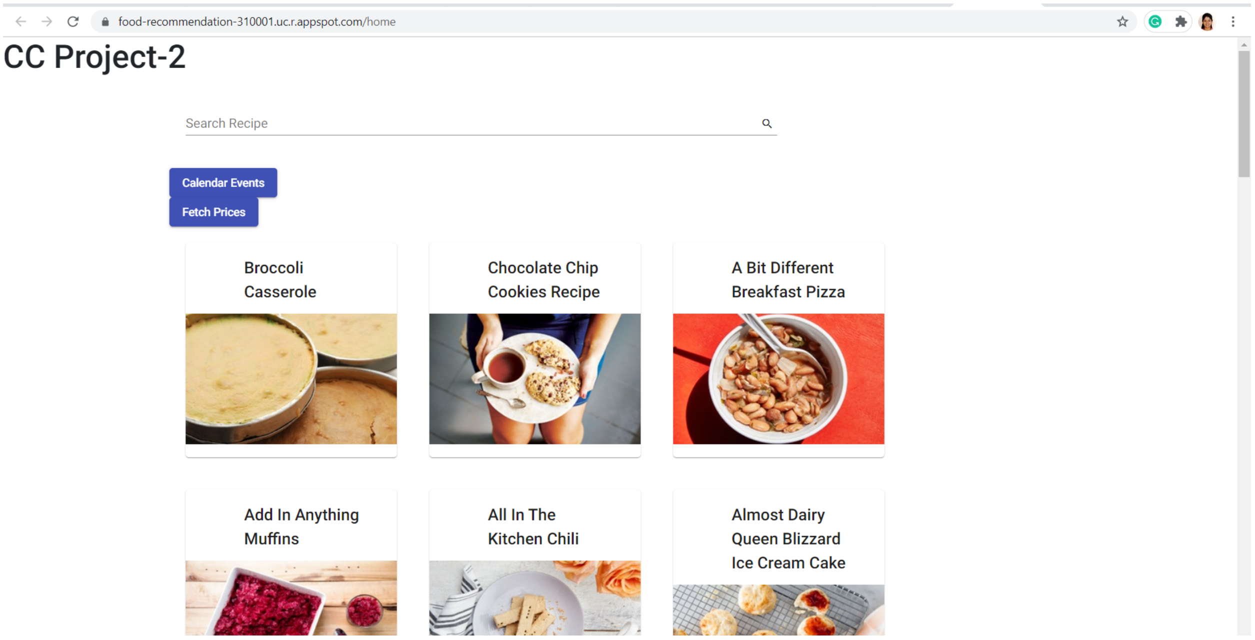
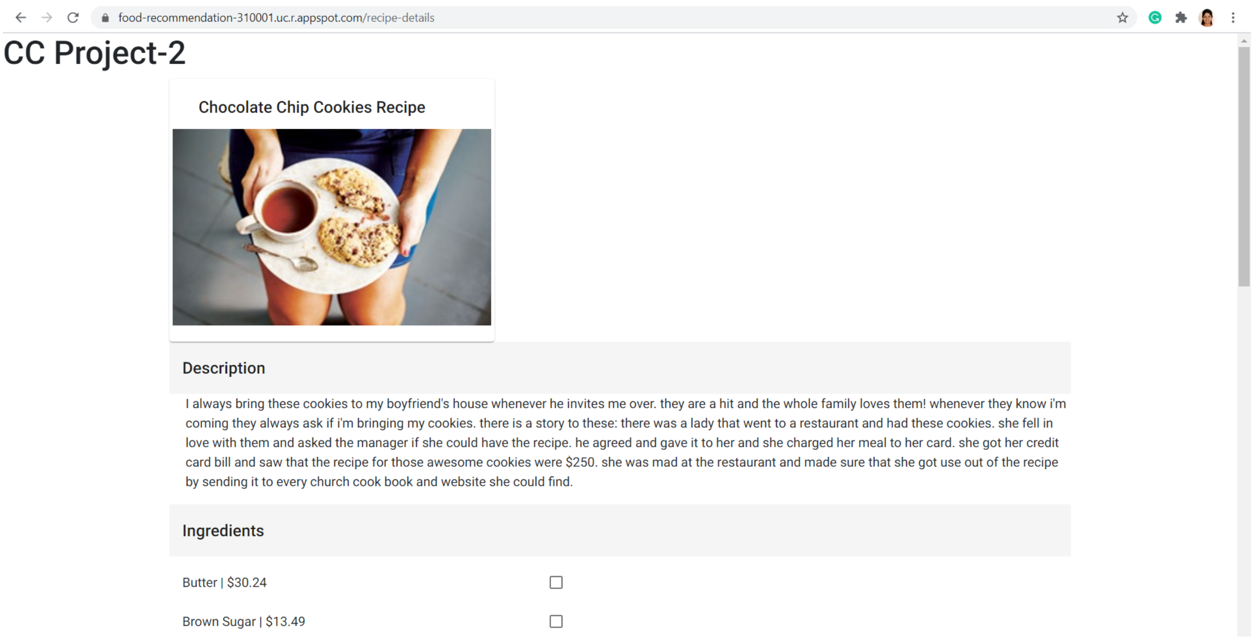

food-recommendation-310001.uc.r.appspot.com/home
CC Project-2
Search Recipe
Calendar Events
Fetch Prices
Broccoli Casserole
Chocolate Chip Cookies Recipe
A Bit Different Breakfast Pizza
Add In Anything Muffins
All In The Kitchen Chili
Almost Dairy Queen Blizzard Ice Cream Cake

Figure 3 : Users home page


food-recommendation-310001.uc.r.appspot.com/recipe-details
CC Project-2
Chocolate Chip Cookies Recipe

Description

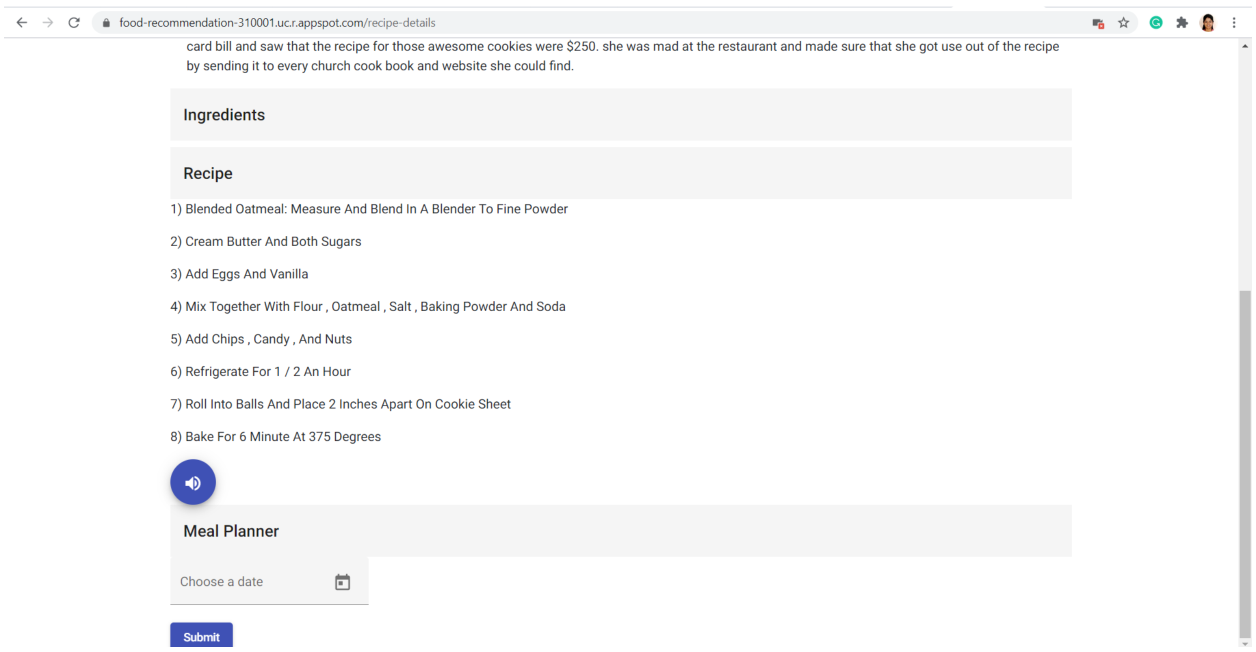
I always bring these cookies to my boyfriend's house whenever he invites me over. they are a hit and the whole family loves them! whenever they know i'm coming they always ask if i'm bringing my cookies. there is a story to these: there was a lady that went to a restaurant and had these cookies. she fell in love with them and asked the manager if she could have the recipe. he agreed and gave it to her and she charged her meal to her card. she got her credit card bill and saw that the recipe for those awesome cookies were $250. she was mad at the restaurant and made sure that she got use out of the recipe by sending it to every church cook book and website she could find.

Ingredients

Butter | $30.24 ☐

Brown Sugar | $13.49 ☐

Figure 4 : Details of the recipe selected by the user


food-recommendation-310001.uc.r.appspot.com/recipe-details

card bill and saw that the recipe for those awesome cookies were $250. she was mad at the restaurant and made sure that she got use out of the recipe by sending it to every church cook book and website she could find.

Ingredients

Recipe

1) Blended Oatmeal: Measure And Blend In A Blender To Fine Powder

2) Cream Butter And Both Sugars

3) Add Eggs And Vanilla

4) Mix Together With Flour , Oatmeal , Salt , Baking Powder And Soda

5) Add Chips , Candy , And Nuts

6) Refrigerate For 1 / 2 An Hour

7) Roll Into Balls And Place 2 Inches Apart On Cookie Sheet

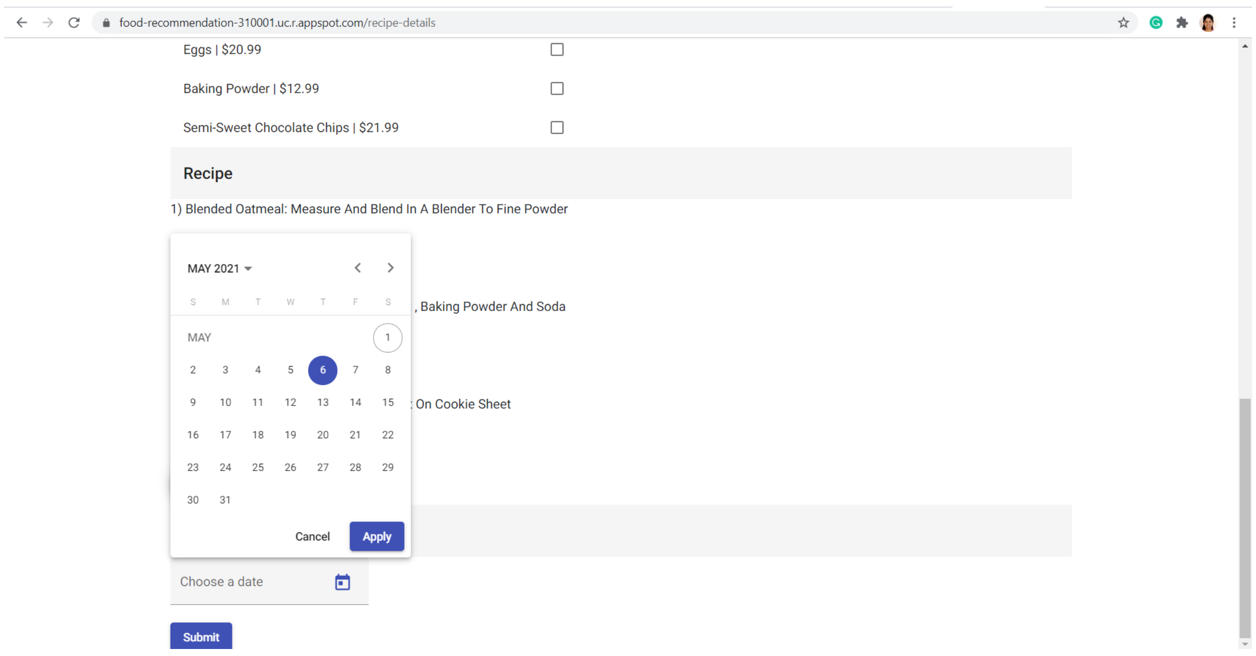
8) Bake For 6 Minute At 375 Degrees

Meal Planner

Choose a date

Submit

Figure 5 : step by step recipe of the dish selected


food-recommendation-310001.uc.r.appspot.com/recipe-details

Eggs | $20.99

Baking Powder | $12.99

Semi-Sweet Chocolate Chips | $21.99

Recipe

1) Blended Oatmeal: Measure And Blend In A Blender To Fine Powder

MAY 2021

S	M	T	W	T	F	S
MAY						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Cancel Apply

, Baking Powder And Soda

On Cookie Sheet

Choose a date

Submit

Figure 6 : Adding the selected recipe to weekly meal plan

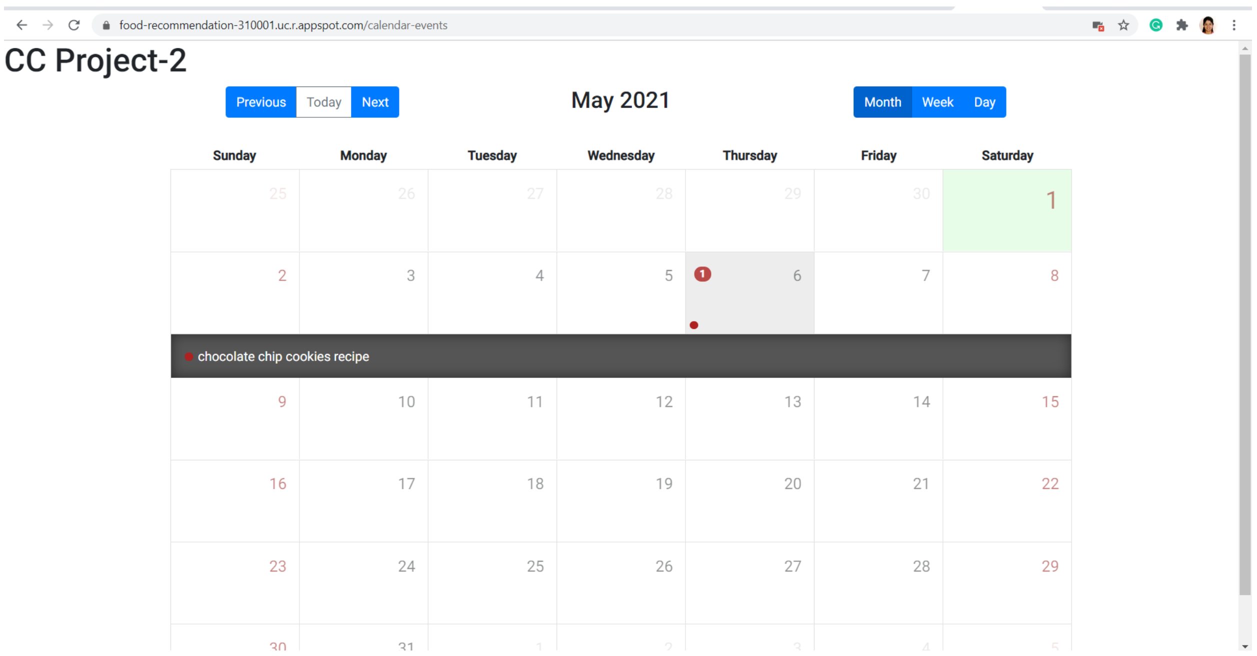


Figure 7 : Checking the calendar for weekly meal plan

,

5. **Code:**
    - Backend:
        - Node Server
            - The node server was deployed and hosted using the Google App Engine.
            - Database.js : Declaration of the configurations for connecting google mysql database.Creates an open connection and exports the file.
            - App.yaml:
                - Used for hosting the node server with runtime and environments declared. Also used for declaring automatic_scaling details.
            - Google_Calendar.js :
                - Responsible for storing events on particular dates in the google calendar.
                - In the start declaring authorization using the google auth jwt token with given secret keys as inputs.
                - dateTimeForCalander(date): Converts the given date into the specific date format required by google calendar.
                - insertCalendarEvent(date, title, description) :
                    - Responsible for querying google calendar API with given authorization and passing in the events specified.
                    - Creates an event for a given date, title and description.
                    - Pass this event to the google calendar API.
                    - Returns true if the call is successful, else returns false with error displayed.
            - Google-sheets.js:
                - Responsible for storing ingredients list inside google sheets and retrieving prices and titles of the given ingredients.
                - In the start we declare a specific ID of the sheet along with their credentials . The credentials are imported from the JSON key provided for access to google sheets api.
                - storeValuesinSpreadSheet(Array ingredientsList):
                    - It stores the given ingredients values individually at each row then it returns true once the job is completed.
                - readSpreadSheet():
                    - It fetches the given ingredients value, their title and prices from the sheets and returns it separately.

- Google-Text-To-Speech.js
    - Responsible for converting the given input text to audio format using **google cloud text-to-speech api**.
    - After the conversion the given binary file is saved as a mp3 and stored at given google cloud storage.
    - During the declaration it creates a connection to cloud storage with the given parameters.The connection is established using **google cloud storage api** along with given authentication keys.
    - recepieAudio() :
        - It takes text and index as an input.
        - It makes a call to synthesizeSpeech with the given request. In return we get an audioContect response.
        - The response is then stored on the google cloud storage with the given Step name and index specified.
        - It then returns true if calls are successful else it returns false with the error message.
- Google-speech-to-text.js
    - Responsible for converting a given audio buffer into specific text and returning it .
    - googleTextToSpeech(audioBuffer):
        - It takes audioBuffer as an input which is recorded on the front end side and it's fetched into the google speech to text api.
        - It then returns the given transcription back.
- Google-dialog-flow.js
    - Responsible for understanding and matching a given query provided by the user for a specific response.
    - The dialog flow is trained in which multiple inputs are fetched in and mapping is done to specific responses.
    - In the code we make a connection to google dialogflow api using the given keys and project id.
    - runDialogFlow():
        - Takes user queries as an input. It then makes a call to api and fetches the response text. The response text is then sent back to the users.
- Server.js
    - Contains all dependencies and routing of functionalities.
    - It imports all the functionalities.

        - It uses express to host the application over port 8080.
        - Adding cors to bypass the Cross-Origin Resource Sharing issue.
        - Adding multiple get and post requests for given functionalities.
        - Requests used:
            - GET -> api/recipeList : Returns a list of all recipes in the cloud SQL database.
            - GET -> /api/recipe/:Title : Returns the details of a given specific recipe based on the title from the database.
            - POST -> /api/calendar : Passes the title and description details to google calendar file.
            - GET -> /api/getIngredientsPrice : Reads the details from google sheets and returns it.
            - POST -> /api/storeIngredients : Stores the given ingredients onto the google sheets.
            - POST -> /api/upload_sound : Reads in the given audio buffer and passes it to googleTextToSpeech. Based on the response it is given back to the dialog flow and response is sent back to the user.
            - POST -> /api/send_recipe_step :Passes the input string to Google-Text-To-Speech file.

- Frontend
    - The frontend was created in Angular 11 and was deployed over the google app engine . URL : https://food-recommendation-310001.uc.r.appspot.com/
    - User Login Page
        - It consists of a Username and Password field.
        - Submit button directs user to the home page
    - User Registration Page
        - It consists of personal information of the user.
            - First Name
            - Last Name
            - Gender
            - Age
            - Email
            - Mobile
            - Current Region
            - Nationality
            - Favorite Food

        - Allergies
        - Calorie Intake
        - Diet Plan
    - Submit Button: it stores the data into the database.
- Home Component:
    - First it displays 2 buttons Calendar Events and Fetch Prices.
    - Calendar Events -> Calls Calendar Component
    - Fetch Prices -> Calls state service to fetch the latest prices of specific ingredients.
    - The home page then fetches all the recipe list from the state service and displays it in card format to the users. The state service makes a call to the node server to fetch all recipe details and returns it back to the UI.
- Search Component:
    - Allows users to type in and search any given recipe.
    - When a user selects a recipe a call is made to the state service to fetch details of that recipe and then the user is redirected to the recipe details page.
- Calendar Component:
    - Displays a UI for calendar with Month, Day and Week parameter selections.
    - The recipe events selected by the user for a weekly plan are stored in the given specific dates.
    - A page for a user to browse through the selected recipes.
- Recipe Detail Component
    - It first fetches the given recipe data from state service then it fetches the price list for the recipes ingredients.
    - Then in the UI we display the:
        - Recipe Title and Image in a card format
        - Recipe description below it.
        - List of ingredients along with their price.
        - Then the steps for the recipe.
        - A volume button to activate the personal assistant.
        - A calendar to select the date for the weekly plan.
        - Submit button
    - Personal Assistant :
        - When the user clicks on the personal assistant button we make a call to the state service to convert the text to speech and read it out to the user. Then once the audio is finished playing we record

what the user wants to say. Once an audio buffer is captured we send it to state service to understand the meaning of it. It either responds with Next, Pause, Stop, Previous.

        - If we get any of the responses the code then handles it in the given way:
            - Next: In this we take the recipe to the next step and continue the recipe reading process.
            - Pause: We pause the recipe reading process and wait for the user to continue.
            - Stop : We kill the given process and reset the recipe step count to 0.
            - Previous: If any previous steps are there we go to the previous step in the recipe step process.
        - In this way the user can interact with the bot with audio commands only.
    - Submit:
        - When the submit button is pressed the calendar event in state service is called with the select date and it stores the date inside google calendar. Also the ingredients are added to the google sheets using the node server api call.
- State Service
    - Responsible for handling the caching part of data, redirecting calls to server API and returning values when needed.
    - The state service contains multiple functions which make a HTTP call to the hosted server to fetch data for the given functionalities.
    - fethchRecipeListFromServer() : Makes a HTTP call to fetch the list of all recipes.
    - fetchRecipeByName() : It fetches the details of the recipe based on the title provided.
    - setCalendarEvent() : Stores the calendar event locally and makes a call to the server to store it on the google calendar as well.
    - fetchRecipeSteps() : Sends the data and file name to the server to convert it to audio file.
    - sendAudioCommands(): Sends the audio commands specified by the user and returns a text translation of it from the server.
    - fetchIngredientsPrices() : Fetches the price list for all ingredients from the google sheets and returns it back.

- Web Scraping:
    - Send request to Amazon Webpage that needs to be scrapped
        - The request is sent via the “requests” library of python.
        - The requested page contains the product information. Using a python script, product name and price details are extracted.
    - Create a soup function that contains responses received from the website.
        - Response is received from the website and it is passed on to a BeautifulSoup function that parses the page.
    - Using lxml parser, break down the HTML page
        - It receives 4 kinds of python objects after parsing the page out of which Tag object is required for obtaining product name and price
    - Locate the exact tags for extracting the object
        - Using web browser we obtain the exact tags of the product
    - Locate the title and price of the product.
        - find() function is used for searching specific tags and their attributes.
    - Send a request to the Google Sheet and append the price and product name into the sheet.
        - Product title and price is appended to the Google Sheet.

- ML model
    - App.yaml:
        - Used for hosting the flask server with runtime and environments declared. Also used for declaring automatic_scaling details.
    - Topic_modelling.py
        - Read the data from the dataset, perform the cleaning of data
        - Save the data into required pickle file to perform further analysis
        - Train LDA model to extract the topics from the content of data
        - Save the trained model file.
    - model_result.py
        - Load the trained model from model file
        - On the http request for fetching the recipes for recommendation, fetch the recipes data which fall under the same topics whose recipes user had preferred before.
        - Send the recommended recipes as a response to the flask server.
    - main.py
        - Responsible for hosting the flask application
        - It handles the http request for fetching the recommended recipes from the model and send it back as a http response

6. Conclusions:

We were able to create a web application that maintained and analysed the user's weekly meal schedule. It fetches the prices of the items on the list across a shopping site and displays them so that the users can add them to the cart. Using an LDA model we were able to suggest the user recommendations based on personal preferences. Further adding features such as personal assistant bot made it more interactive and fun. Furthermore GAE was successfully able to autoscale the resources in order to meet the user requirements. Therefore, as the load on the website increased, GAE scaled up and down the instances.

As a future scope we can create a database that stores information of multiple users. And for each user we can provide their personalised recommendation system. Moreover, we can provide a price comparison system wherein prices from multiple websites like Amazon, Walmart, Safeway can be compared and the lowest price will be added to the cart.

7. Individual contributors :

Sayali Papat (ASU ID: 1219725975)

Design:
The design phase involved deciding the architecture and which services to be used from GCP. What would be the role of each component and how they will communicate with each other. Me and my team discussed different models and finalized the architecture of the Project, with the goal of low latency for customers in mind. I worked on finalizing the design of the ML model.

Implementation:
The implementation involved the deployment of an google app engine using python as runtime environment, for the ML model. In the start we faced a lot of issues in deploying the flask webapp, because of package dependencies and their versions, I debugged the issues and found out what all dependencies are required for successful execution of the code, added it to the requirements.txt file and executed the web app deployment.
I further worked on cleaning the recipe dataset collected from Kaggle, using cloud storage via python modules, downloading and uploading files from and to the cloud storage, connecting to the Google cloud MySQL instance. I worked on training the recipe data from the dataset using an LDA model to extract features from recipe ingredients. At start I tried to train the data with default parameters of the model, but later on I checked the model performance by changing them, in order to fit the model with best feasible values of parameters. Once the model was trained I worked on utilizing the topics extracted by the model from the dataset to provide a personalized recommendation to users, depending on their previous preferences.
Setting up the Google cloud environment including the firewall rules, Identity Access Management, setting up the MySQL instance, importing the dataset to the database, managing and deployment of Google App Engine for the ML model was done by me.

Testing:
Testing phase involved the testing of individual models and then integration testing. Me and my team performed end to end testing with different sizes of input image sets. I tested the ML model functionality, which involved checking the correctness of the individual models and code correctness. I tested the recommendation provided by the model for different user preferences, multiple times.
While testing auto scaling logic, we encountered many corner case scenarios, to address that I optimized a few parameters and performed stress testing multiple times. In this testing and debugging phase, I helped in fixing the bug and improved performance.

Shivangi Singh (ASU ID: 1219123646)

I helped design the architecture of the system. I contributed to the front end of the application which includes the User Login and Registration page. I further implemented the item information service. It takes the items information from the recipe and scrapes the current price of the items from a shopping portal.

Design:

User authentication is an important step in modern web applications. It provides a security mechanism that restricts unauthorized access to the site. Users need to enter their personal information including name, email address, phone number, nationality, and other food preferences. The information is stored in the database. After registration, next time when the user wants to log into the application, they enter their username and password. When users submit the form these inputs will be verified against the credentials stored in the database, if they match, the user is authorized and granted access to the site.

I also designed an item information service, which fetches recipe details from Google Sheets and web scaps the current price of the items from a shopping portal. It then provides the user the best price of the ingredients. This is added to the user’s cart and the total price of the food item is displayed.

Implementation :

I created a registration system that allows users to create a new account by filling out the registration page. It generates errors if a user tries to submit the form without entering any value, or if the username entered by the user is already taken by another user. For this purpose, I created a database using Google Cloud MYSQL 8.0 that will hold all the user data. Angular was used for creating the web page using typescript. NodeJS was used to create the back-end of the app and it was hosted on a separate Google App Engine instance. Similarly, a login page was created that validated the user inputs against the credentials stored in the database.

In the item information service, Google Sheets API was used to collect ingredients details of the recipe. Libraries like BeautifulSoup and Requests were used to scrap the shopping portals. The price and product name was then returned to the user into the google sheets which was updated into the web portal.

Testing

I performed Load Testing to check if the application is able handle high amounts of requests, that is it can manage auto scaling on its own. I further tested the web scraper if it is giving the minimum priced product or not. It was displaying the least priced product to the user which was added to the cart.

Yash Ravindra Bhokare (ASU ID: 1219496304)

Design: I was the one to pitch in the basic framework of the given idea. I helped the team finalize the basic resource requirements. I suggested the need of a virtual assistant while cooking a recipe. During the first phase of this project I designed the basic architecture for connecting the backend services with the front end and the need of auto scaling of back end services. UX designs and the UI flow were suggested and implemented by me.

Implementation:
Back-end: I implemented and deployed the backend REST API in the form of a node server. The node server was created using express.js and deployed over the Google App Engine. In the node server I took care of the CORS implementation to allow specific cross-origin requests. At first I made a connection to the google cloud sql model where the recipe database was stored. After the connection was successful, I queried the database and opened an endpoint to return the data to the front end. I also added a google calendar api to store the recipe name on a selected date of the calendar. Then I started working on the personal assistant bot. For this I first implemented google speech to text and text to speech apis to understand the user's input and also to read him out the recipes step by step. Once the basic integration was completed with the front end I added Google Dialog Flow to interpret multiple sentences and give back a response according to the interpretation. During the end phases when the web crawler (for ingredient prices) implementation was completed, I integrated the Google Sheets API to store ingredients and fetch prices from the sheets and display them to the user on the web app.

Front-end: I designed the UI of our web application. Started with implementing the home page which would display the list of recipes for users to select. Then I made a connection between the angular app and the node server using http requests to fetch data. After this I implemented a state service which would handle the calls made to node server. All responses were cached locally in the web app. I implemented the recipe detail page that contained the ingredients list along with it's prices fetched from the backend. Also added the calendar selection component into it. Along with this I added the interaction flow of a Personal Cooking Assistant. I deployed the angular application on the GAE.

Testing:
At first I started running multiple tests from the web client to the node server to fetch the recipe lists. After testing this thoroughly I started integration of other google apis in the node server. In this while implementing the text-to-speech API I faced a lot of issues. One problem was to fetch the mp3 file generated at the server end and return it to the front end. I added one more endpoint for this. While testing this worked fine locally, but when I deployed it I got access issues as the file can be created on the node server over cloud. Then I had to switch to google cloud storage which would save these files and return it back to the user on the front end. The second major issue was to pass the audio buffer from front end to back end. I had to convert the audio stream to uint8 array and then convert it to base64 string which is accepted by the Google API of speech-to-text. After these issues were solved I deployed the code over the server and stress tested it multiple times to check if APIs implemented were giving an issue. Later on during the last phases we started passing multiple requests using an apache bench to stress test the application for its scalability and latency management.

Github links: https://github.com/yash4596/node-server-food-recommendation
https://github.com/yash4596/angular-food-recommendation-app

8. References:

[1] Cloud SDK guide: https://cloud.google.com/sdk/
[2] Kaggle dataset: https://www.kaggle.com/
[3] LDA model documentation: https://radimrehurek.com/gensim/models/ldamodel.html
[4] Google App Engine documentation: https://cloud.google.com/appengine/docs
[5] Google Dialog Flow documentation: https://cloud.google.com/dialogflow/es/docs