Traverse | Tinder-Like Itinerary Planner
Project Overview

Team Name: CC Legends
Application Name: Traverse
Team Focal Point: Anuar Burkitbayev (ab5465@columbia.edu)
Team Members: Anuar Burkitbayev (ab5465), Justin Clarke (jac2475), Harshul Gupta (hg2729), Arjun Bhan (ab5666), Alisha Varma (av3120), Mariam Naser (man2182)

Document version: FINAL, 2024-DEC-15

Document Roles

Role	Name	RACI Role(s)
Focal point	Anuar Burkitbayev	R
Team member	Justin Clarke	R
Team member	Harshul Gupta	R
Team member	Arjun Bhan	R
Team member	Alisha Varma	R
Team member	Mariam Naser	R

Project Summary

Introduction

Traverse | Tinder-Like Itinerary Planner

Many folks live in and visit NYC very often but due to the vast number of options of things to do and places to see it is often overwhelming to decide one's day. Traverse aims to provide a tinder-like swiping platform to generate itineraries for uses on things to do in NYC (based on businesses) by leveraging advanced microservices tailored to the unique needs and wants of residents and visitors alike.

Core Features to Find and Personalize an Itinerary Experience:

- **User Management Microservice:** holds basic user data for app functionality
- **List Management Microservice:** holds users' lists and their itineraries
- **Business Management Microservice:** holds business data such as location, address, category, and name
- **Composite Service:** orchestrates list and business management services to serve and add itineraries to lists

Enhanced Features to Expand Scope (Wish List):

- **Timing Scope:** allowing users to input a specific time frame for which they would like an itinerary for would allow selection of businesses and generations of itineraries based on expected times spent at each business location.
- **Neighborhood Selection:** Traverse would allow the user to select and narrow down the specific neighborhoods that they would like an itinerary generation from, which is prompted before swiping.
- **Itinerary Reshuffling:** Traverse would allow users to generate and reshuffle the current itineraries that have been generated for them.

Three-Tier Architecture Approach

1. **Presentation Layer (Frontend):**
    - **User Interface (UI):** a responsive, user-friendly web application where users can interact with Traverse. The UI features:
        - A dynamic selection swiping page that allows users to “swipe” on business that they are interested in
        - A list view of the current itineraries that have been selected
        - A login page
    - **Technologies:** Next.js
2. **Logic Layer (Backend/Microservices):**
    - **Microservices:** Design discrete services to handle specific user needs. Examples include:
        - **User Management Microservice:** holds basic user data for app functionality
        - **List Management Microservice:** holds users’ lists and their itineraries
        - **Business Management Microservice:** holds business data such as location, address, category, and name
        - **Composite Service:** orchestrates list and business management services to serve and add itineraries to lists
    - **Technologies:** Python, FastAPI
3. **Data Layer (Database):**
    - **Tables:**
        - **Users Table:** maintains personal information on the user
        - **Lists Table:** holds list data and owner’s user ID
        - **Itineraries Table:** links businesses to lists
        - **Businesses Table:** holds information for all businesses
    - **Technologies:** MySQL

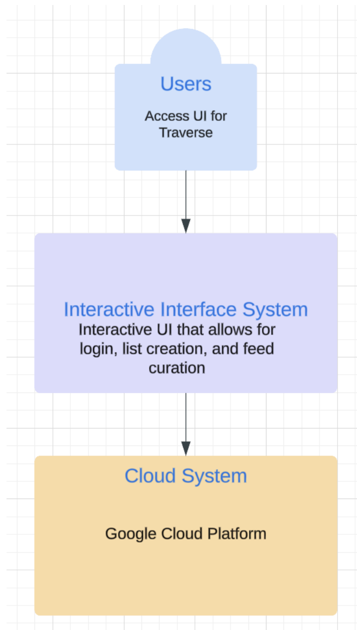
Current Implementation Stack -

Next.js
Python
MySQL

This architecture ensures a modular, scalable, and adaptable system that can grow with the evolving needs and requests of users.

Solution Overview

Domains



Traverse is designed to provide users with location-based itinerary generation. The system incorporates business data and location data to generate users' itinerary queues.

Roles:

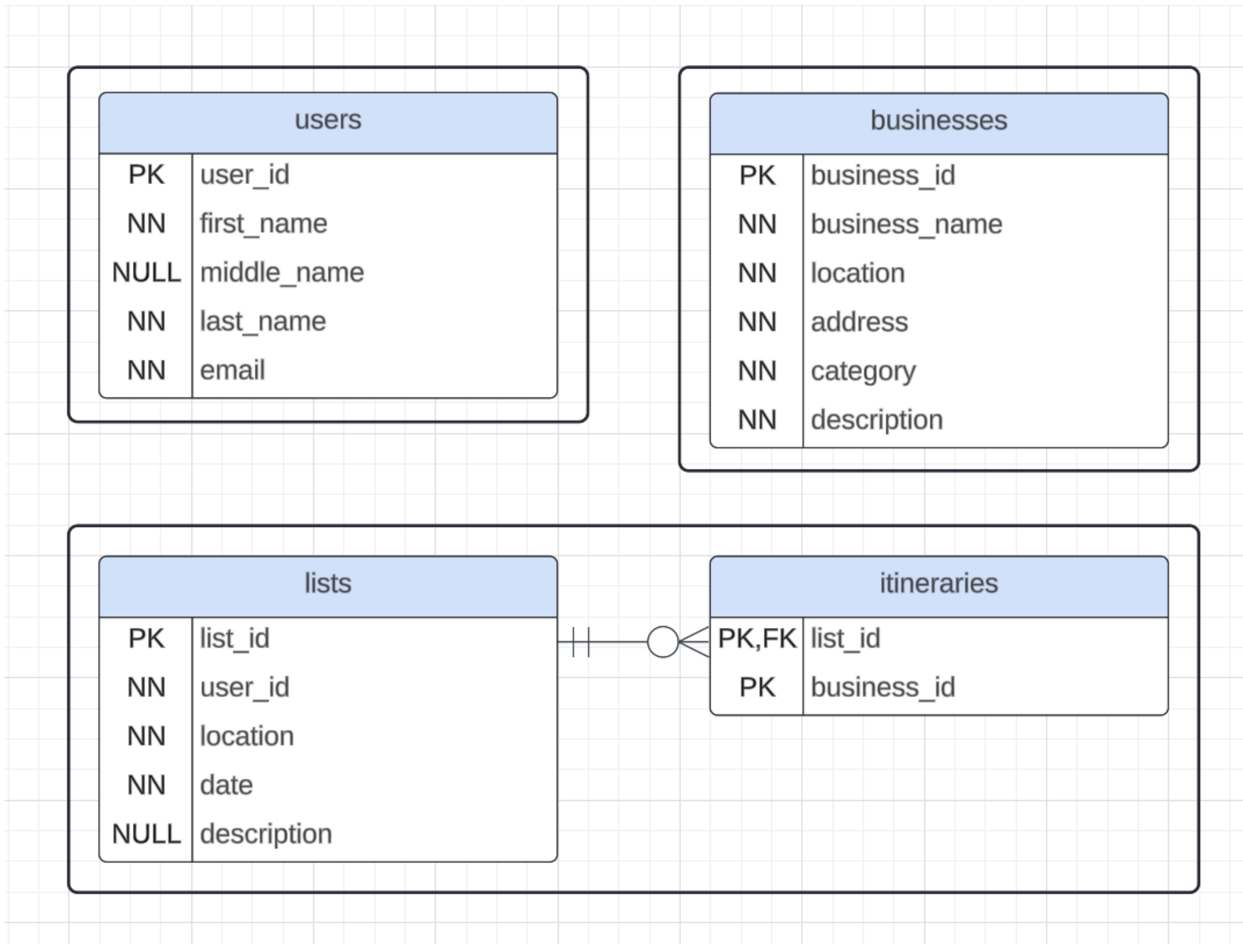
- Users – Access the interactive UI system in order to login, swipe on businesses and view the generated itineraries

Major Systems/Subsystems:

- Interactive interface system – UI is hosted on a Cloud application and accesses data from the database system for display and authentication. This allows users to utilize Traverse
- The database system is hosted on the cloud. The database holds three separate schemas for users, lists, and businesses

Lucid Chart:
https://lucid.app/lucidchart/8b7502aa-8250-46f7-a876-d2681638b855/edit?invitationId=inv_b062d717-b2aa-4c85-8a3f-7f3a0669caaf&page=0_0#

Resource Model



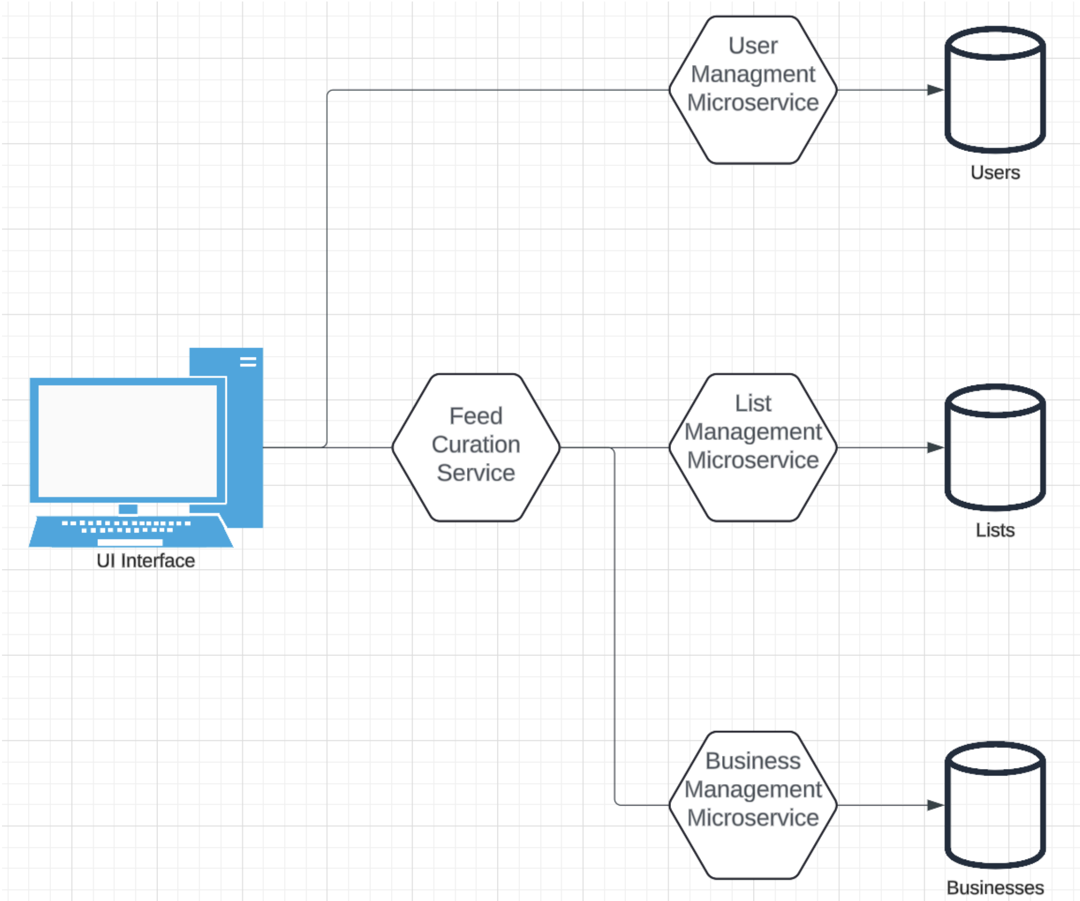
Basic user information is stored for functionality purposes.

Every user has a set of lists with specific itineraries detailing businesses that they will be visiting.

The Business table contains business specific information that will be displayed and used to aid in itinerary generation.

Lucid Chart:
https://lucid.app/lucidchart/8b7502aa-8250-46f7-a876-d2681638b855/edit?invitationId=inv_b062d717-b2aa-4c85-8a3f-7f3a0669caaf&page=0_0#

Interactive and Operations System



This is a three tier architecture approach.

Presentation layer, is a web application UI on a computer that interacts with the UI hosting infrastructure and the composite and three microservices currently in place.

Logic layer, is a set of a composite and three microservices that interact with both the data layer and the UI presentation layer. Traverse services interact with both layers in order to access and house user data as well as business and list generation data.

Data layer is made up of 4 different tables that are housed in separate databases that interact with the microservices.

Lucid Chart:
https://lucid.app/lucidchart/8b7502aa-8250-46f7-a876-d2681638b855/edit?invitationId=inv_b062d717-b2aa-4c85-8a3f-7f3a0669caaf&page=0_0#

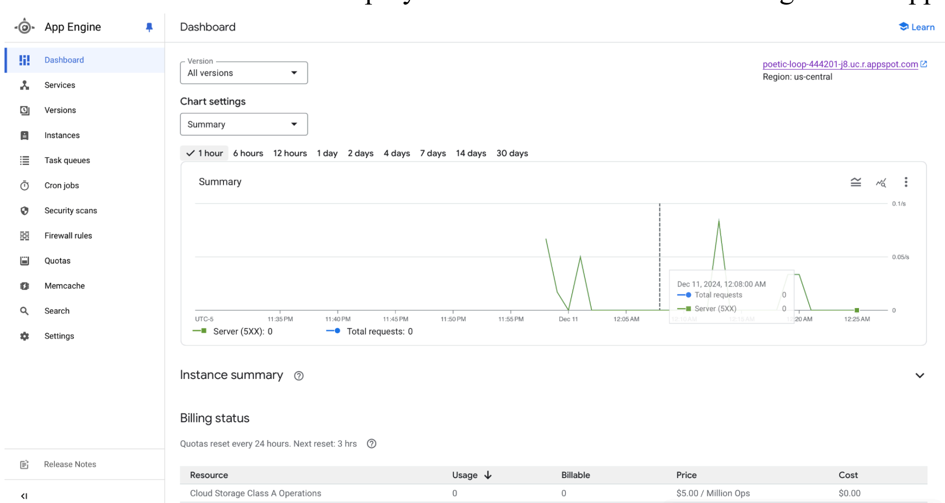
Deployment | Description Overview

Database

Traverse's database uses MySQL and is stored on a single Google CloudSQL instance, which houses each schema (users, lists, and businesses) separately.

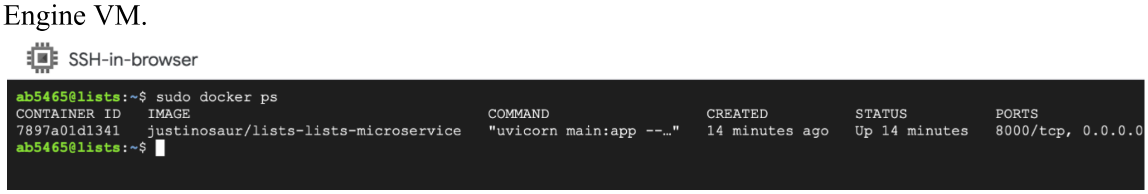
Business Management Microservice

Traverse's business management microservice is running from Google's PaaS service, App Engine. The code for it is committed and deployed from the GCP terminal along with an app.yaml file.



List Management Microservice

Traverse's list management microservice is running from a docker container on a Google Compute Engine VM.



User Management Microservice

Traverse’s user management microservice is running from a Google Compute Engine VM.

SSH-in-browser

```
ab5465@users:~/src$ find . | sed -e "s/[^-][^\/]*\// |/g" -e "s/|\([^ ]\)/|-\1/"
.
 |-schemas.py
 |-requirements.txt
 |-env.py
 |-__pycache__
 |  |-schemas.cpython-311.pyc
 |  |-env.cpython-311.pyc
 |  |-routers.cpython-311.pyc
 |  |-database.cpython-311.pyc
 |-main.py
 |-database.py
 |-routers.py
ab5465@users:~/src$ ~/bin/python3 main.py
INFO:     Started server process [71111]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

Composite Service

Traverse’s composite service is also running on a Google Compute Engine VM.

SSH-in-browser

```
ab5465@composite:~/src$ find . | sed -e "s/ [^-][^\/]*\// |/g" -e "s/|\([^ ]\)/|-\1/"
.
./main.py
./requirements.txt
ab5465@composite:~/src$ cat main.py
import asyncio
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from typing import List
import time
import logging
from starlette.middleware.base import BaseHTTPMiddleware
from starlette.requests import Request
from fastapi.responses import Response
import uuid
import uvicorn
import httpx
import json

app = FastAPI()

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    handlers=[
        logging.FileHandler("tracing.log"),  # Log to a file
        logging.StreamHandler(),            # Log to the console
    ],
)

# Custom Logging, Tracing, and Correlation ID Middleware
class TracingMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request: Request, call_next):
```

*All Traverse VMs:

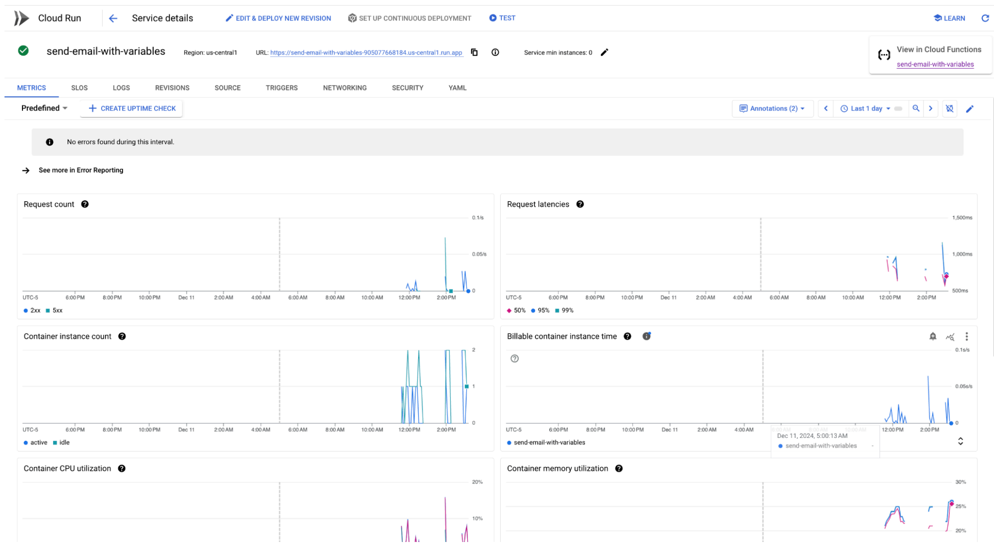
VM instances

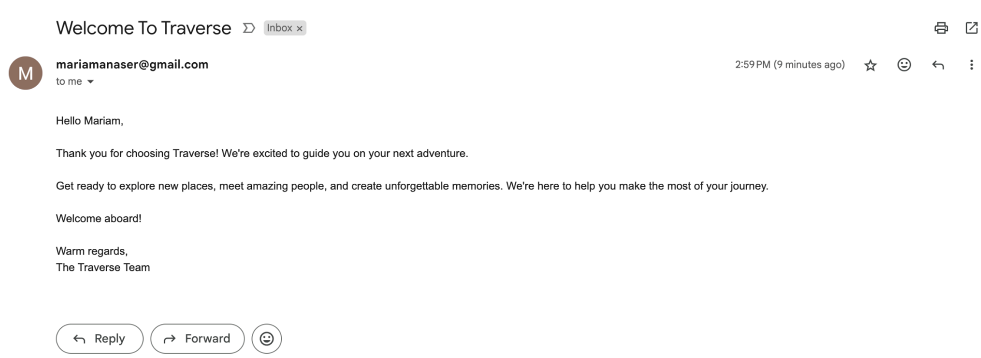
Filter Enter property name or value

Status	Name	Zone	Recommendations	In use by	Internal IP	External IP	Connect
✓	composite	us-central1-a			10.128.0.6 (nic0)	34.31.166.178 (nic0)	SSH
✓	lists	us-central1-c			10.128.0.5 (nic0)	34.67.161.129 (nic0)	SSH
✓	users	us-central1-c			10.128.0.2 (nic0)	35.225.241.51 (nic0)	SSH

End-user Notification Service

Traverse's end-user notification service is running from Google's FaaS, Cloud Functions. The service sends emails to new users.






```
    # Build the Gmail service
    service = build('gmail', 'v1', credentials=credentials)
    logging.info("Email service built successfully.")

    # Hard-coded email content
    to_email = user_email
    subject = 'Welcome To Traverse'
    body = f"""
Hello{user_name},

Thank you for choosing Traverse! We're excited to guide you on your next adventure.

Get ready to explore new places, meet amazing people, and create unforgettable memories. We're here to help you make the most of your journey.

Welcome aboard!

Warm regards,
The Traverse Team
"""

    # Create the email
    message = MIMEText(body)
    message['to'] = to_email
    message['from'] = 'me'
    message["subject"] = subject

    # Encode the message
    raw = base64.urlsafe_b64encode(message.as_bytes()).decode()

    # Send the email
    message_body = {'raw': raw}
    try:
        message = service.users().messages().send(userId='me', body=message_body).execute()
        logging.info(f'Email sent successfully: Message Id: {message["id"]}')
        return f"Email sent successfully: {message['id']}", 200
    except Exception as error:
        logging.error(f"An error occured: {error}")
```

UI | Description Overview

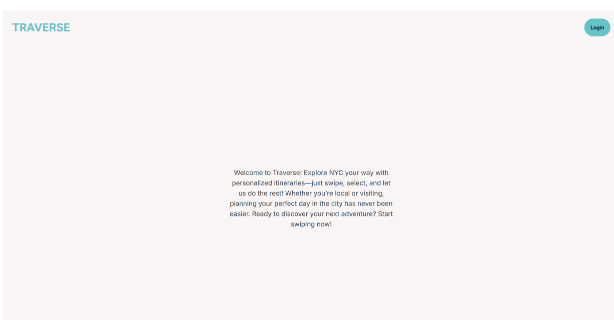
The UI includes the following:

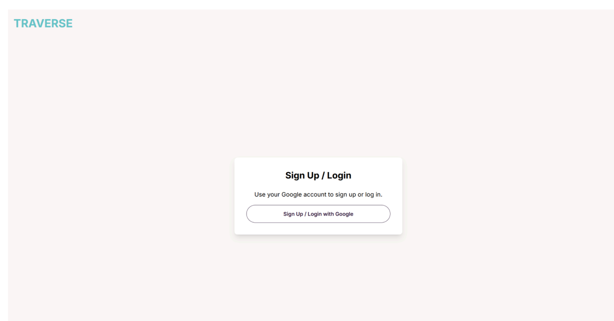
- Landing page giving details about Traverse
- Login/sign up page that allows users to authenticate with Google
- Dashboard page that allows users to view their existing lists or create a new list
- List creation page that allows the user to specify the list name, date and location of travel and create the particular travel list
- Swiping page that allows users to use the “Tinder-like” feature and determine what the like (swipe right) and what they don’t (swipe left)
- Users’ lists page that shows all the lists that the user has created
- Itinerary Details page that displays all the details of the generated itinerary from the users’ past interactions, also has a weather api to get the weather info for the place on that day.

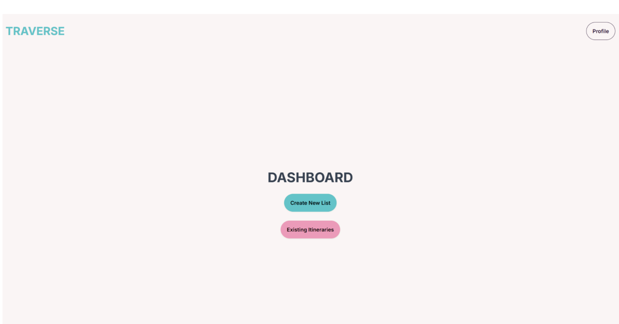
Frameworks & Libraries:

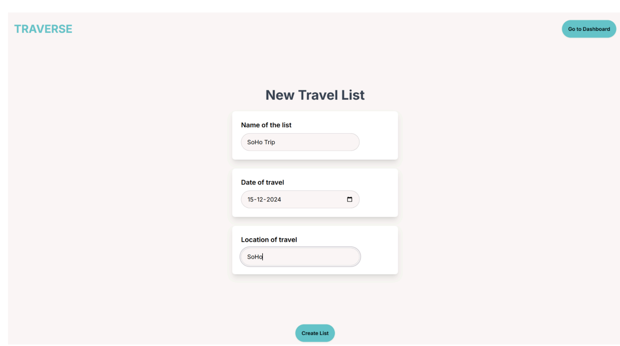
- Next.js: Used for building the UI with React components and server-side rendering.
- React: For creating reusable, interactive components.
- Tailwind CSS: For styling with utility-first classes and responsive design.

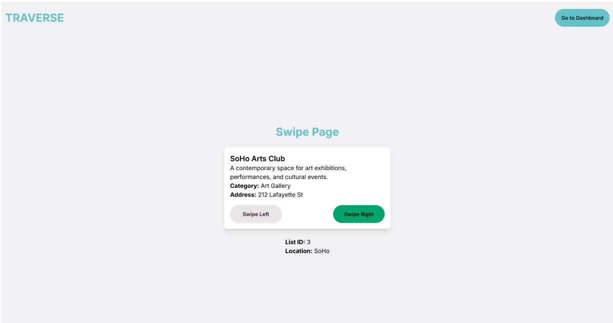
Screenshots:

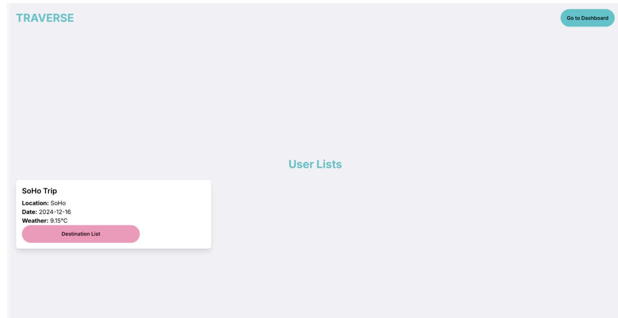


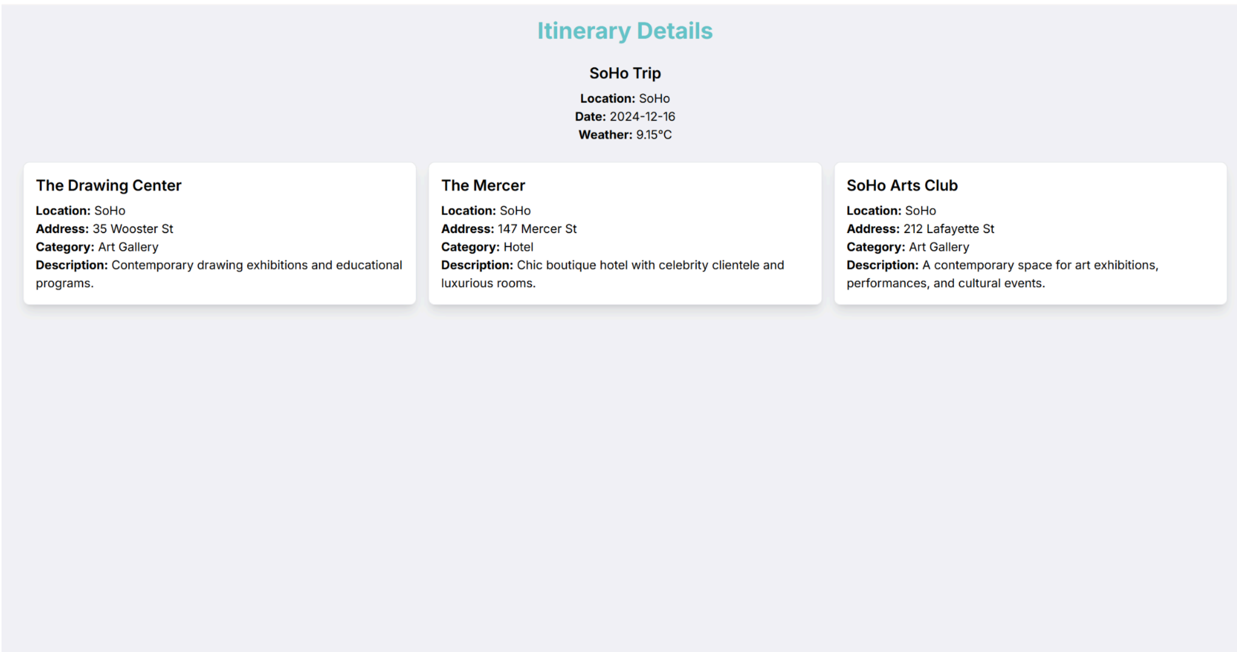







TRAVERSE
Go to Dashboard
Swipe Page
SoHo Arts Club
A contemporary space for art exhibitions, performances, and cultural events.
Category: Art Gallery
Address: 212 Lafayette St
Swipe Left
Swipe Right
List ID: 3
Location: SoHo


TRAVERSE
Go to Dashboard
User Lists
SoHo Trip
Location: SoHo
Date: 2024-12-16
Weather: 9.15°C
Destination List


Itinerary Details
SoHo Trip
Location: SoHo
Date: 2024-12-16
Weather: 9.15°C
The Drawing Center
Location: SoHo
Address: 35 Wooster St
Category: Art Gallery
Description: Contemporary drawing exhibitions and educational programs.
The Mercer
Location: SoHo
Address: 147 Mercer St
Category: Hotel
Description: Chic boutique hotel with celebrity clientele and luxurious rooms.
SoHo Arts Club
Location: SoHo
Address: 212 Lafayette St
Category: Art Gallery
Description: A contemporary space for art exhibitions, performances, and cultural events.

Other Requirements | Description Overview

Security:

- Google login using OAuth 2.0
- Issue own JWT tokens with "grants."
- Validate tokens in middleware and propagate on calls to other services

```python
@router.get('/login')
async def login(request: Request):
    # Redirect the user to Google's OAuth2 authorization URL
    redirect_uri = 'http://localhost:8000/auth/callback'  # The callback URL
    return await oauth.google.authorize_redirect(request, redirect_uri)

@router.get('/auth/callback')
async def auth(request: Request, response: Response):
    try:
        # Retrieve token and user information
        token = await oauth.google.authorize_access_token(request)
        user_info_feedback = await google.get("userinfo", token=token)
        user_info = user_info_feedback.json()

        # Extract user details
        given_name = user_info.get("given_name")
        family_name = user_info.get("family_name")
        email = user_info.get("email")

        # Validate user information
        if not email:
            raise HTTPException(status_code=400, detail="Invalid user information received from G

        # Save user to the database or return the user_id if the user already exists
        with get_connection() as conn:
            with conn.cursor() as cursor:
                # Check if the email already exists and get the user_id
                cursor.execute("SELECT user_id FROM log_in.users WHERE email = %s", (email,))
                existing_user = cursor.fetchone()

                if existing_user:
                    user_id = existing_user[0]  # Extract user_id from the result
                else:
```

As can be seen in a sample code snippet above, we implemented Google login using OAuth 2.0 by redirecting users to Google's authorization URL, retrieving their access token, and extracting user information. It validates the email and handles the user data securely. It then creates and sends a custom JWT to the UI which is used for user ID verification.

```python
class TracingMiddleware(BaseHTTPMiddleware):  1 usage  new *
    async def dispatch(self, request: Request, call_next):  new *
        # Extract and validate the token from the header
        raw_token = request.headers.get("X-Token")
        if not raw_token:
            logging.error("Access denied: Missing X-Token header.")
            raise HTTPException(status_code=400, detail="Access denied: Missing token.")

        try:
            # Parse the token value as JSON
            token_data = json.loads(raw_token)
            user_token = token_data.get("user_token")
            user_id = token_data.get("user_id")

            # Validate that both user_token and user_id are present
            if not user_token or not user_id:
                logging.error("Access denied: Malformed token.")
                raise HTTPException(status_code=400, detail="Access denied: Malformed token.")

            # Store the token in request.state for propagation
            request.state.user_token = user_token
            request.state.user_id = user_id

        except (json.JSONDecodeError, TypeError):
            logging.error("Access denied: Invalid token format.")
            raise HTTPException(status_code=400, detail="Access denied: Invalid token format.")
```

This sample snippet above shows a middleware that validates the custom tokens (JWTs) by extracting them from the request header, ensuring proper format and required fields like user_id and user_token. Valid tokens are propagated through the request state for downstream services.

Observability:

- Basic log and trace in microservices, correlation ID and propagation, and middleware
- Demonstrate in a cloud's logging tools

The snippet above shows a demonstration of the GCP's logging tools

```python
# Custom Logging, Tracing, and Correlation ID Middleware
class TracingMiddleware(BaseHTTPMiddleware):  1 usage  Justin Clarke *
    async def dispatch(self, request: Request, call_next):  Justin Clarke *
        correlation_id = request.headers.get("X-Correlation-ID", str(uuid.uuid4()))
        request.state.correlation_id = correlation_id

        # Start timing the request
        start_time = time.time()

        # Log request details
        logging.info(f"Request Start: {request.method} {request.url}")
        logging.info(f"Correlation ID: {correlation_id} - {request.method} {request.url}")
        logging.info(f"Headers: {request.headers}")
        if request.method in ["POST", "PUT", "PATCH"]:
            body = await request.body()
            logging.info(f"Body: {body.decode('utf-8') if body else None}")

        # Process the request and get the response
        response = await call_next(request)
        response.headers["X-Correlation-ID"] = correlation_id

        # End timing and log response details
        process_time = time.time() - start_time
        logging.info(
            f"Request End: {request.method} {request.url} - "
            f"Status Code: {response.status_code} - Time: {process_time:.4f}s"
        )

        return response
```

This code is an example of how we implemented basic logging and tracing in microservices by recording request details such as method, URL, headers, and body. It uses a correlation ID for tracing, either extracting it from incoming headers or generating a new one, and propagating it in the response headers. The middleware also logs response times and status codes, providing end-to-end visibility for requests.

REST API

- OpenAPI documents for all microservices
- Documented and followed a basic REST API best practices guidance
- HATEOAS and support for links
- Pagination for at least one of the services
- For POST, at least one of the services should implement 201 created with a link header
- At least one path must implement an asynchronous execution pattern, eg 202 Accepted

```
/businesses/{business_id}
@app.get( path: "/businesses/{business_id}", response_model=schemas.Business)  Justin Clarke
def get_business(business_id: int, db: Session = Depends(get_db), request: Request = None):
    correlation_id = request.state.correlation_id
    business = crud.get_business(db, business_id=business_id, correlation_id=correlation_id)
    if business is None:
        raise HTTPException(status_code=404, detail="Business not found")
    return {
        **business.__dict__,
        "links": {
            "self": f"/businesses/{business.business_id}",
            "update": f"/businesses/{business.business_id}",
            "delete": f"/businesses/{business.business_id}",
        },
    }
```

This code implements HATEOAS (Hypermedia as the Engine of Application State) by including navigational links in the response. The links section provides self, update, and delete links for the resource, allowing clients to discover actions dynamically. This supports RESTful API principles by enhancing resource interaction.

```
/lists/
@app.get( path: "/lists/", response_model=List[schemas.List])  Justin Clarke
def get_lists(skip: int = 0, limit: int = 10, db: Session = Depends(get_db)):
    return crud.get_lists(db=db, skip=skip, limit=limit)

/lists/{list_id}
@app.delete( path: "/lists/{list_id}", response_model=schemas.List)  Justin Clarke
def delete_list(list_id: int, db: Session = Depends(get_db)):
    deleted_list = crud.delete_list(db=db, list_id=list_id)
    if deleted_list is None:
        raise HTTPException(status_code=404, detail="List not found")
    return deleted_list

/lists/{list_id}/itineraries/
@app.post( path: "/lists/{list_id}/itineraries/", response_model=schemas.Itinerary, status_code=202)  Justin Clarke
def add_itinerary_to_list(list_id: int, business_id: int, db: Session = Depends(get_db)):
    return crud.add_itinerary(db=db, list_id=list_id, business_id=business_id)

/lists/{list_id}/itineraries/
@app.get( path: "/lists/{list_id}/itineraries/", response_model=List[schemas.Itinerary])  Justin Clarke
def get_itineraries_for_list(list_id: int, skip: int = 0, limit: int = 10, db: Session = Depends(get_db)):
    return crud.get_itineraries(db=db, list_id=list_id, skip=skip, limit=limit)
```

This code meets the pagination requirement with the get_lists and get_itineraries_for_list endpoints, which support skip and limit parameters to control the number of results returned. For the asynchronous execution requirement, the add_itinerary_to_list endpoint uses status_code=202 to indicate asynchronous processing. This signifies that the request has been accepted for execution, but the operation may complete later, aligning with the 202 Accepted response pattern.

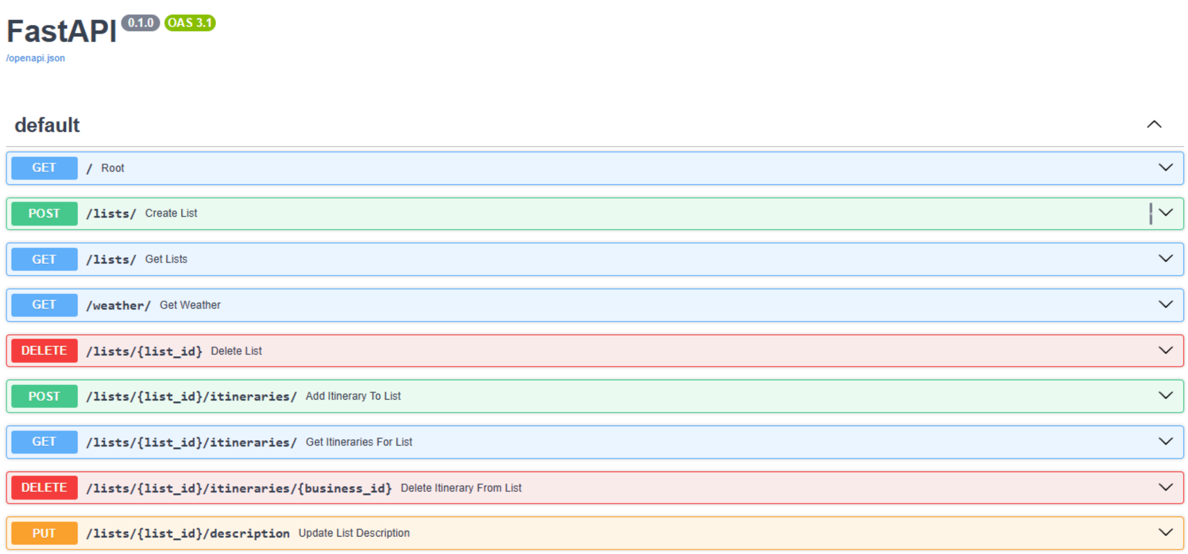
```python
@app.post("/lists/", response_model=schemas.List, status_code=201)
def create_list(user_id:int,location:str,date:str,description:str, db: Session = Depends(get_db)):
    # Call the CRUD function to create the resource
    list_data = schemas.ListCreate(
        user_id=user_id,
        location=location,
        date=date,
        description=description
    )
    created_list = crud.create_list(db=db, list_data=list_data)
    """
    # Generate the link to the newly created resource
    resource_id = created_list.list_id
    resource_url = f"/lists/{resource_id}"
    """

    # Add the link or location header
    #response.headers["Location"] = resource_url # Used to point to the new resource

    return created_list
```

This code fulfills the 201 Created with a link header requirement. The create_list endpoint creates a new resource and returns a 201 status code. It dynamically generates the resource URL (/lists{resource_id}) and prepares a Location header to point to the newly created resource, ensuring RESTful API compliance.

The REST API Documentation & openAPI documents can be seen below

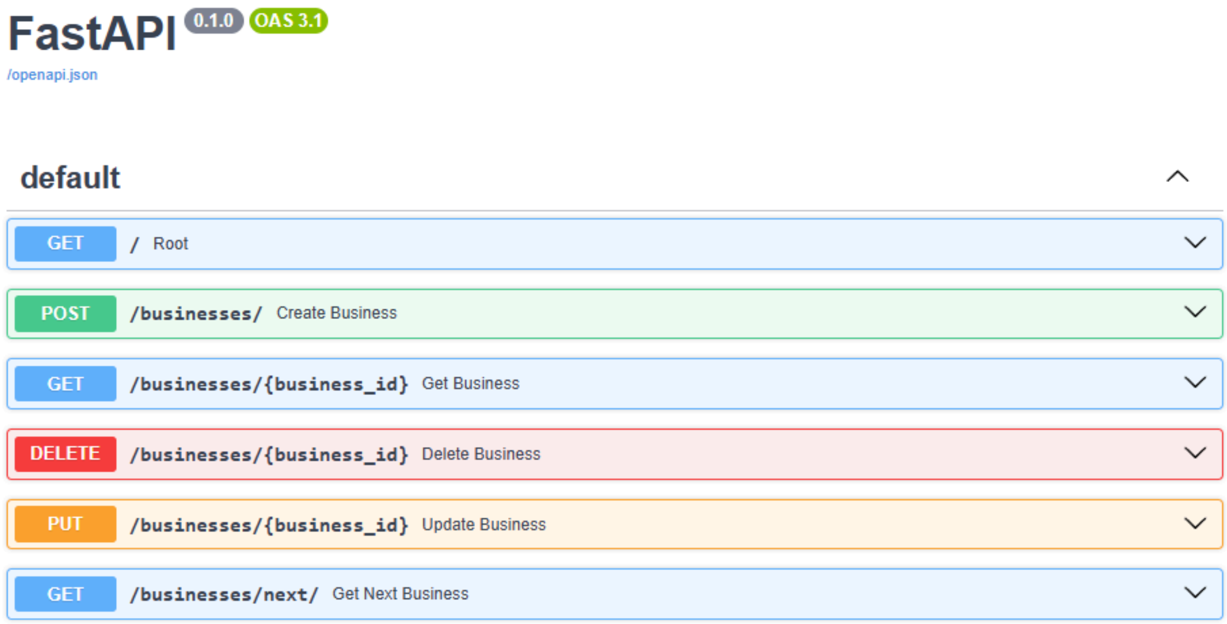
Lists Microservice

Description: The lists microservice manages user lists and their associated itineraries, providing endpoints to create, retrieve, update, and delete lists and itineraries. It supports pagination for retrieving lists or itineraries and implements a 202 Accepted response for adding itineraries asynchronously. Additionally, it includes a get_weather endpoint to fetch weather data based on location and date, making the service versatile and comprehensive for managing list-related functionalities.



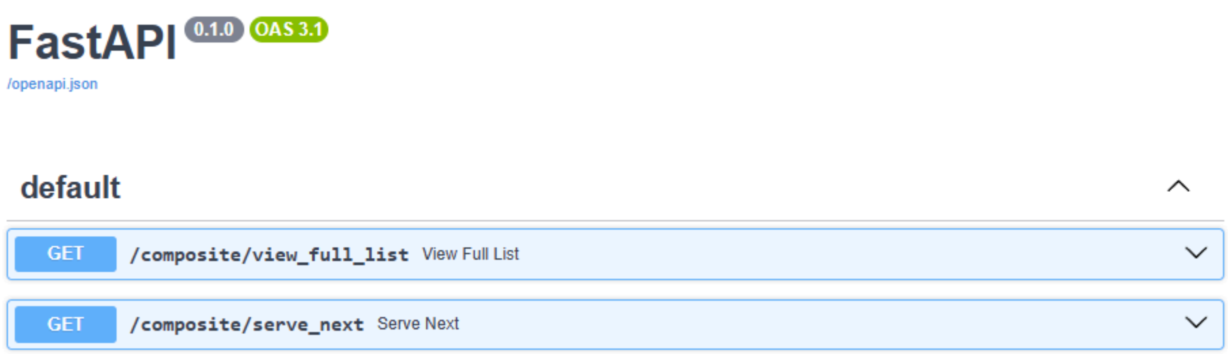
Businesses microservice

Description: The business microservice manages business data, offering endpoints for CRUD operations and advanced queries. It supports creating, retrieving, updating, and deleting businesses, while also providing the get_next_business endpoint to fetch the next available business not included in a specified list, filtered by location. This service ensures robust responses and error handling, making it suitable for integration into composite workflows.



Composite microservice

Description: This composite microservice integrates functionalities from two other microservices: a list microservice and a business microservice. The view_full_list endpoint asynchronously fetches all business IDs associated with a list from the list service and retrieves detailed information about each business concurrently using asynchronous HTTP requests. The serve_next endpoint synchronously fetches existing business IDs from the list service and requests the next business recommendation from the business microservice, excluding already listed businesses. Together, these endpoints efficiently combine data from multiple services to provide comprehensive and optimized responses.



At least one composite service

- Composition using code and synchronous API calls
- Composition using code and asynchronous API calls

```python
@app.get("/composite/view_full_list", response_model=List[dict])
async def view_full_list(list_id: int, request: Request):
    token = {"X-Token": json.dumps({"user_token": request.state.user_token, "user_id": request.state.user_id})}
    correlation_id = {"X-Correlation-ID": request.state.correlation_id}

    headers = {**token, **correlation_id}

    async with httpx.AsyncClient() as client:
        # Step 1: Get all business_ids from the list microservice
        list_url = f"{LIST_SERVICE_URL}/{list_id}/itineraries"
        list_response = await client.get(list_url, headers=headers)

        if list_response.status_code != 200:
            raise HTTPException(status_code=list_response.status_code, detail="Failed to fetch business IDs.")

        business_ids = [item["business_id"] for item in list_response.json()]

        # Step 2: Fetch business details asynchronously from the business microservice
        tasks = [client.get(f"{BUSINESS_SERVICE_URL}/{business_id}", headers=headers) for business_id in business_
        responses = await asyncio.gather(*tasks)

        # Step 3: Compile the details for valid responses
        businesses = [response.json() for response in responses if response.status_code == 200]

    return businesses
```

This code demonstrates composition by integrating data from multiple microservices using asynchronous API calls. It first fetches business IDs from the list microservice and then concurrently retrieves detailed business information from the business microservice using asyncio.gather. This approach ensures efficient execution and seamless aggregation of data across services.

```python
@app.get("/composite/serve_next", response_model=dict)
def serve_next(list_id: int, location: str, request: Request):
    token = {"X-Token": json.dumps({"user_token": request.state.user_token, "user_id": request.state.user_id})}
    correlation_id = {"X-Correlation-ID": request.state.correlation_id}

    headers = {**token, **correlation_id}

    with httpx.Client() as client:
        # Step 1: Get all business_ids from the list microservice
        list_url = f"{LIST_SERVICE_URL}/{list_id}/itineraries"
        list_response = client.get(list_url, headers=headers)

        if list_response.status_code != 200:
            raise HTTPException(status_code=list_response.status_code, detail="Failed to fetch business IDs.")

        existing_business_ids = [item["business_id"] for item in list_response.json()]

        # Step 2: Request the next business from the business microservice
        next_business_url = f"{BUSINESS_SERVICE_URL}/next"
        payload = {"list_id": list_id, "location": location, "existing_ids": existing_business_ids}
        next_business_response = client.get(next_business_url, headers=headers, params=payload)

        if next_business_response.status_code != 200:
            raise HTTPException(status_code=next_business_response.status_code, detail="No next business found.")

    return next_business_response.json()
```

This code achieves composition using synchronous API calls by integrating data from two microservices. It first retrieves all business IDs from the list microservice and then sends a subsequent request to business microservices to fetch the next available business, excluding existing IDs. The use of httpx.Client ensures sequential, synchronous execution while maintaining logical flow across services.

Calling / using at least one external cloud service

```python
def fetch_lat_lon(location: str, correlation_id: str) -> tuple:  1 usage  new *
    base_url = "https://nominatim.openstreetmap.org/search"
    headers = {"X-Correlation-ID": correlation_id}
    params = {
        "q": location,
        "format": "json",
        "limit": 1,
    }
    response = httpx.get(base_url, headers=headers, params=params)
    if response.status_code == 200 and response.json():
        data = response.json()[0]
        latitude = float(data["lat"])
        longitude = float(data["lon"])
        return latitude, longitude
    raise Exception("Could not fetch latitude and longitude for the given location")

def fetch_weather(latitude: float, longitude: float, date: str, correlation_id: str) -> str:  1 usage  new *
    api_url = f"https://api.open-meteo.com/v1/forecast?latitude={latitude}&longitude={longitude}&start_date={date}&end_date={date}&daily=temperature_2m_max,temperature_2
    headers = {"X-Correlation-ID": correlation_id}
    response = httpx.get(api_url, headers = headers)
    if response.status_code == 200:
        data = response.json()
        avg_temp = (data["daily"]["temperature_2m_max"][0] + data["daily"]["temperature_2m_min"][0]) / 2
        return str(avg_temp)
```

This code fulfills the requirement of calling an external cloud service by integrating third-party APIs. The fetch_lat_lon function uses OpenStreetMap's Nominatim API to retrieve latitude and longitude for a given location. The fetch_weather function calls Meteo API to fetch weather data based on the latitude, longitude, and date, demonstrating interaction with cloud-hosted services to provide geolocation and weather information.

GitHub Project (with Kanban Board)

Project Board: https://github.com/users/alishavarma/projects/2

GitHub Repositories

- UI Repo: https://github.com/alishavarma/traverse_ui
- Database Repo: https://github.com/alishavarma/traverse_database
- User Management Repo: https://github.com/alishavarma/traverse_user_managment
- Businesses Repo: https://github.com/alishavarma/traverse_businesses
- Lists Repo: https://github.com/alishavarma/traverse_lists
- Composite Repo: https://github.com/alishavarma/traverse_composite

Demo Recordings & Presentations

Demo Recording:
https://drive.google.com/file/d/1x6IkI8oGFIQGnT5knYbkLqiDymFiNtRh/view?usp=share_link

Presentation:
https://docs.google.com/presentation/d/14VE1tHeMs-mMaj9DfJMfytkAhZsw3Ft2JFCf0RAPps8/edit?usp=sharing

Contributions

Backend: Justin, Anuar, Arjun, Mariam
Frontend: Harshul, Alisha
Database: Anuar, Arjun
Deployment: Anuar, Mariam
Documentation: Alisha, Anuar, Justin, Harshul