Microservices-Based Translation and Summarization Platform

Abstract

This report outlines the development and deployment of core microservices for translation and text summarization services. The project's primary goal is to create scalable and efficient APIs for language translation (English to Arabic and Arabic to English) and text summarization . Utilizing modern tools like FastAPI, Django, and Kafka, the project emphasizes modularity, seamless integration, and robust performance. The outcomes include successfully containerized services, tested APIs, and a deployed infrastructure that supports scalability and high reliability.

Introduction

In an era of global connectivity, efficient language processing tools have become critical for bridging communication gaps. This project addresses this need by developing microservices for real-time language translation and text summarization. By leveraging state-of-the-art natural language processing (NLP) models and scalable backend technologies, the project aims to provide a robust solution for diverse user requirements. These microservices are designed with modularity, allowing seamless integration into larger systems, while maintaining high performance and reliability. The significance of this project lies in its potential to enhance multilingual communication and information accessibility, catering to both individual users and enterprises.

Methodology

The project employs a systematic approach to design, development, and deployment, ensuring modularity and scalability.

Technology Selection:

- FastAPI and Django were chosen for their high performance and ease of use in developing RESTful APIs.
- Kafka was implemented for message brokering, ensuring efficient communication between microservices.
- PostgreSQL was selected for its reliability and scalability as the primary database.

Development Process:

Each microservice was developed independently, adhering to a modular architecture.

Tools like Docker and Docker Compose were utilized for containerization, ensuring consistency across development and production environments.

Integration and Communication:

- API Gateway with NGINX was implemented to manage routing and load balancing.
- Kafka topics were established for structured communication between services, enabling asynchronous processing.

Testing and Validation:

- Comprehensive unit and integration testing ensured the functionality and reliability of APIs.
- Dockerized environments were tested across various platforms to verify portability and scalability.
- This methodology ensures the project's alignment with its goals of efficiency, scalability, and user-centric design.

Phase 1: Core Microservices Development

Translation Services

A. EN2AR-Service

1.Enabling Auto-Reload for Fast API:

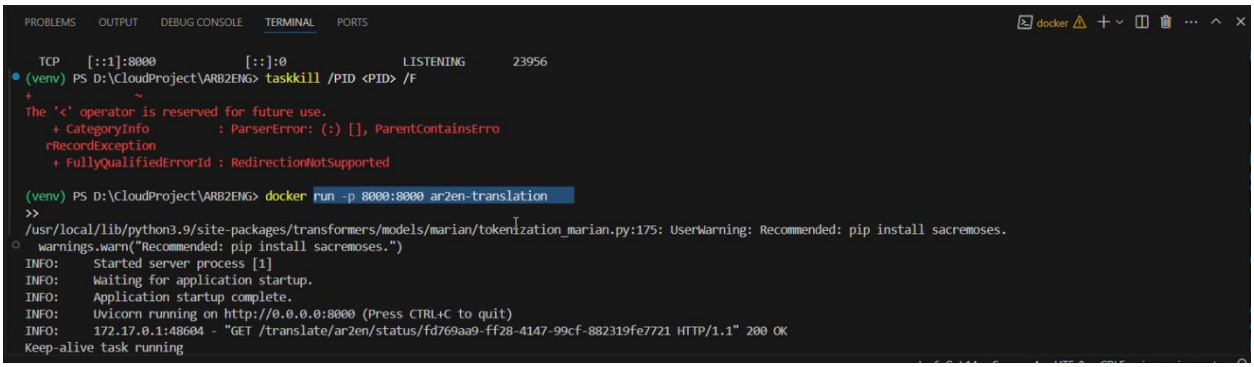
To enhance the development experience, auto-reloading allows the Fast API application to restart automatically when changes are made to the codebase. This feature is achieved using the --reload flag with Unicorn, the SQL server.

2. I make (app.py , dockerFile , requirements) files
and make i make virtual environment (venv) to isolate the dependencies of your Fast API project.

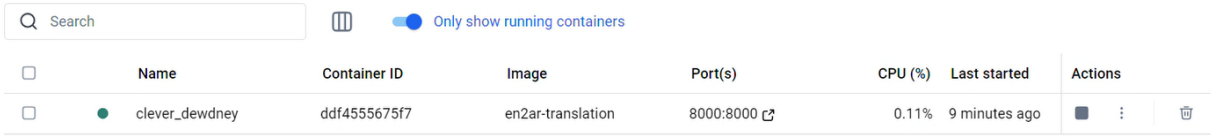
3. Start Your FastAPI Server to test the translation process through the postman program

By (uvicorn app:app --reload)

4. After making sure the code is working correctly I Run the Docker Container
by (docker run -d -p 8000:8000 eng2arb-translator)



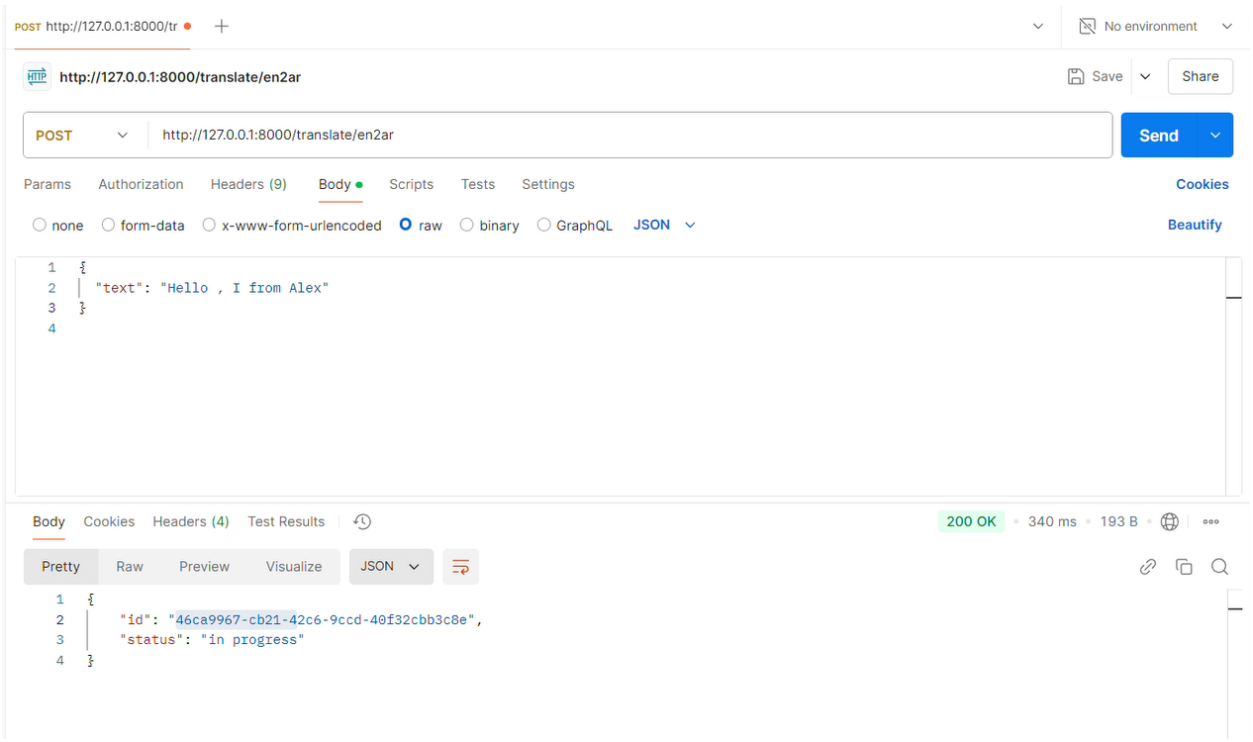
Making sure that is now working at docker hub



5. The post Endpoint
Configure the post request:

Method: post
URL: http://127.0.0.1:8000/translate/en2ar
Body: Select raw and JSON and write What you want to translate
For example:

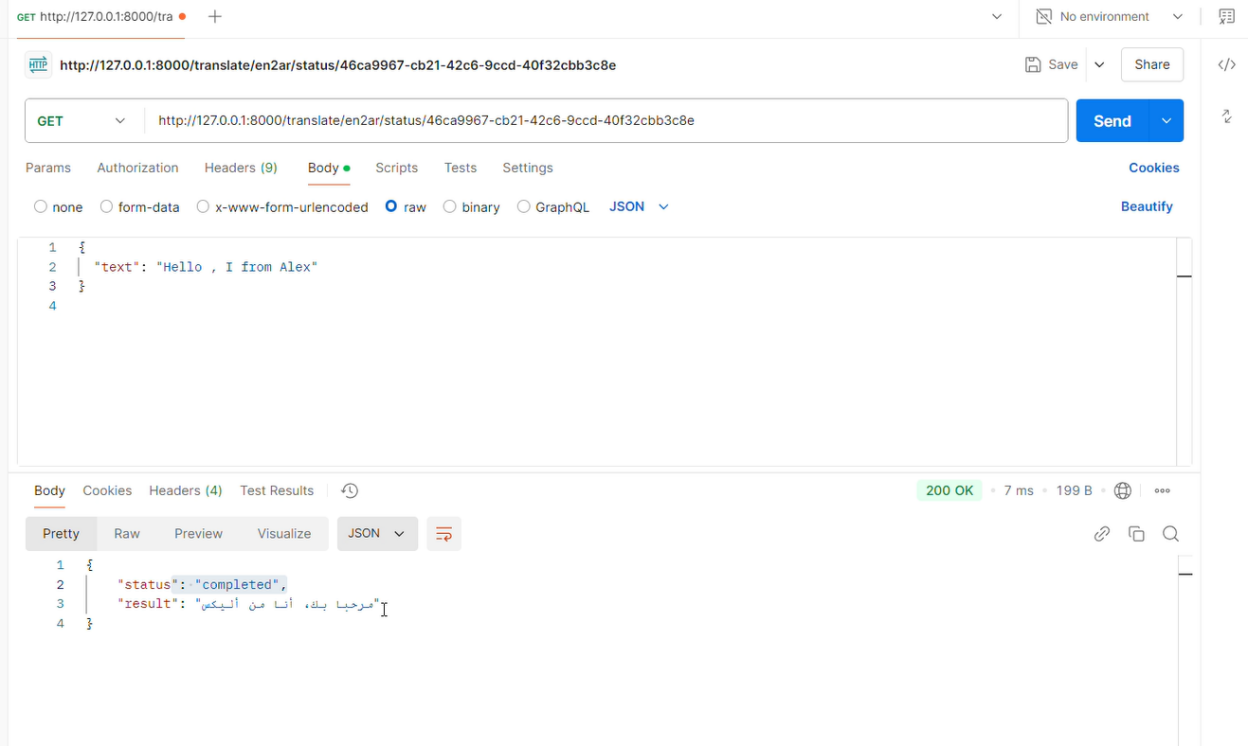
```
{
  "text": "Hello, how are you?"
}
```



6. The get Endpoint

Configure the get request:

Method: get
URL: http://127.0.0.1:8000/translate/en2ar/status/{id}
{id} with the id you received from the POST request.



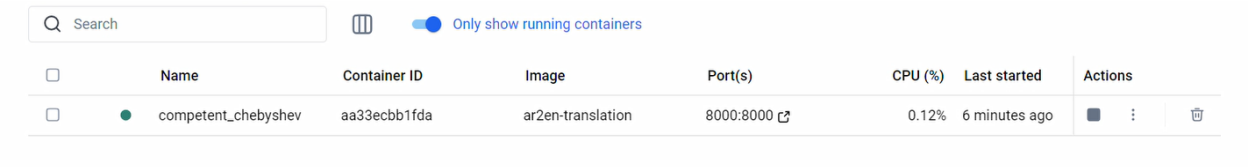
7. Confirmation arrives (ok 200) from postman on the post and get Process

```
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/yugzdvv0yvz1nen9sszvr1tqt
(venv) PS D:\CloudProject\ENG2ARB> docker run -p 8000:8000 en2ar-translation
/usr/local/lib/python3.9/site-packages/transformers/models/marian/tokenization_marian.py:175: UserWarning: Recommended: pip install sacremoses.
  warnings.warn("Recommended: pip install sacremoses.")
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     172.17.0.1:34560 - "POST /translate/en2ar HTTP/1.1" 200 OK
Keep-alive task running: Server is alive.
INFO:     172.17.0.1:55406 - "POST /translate/en2ar HTTP/1.1" 200 OK
INFO:     172.17.0.1:40662 - "GET /translate/en2ar/status/670e4eac-dc62-4f6a-8c38-0f0d0238535d HTTP/1.1" 200 OK
Keep-alive task running: Server is alive.
INFO:     172.17.0.1:38668 - "POST /translate/en2ar HTTP/1.1" 200 OK
Keep-alive task running: Server is alive.
INFO:     172.17.0.1:33592 - "GET /translate/en2ar/status/46ca9967-cb21-42c6-9ccd-40f32cbb3c8e HTTP/1.1" 200 OK
```

B. R2EN-Service

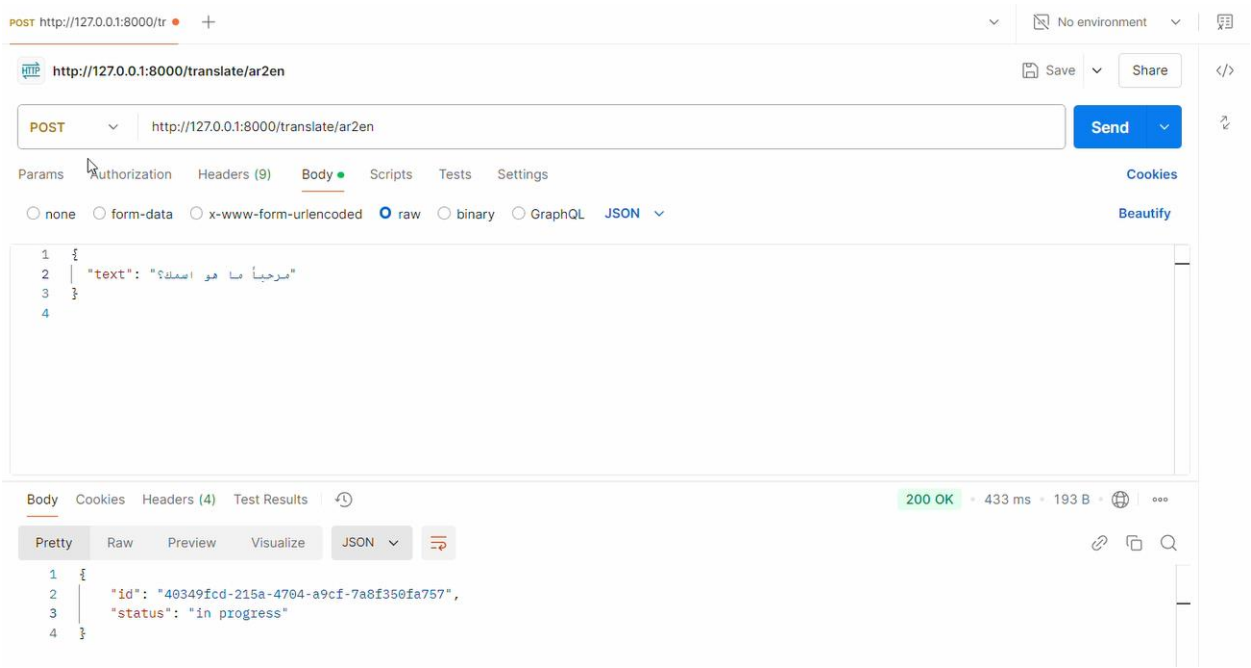
Like I did in the translation from English to Arabic I will do here

1. Making sure that is now working at docker hub



2. The post Endpoint
Configure the post request:

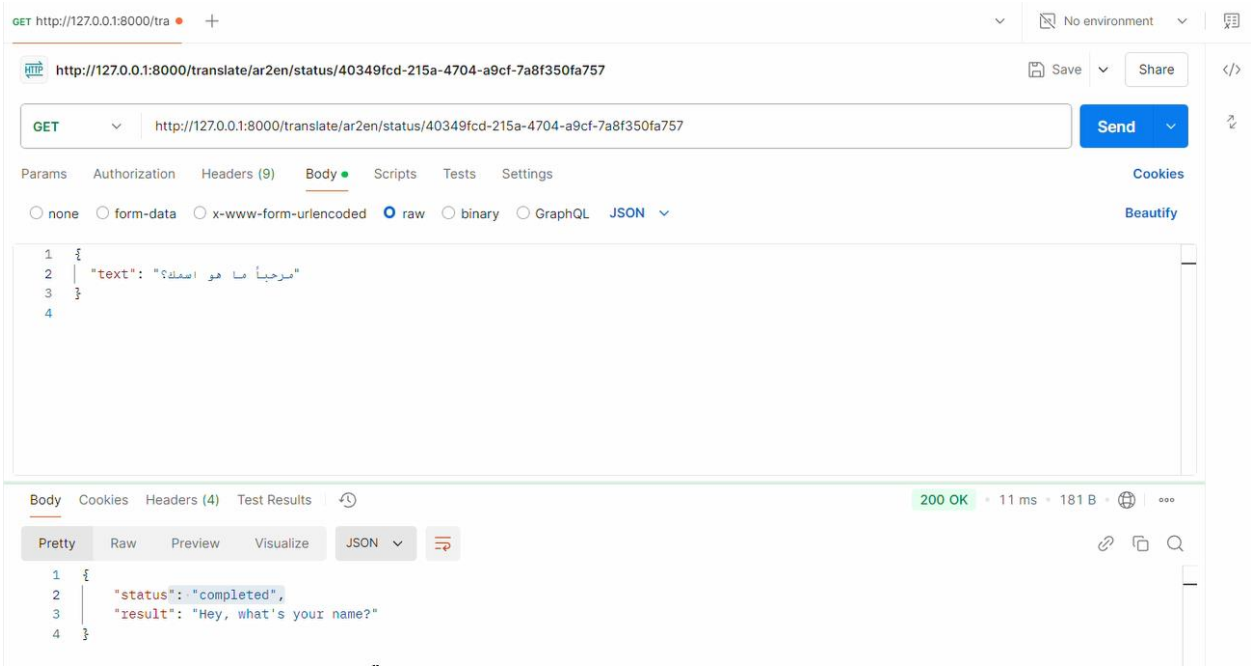
Method: post
URL: http://127.0.0.1:8000/translate/ar2en
Body: Select raw and JSON and write What you want to translate



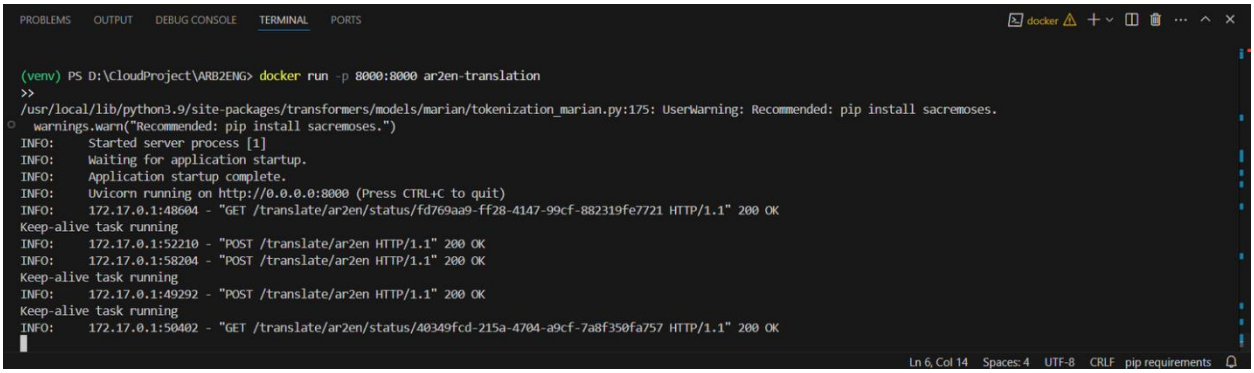
3. The get Endpoint

Configure the get request:

Method: get
URL: http://127.0.0.1:8000/translate/en2ar/status/{id}
{id} with the id you received from the POST request.



4. Confirmation arrives (ok 200) from postman on the post and get Process



Text Summarization Service

Project Details

1. Core Features

- Text Summarization Functionality:
  - Utilize Hugging Face Transformers for Natural Language Processing (NLP).
  - Provide concise and accurate summaries based on input text.
- API Endpoints:
  - POST /summarize: Accepts a text payload and returns its summarized version.
  - GET /summarize/status/{id}: Retrieves the status of a summarization task using a unique task ID.

2. Optional Customization Features

- Support various summarization styles, including:
    - Formal: Suitable for academic or professional use.
    - Informal: Simplified and conversational summaries.
    - Technical: Detailed summaries for specific fields like engineering or medicine.

3. Implementation Steps

Backend Development

1. Set up Django Framework:
    a. Create a Django project and define necessary models and views.
    b. Integrate Django REST Framework (DRF) for handling API requests and responses.

2.Integrate Hugging Face Transformers:

- Use pre-trained models for summarization tasks (e.g., facebook/bart-large-cnn or t5-small).
- Optionally fine-tune models for better results.

3.Handle Asynchronous Tasks:

- Use tools like Celery or Django-Q for processing long-running summarization tasks.
- Store task statuses in a database to track and retrieve summaries using the /status/{id} endpoint.

4.Database Integration:

- Use PostgreSQL as the database, with the schema designed for storing task information and summaries.
- Leverage the psycopg2-binary library for seamless interaction with the database.

Customization

- Add parameters to the API for selecting summarization styles.
- Implement logic to adapt model outputs to fit the chosen style.

4. Containerization with Docker

1. Create a Dockerfile:
    a. Define instructions to build a lightweight image for the Django service.
    b. Install required dependencies using requirements.txt.
2. Set Up Docker Compose:
    a. Include a web service for the Django application and a db service for PostgreSQL.
    b. Expose necessary ports (8000) for API accessibility.
3. Volume Management:

a. Mount local files to the Docker container for persistent development.
b. Use volumes for PostgreSQL data to maintain database integrity.

5. Testing and Deployment

Testing

- Unit Testing: Test API endpoints using Django's built-in testing framework or pytest.
- Functional Testing: Validate the summarization quality for different styles and verify API reliability.
- Container Testing: Ensure the Dockerized application works across environments and supports scaling.

Deployment

- Dockerized Deployment: Use Docker containers for seamless deployment in production environments.
- Cloud Hosting: Deploy the application on platforms like AWS, Google Cloud Platform, or Heroku for scalability.
- Monitoring: Implement logging and monitoring tools to ensure service availability and performance.

6. Additional Enhancements

- API Documentation: Provide clear and detailed documentation for the API endpoints using tools like Swagger or Postman.
- Batch Processing: Allow users to submit multiple texts for summarization in a single request.

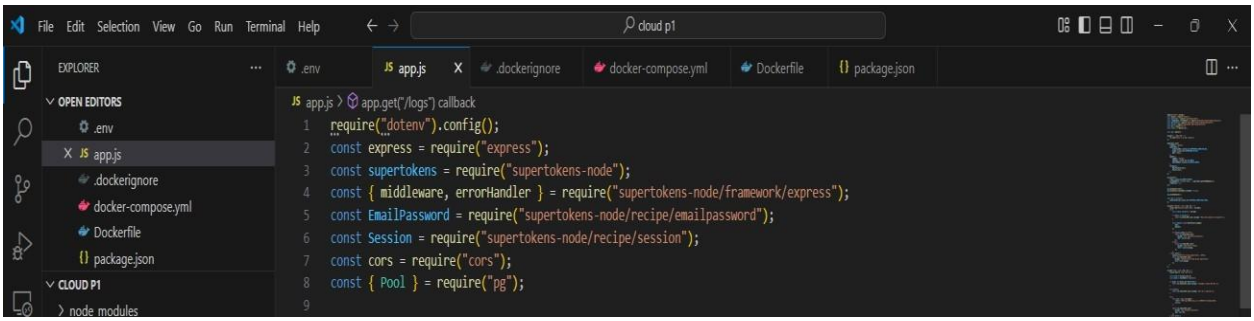
Deliverables

- Fully functional summarization service with REST API endpoints.
- Dockerized application for easy deployment.
- Customization features for tailored summarization styles.
- Optional advanced features for scalability and usability.

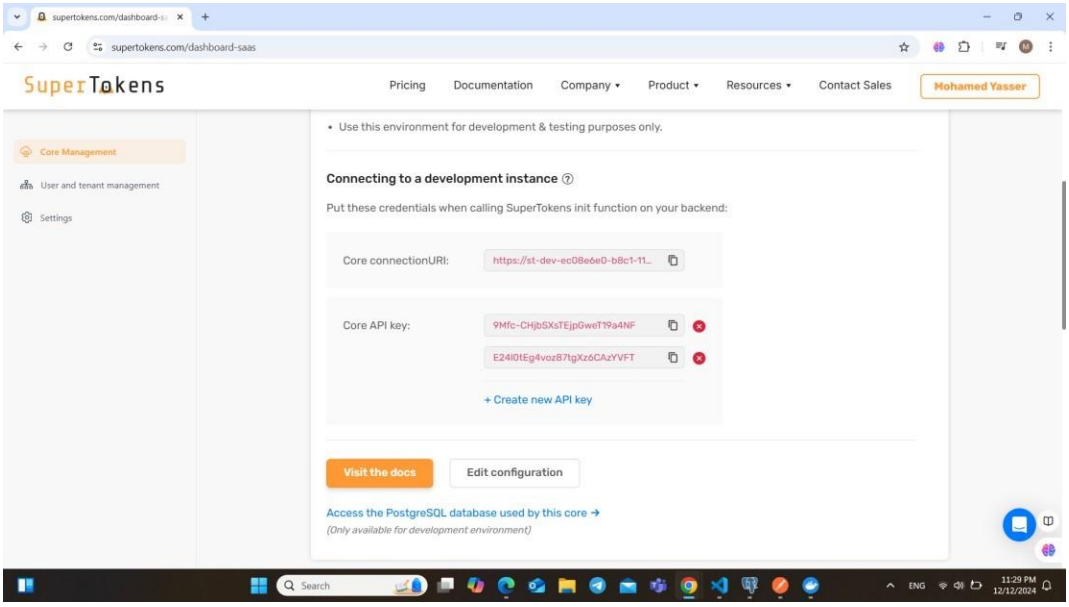
User Management and Database Services

Develop User-Service with Node.js and Super tokens for authentication and user management.:

1st: import required packages:



2nd: Super Tokens Initialization:



3rd: Enable CORS

Cors is used to interact with the Api

4th: register code

```
app.post("/register", async (req, res) => {
    console.log("Received request body:", req.body);
    try {
        const { email, password } = req.body;

        if (!email || !password) {
            return res.status(400).json({ message: "Email and password are required" });
        }

        const response = await EmailPassword.signUp({
            email,
            password,
        });

        if (response.status === "OK") {
            return res.status(200).json({
                message: "User registered successfully",
                user: response.user,
            });
        } else {
            return res.status(400).json({
                message: "Registration failed",
                error: response.message,
            });
        }
    } catch (error) {
        console.error("Error during registration:", error);
        return res.status(500).json({
            message: "Something went wrong during registration",
            error: error.message,
        });
    }
```

He post his request in req.body

If email or password is missing he will return an error

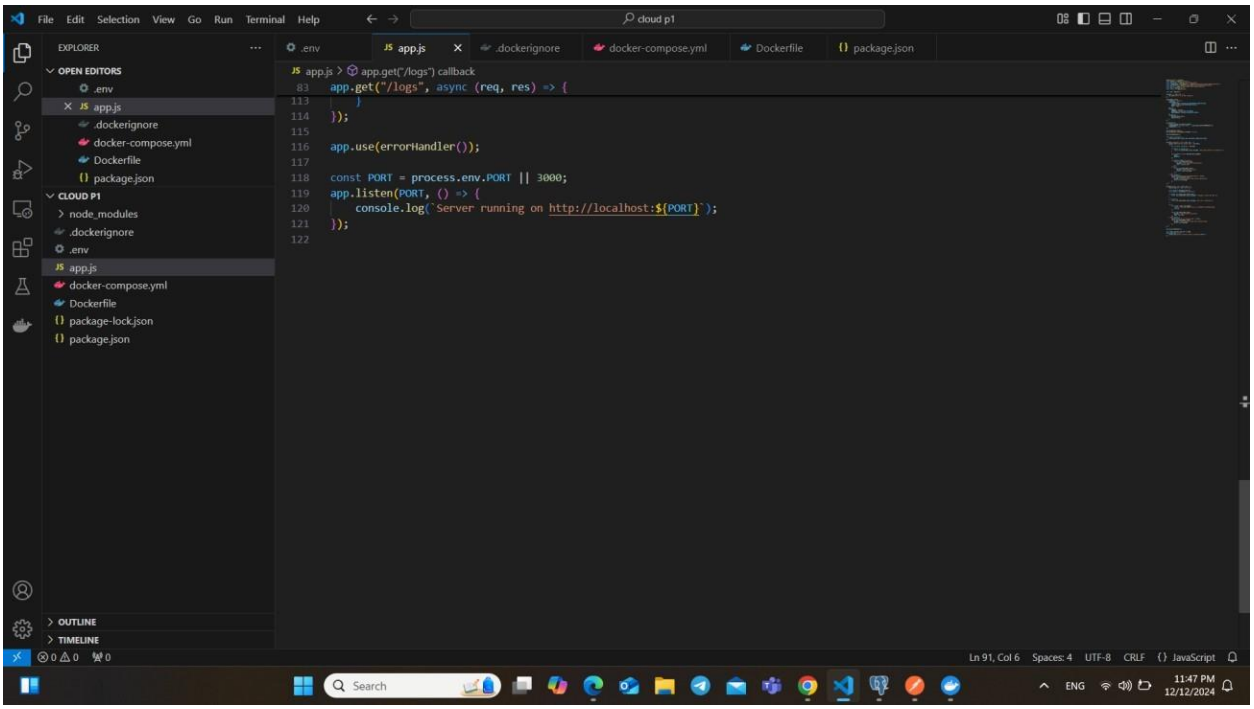
Emalpassword.signup is a super token function

If it succed he will return a succes message other wise if failed 5th:

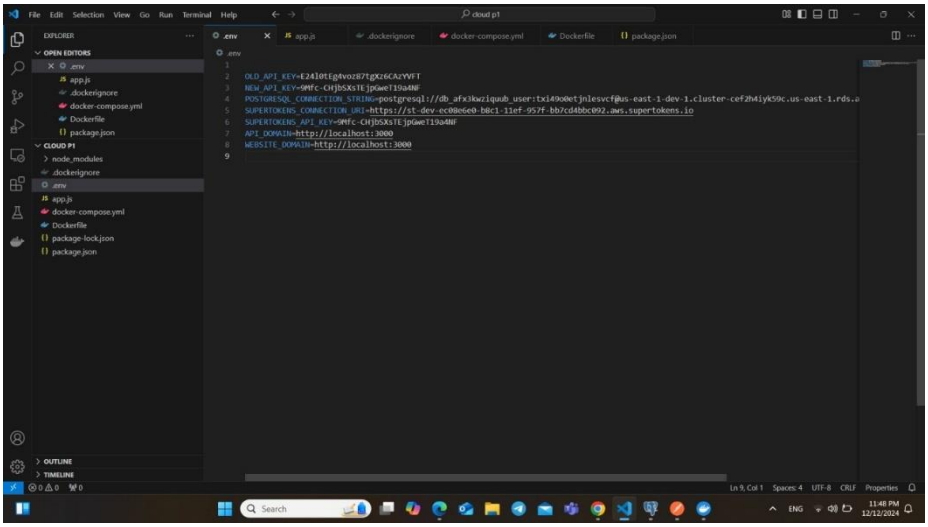
get logs

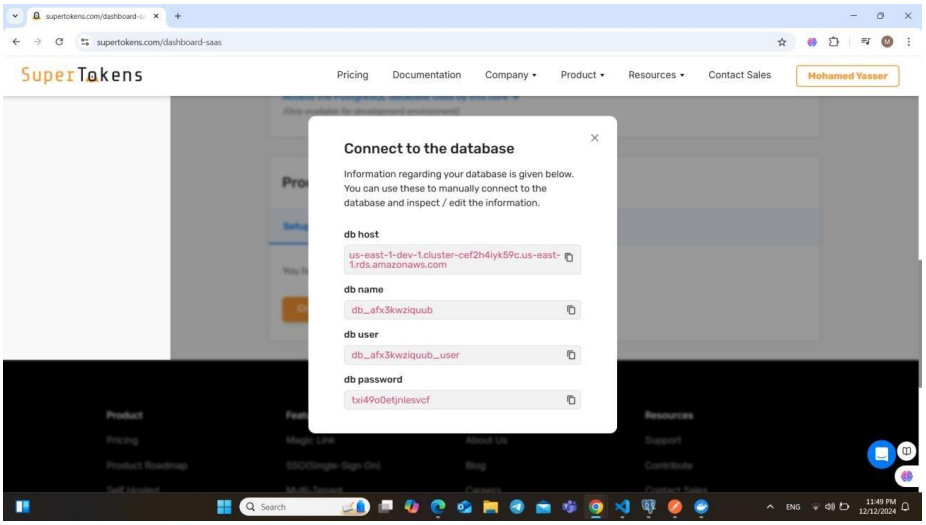
He gets the logs after posting

6th: finally he the port :

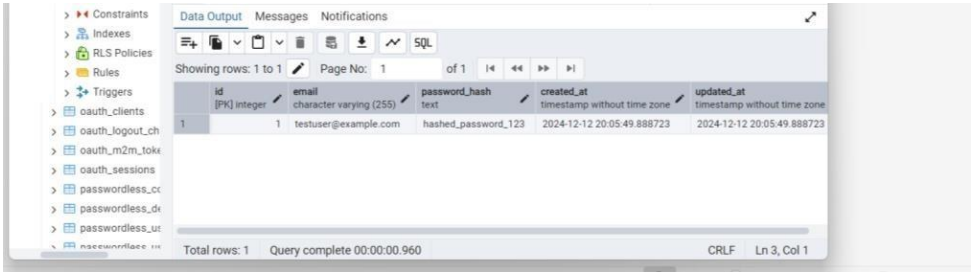


Second : Design and implement the database schema for user profiles and logs using PostgreSQL

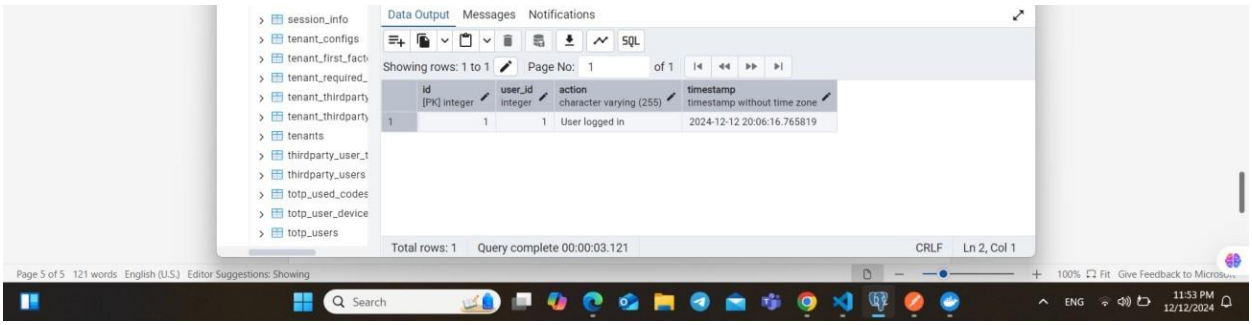




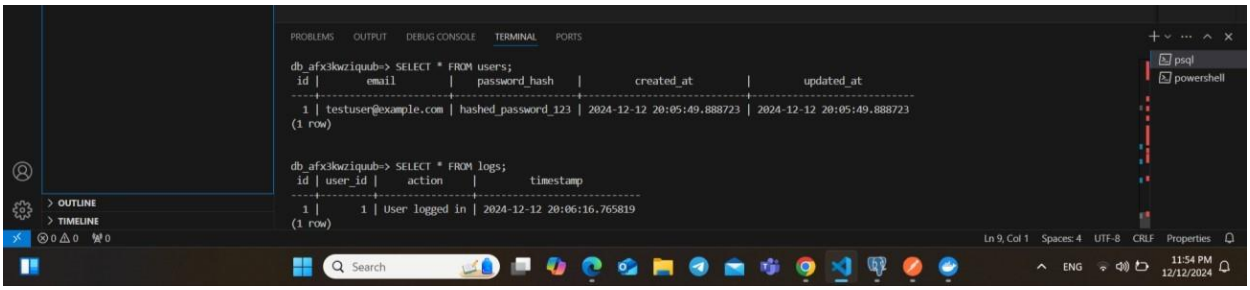
User table :



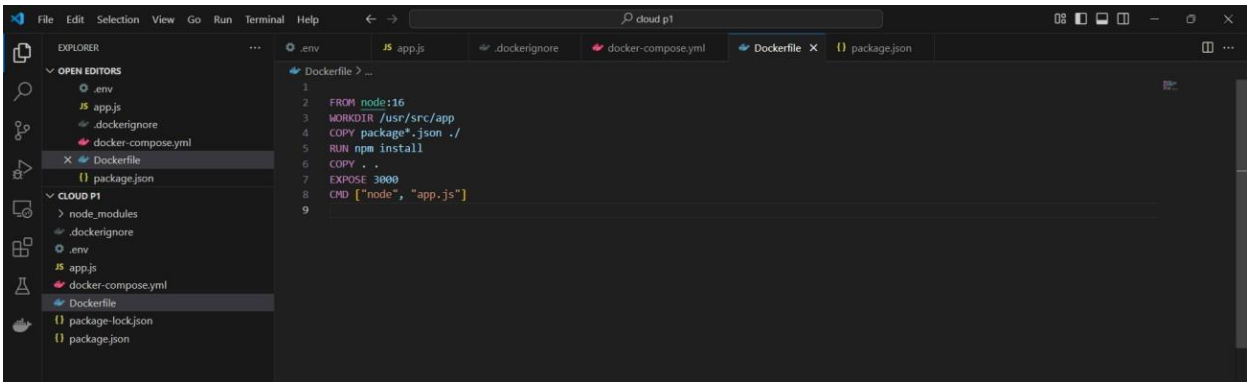
Logs table :



Third : Secure data storage and integrate the database with User-Service



Fourth : Containerize User-Service and DB-Service using Docker.



Pull a node js

image Set a work

directory

Copy the package .json

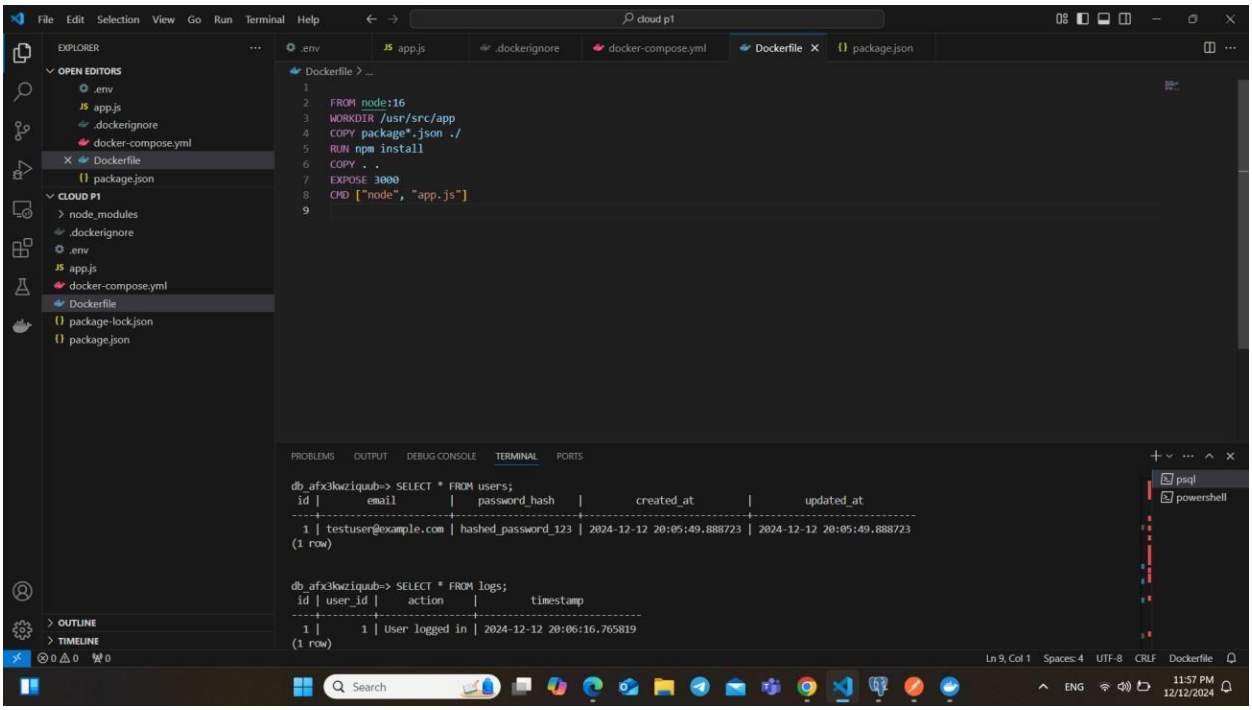
Install all dependencies

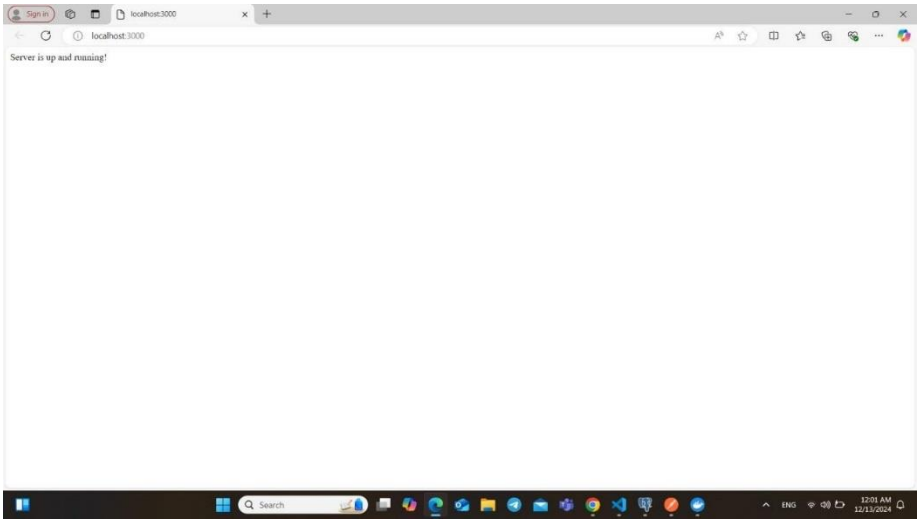
Copy the rest of the code

Set the port

Set the entry point

Output:





Phase 2: Core Infrastructure Setup

API Gateway

This Dockerfile creates a simple Nginx container image.

- FROM nginx:latest: This line pulls the latest Nginx image from Docker Hub and sets it as the base image.

- COPY nginx.conf /etc/nginx/nginx.conf: This line copies the nginx.conf file from the current directory into the Nginx configuration directory on the container.

- EXPOSE 80: This exposes port 80 to allow external access to the Nginx server running inside the container.

- CMD ["nginx", "-g", "daemon off;"]: This sets the command that runs Nginx in the container, with -g "daemon off;" ensuring that Nginx runs in the foreground (non-daemon mode).

General Overview:

This configuration sets up NGINX to act as a reverse proxy for multiple services, including translation and summarization services. It also handles logging for access and errors.

Configuration Breakdown:

1. worker_processes 1:
    a. Defines the number of worker processes for handling requests. In this case, it is set to 1.
2. events:
    a. Specifies the number of worker connections per worker process. Here, worker_connections is set to 1024.
3. http:
    a. Contains the main configuration for handling HTTP requests.
4. server block:
    a. Configures the settings for the NGINX server.
5. upstream translation_services:
    a. Defines an upstream group for translation services. Requests to /translate/ will be proxied to these services.
    b. en2ar-service:8001 and ar2en-service:8002 represent two different services for translation between English and Arabic.
6. upstream summarization_services:
    a. Defines an upstream group for summarization services. Requests to /summarize/ will be proxied to summary-service:8003.
7. location /translate/:
    a. Handles requests starting with /translate/.
    b. Uses proxy_pass to forward requests to the translation_services upstream group.
8. location /summarize/:
    a. Handles requests starting with /summarize/.

    b. Uses proxy_pass to forward requests to the summarization_services upstream group.
9. proxy_set_header:
    a. Sets various headers to maintain request information (e.g., Host, X-Real-IP, X-Forwarded-For, X-Forwarded-Proto).
10. Logging:
    a. Optional access and error logging are configured with paths /var/log/nginx/access.log and /var/log/nginx/error.log.

Message Queue Architecture:

Kafka Integration with Flask and FastAPI:

1. Kafka as a Messaging Broker:
    a. Kafka acts as a message broker for sending and receiving messages between different services (e.g., translation or summarization services).
    b. Topics are created such as:
        i. translation-requests
        ii. translation-responses
        iii. summarization-requests
        iv. summarization-responses
2. Flask Service:
    a. In Flask, a Kafka producer (KafkaProducer) is used to send requests to Kafka (e.g., translation text).
    b. A Kafka consumer (KafkaConsumer) is used to listen for responses from Kafka and process them.
3. FastAPI Service:
    a. Similar to Flask, Kafka is used to handle requests and responses between services.
    b. Requests are sent to Kafka using FastAPI, and responses are consumed to process the results.
4. Message Exchange:
    a. Different components (services) send data as JSON messages to Kafka under specific topics.
    b. These messages are consumed by the appropriate service to process requests and provide responses.

Frontend Service

- Project Name: frontend
- Version: 1.0.0
- Lockfile Version: 3

- Requires: Indicates tools like Node.js are required to use this lockfile.

Dependencies:

- Dependencies: These are packages that your project relies on for features like styling, UI components, and core functionality.
    - Example: @emotion/react, @mui/material, react.

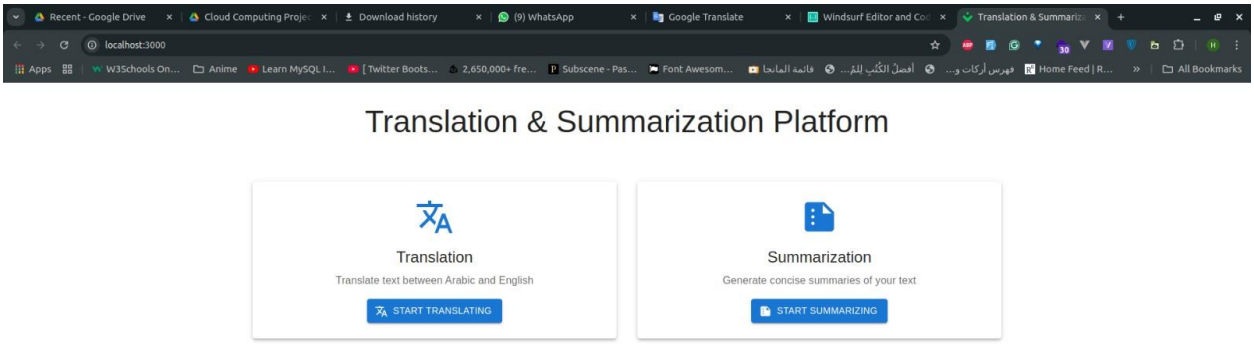
Development Dependencies:

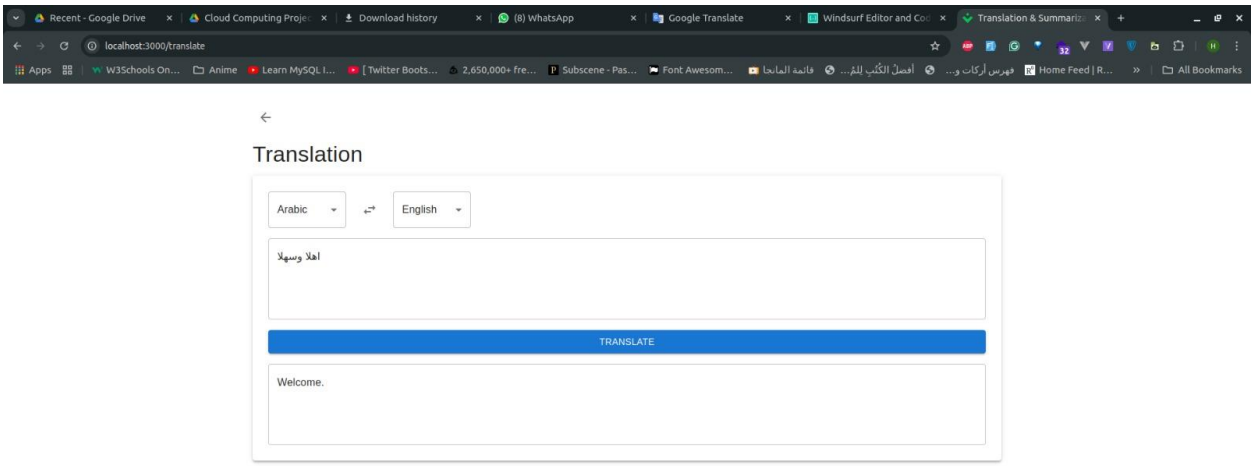
- Dev Dependencies: These packages are used during development for tasks like testing and building the project.
    - Example: @testing-library/react, typescript.

Node Modules:

- Node Modules: Lists individual packages with details like version, URLs, and dependencies.
    - Example: @babel/core with its specific version, dependencies like @babel/parser and @babel/generator.

Integration Testing





Demo link for front-end and services: https://vimeo.com/1042315378?share=copy

Demo link for user-service: https://vimeo.com/1042534873?share=copy

Team names :

Name	id
Abdulrahman Mohamed Eltahan	21100958
Abdelrahman Ahmed Haleem	21100829
Rawan Ayman Adli	21100867
Mohamed Yasser Mohamed	21100838