NAME: ADITI SHARMA

COLLEGE: KIET GROUP OF INSTITUTIONS

Blood Bridge: optimizing Lifesaving resources using AWS Services

📑 CONTENTS

Project Description:

"BloodBridge" is a comprehensive web-based blood bank management system designed to streamline the process of blood donation and distribution. The project leverages Amazon Web Services (AWS) for robust and scalable infrastructure, utilizing

Amazon DynamoDB for secure and efficient data storage and Amazon EC2 for reliable web hosting. The user-friendly web interface allows individuals to register and log in to their personal accounts, creating a seamless experience for both donors and recipients. Once logged in, users are presented with a dashboard that serves as a central hub for all blood-related activities.

The dashboard prominently features current blood requests, allowing users to view real-time needs in their community. Additionally, registered users can easily submit their own blood requests, specifying blood type, quantity, and urgency. This system not only facilitates quick responses to critical blood needs but also fosters a sense of community engagement in the life-saving act of blood donation. By combining modern cloud technology with an intuitive user interface, "BloodBridge" aims to bridge the gap between blood donors and those in need, ultimately saving lives and improving healthcare outcomes.

Scenarios

Scenario 1: Emergency Blood Request

Sarah, a hospital administrator, logs into Life Link during a critical situation. A patient needs a rare blood type urgently. Using her dashboard, Sarah quickly submits a high-priority blood request, specifying the required blood type and quantity. The system immediately notifies potential donors in the area, significantly reducing the time to find a match and potentially saving the patient's life.

Scenario 2: Regular Donor Management

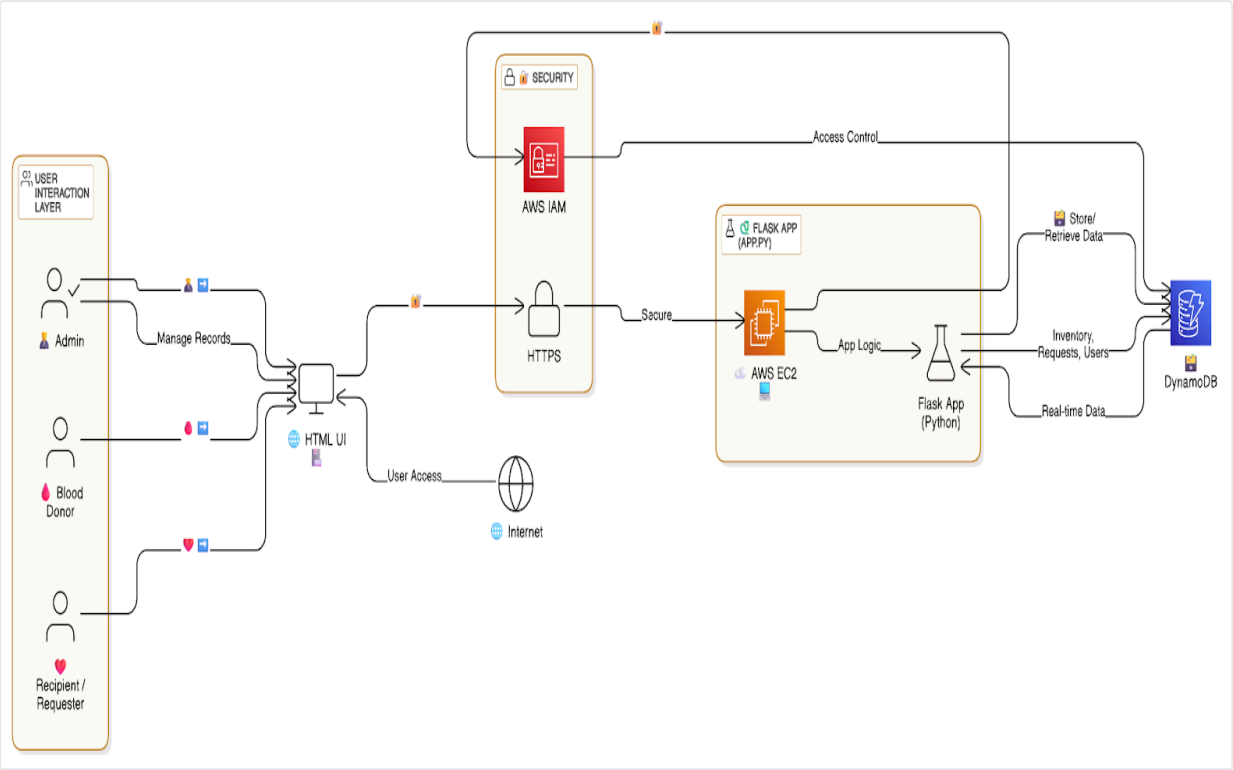
John, a regular blood donor, uses Life Link to manage his donations. After logging in, he checks his dashboard to see when he's eligible to donate again. He notices a nearby blood drive event listed in the requests section. John uses the system to schedule his next donation, helping maintain a steady supply of blood for the local hospitals.

Scenario 3: Blood Bank Inventory Update

A blood bank manager, Lisa, uses Life Link to update the current blood inventory. She logs into her specialized account and accesses a feature to input the latest stock levels for each blood type. The system automatically updates the dashboard for all users, reflecting the current needs. This real-time update helps prioritize requests for blood types that are running low, ensuring efficient distribution of this vital resource.

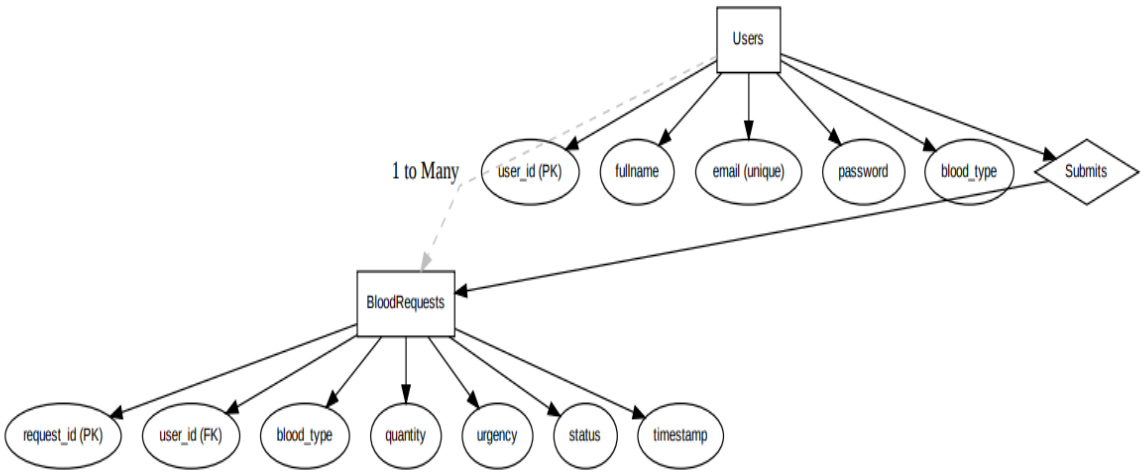
Architecture

This AWS-based architecture powers a scalable and secure web application using Amazon EC2 for hosting the backend, with a lightweight framework like Flask handling core logic. Application data is stored in Amazon DynamoDB, ensuring fast, reliable access, while user access is managed through AWS IAM for secure authentication and control. Real-time alerts and system notifications are enabled via Amazon SNS, enhancing communication and user engagement.



Entity Relationship (ER) Diagram

An ER (Entity-Relationship) diagram visually represents the logical structure of a database by defining entities, their attributes, and the relationships between them. It helps organize data efficiently by illustrating how different components of the system interact and relate. This structured approach supports effective database normalization, data integrity, and simplified query design.



Project Flow

Milestone 1. Backend Development and Application Setup

- Develop the Backend Using Flask.
- Integrate AWS Services Using boto3.

Milestone 2. AWS Account Setup and Login

- Set up an AWS account if not already done.
- Log in to the AWS Management Console

Milestone 3. DynamoDB Database Creation and Setup

- Create a DynamoDB Table.
- Configure Attributes for User Data and Book Requests.

Milestone 4. SNS Notification Setup

- Create SNS topics for book request notifications.
- Subscribe users and library staff to SNS email notifications.

Milestone 5. IAM Role Setup

- Create IAM Role
- Attach Policies

Milestone 6. EC2 Instance Setup

- Launch an EC2 instance to host the Flask application.
- Configure security groups for HTTP, and SSH access.

Milestone 7. Deployment on EC2

- Upload Flask Files

- Run the Flask App

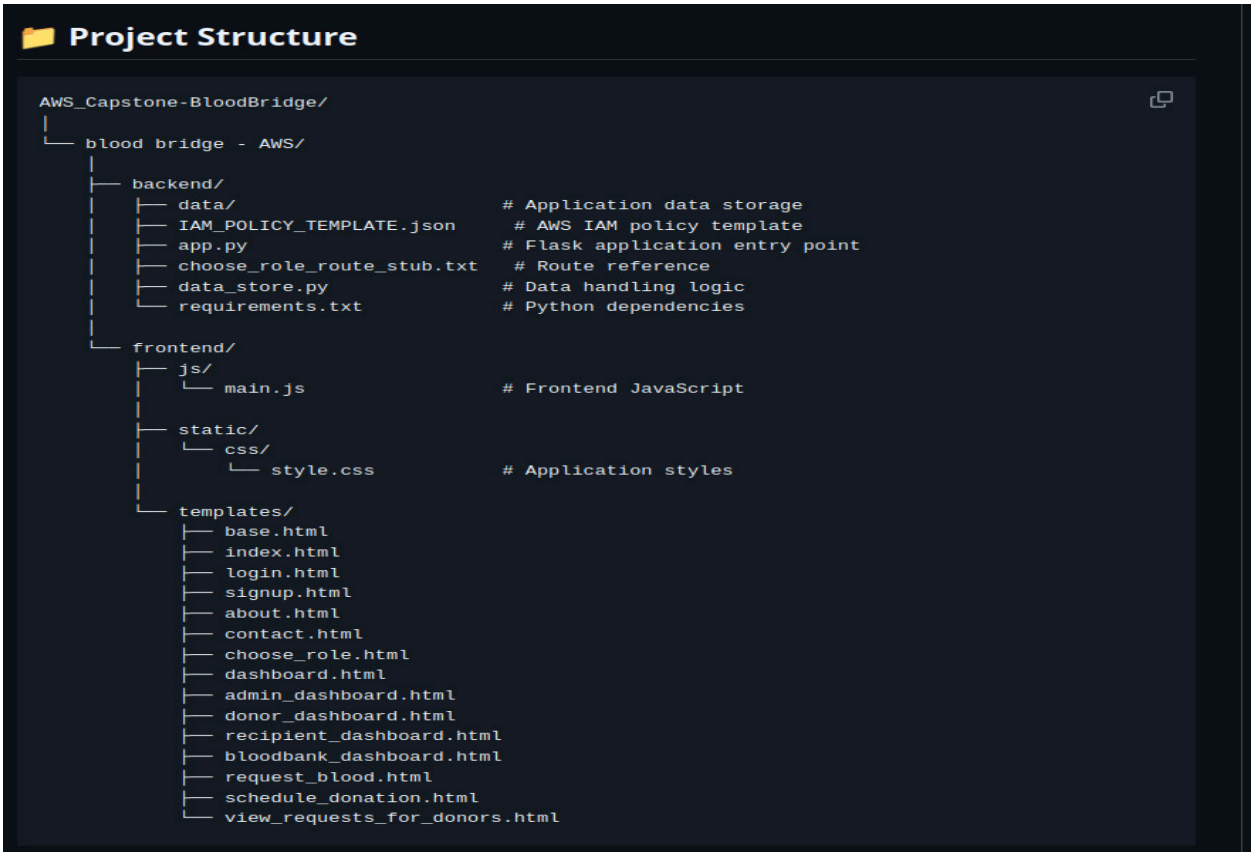
Milestone 8. Testing and Deployment

- Conduct functional testing to verify user signup, login, buy/sell stocks and notifications.

Milestone 1: Web Application Development And Setup

Backend Development and Application Setup focuses on establishing the core structure of the application. This includes configuring the backend framework, setting up routing, and integrating database connectivity. It lays the groundwork for handling user interactions, data management, and secure access.

● Activity 4.1: Develop the backend using Flask.



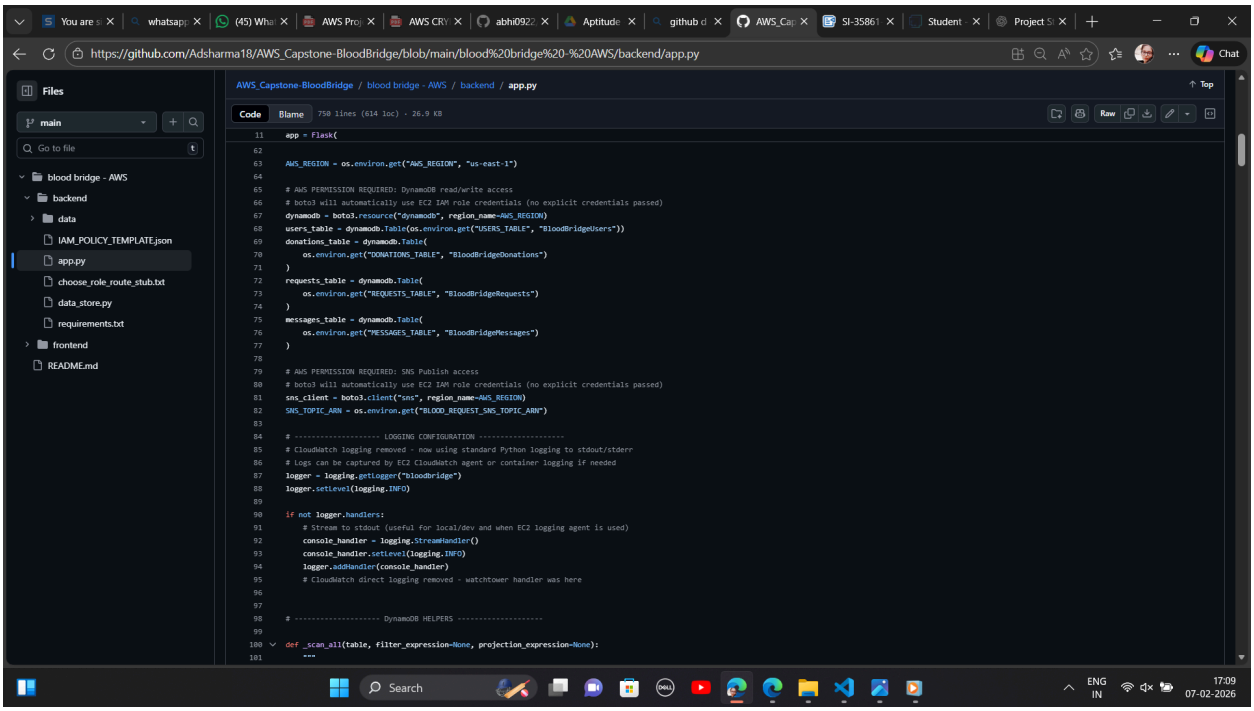
Description of the Code Flask App Initialization

● Imports and Configuration

- The application starts by importing necessary modules such as Flask, boto3, session handling, and other utilities like datetime.
- app = Flask(__name__): Initializes a Flask web application.
- app.secret_key = "your_secret_key": This key is necessary for securely handling sessions and flash messages (temporary notifications).

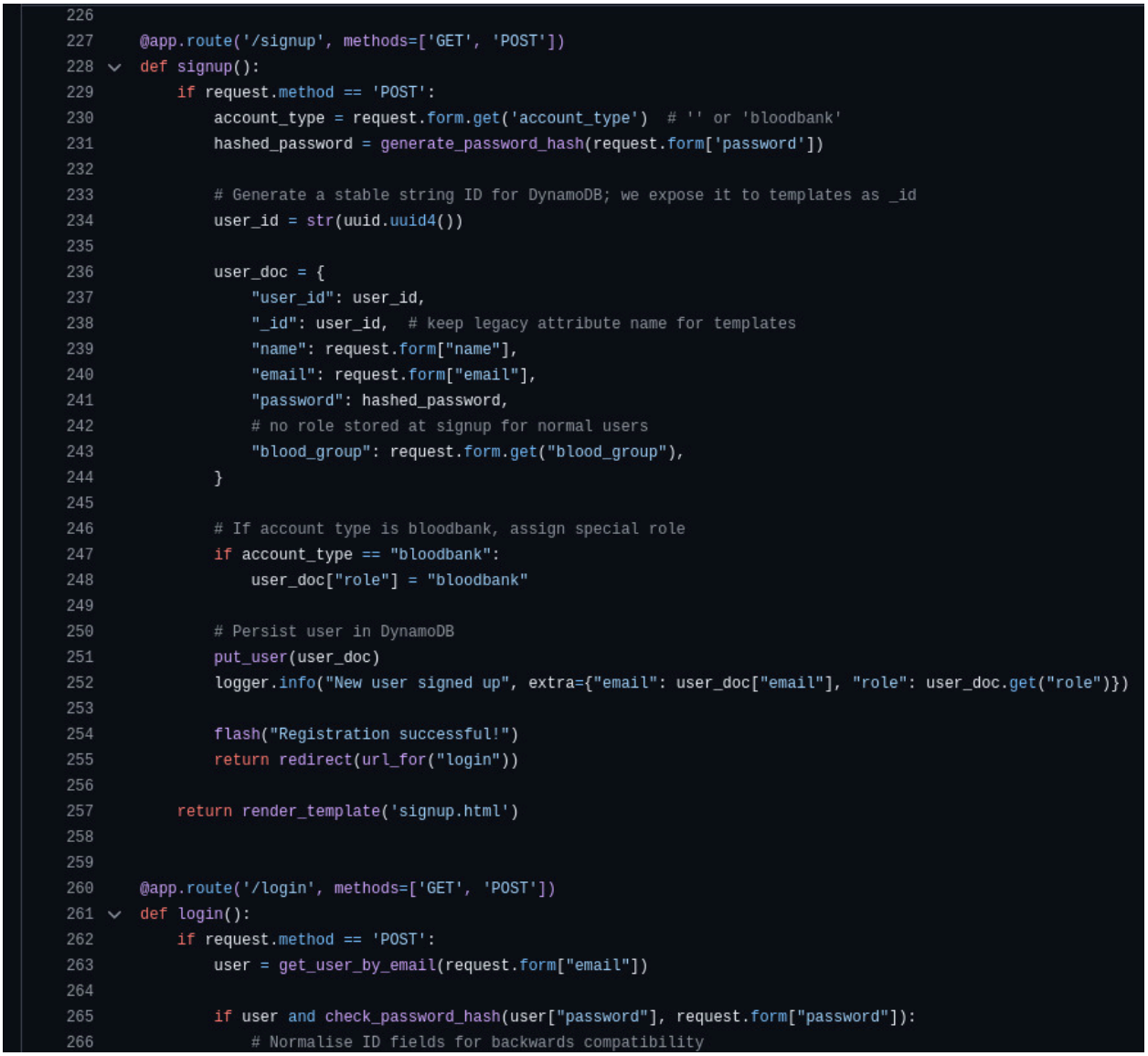
```python
import logging
import os
import uuid
from datetime import datetime

import boto3
# CloudWatch removed: watchtower import was here
from flask import Flask, render_template, request, redirect, url_for, session, flash
from werkzeug.security import generate_password_hash, check_password_hash

app = Flask(
    __name__,
    template_folder="../frontend/templates",
    static_folder="../frontend/static",
)

app.secret_key = os.environ.get("FLASK_SECRET_KEY", "blood_bridge_secret_key")

# ------------------- AWS CONFIGURATION -------------------
# IAM ROLE-BASED AUTHENTICATION:
# This application uses EC2 Instance IAM Role for AWS authentication.
# NO hardcoded AWS credentials (access keys/secrets) are used.
#
# CREDENTIAL HANDLING:
# - boto3 automatically uses the EC2 instance's IAM role credentials via
#   the default credential chain (EC2 instance metadata service)
# - No explicit credentials are passed to boto3 clients/resources
# - When running on EC2, the instance must have an IAM role attached
#
# REQUIRED IAM ROLE PERMISSIONS:
# The EC2 instance IAM role must have the following permissions:
```

Description: The section shows a Flask app (`dynamo_app.py`) that connects to AWS DynamoDB using `boto3`. It sets up a secret key, defines the AWS region, accesses `"users"` and `"requests"` tables, and renders `index.html` at the root route.



Sign Up (/signup):

- In this route the user submits a registration form with details like fullname, email, password, and blood_type.
- It checks if the email already exists in the database.
- If the email exists, the user is redirected to the login page, and a flash message is shown.
- If the user is new, their details are inserted into the database.



```python

@app.route('/signup', methods=['GET', 'POST'])
def signup():
    if request.method == 'POST':
        account_type = request.form.get('account_type')  # '' or 'bloodbank'
        hashed_password = generate_password_hash(request.form['password'])

        # Generate a stable string ID for DynamoDB; we expose it to templates as _id
        user_id = str(uuid.uuid4())

        user_doc = {
            "user_id": user_id,
            "_id": user_id,  # keep legacy attribute name for templates
            "name": request.form["name"],
            "email": request.form["email"],
            "password": hashed_password,
            # no role stored at signup for normal users
            "blood_group": request.form.get("blood_group"),
        }

        # If account type is bloodbank, assign special role
        if account_type == "bloodbank":
            user_doc["role"] = "bloodbank"

        # Persist user in DynamoDB
        put_user(user_doc)
        logger.info("New user signed up", extra={"email": user_doc["email"], "role": user_doc.get("role")})

        flash("Registration successful!")
        return redirect(url_for("login"))

    return render_template('signup.html')


@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        user = get_user_by_email(request.form["email"])

        if user and check_password_hash(user["password"], request.form["password"]):
            # Normalise ID fields for backwards compatibility
```

User Account Confirmation (/confirm):

- The `/confirm` route retrieves the user from the session and renders a confirmation page

User Login (/login):

- A login form accepts the email and password.
- These credentials are verified by querying the database. If they match, the user is redirected to the dashboard, and session data is stored:

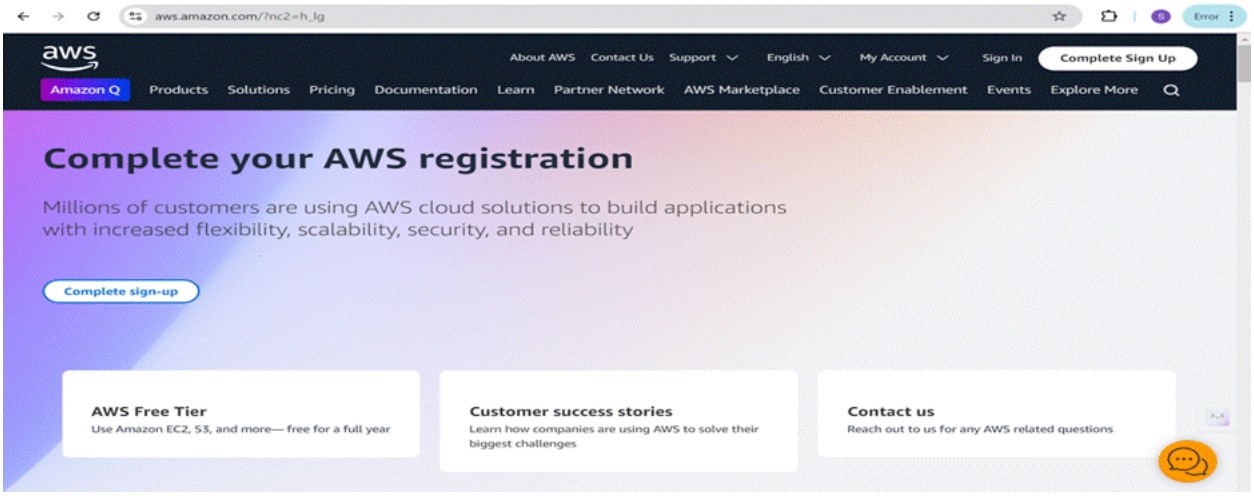
```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        user = get_user_by_email(request.form["email"])

        if user and check_password_hash(user["password"], request.form["password"]):
            # Normalise ID fields for backwards compatibility
            uid = user.get("user_id") or user.get("_id")
            session["user_id"] = uid
            session["name"] = user["name"]
            session["user_email"] = user.get("email")
            # Preserve any special roles (admin / bloodbank) if they exist
            session["role"] = user.get("role")

            logger.info(
                "User logged in",
                extra={"user_id": uid, "email": user.get("email"), "role": user.get("role")},
            )

            # Normal users: go to role selection page
            if user.get("role") in ["admin", "bloodbank"]:
                return redirect(url_for("dashboard"))
            else:
                return redirect(url_for("choose_role"))

        flash("Invalid email or password")

    return render_template('login.html')
```

Description: This block runs the Flask app on the specified host (0.0.0.0) and port (80) in debug mode, allowing for real-time error tracking and hot reloading during development. It ensures the application is accessible from any network interface on the machine.

```python
if __name__ == '__main__':
    # Bind to 0.0.0.0 so the app is reachable on EC2.
    # In production, disable debug and run behind a WSGI server like gunicorn.
    app.run(host='0.0.0.0', port=int(os.environ.get("PORT", 5000)), debug=bool(os.environ.get("FLASK_DEBUG", False)), use_reloader=False)
```

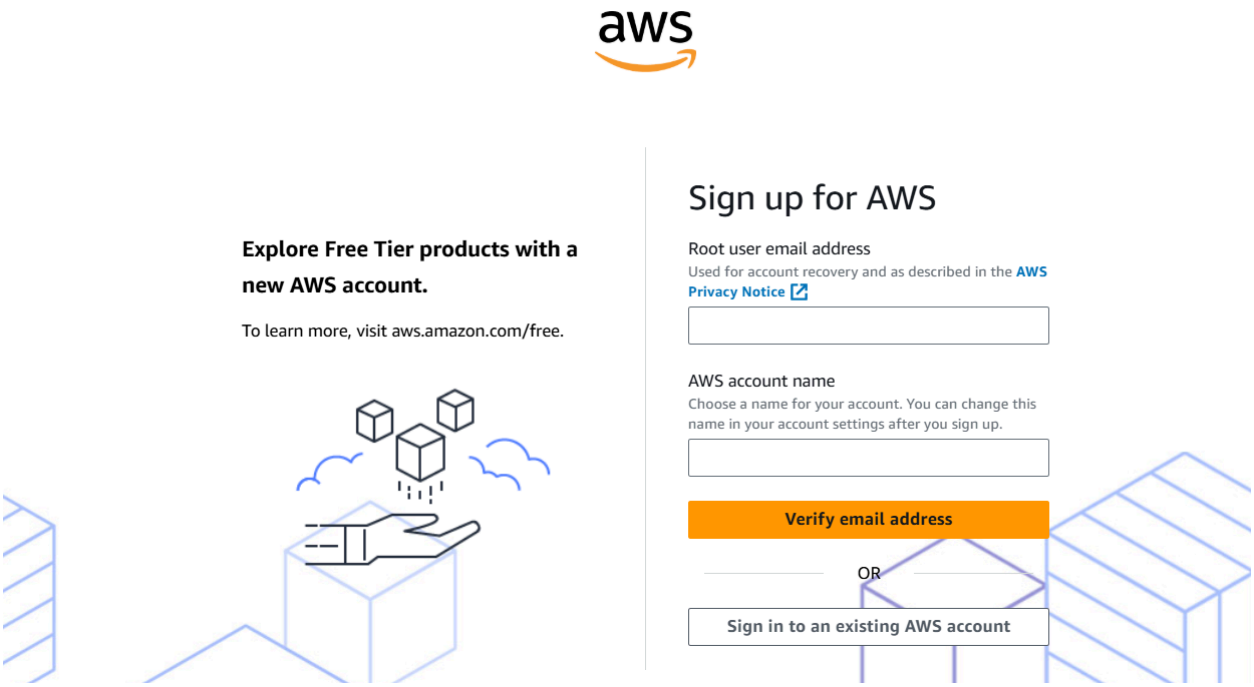
Milestone 2 : AWS Account Setup

An AWS account was created by registering on the official AWS website. During the setup, required details were provided and billing information was configured to enable access to AWS services under the Free Tier.



Activity 1.2: Log in to the AWS Management Console

After successful account creation, the AWS Management Console was accessed using the registered credentials. The console serves as the central platform for managing AWS services used in the Blood Ridge.

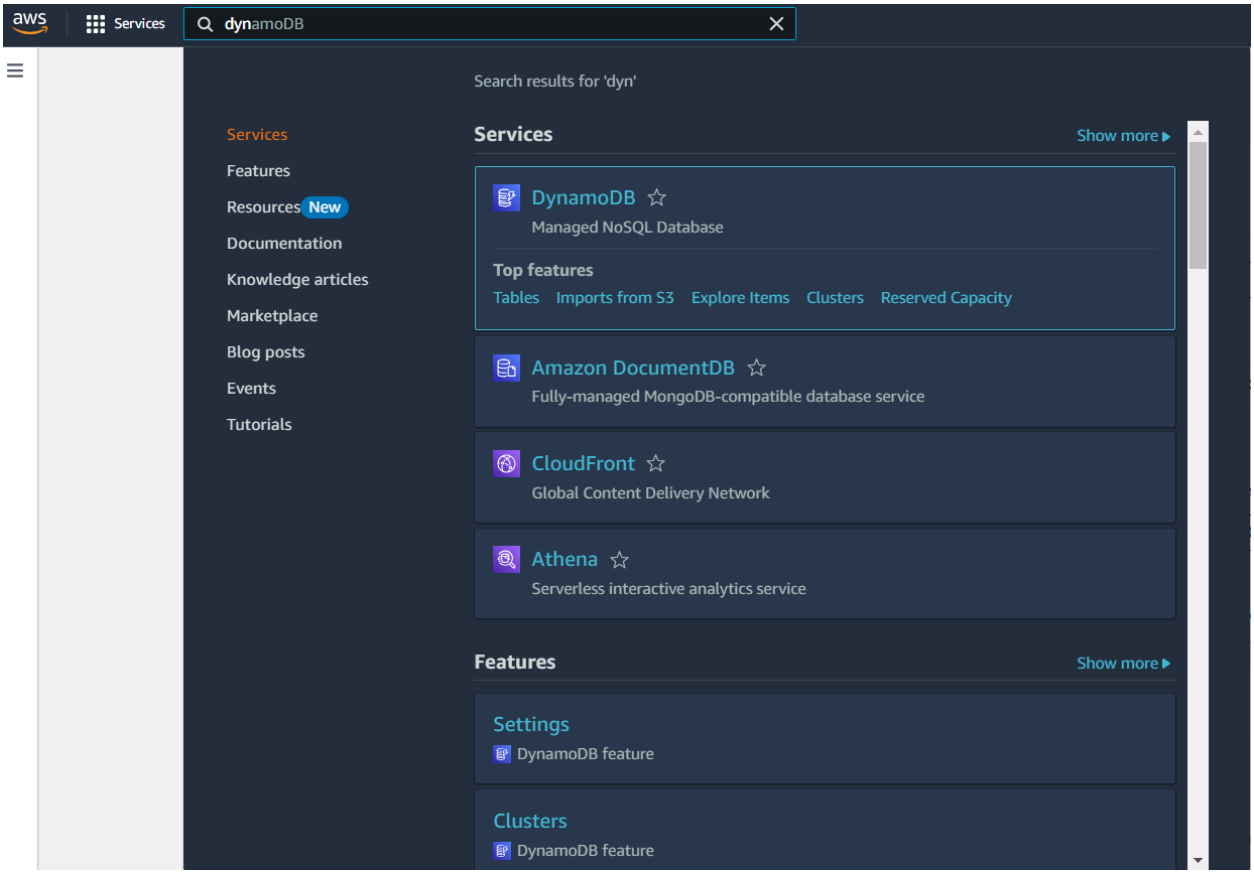


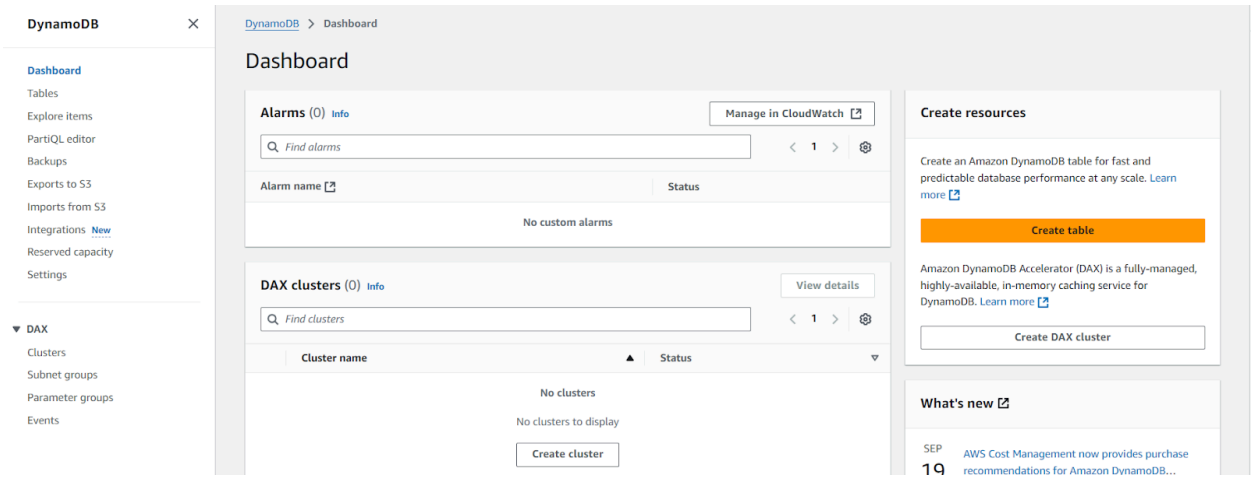
Milestone 3 : DynamoDB Database Creation and

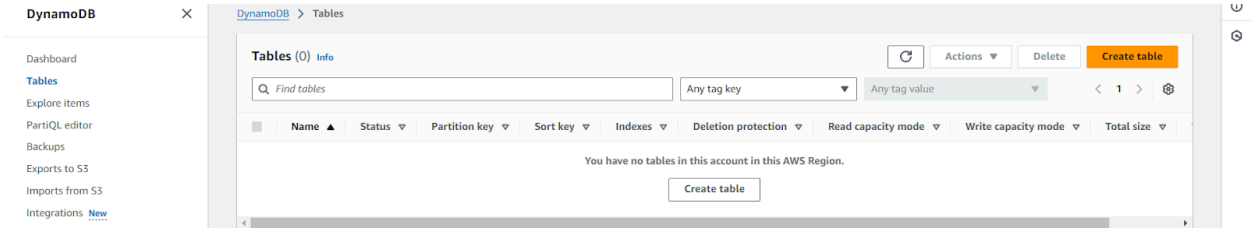
Setup

Database Creation and Setup involves initializing a cloud-based NoSQL database to store and manage application data efficiently. This step includes defining tables, setting primary keys, and configuring read/write capacities. It ensures scalable, high-performance data storage for seamless backend operations.

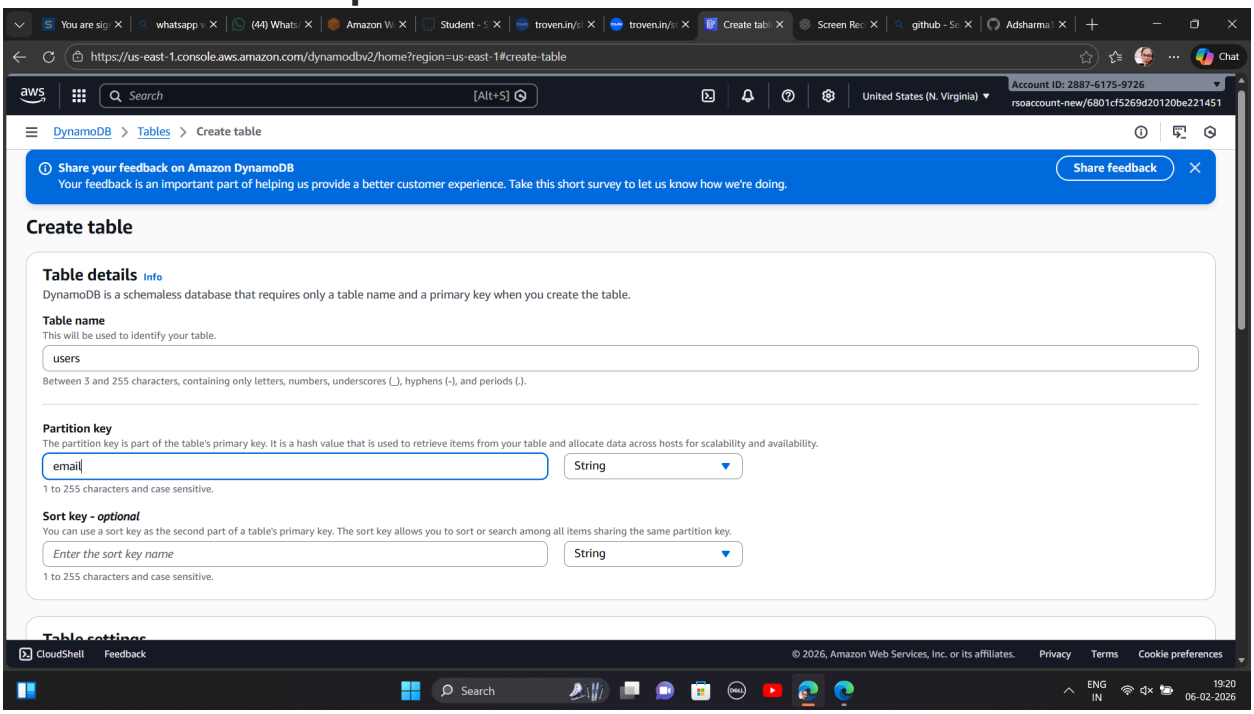
Navigate to the DynamoDB

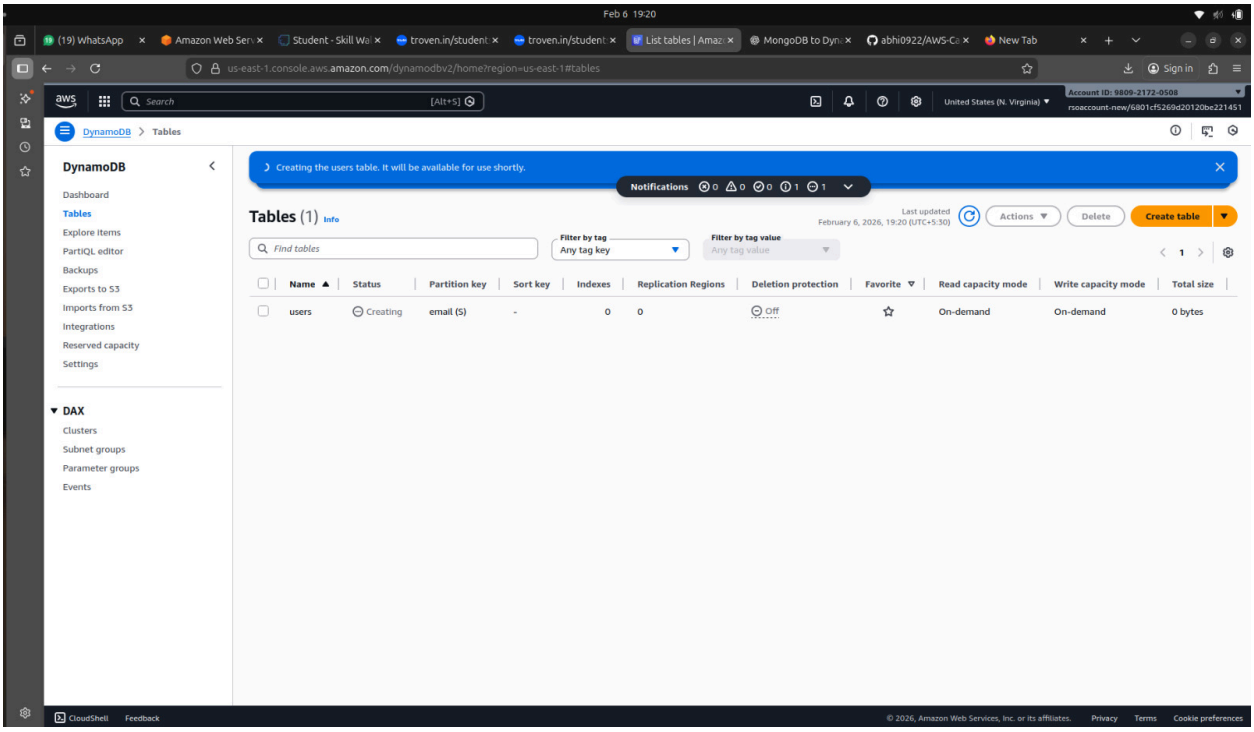


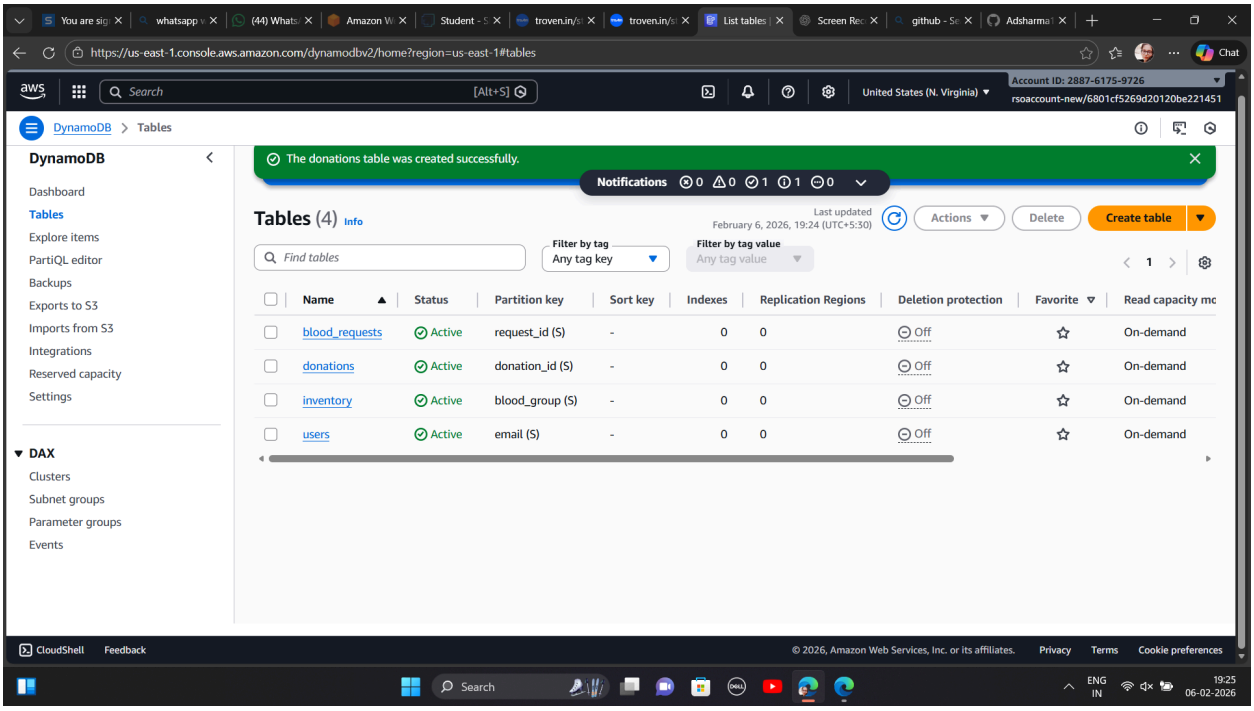




Create a DynamoDB table for storing registration details and requests.





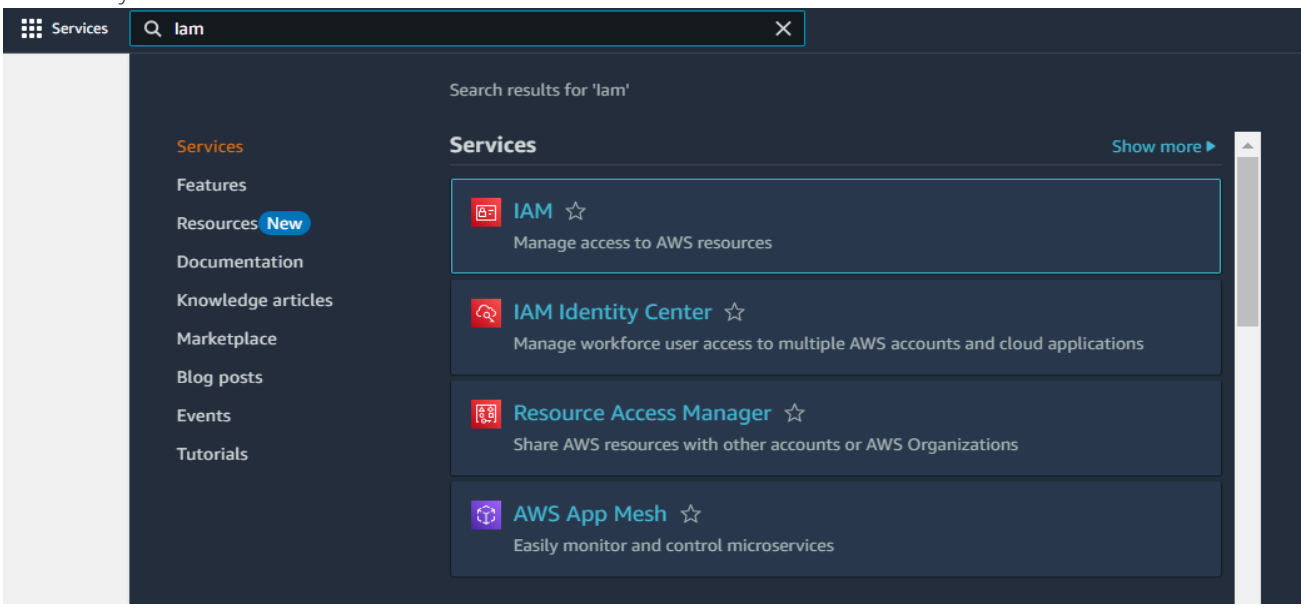


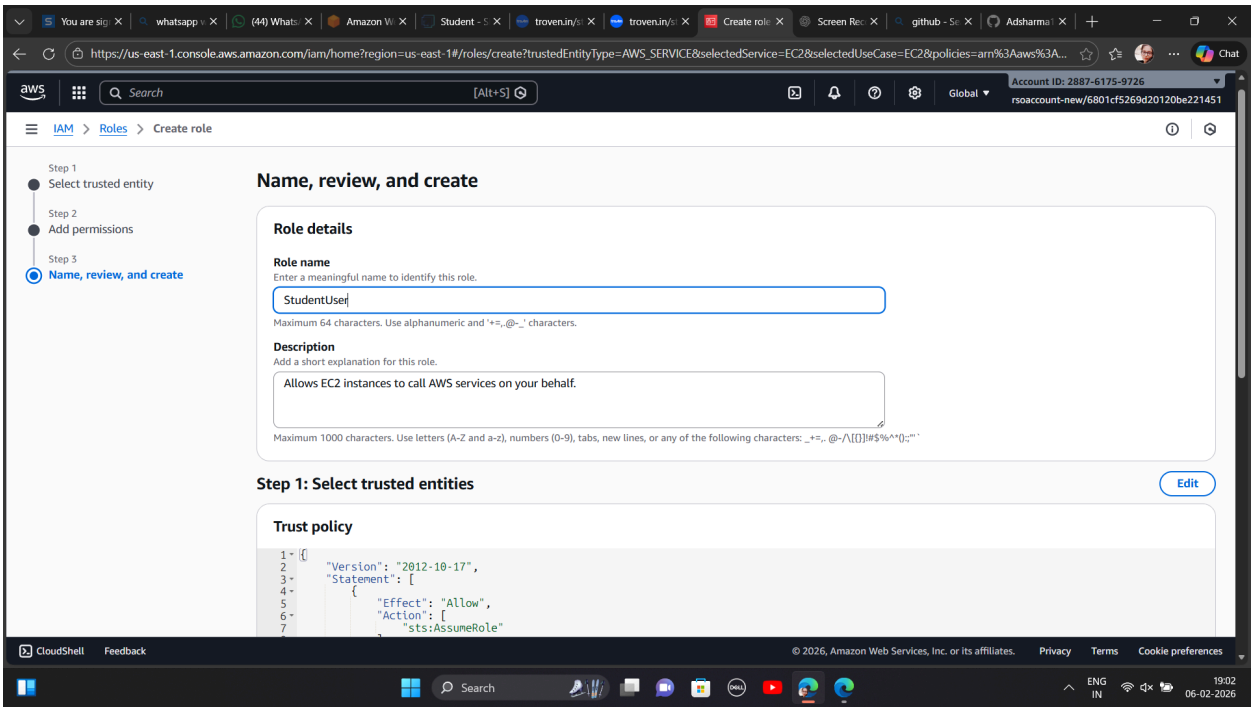
Milestone 4 : IAM Role Setup

Create IAM Role

- In the AWS Console, go to IAM and create a new IAM Role for EC2 to interact with

DynamoDB and SNS.





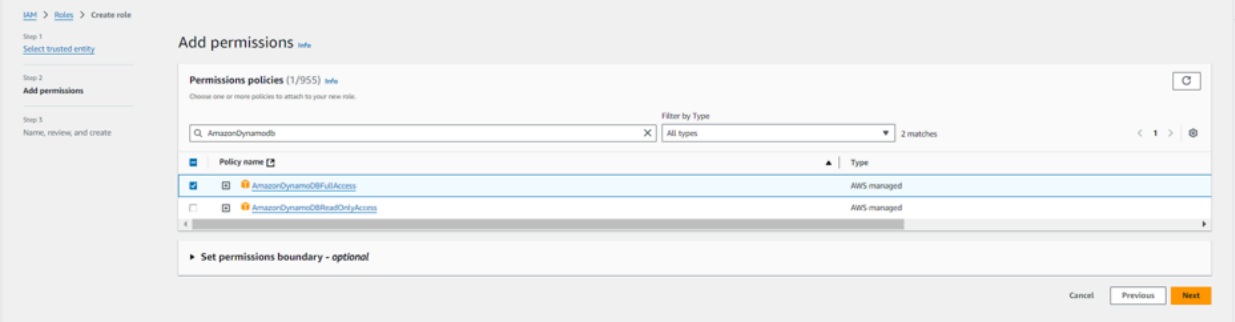
Attach Policies

Attach the following policies to the role:

- AmazonDynamoDBFullAccess: Allows EC2 to perform read/write operations on

DynamoDB.

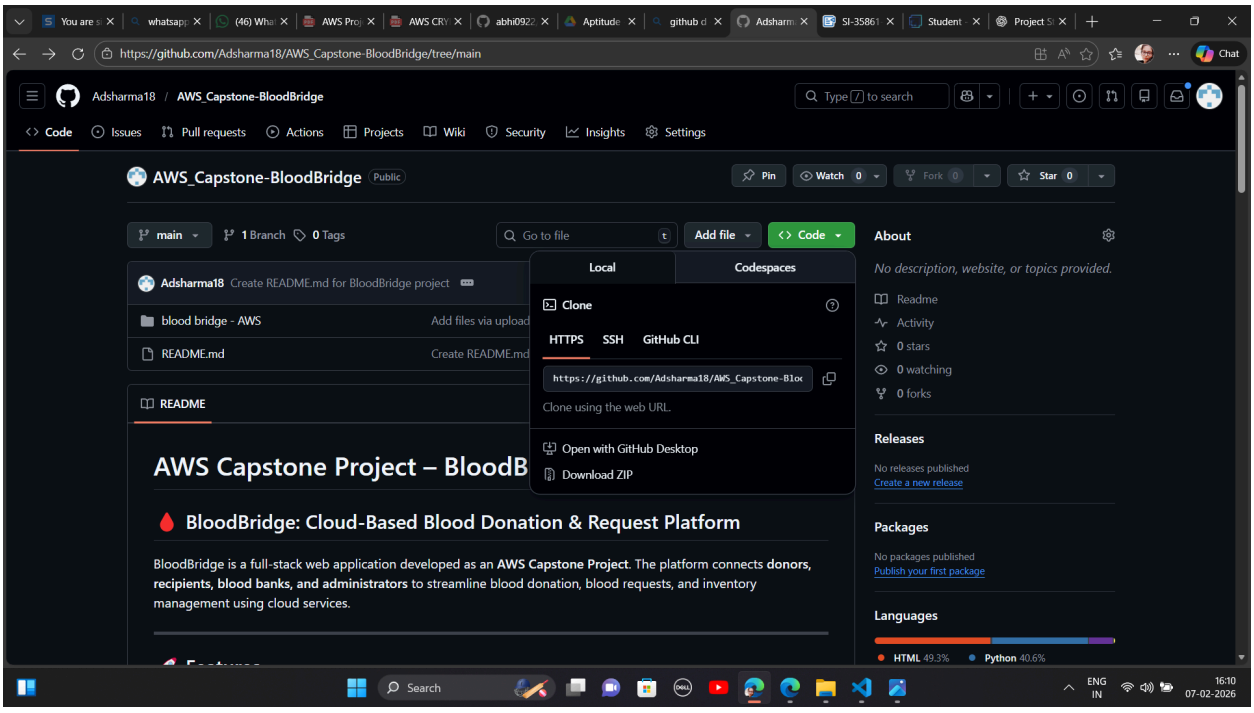
- AmazonSNSFullAccess: Grants EC2 the ability to send notifications via SNS

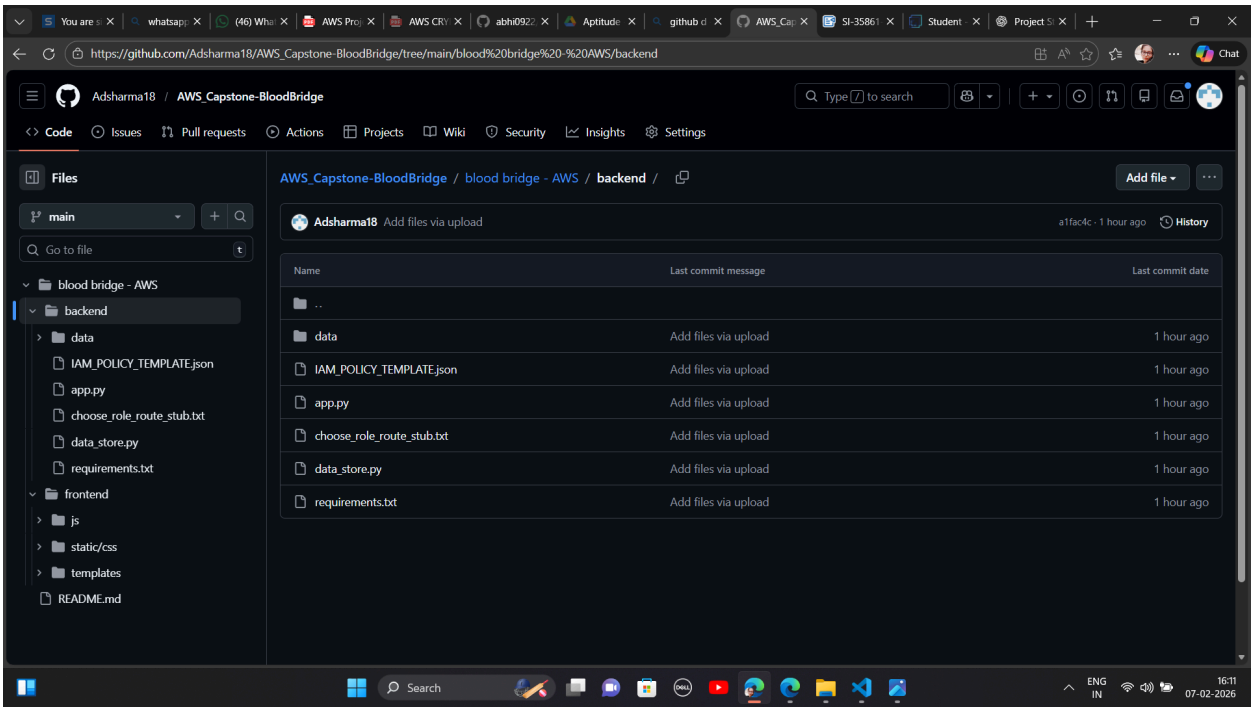


Milestone 5 : EC2 instance Setup

To set up a public EC2 instance, choose an appropriate Amazon Machine Image (AMI) and instance type. Ensure the security group allows inbound traffic on necessary ports (e.g., HTTP/HTTPS for web applications). After launching the instance, associate it with an Elastic IP for consistent public access, and configure your application or services to be publicly accessible.

Milestone 6 : Deployment using EC2

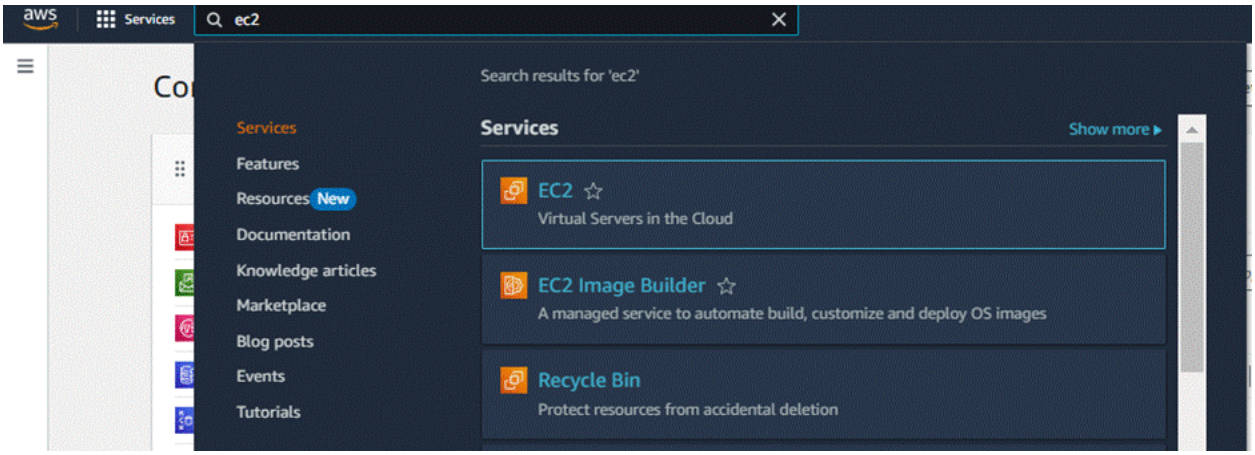


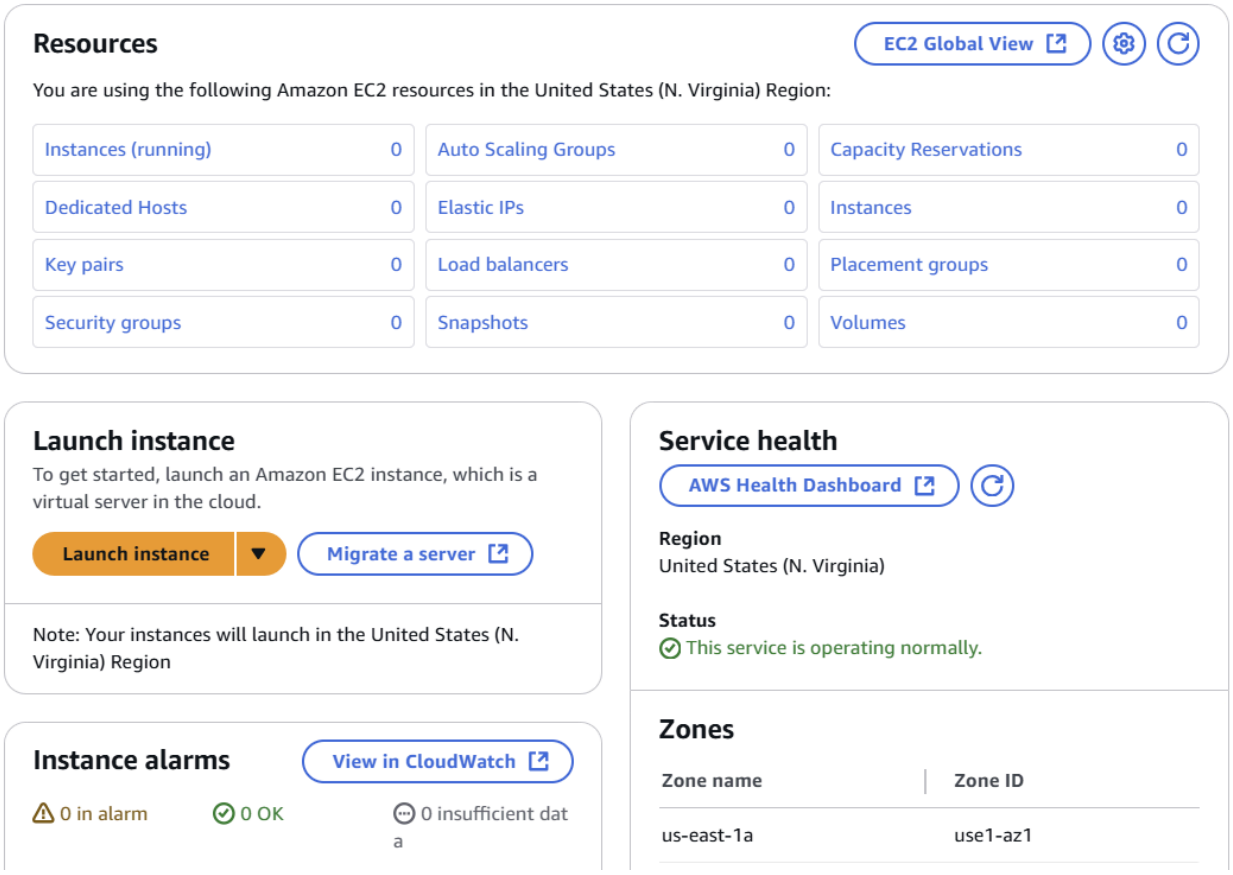


Launch an EC2 instance to host the Flask application.

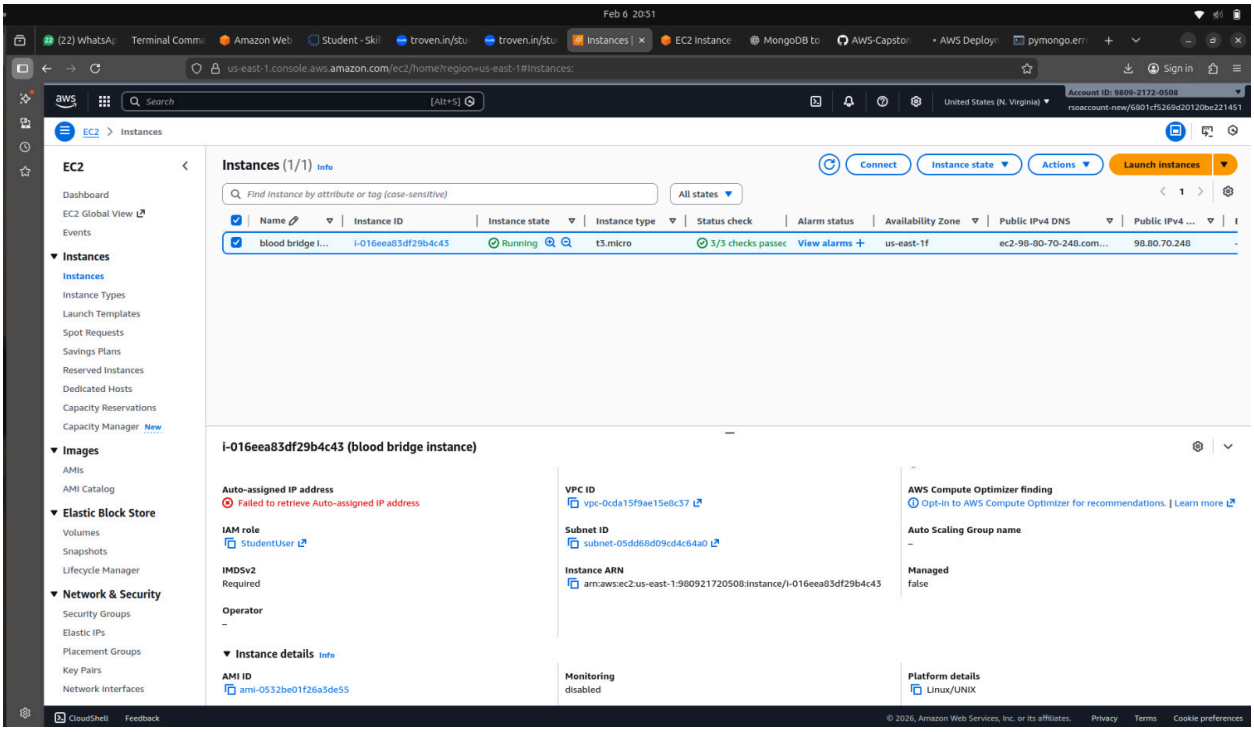
Launch EC2 Instance

- In the AWS Console, navigate to EC2 and launch a new instance.





- This is the EC2 instance console page where you need to click on the launch instance to create a new instance or click on the instances to go with the running instances.



Configure security groups for HTTP, and SSH access.

- Select the no key pair for the authentication process and proceed to the networking settings
- In the networking settings select the VPC you have created for your project if not select the default VPC(make sure to stop the services after using them, VPC also will charge you based upon the As per **New – AWS Public IPv4 Address Charge + Public IP Insight**, you'll get billed $0.005/hour = 0.12/day.)
- Edit the VPC security settings, edit the security configurations of the ec2 which you need to define to allow the incoming and outgoing traffic from your website.

Connect to instance Info

Connect to your instance i-001861022fbcac290 (InstantLibraryApp) using any of these options

EC2 Instance Connect | Session Manager | SSH client | EC2 serial console

Port 22 (SSH) is open to all IPv4 addresses
Port 22 (SSH) is currently open to all IPv4 addresses, indicated by **0.0.0.0/0** in the inbound rule in your security group. For increased security, consider restricting access to only the EC2 Instance Connect service IP addresses for your Region: 13.233.177.0/29. Learn more.

Instance ID
i-001861022fbcac290 (InstantLibraryApp)

Connection Type

- (selected) Connect using EC2 Instance Connect
  Connect using the EC2 Instance Connect browser-based client, with a public IPv4 or IPv6 address.
- ( ) Connect using EC2 Instance Connect Endpoint
  Connect using the EC2 Instance Connect browser-based client, with a private IPv4 address and a VPC endpoint.

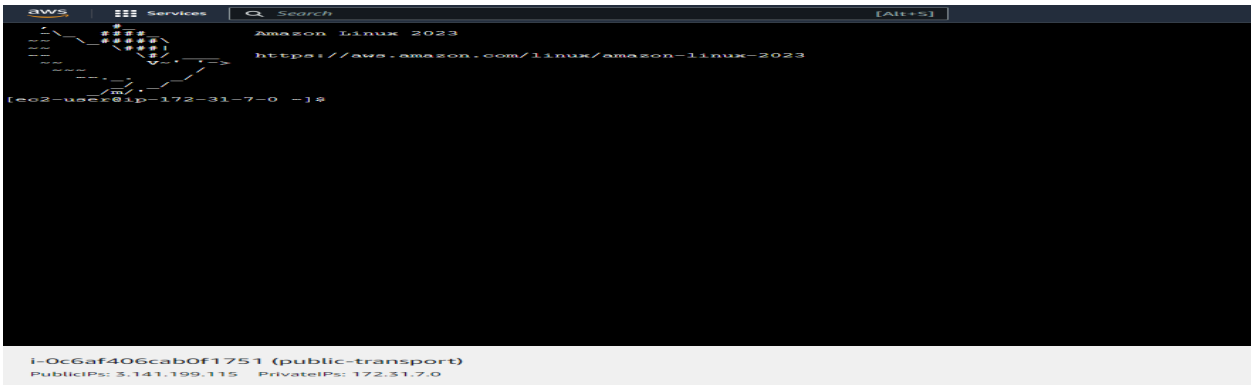
- (selected) Public IPv4 address
  13.200.229.59
- ( ) IPv6 address
  –

Username
Enter the username defined in the AMI used to launch the instance. If you didn't define a custom username, use the default username, ec2-user.

ec2-user

Note: In most cases, the default username, ec2-user, is correct. However, read your AMI usage instructions to check if the AMI owner has changed the default AMI username.

Cancel | Connect



Milestone 6 : Deployment using EC2

Deployment on an EC2 instance involves launching a server, configuring security groups for public access, and uploading your application files. After setting up necessary dependencies and environment variables, start your application and ensure it's running on the correct port. Finally, bind your domain or use the public IP to make the application accessible online.

Install Software on the EC2 Instance

Install Python3, Flask, and Git:

- **On Amazon Linux 2:**

  sudo yum update -y

  sudo yum install python3 git

  sudo pip3 install flask boto3

- **Verify Installations:**

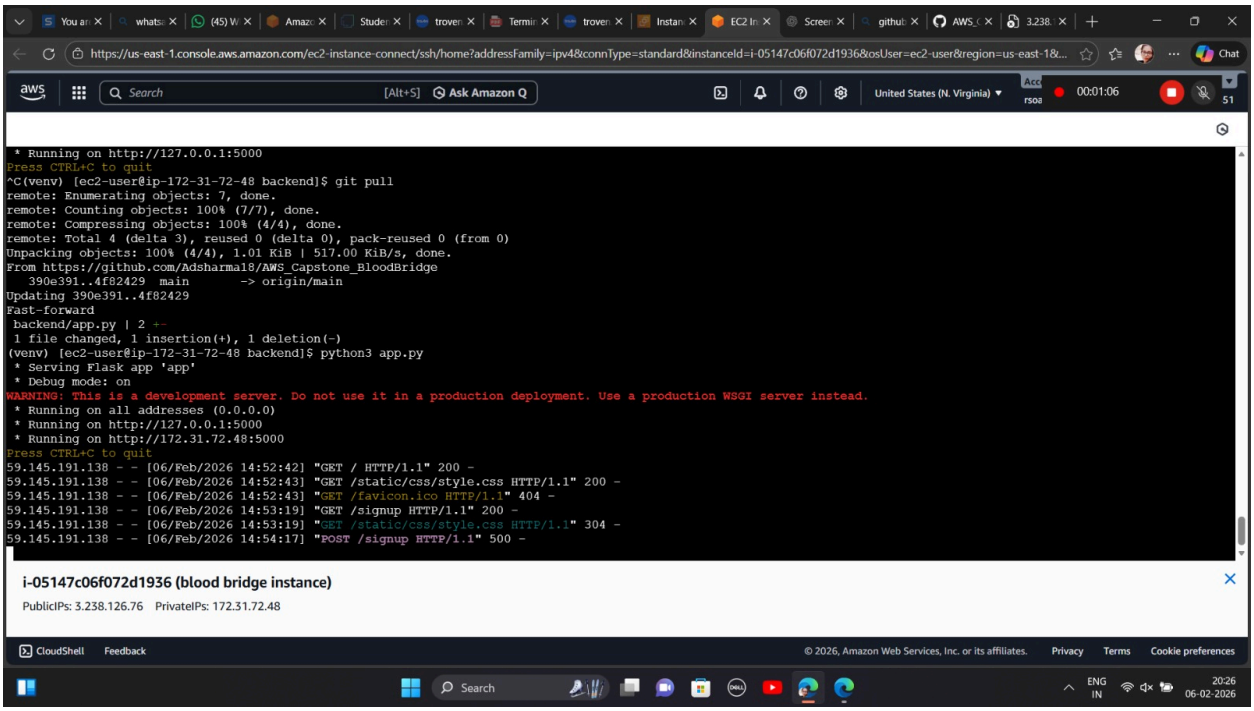
  flask --version

  git --version

Clone your project repository from GitHub into the EC2 instance using Git.

- Run: 'git clone https://github.com/Adsharma18/AWS_Capstone-BloodBridge.git'
  Note: change your-github-username and your-repository-name with your credentials

- here: 'git clone https://github.com/suryavarma03/BloodBridge.git'



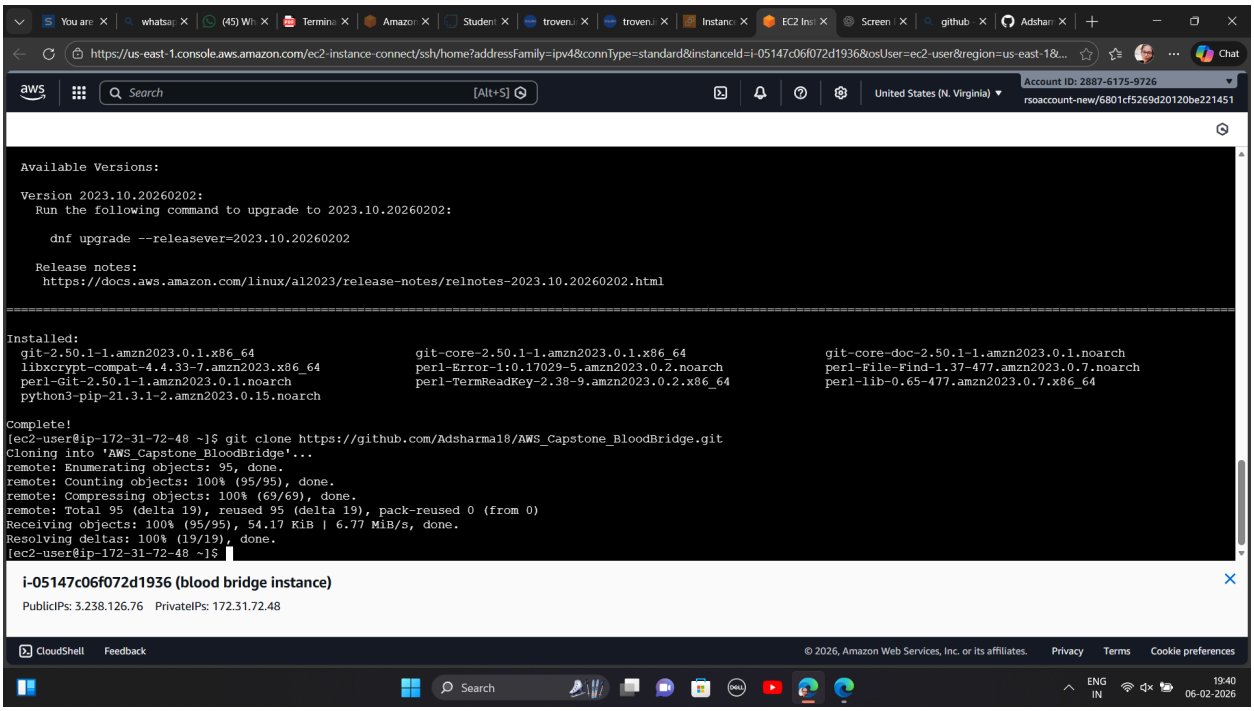
This will download your project to the EC2 instance.

- To navigate to the project directory, run the following command: cd <your files path>
- Once inside the project directory, configure and run the Flask application by executing the following command with elevated privileges:
- Run the Flask Application:
  sudo flask run --host=0.0.0.0 --port=80

Verify the Flask app is running:

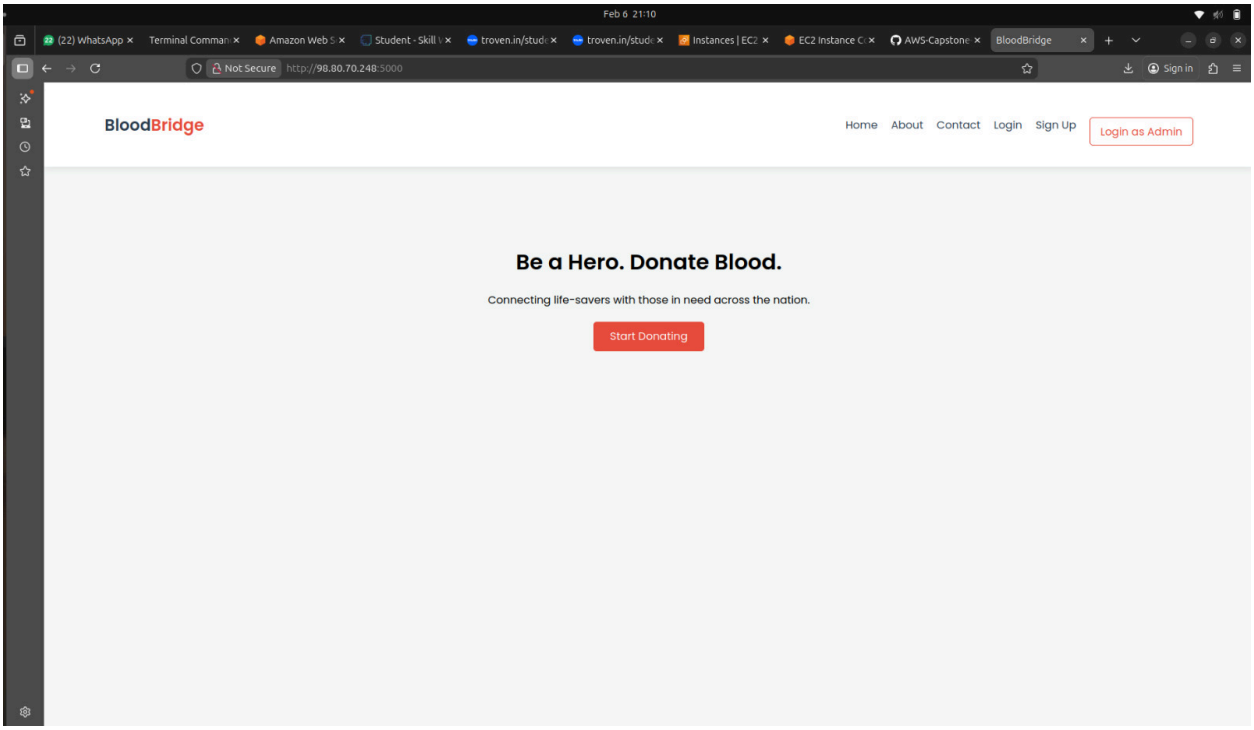
http://your-ec2-public-ip

- Run the Flask app on the EC2 instance

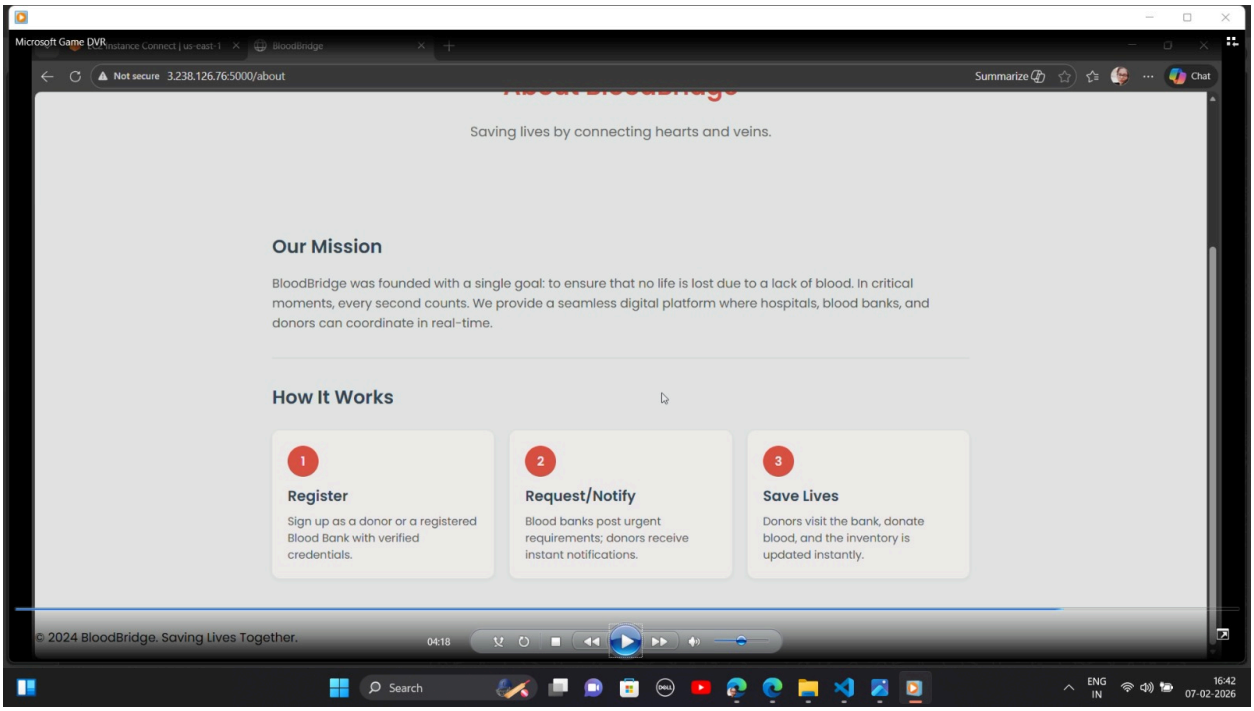


Functional Testing To Verify The Project

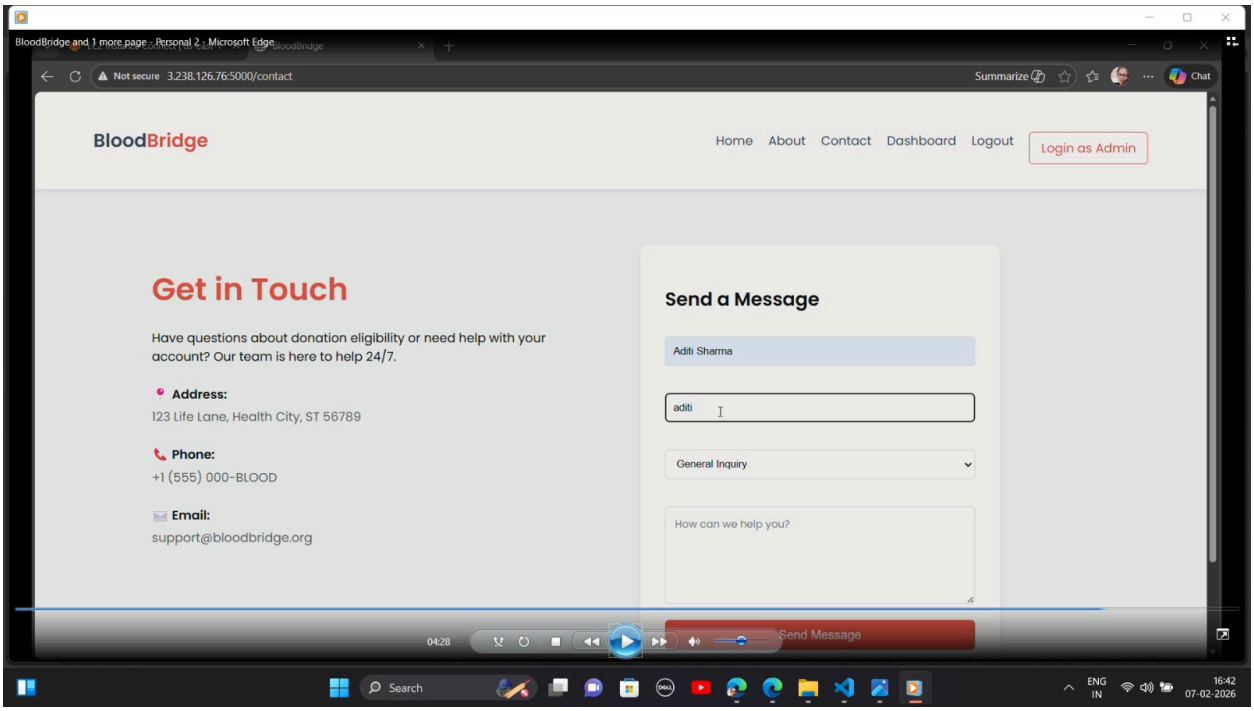
- **Home Page:**



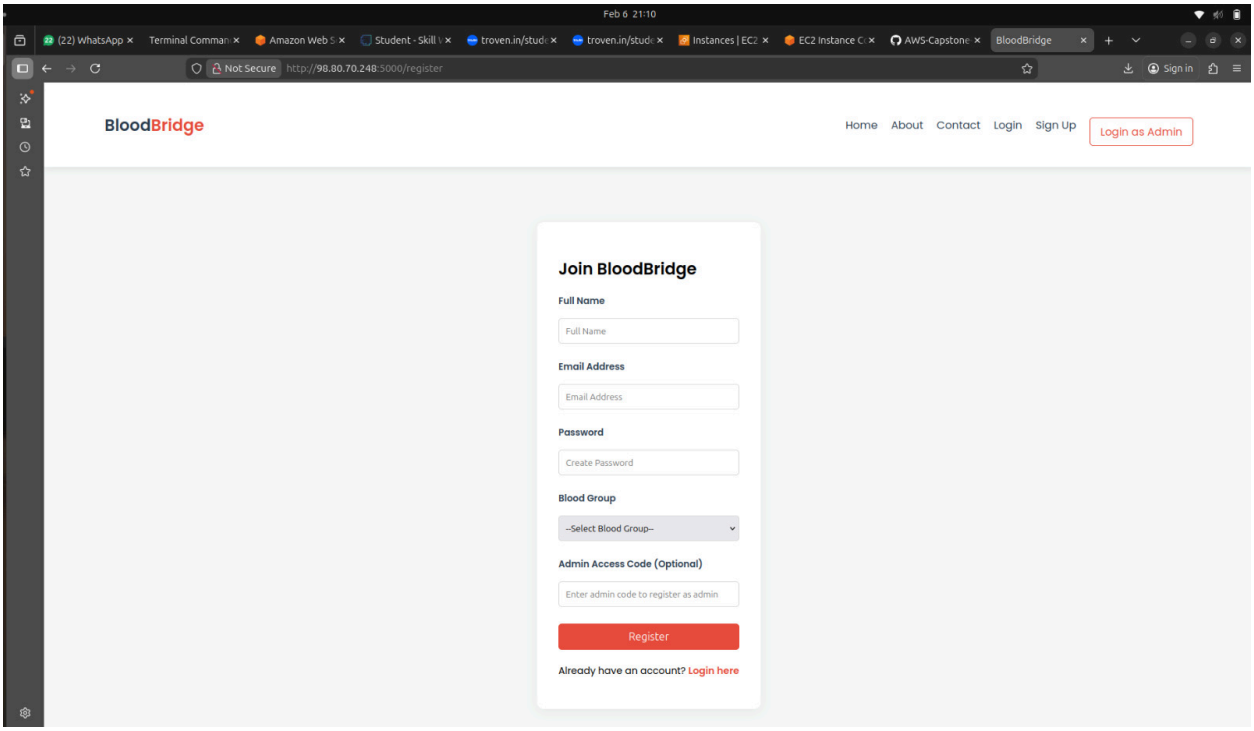
- **About Us page:**



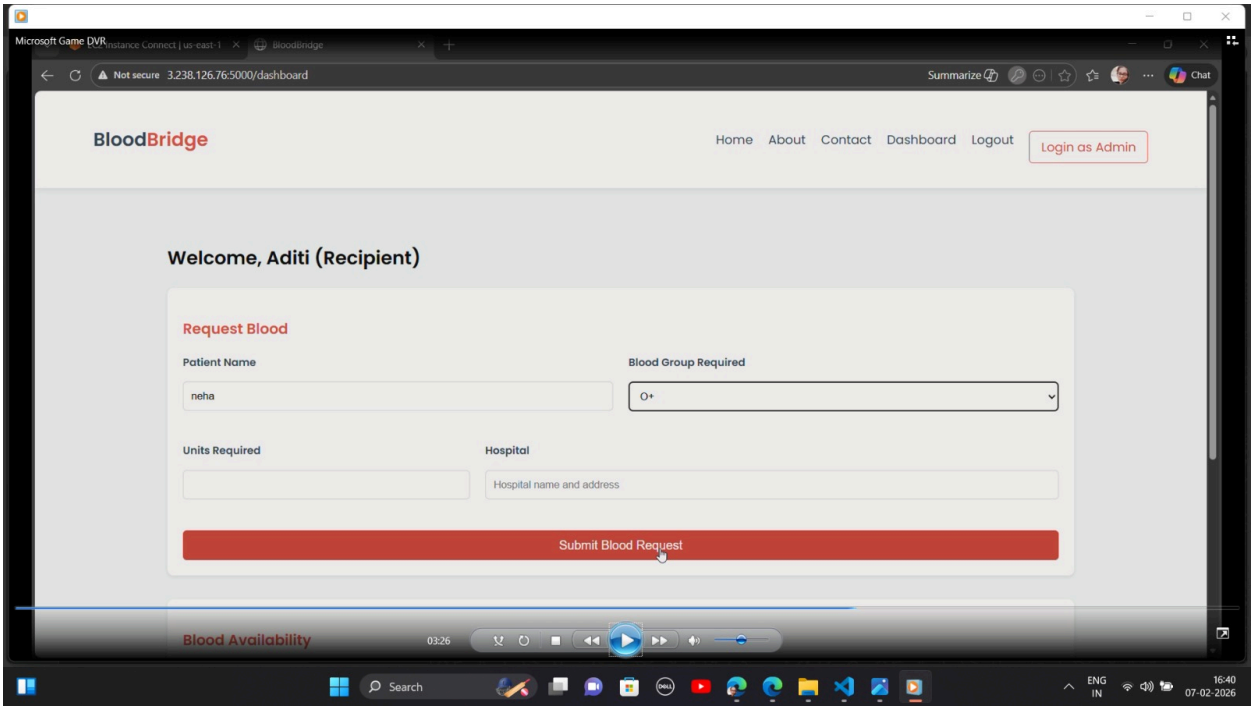
- **Contact Us page:**

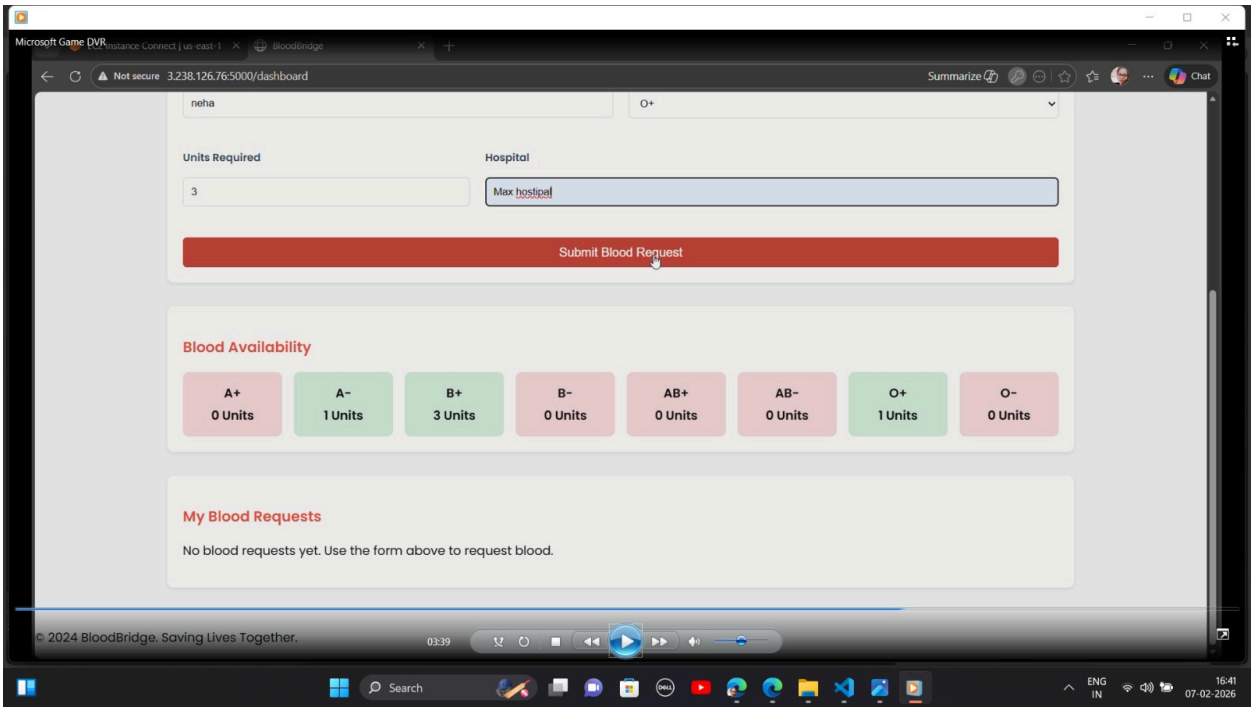


- **Register page:**



- **Recipient page:**





Conclusion

This document presents a comprehensive guide for designing, developing, and deploying the **BloodBridge** platform using AWS services. It highlights the critical steps involved in configuring **Amazon DynamoDB**, building a **Flask-based web application**, and deploying the system on an **EC2 instance**. By following these milestones and implementation activities, the platform enables efficient management of blood donation requests while ensuring reliability and security.

Utilizing **EC2** for application hosting and **DynamoDB** for database management provides a scalable, high-performance cloud infrastructure capable of adapting to increasing user demand. The integration of AWS services not only enhances system availability and fault tolerance but also supports future expansion and optimization. Overall, BloodBridge demonstrates how cloud computing can be effectively leveraged to build a robust, real-world healthcare support application.