MedTrack: AWS Cloud-Enabled Healthcare Management System

Project Description:

In today's fast-evolving healthcare landscape, efficient communication and coordination between doctors and patients are crucial. MedTrack is a cloud-based healthcare management system that streamlines patient doctor interactions by providing a centralized platform for booking appointments, managing medical histories, and enabling diagnosis submissions. To address these challenges, the project utilizes Flask for backend development, AWS EC2 for hosting, and DynamoDB for managing data. MedTrack allows patients to register, log in, book appointments, and submit diagnosis reports online. The system ensures real-time notifications, enhancing communication between doctors and patients regarding appointments and medical submissions. Additionally, AWS Identity and Access Management (IAM) is employed to ensure secure access control to AWS resources, allowing only authorized users to access sensitive data. This cloud-based solution improves accessibility and efficiency in healthcare services for all users.

Scenario 1: Efficient Appointment Booking System for Patients:

In the MedTrack system, AWS EC2 provides a reliable infrastructure to manage multiple patients accessing the platform simultaneously. For example, a patient can log in, navigate to the appointment booking page, and easily submit a request for an appointment. Flask handles backend operations, efficiently retrieving and processing user data in real-time. The cloud-based architecture allows the platform to handle a high volume of appointment requests during peak periods, ensuring smooth operation without delays.

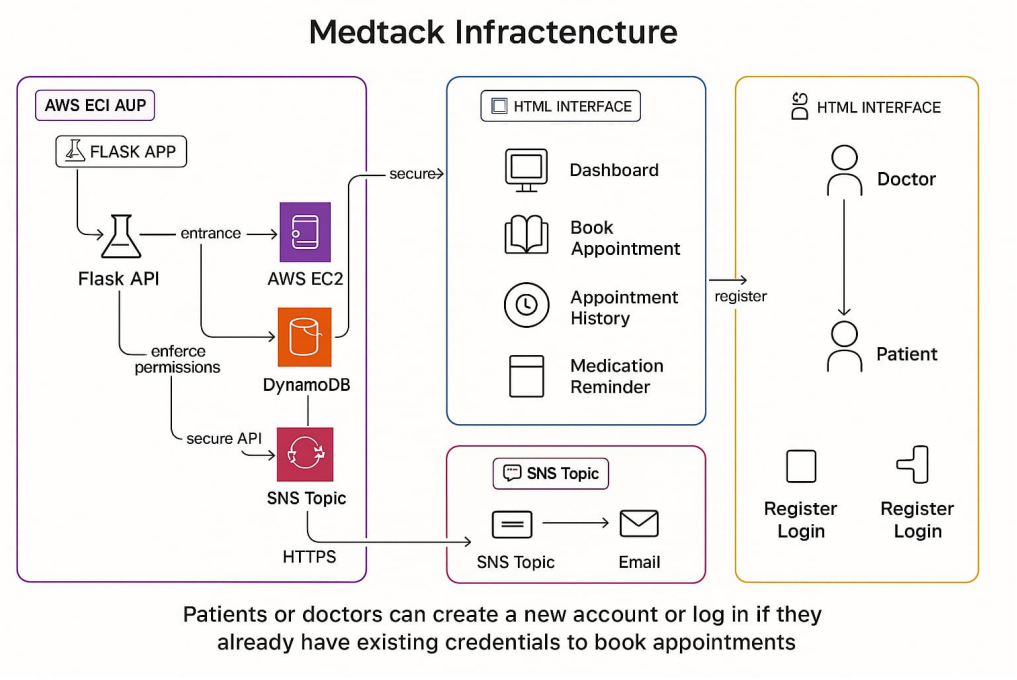
Scenario 2: Secure User Management with IAM:

MedTrack utilizes AWS IAM to manage user permissions and ensure secure access to the system. For instance, when a new patient registers, an IAM user is created with specific roles and permissions to access only the features relevant to them. Doctors have their own IAM configurations, allowing them access to patient records and appointment details while maintaining strict security protocols. This setup ensures that sensitive data is accessible only to authorized users.

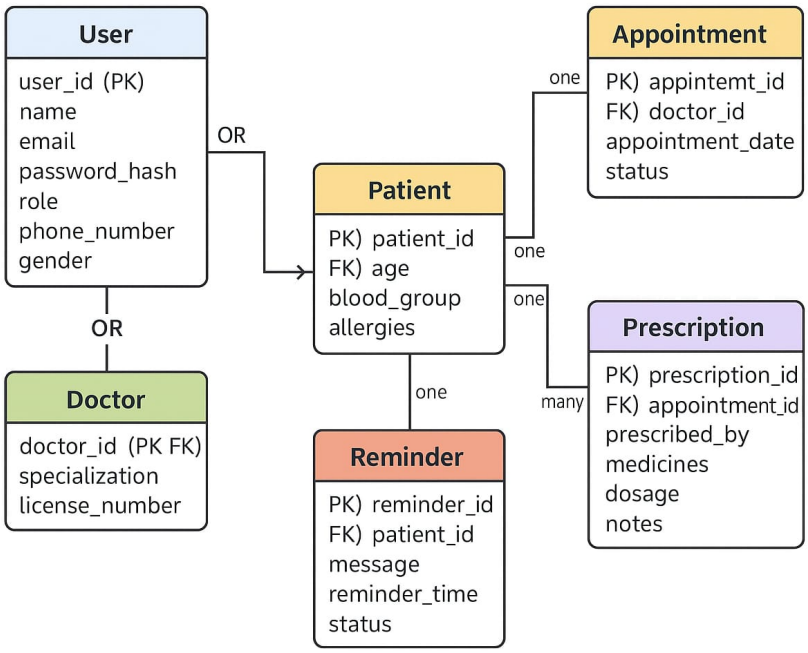
Scenario 3: Easy Access to Medical History and Resources:

The MedTrack system provides doctors and patients with easy access to medical histories and relevant resources. For example, a doctor logs in to view a patient's medical history and upcoming appointments. They can quickly access, and update records as needed. Flask manages real-time data fetching from DynamoDB, while EC2 hosting ensures the platform performs seamlessly even when multiple users access it simultaneously,offering a smooth and uninterrupted user experience.

AWS ARCHITECTURE:



ER DIAGRAM:



Pre-requisites:

1. .**AWS Account Setup**: AWSAccount Setup
2. **Understanding IAM**: IAMOverview
3. **Amazon EC2 Basics**: EC2Tutorial
4. **DynamoDB Basics**: DynamoDB Introduction
5. **SNS Overview**: SNSDocumentation
6. **Git Version Control**: Git Documentation

PROJECT WORK FLOW

1.AWS Account Setup and Login:

Activity1.1: SetupanAWSaccountifnotalreadydone**.**

Activity1.2: LogintotheAWSManagementConsole

2.DynamoDB Database Creation and Setup:

Activity2.1:CreateaDynamoDBTable.

Activity2.2:ConfigureAttributesforUserDataandBookRequests.

3.SNS Notification Setup:

Activity3.1:CreateSNStopicsforbookrequestnotifications.

Activity3.2:SubscribeusersandlibrarystafftoSNSemailnotifications.

4.Backend Development and Application Setup:

Activity4.1:DeveloptheBackendUsingFlask.

Activity4.2:IntegrateAWSServicesUsingboto3.

5.IAM Role Setup:

Activity5.1:CreateIAMRole

Activity5.2:AttachPolicies

6.EC2 Instance Setup

Activity6.1:LaunchanEC2instancetohosttheFlaskapplication.

Activity6.2:ConfiguresecuritygroupsforHTTP,andSSHaccess.

7.Deployment on EC2

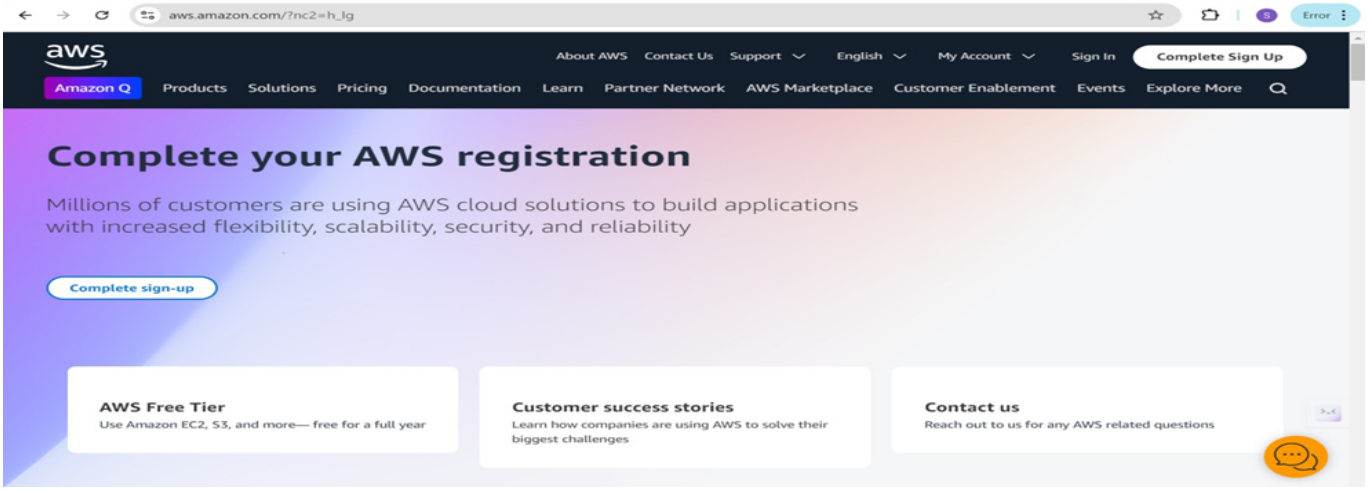
Activity7.1:UploadFlaskFiles

Activity7.2:RuntheFlaskApp

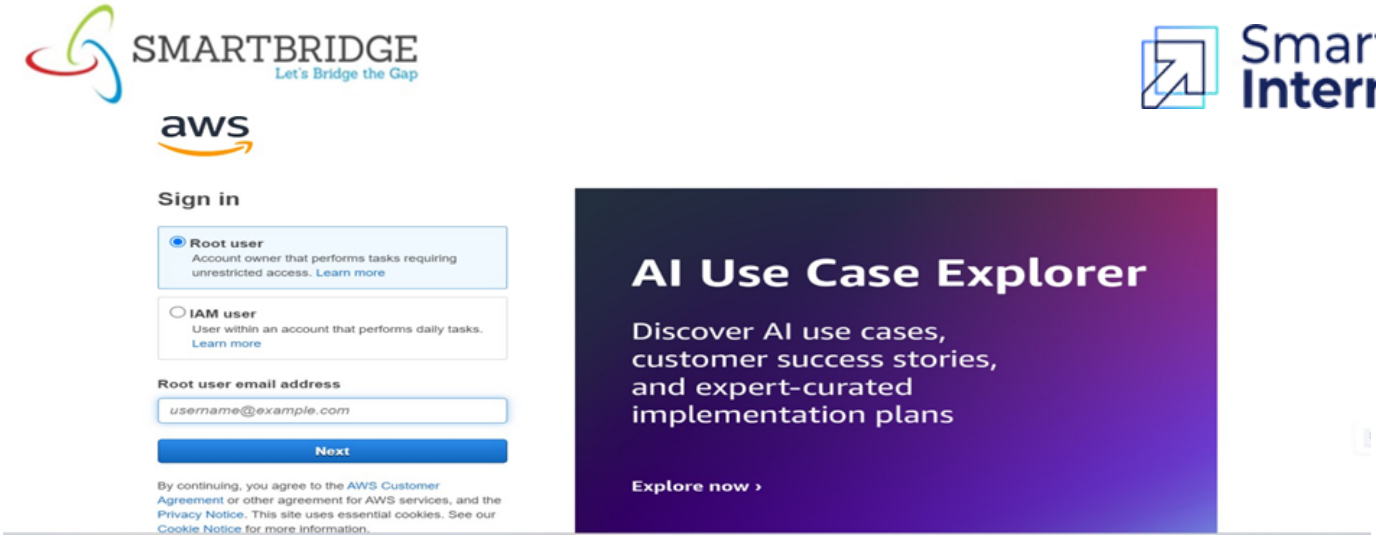
8. Testing and Deployment

Activity 8.1: Conduct functional testing to verify user registration, login, book requests.

Milestone 1: AWS AccountSetup and Login

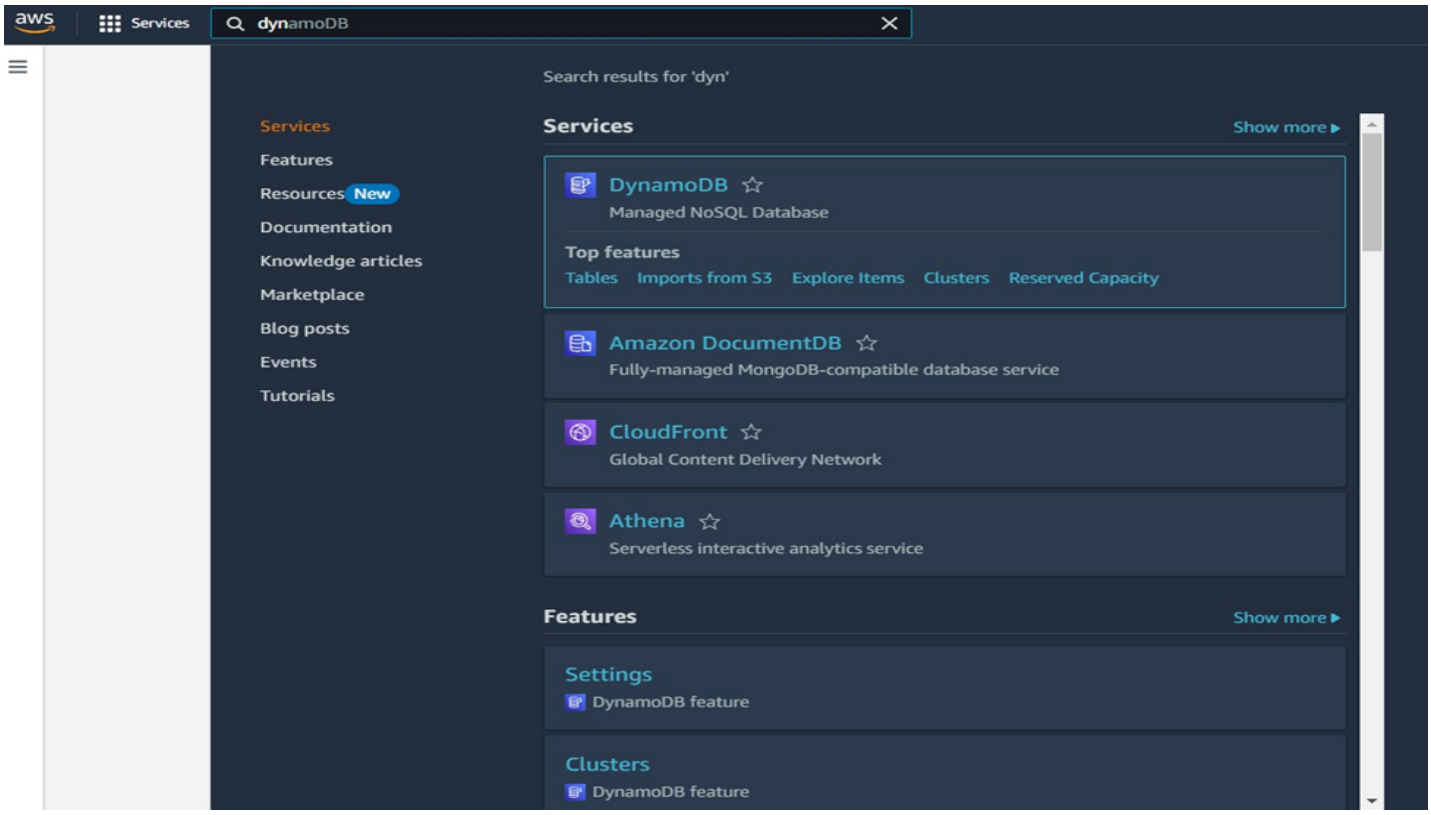
- **Activity 1.1: Set up an AWS account if not alreadydone.**
  - Sign up for an AWS account and configure billingsettings.
  - 

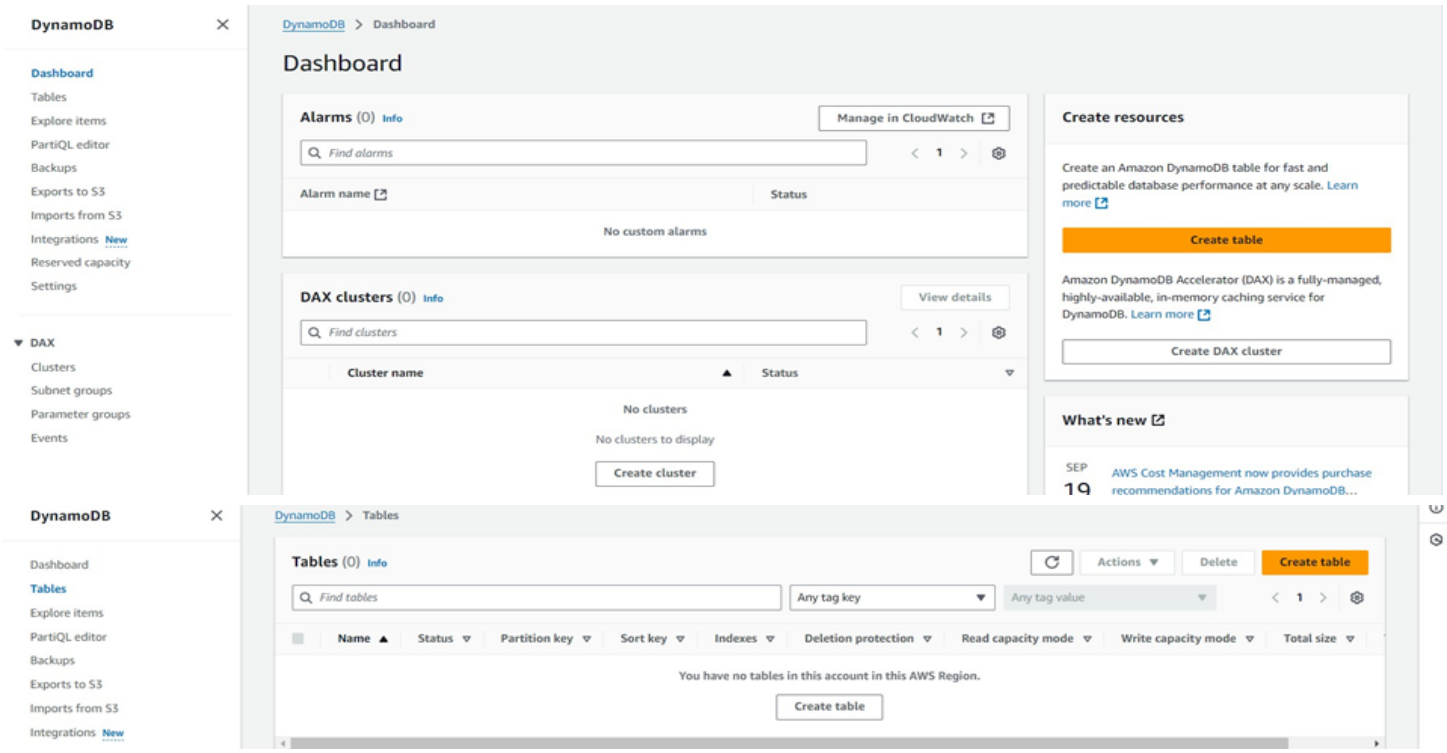
- **Activity 1.2: Log in to the AWS Management Console**
  - After setting up your account,log in to the AWS Management Console.



Milestone 2: DynamoDB Database Creation and Setup

- **Activity 2.1:Navigate to the DynamoDB**
  - In the AWS Console, navigate to DynamoDB and click on create tables





- **Activity 2.2: Create a DynamoDBtable for storingregistration details and book requests.**
  - Create Users table with partition key “Email” with type String and click on create tables.

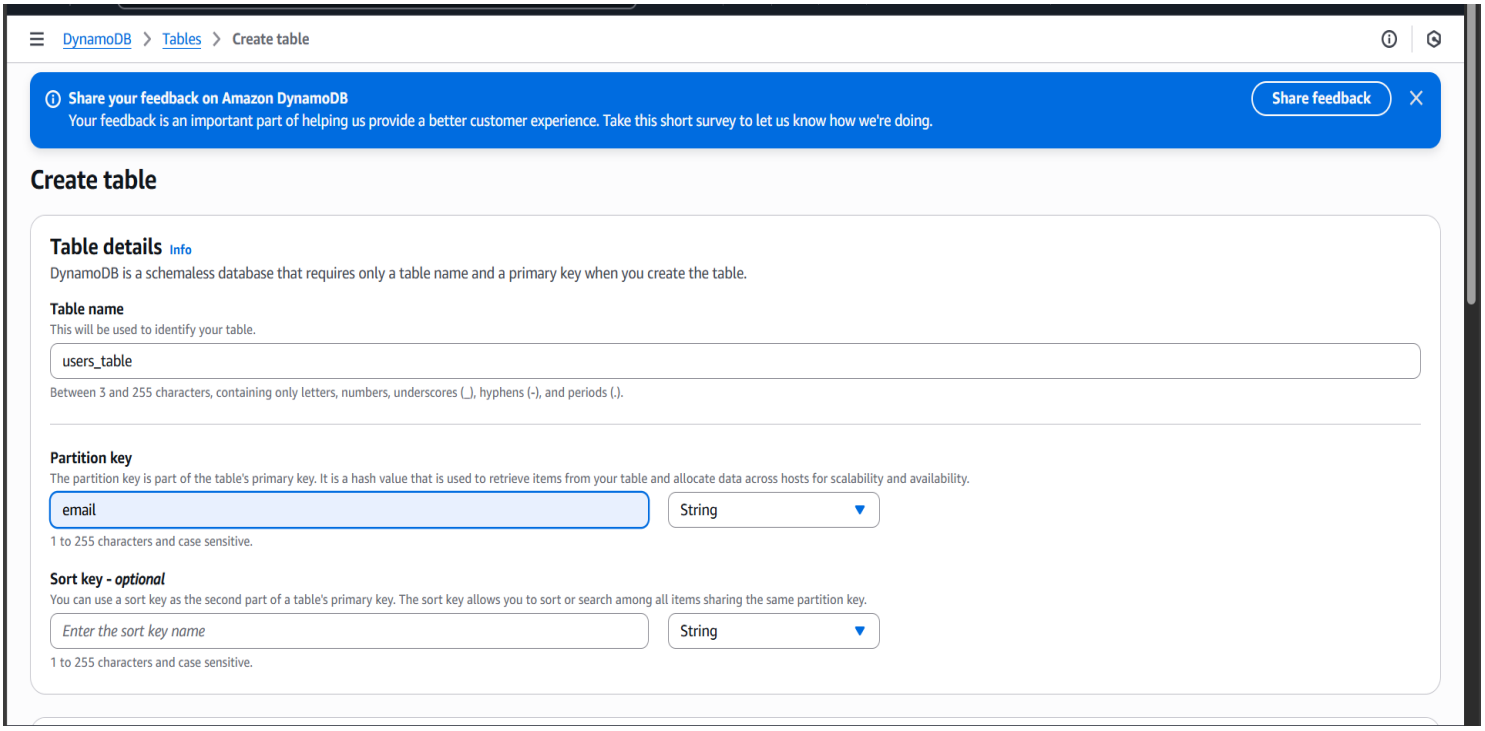


Table class	DynamoDB Standard	Yes
Capacity mode	Provisioned	Yes
Provisioned read capacity	5 RCU	Yes
Provisioned write capacity	5 WCU	Yes
Auto scaling	On	Yes
Local secondary indexes	-	No
Global secondary indexes	-	Yes
Encryption key management	Owned by Amazon DynamoDB	Yes
Deletion protection	Off	Yes
Resource-based policy	Not active	Yes

Tags

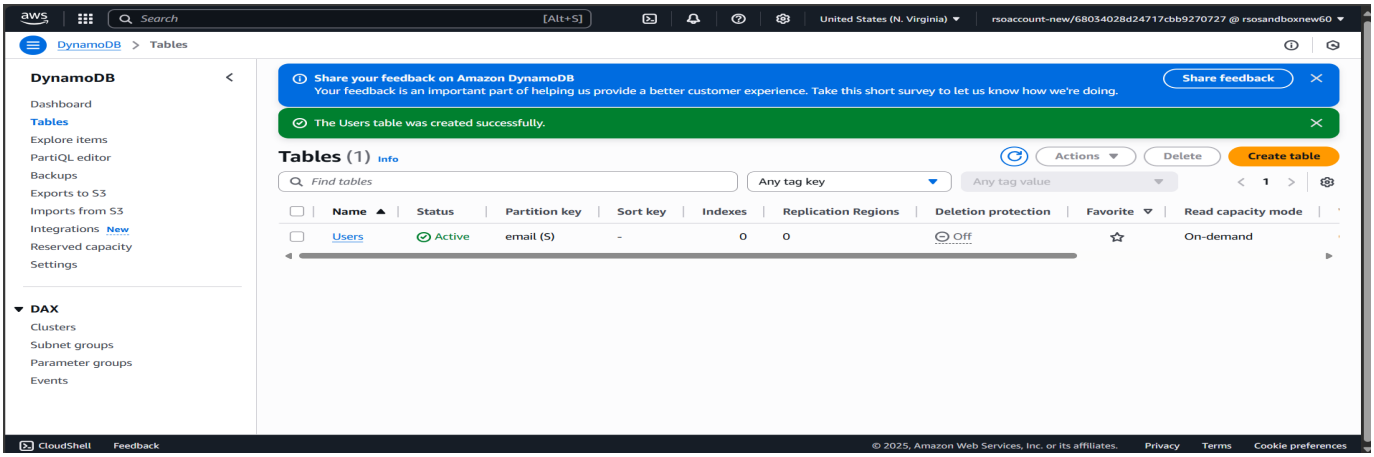
Tags are pairs of keys and optional values, that you can assign to AWS resources. You can use tags to control access to your resources or track your AWS spending.

No tags are associated with the resource.

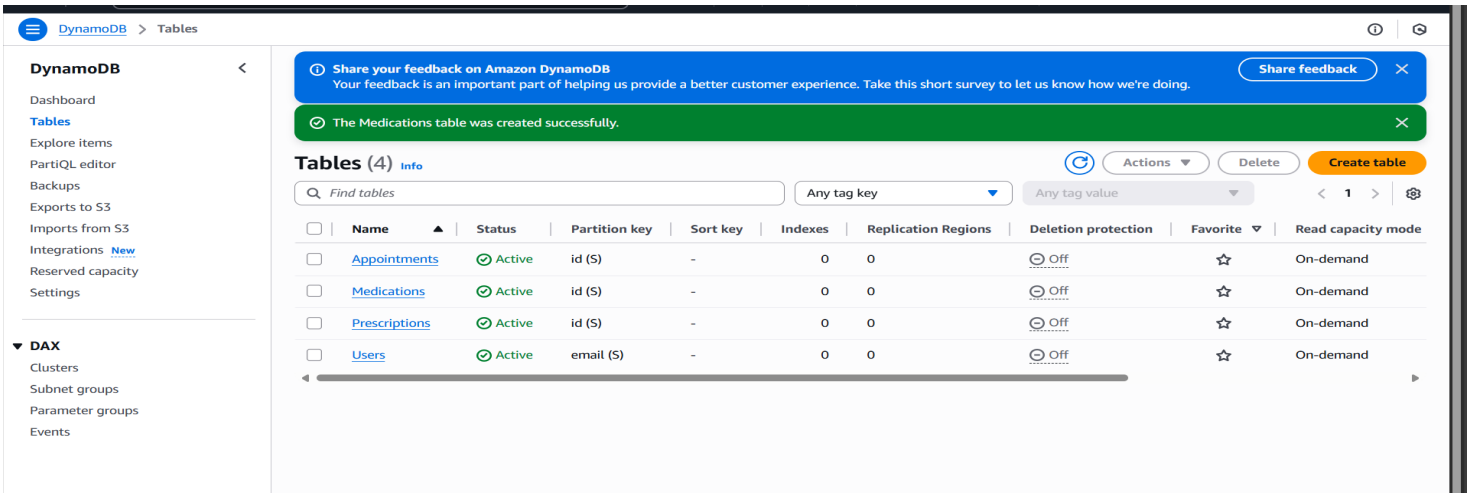
Add new tag

You can add 50 more tags.

Cancel Create table

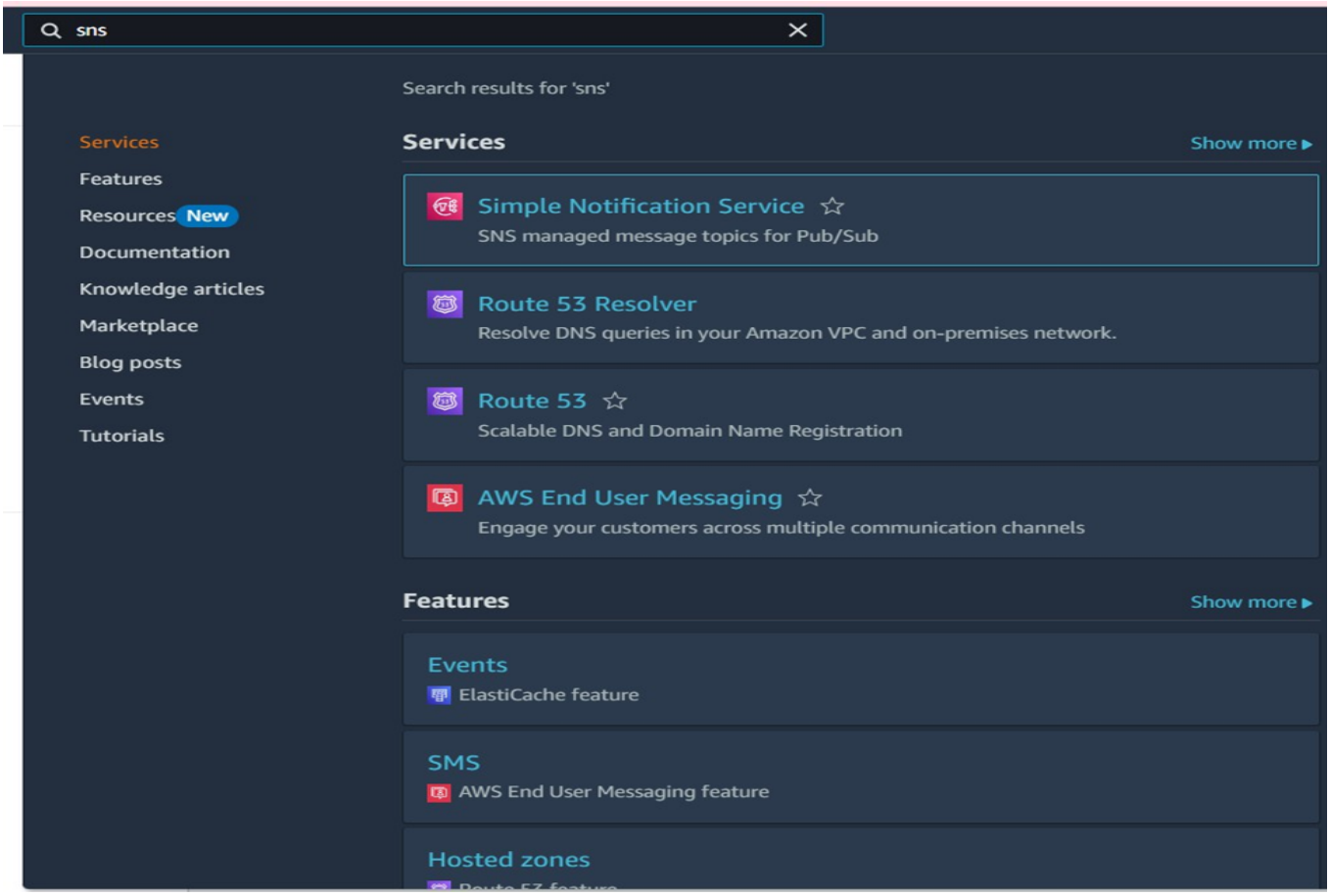


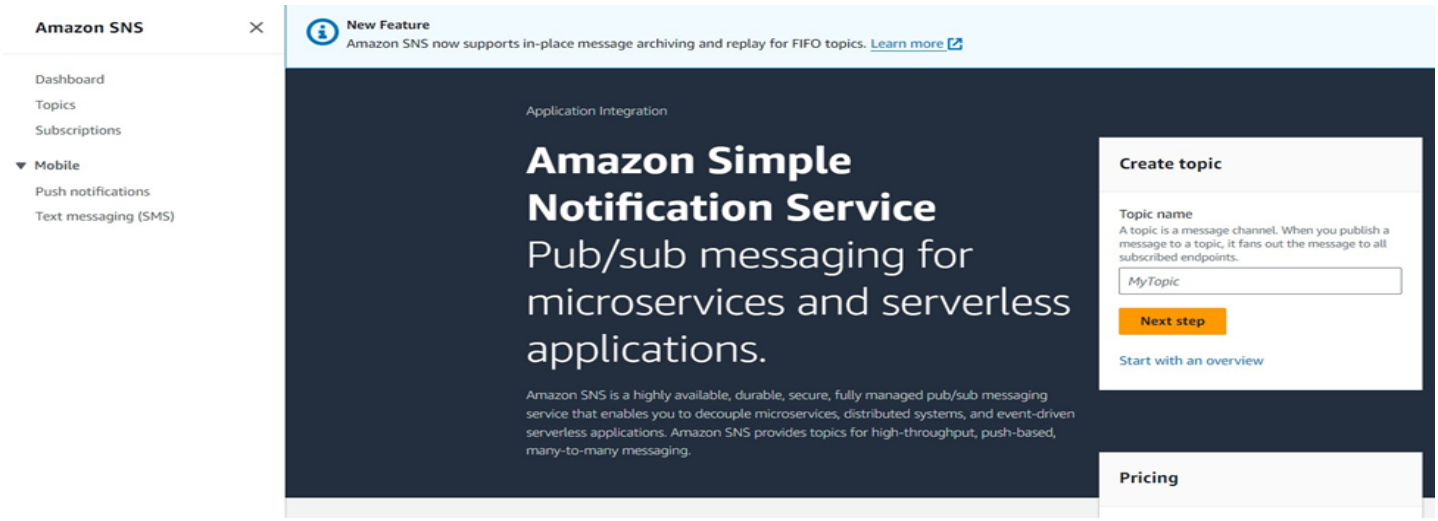
- Follow the same steps to create an Appointments,Prescriptions,Medications tablewith ID as the primary key for book requests data.



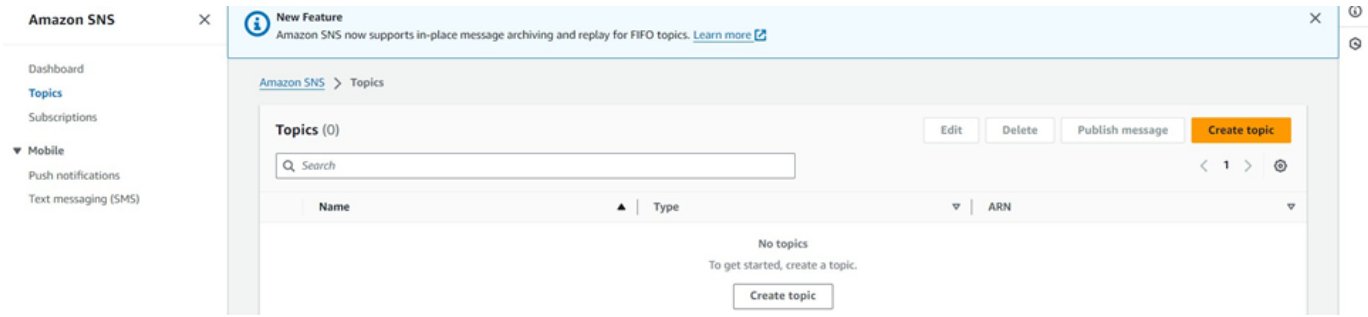
Milestone 3: SNS Notification Setup

- **Activity 3.1: Create SNS topics for sending email notifications to users and library staff.**
    - In the AWS Console,search for SNS and navigateto the SNS Dashboard.

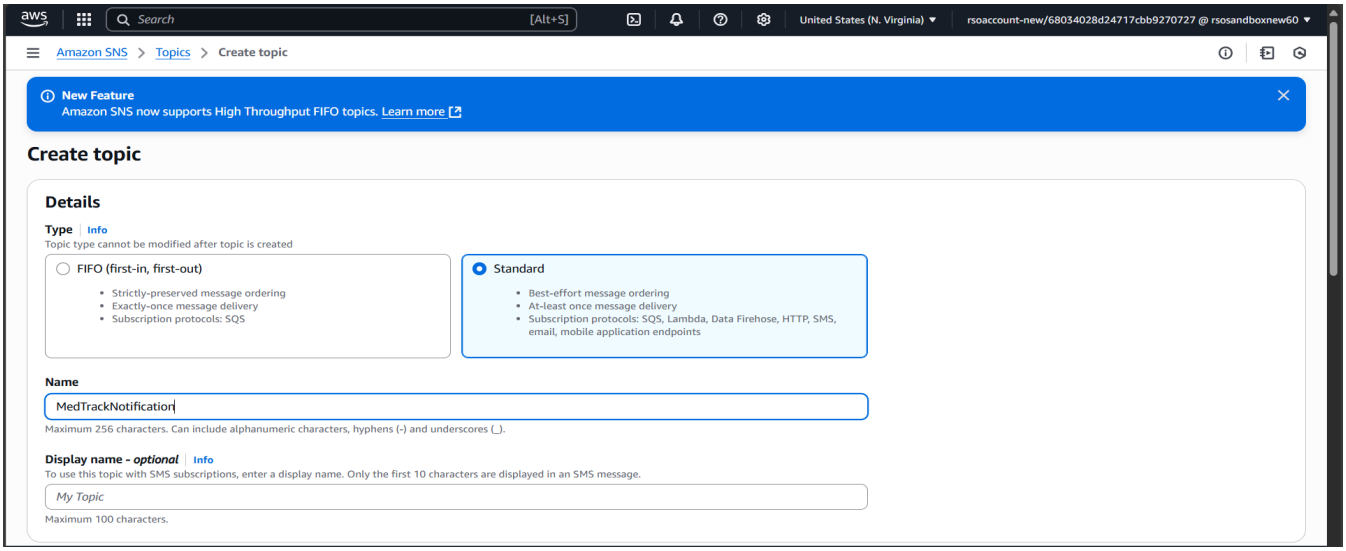


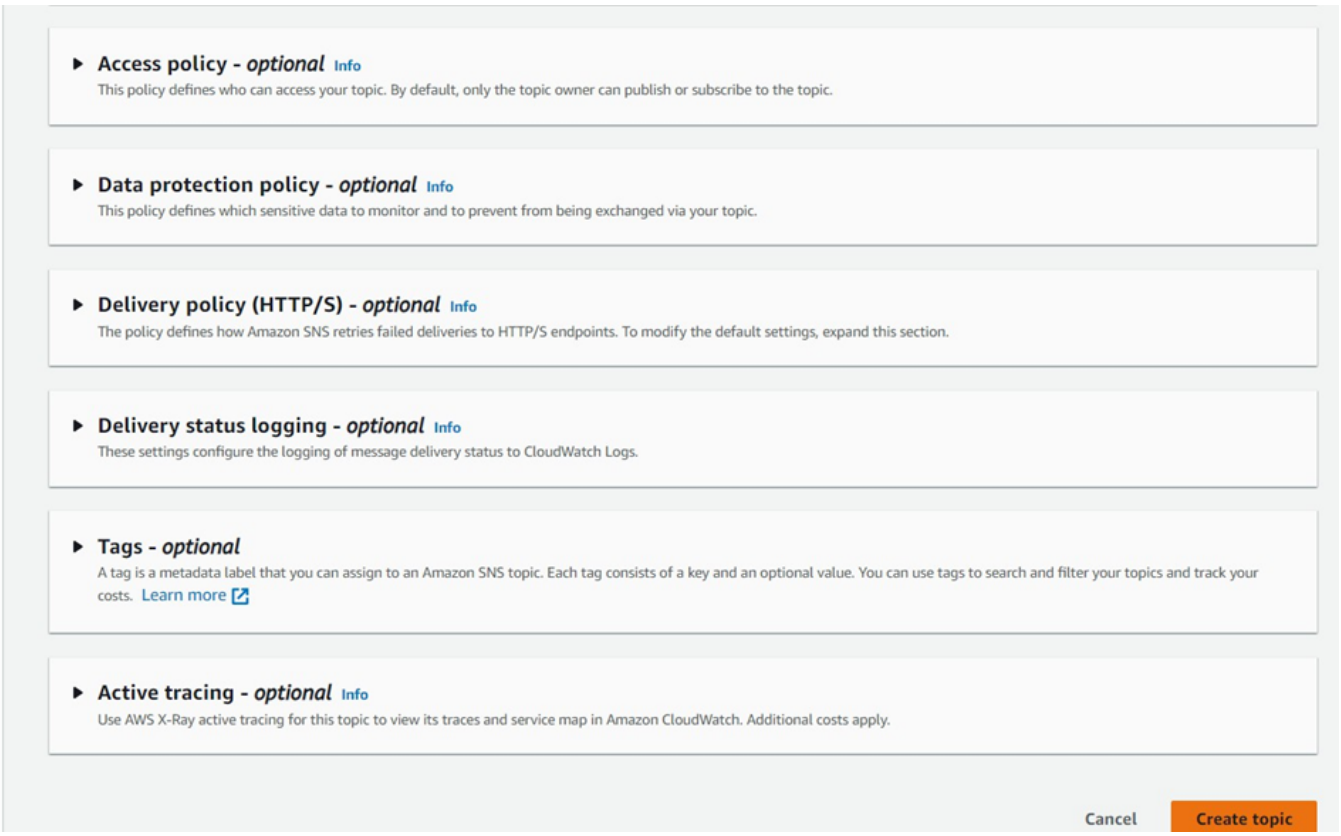


- Click on **Create Topic** and choose a name for the topic.

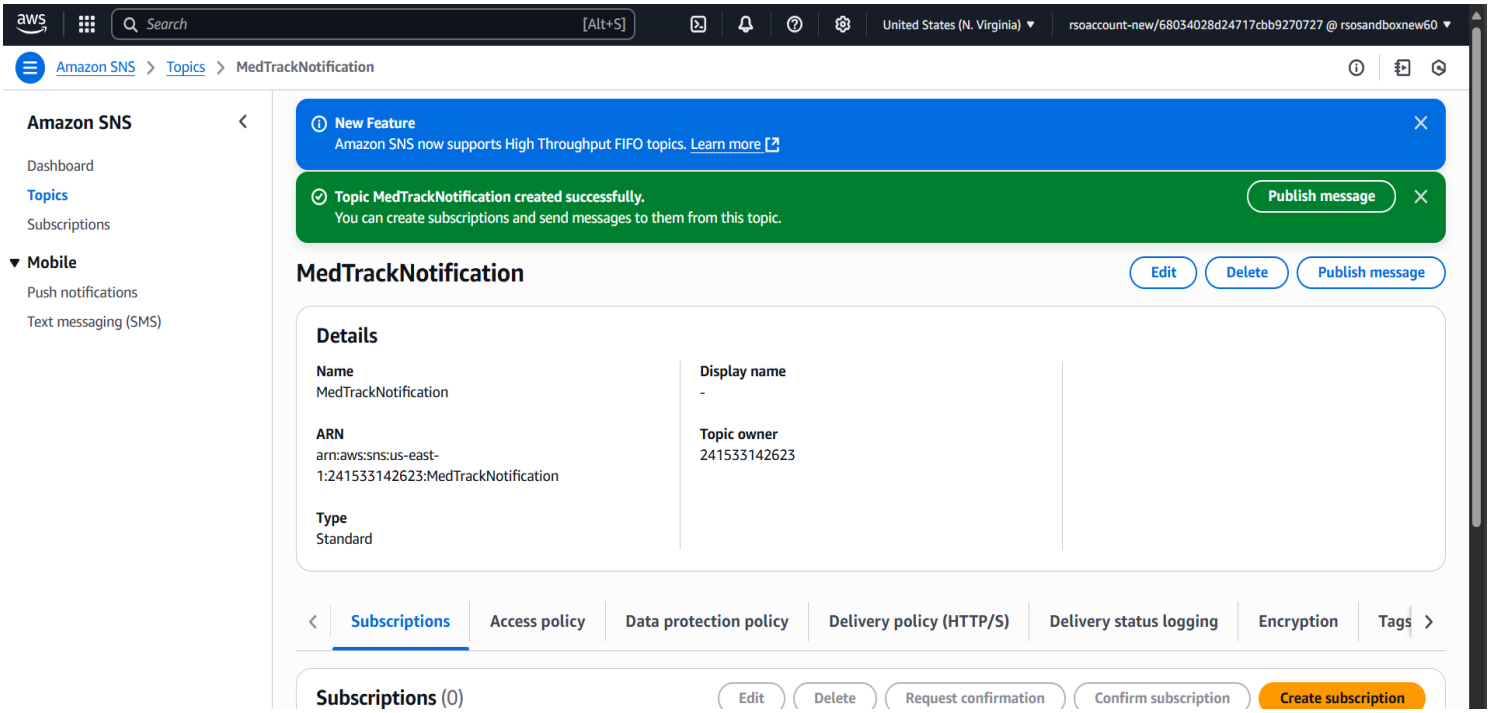


- Choose Standardtype for generalnotification use casesand Click on Create Topic.



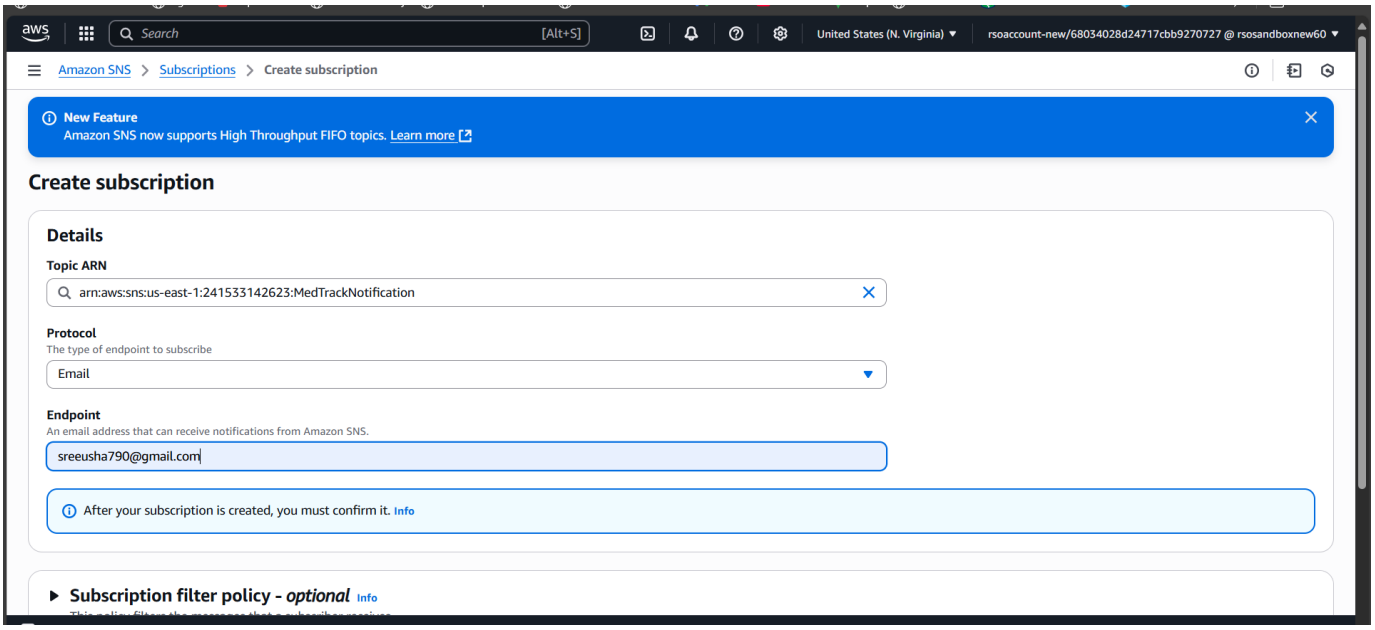


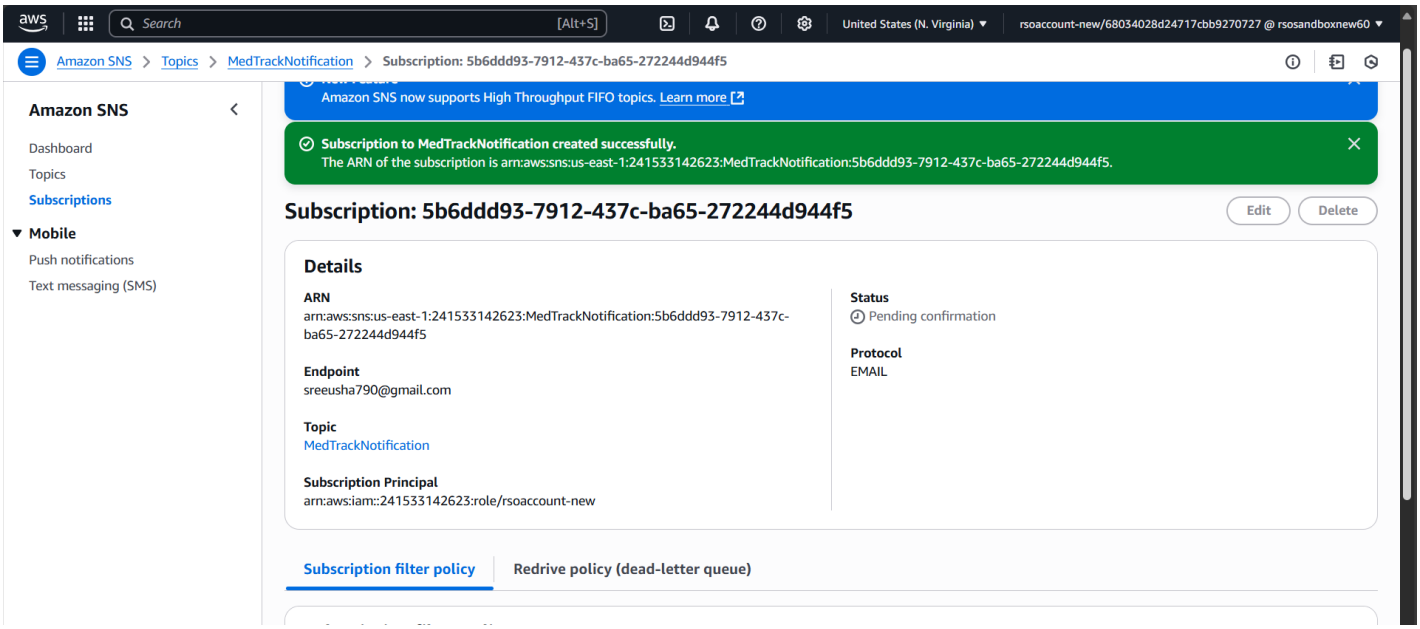
- Configure the SNS topic and note down the **Topic ARN**.



- **Activity 3.2: Subscribe users and staff to relevantSNS topics to receive real-time notifications when a book request is made.**

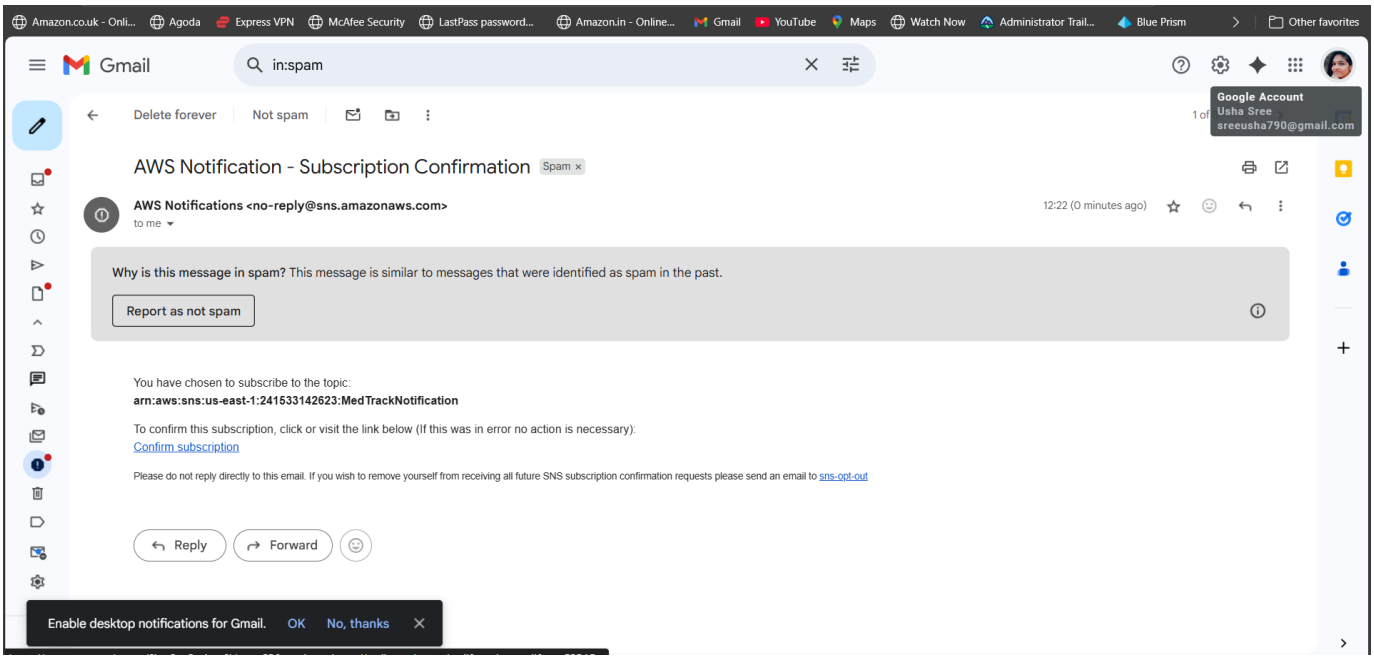
- Subscribe users (or admin staff) to this topic via Email. When a book request is made, notifications will be sent to the subscribed emails.

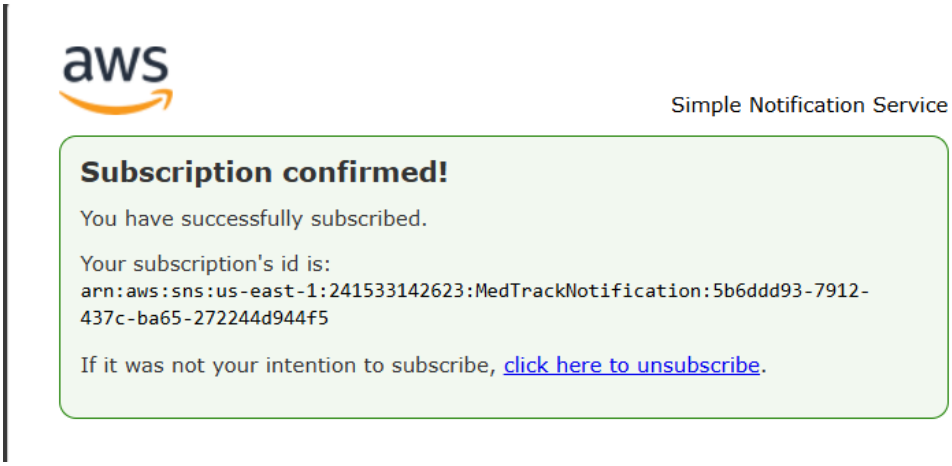




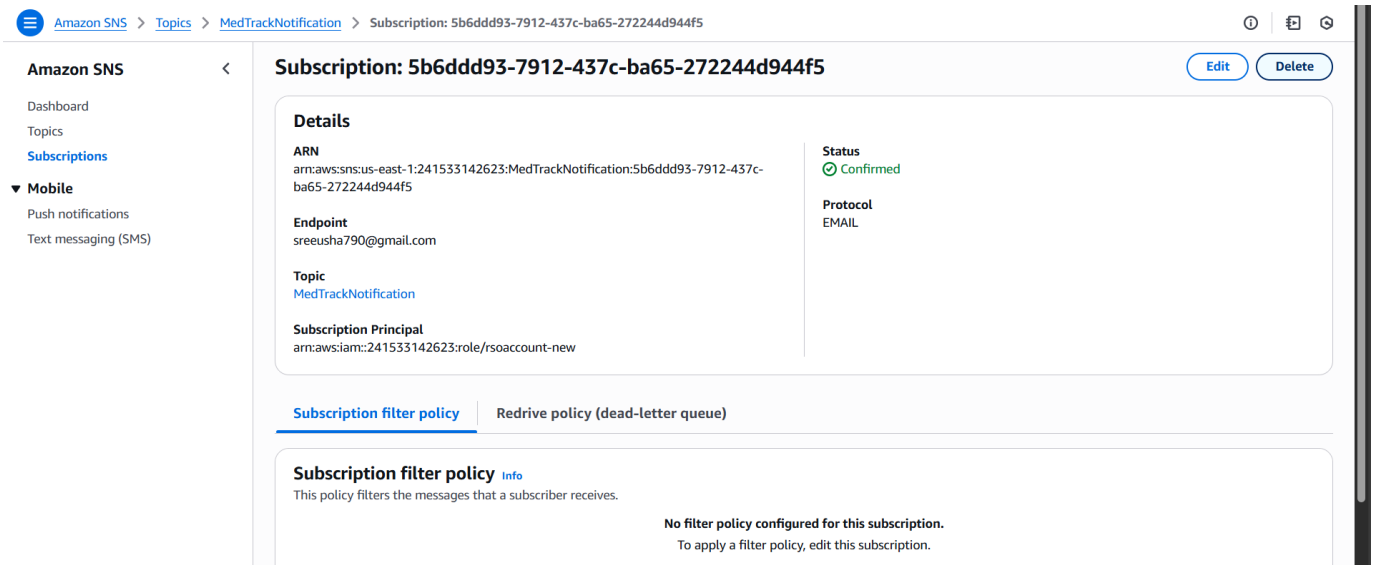
- After subscription requestfor the mail confirmation

- Navigate to the subscribed Emailaccount and Clickon the confirm subscription in the AWS Notification- Subscription Confirmation mail.



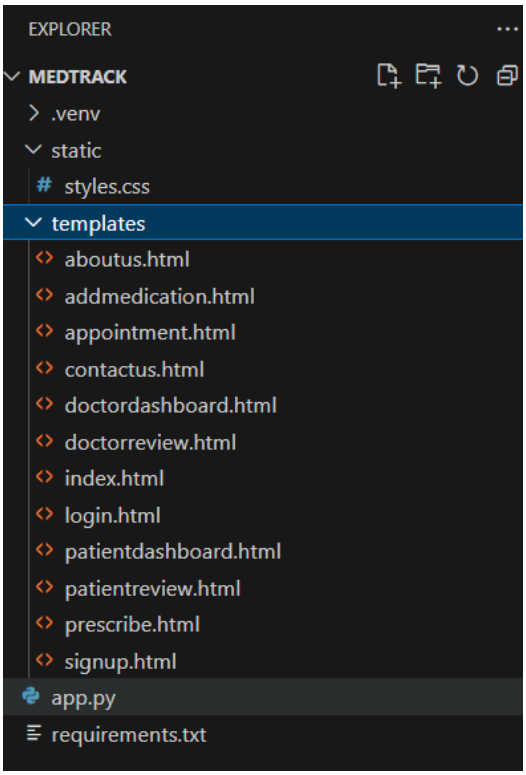


- Successfully done with the SNS mail subscription and setup, now store the ARN link



Milestone 4:Backend Development and Application Setup

- **Activity 4.1: Develop the backend using Flask**
  - File Explorer Structure



Description: set up the INSTANT LIBRARY project with an app.py file, a static/ folder for assets, and a templates/ directory containing all required HTML pages like index, login, signup, subject-specific pages (e.g., Doctordashboard.html, patientdashboard.html), and utility pages.

Description of the code :

- **Flask App Initialization**

```python
app.py > ...
from flask import Flask, render_template, request, redirect, url_for, session, flash
import boto3
from boto3.dynamodb.conditions import Key
import os
import uuid
from datetime import datetime
import smtplib
from email.mime.text import MIMEText

app = Flask(__name__)
app.secret_key = 'sreeusha790'

USE_DYNAMODB = False  # Set to True when deploying on AWS
AWS_REGION = 'us-east-1'
```

Description: This project begins by importing all essential libraries required for MedTrack. Flask utilities are used to handle routing, sessions, and template rendering. **Boto3** is used to interact with AWS **DynamoDB** for storing user, appointment, and prescription data. **SMTP** and the **email.mime** modules are utilized for sending custom email notifications to users. Additionally, libraries like uuid and datetime assist in generating unique IDs and managing date/time operations.

```python
app = Flask(__name__)
```

Description: initialize the Flask application instanceusing Flask(_name) to start building the web app.

- **Dynamodb Setup:**

```python
USE_DYNAMODB = True  # Set to True when deploying on AWS
AWS_REGION = 'us-east-1'

if USE_DYNAMODB:
    dynamodb = boto3.resource('dynamodb', region_name=AWS_REGION)
    users_table = dynamodb.Table('Users')
    appointments_table = dynamodb.Table('Appointments')
    prescriptions_table = dynamodb.Table('Prescriptions')
    medications_table = dynamodb.Table('Medications')
```

Description: Initialize the **DynamoDB** resource for the **us-east-1** region and set up access to the Users, Appointments, Prescriptions, and Medications tables. These tables are used to securely store user information, appointment bookings, doctor-issued prescriptions, and patient medication details for the MedTrack healthcare application.

- **SNS Connection:**

```python
AWS_REGION = "ap-south-1"
SNS_TOPIC_ARN = "arn:aws:sns:ap-south-1:123456789012:MedTrackNotifications"
sns = boto3.client('sns', region_name=AWS_REGION)
def send_sns_email(subject, message):
    try:
        sns.publish(
            TopicArn=SNS_TOPIC_ARN,
            Message=message,
            Subject=subject
        )
        print("📧 SNS email sent successfully.")
    except Exception as e:
        print(f"❌ Failed to send SNS email: {e}")
def send_local_sms(phone, message):
    print(f"[SIMULATED SMS to {phone}]: {message}")
def send_local_email(to_email, subject, body):
    try:
        smtp_server = 'smtp.gmail.com'
        smtp_port = 587
        sender_email = os.getenv('SENDER_EMAIL')
        sender_password = os.getenv('SENDER_PASSWORD')    # Use Gmail App Password (not regular password)
        msg = MIMEText(body)
        msg['Subject'] = subject
        msg['From'] = sender_email
        msg['To'] = to_email
        server = smtplib.SMTP(smtp_server, smtp_port)
        server.starttls()
        server.login(sender_email, sender_password)
        server.send_message(msg)
        server.quit()
        print(f"Email sent to {to_email}")
    except Exception as e:
        print(f"Email sending failed: {e}")
```

Description:

Configure **Amazon SNS** with your Topic ARN and region (e.g., us-east-1) to send medical alerts. Set up **SMTP** (like Gmail) in SENDER_EMAIL and SENDER_PASSWORD using an app password to send local email notifications for reminders and signups.

- **Routes for Web Pages**

- Home Route:

  Description: Defines the home route `/` to render the **index page**, which contains the **Login** and **Sign Up** buttons for both patients and doctors.

```python
@app.route('/')
def index():
    return render_template('index.html')
```

- Sign Up Route:

  Description: Defines `/signup` route to **collect user details** (name, email, password, phone, role, and reminder frequency), **store them in DynamoDB or local DB**, and **send an SNS Welcome email** upon successful registration.

```python
@app.route('/signup', methods=['GET', 'POST'])
def signup():
    if request.method == 'POST':
        data = {k: request.form[k] for k in ['name', 'email', 'password', 'phone', 'role', 'reminder']}
        users = load_users()
        if data['email'] in users:
            flash("Email already exists", "error")
            return redirect(url_for('signup'))
        db['users'][data['email']] = data

        subject = "🎉 Welcome to MedTrack!"
        message = f"Hi {data['name']},\n\nYou have successfully registered in the MedTrack app.\n\nStay healthy!"
        send_sns_email(subject, message)

        flash("Signup successful! Now login.", "success")
        return redirect(url_for('login'))
    return render_template('signup.html')
```

- Login Route (GET/POST):

Description: Defines `/login` route to validate login credentials against the **user database**, initiate a session, and **redirect doctors to `doctordashboard`** or **patients to `patientdashboard`** based on their role.

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email, password = request.form['email'], request.form['password']
        user = db['users'].get(email)
        if user and user['password'] == password:
            session.update({"email": email, "role": user['role'], "name": user['name'], "phone": user.get('phone', '')})
            return redirect(url_for(f"{user['role']}dashboard"))
        flash("Invalid credentials", "error")
    return render_template('login.html')
```

- Doctor Dashboard Route:

Description: Defines `/doctordashboard` route where doctors can **view, approve, or reject appointments**, and **prescribe medications**. Also displays **alerts for missed medications** by patients.

```python
@app.route('/doctordashboard', methods=['GET', 'POST'])
def doctordashboard():
    if session.get('role') != 'doctor':
        return redirect(url_for('login'))

    name = session['name']
    doctor_appts = [a for a in db['appointments'] if a['doctor'].strip().lower() == name.strip().lower()]

    now = datetime.now()
    for appt in doctor_appts:
        appt_time_str = f"{appt['date']} {appt['time']}"
        try:
            appt_time = datetime.strptime(appt_time_str, "%Y-%m-%d %H:%M")
        except ValueError:
            continue
        if appt['status'] == 'Pending' and now > appt_time:
            appt['status'] = 'Missed'

    alerts = []
    for m in db['medications']:
        if m['status'].lower() == 'missed':
            alerts.append(f"Patient {m['patient']} missed dose of {m['medicine']} at {m['time']}")

    if request.method == 'POST':
        decision = request.form['decision']
        appt_id = request.form['appointment_id']
        for a in db['appointments']:
            if a['id'] == appt_id:
                a['status'] = decision
                break
        flash(f"Appointment {decision.lower()}ed.", "info")

    return render_template('doctordashboard.html',
                           name=name,
                           email=session['email'],
                           appointments=doctor_appts,
```

- **Patient Dashboard Route:**

  Description: Defines `/patientdashboard` route where patients can **view appointments, prescriptions, and medication schedules**, and **mark doses as Taken, Missed, or Pending**

```python
@app.route('/patientdashboard')
def patientdashboard():
    if session.get('role') != 'patient':
        return redirect(url_for('login'))

    name = session.get('name')
    email = session.get('email')

    user_appointments = [appt for appt in db['appointments'] if appt['patient'] == name]

    # Convert date + time to datetime object and check if it's passed
    now = datetime.now()
    for appt in user_appointments:
        appt_time_str = f"{appt['date']} {appt['time']}"
        try:
            appt_time = datetime.strptime(appt_time_str, "%Y-%m-%d %H:%M")
        except ValueError:
            appt_time = now  # fallback if formatting is bad

        if appt['status'] == 'Pending' and now > appt_time:
            appt['status'] = 'Missed'

    user_medications = [med for med in db['medications'] if med['patient'] == name]
    user_prescriptions = [p for p in load_prescriptions() if p['patient'] == name]

    alerts = []
    for med in user_medications:
        med_time_str = f"{med.get('date', now.strftime('%Y-%m-%d'))} {med['time'].split()[0]}"

        try:
            med_time = datetime.strptime(med_time_str, "%Y-%m-%d %H:%M")
        except:
            continue
        if med['status'] == 'Pending' and now > med_time:
            med['status'] = 'Missed'
```

- **Appointment Booking Route:**

  Description: Defines `/appointment` route to **allow patients to book appointments** by submitting required details. Bookings are saved in **DynamoDB or local DB** and shown on dashboards.

```python
@app.route('/appointment', methods=['GET', 'POST'])
def appointment():
    if request.method == 'POST':
        f = request.form
        appt = {
            'id': str(uuid.uuid4()), 'patient': session.get('name'), 'name': f['name'], 'gender': f['gender'],
            'phone': f['phone'], 'age': f['age'], 'email': session.get('email'),
            'department': f['department'], 'problem': f['problem'], 'doctor': f['doctor'],
            'status': 'Pending', 'date': f['date'], 'time': f['time']
        }
        save_appointment(appt)
        flash("Appointment booked successfully!", "success")
        return redirect(url_for('appointment'))
    return render_template('appointment.html')
```

- **Prescription Route:**

Description: Defines `/prescribe/<appt_id>` route for doctors to **add prescriptions linked to a specific appointment**. Medications are saved and reminders are scheduled.

```python
@app.route('/prescribe/<appt_id>', methods=['GET', 'POST'])
def prescribe(appt_id):
    if request.method == 'POST':
        f = request.form
        time = f"{f['time']} {f['ampm']}" if 'ampm' in f else f['time']
        presc = {
            'id': str(uuid.uuid4()), 'appointment_id': appt_id, 'medicine': f['medicine'],
            'dosage': f['dosage'], 'days_left': f['days_left'], 'time': time,
            'status': 'Pending', 'patient': f['patient']
        }
        save_prescription(presc)
        save_medication(presc)
        flash("Prescription added & medication reminder set.", "success")
        return redirect(url_for('doctordashboard'))
    appt = next((a for a in db['appointments'] if a['id'] == appt_id), None)
    return render_template('prescribe.html', appointment=appt)
```

- **Add Medication Route:**

Description: Defines `/addmedication` route for patients to **manually add a medication schedule**, which is later used for reminders.

```python
@app.route('/addmedication', methods=['GET', 'POST'])
def addmedication():
    if session.get('role') != 'patient': return redirect(url_for('login'))
    if request.method == 'POST':
        f = request.form
        med = {
            'id': str(uuid.uuid4()), 'patient': session['name'], 'medicine': f['medicine'],'date': f['date'],
            'dosage': f['dosage'], 'time': f['time'], 'days_left': f['days_left'], 'status': f.get('status', 'Pending')
        }
        save_medication(med)
        flash("Medication added successfully.", "success")
        return redirect(url_for('patientdashboard'))
    return render_template('addmedication.html')
```

- **Notification Route:**

Description: Defines `/send_notifications` route to **send SNS notifications** to patients for scheduled medications based on the current date and time.

```python
@app.route('/send_notifications')
def send_notifications():
    now = datetime.now()
    current_time = now.strftime("%H:%M")
    today = now.strftime("%Y-%m-%d")
    reminders_sent = 0

    for med in db['medications']:
        med_time = datetime.strptime(med['time'], "%I:%M %p").strftime("%H:%M")
        med_date = med.get('date', today)  # Optional date field
        if med['status'].lower() == 'pending' and med_time == current_time and med_date == today:
            # Send SNS email reminder
            subject = f"💊 Medication Reminder: {med['medicine']}"
            message = f"Dear {med['patient']},\n\nThis is your MedTrack reminder to take {med['medicine']} at {med['time']} today.\n\nStay heal
            send_sns_email(subject, message)

            med['status'] = 'Sent'  # Prevent duplicate notifications
            reminders_sent += 1

    return f"{reminders_sent} reminder(s) sent."
```

- **Update Medication Status Route:**

Description: Defines `/update_medication_status` route to **allow patients to update the status** of their medications—Taken, Missed, or Pending.

```python
@app.route('/update_medication_status', methods=['POST'])
def update_medication_status():

    if session.get('role') != 'patient':
        return redirect(url_for('login'))

    for med in db['medications']:
        form_key = f"status_{med['id']}"
        if form_key in request.form:
            new_status = request.form[form_key]
            current_time = datetime.now().strftime("%H:%M")
            if new_status == "Not Yet" and current_time > med['time']:
                med['status'] = "Missed"
            else:
                med['status'] = "Completed" if new_status == "Taken" else "Pending"

    flash("Medication status updated.", "success")
    return redirect(url_for('patientdashboard'))
```

- **Profile Update Route:**

  Description: Defines `/update-profile` route to let **users update their name, email, and phone**, which is reflected in the session and user table.

```python
@app.route('/update-profile', methods=['POST'])
def update_profile():
    if 'email' not in session or 'role' not in session:
        return redirect(url_for('login'))

    email = session['email']
    role = session['role']

    # Support for local or DynamoDB users
    users = load_users()

    if email not in users:
        flash("User not found", "error")
        return redirect(url_for(f'{role}dashboard'))

    user = users[email]
    user['name'] = request.form['name']
    user['email'] = request.form['email']
    user['phone'] = request.form['phone']

    # Update session values
    session['name'] = user['name']
    session['email'] = user['email']
    session['phone'] = user['phone']

    if not USE_DYNAMODB:
        db['users'][email] = user
    else:
        users_table.put_item(Item=user)

    flash("Profile updated successfully.", "success")
    return redirect(url_for(f'{role}dashboard'))
```

- **Static Info Routes:**

`/aboutus` – Displays About Us page.
`/contactus` – Sends a confirmation message after user submits a contact form.

```python
@app.route('/aboutus')
def aboutus():
    return render_template('aboutus.html')

@app.route('/contactus', methods=['GET', 'POST'])
def contactus():
    if request.method == 'POST':
        flash("Message sent successfully. Thank you for contacting MedTrack!", 'success')
        return redirect(url_for('contactus'))
    return render_template('contactus.html')
```

- **Logout Route:**

Description: Defines `/logout` to **clear session data** and safely redirect the user to the login page.

```python
@app.route('/logout')
def logout():
    session.clear()
    flash("You have been logged out.", "info")
    return redirect(url_for('login'))
```

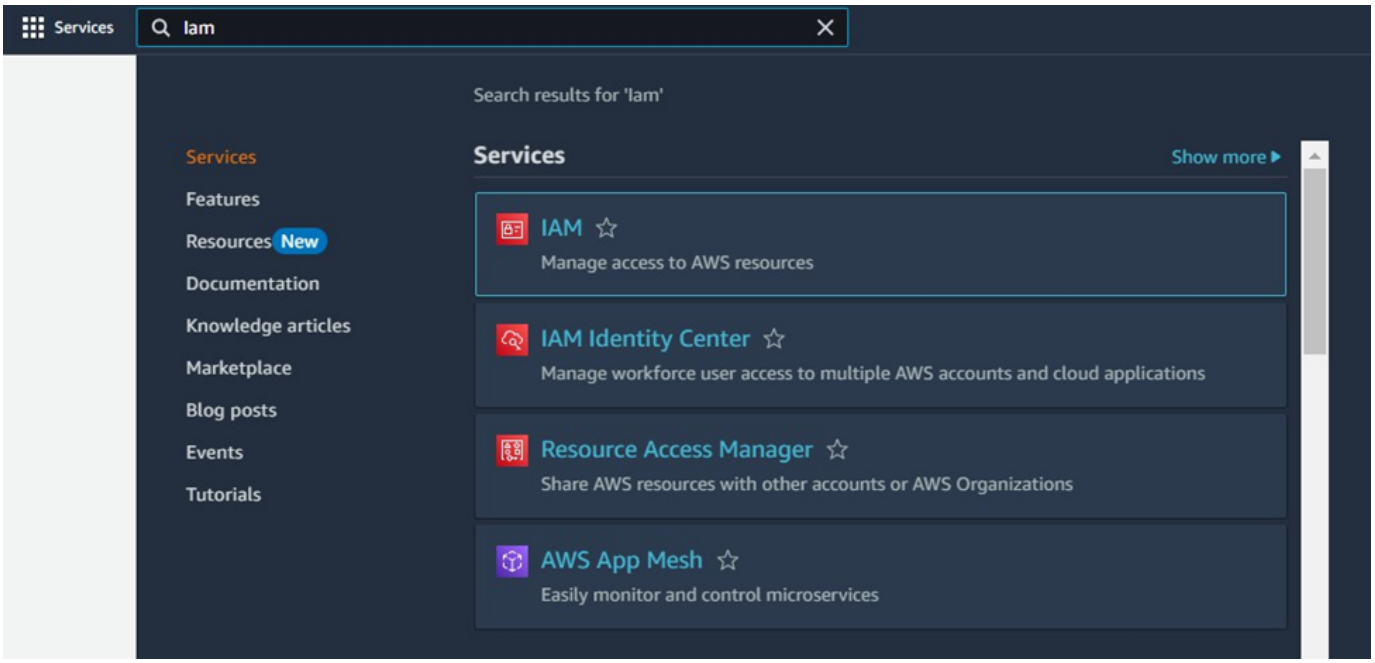
Deployment Code:

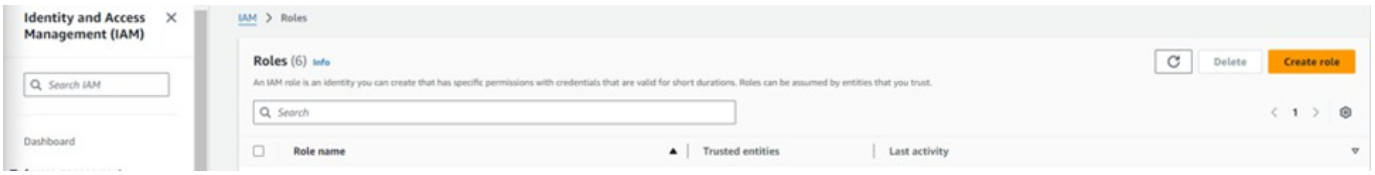
```python
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)
```

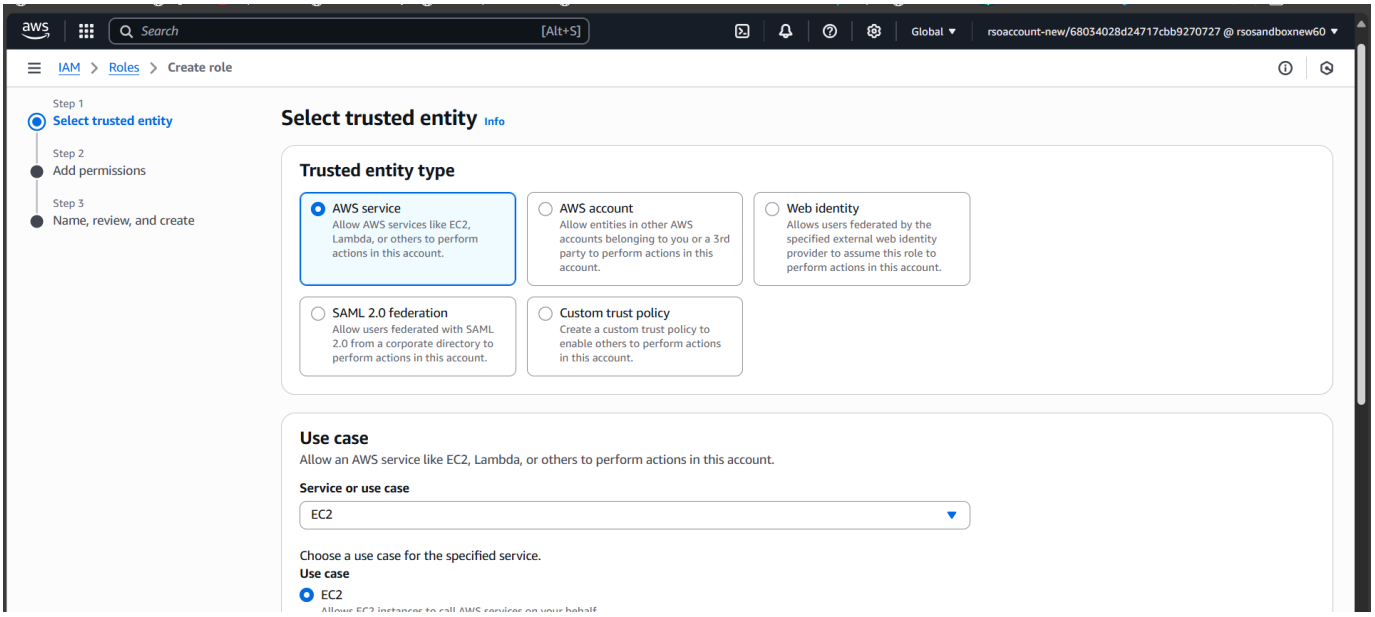
Description: start the Flask serverto listen on all networkinterfaces (`0.0.0.0`) at port `80` with debug mode enabled for development and testing.

Milestone 5: IAM Role Setup

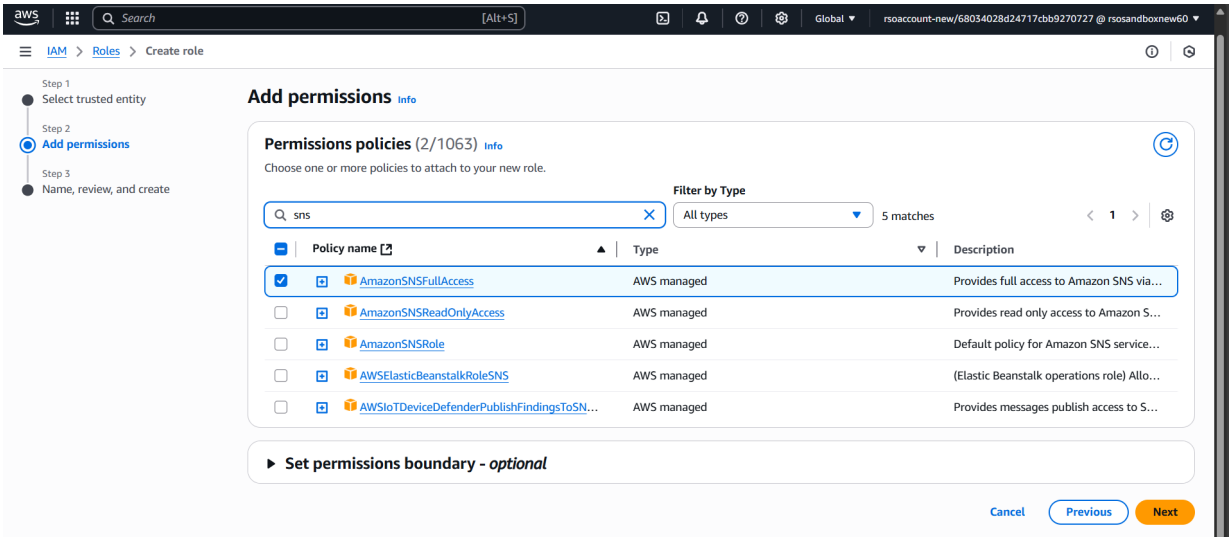
- **Activity 5.1:Create IAM Role.**
  - In the AWS Console,go to IAM and createa new IAM Role for EC2 to interact with DynamoDB and SNS.

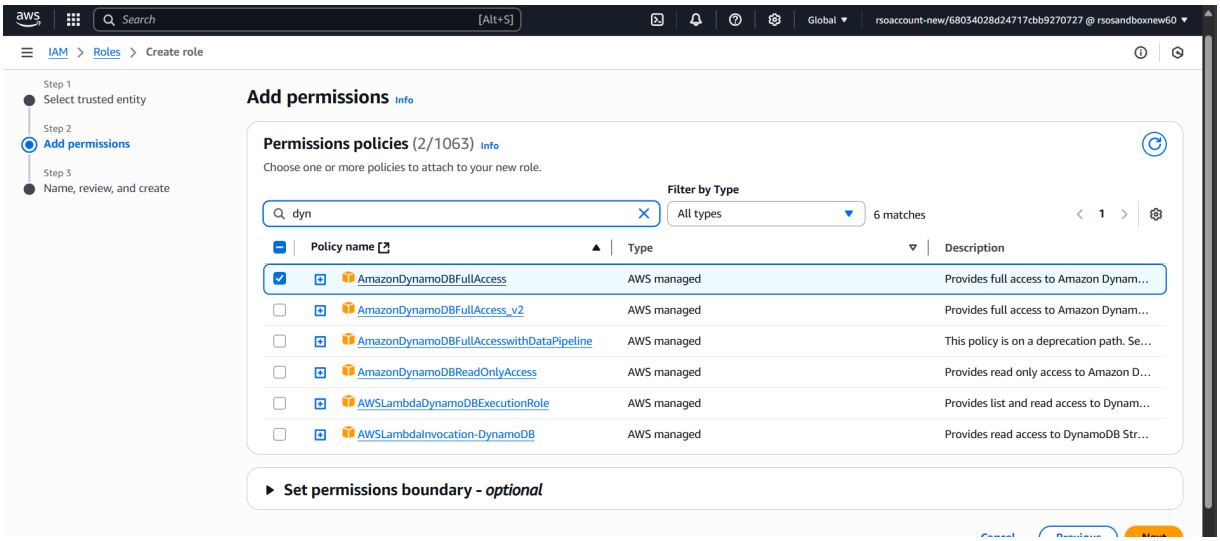


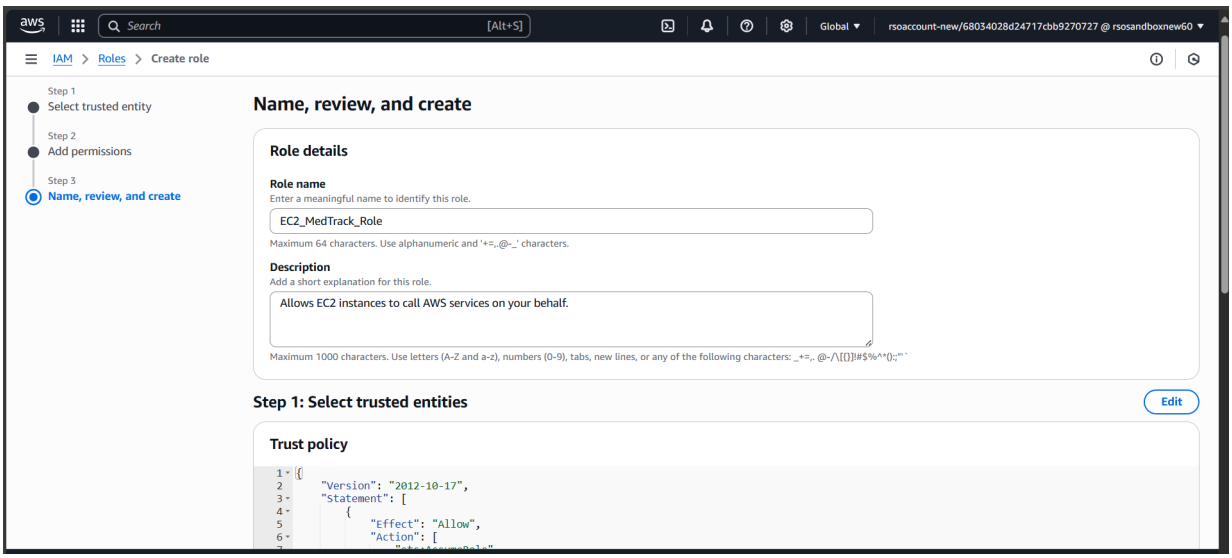


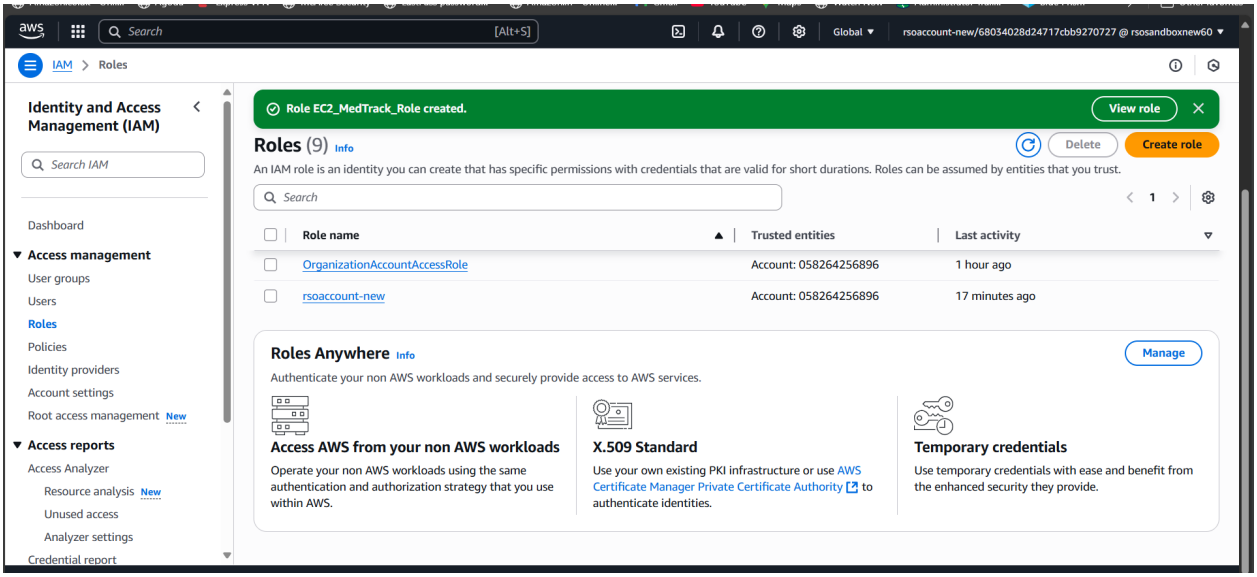


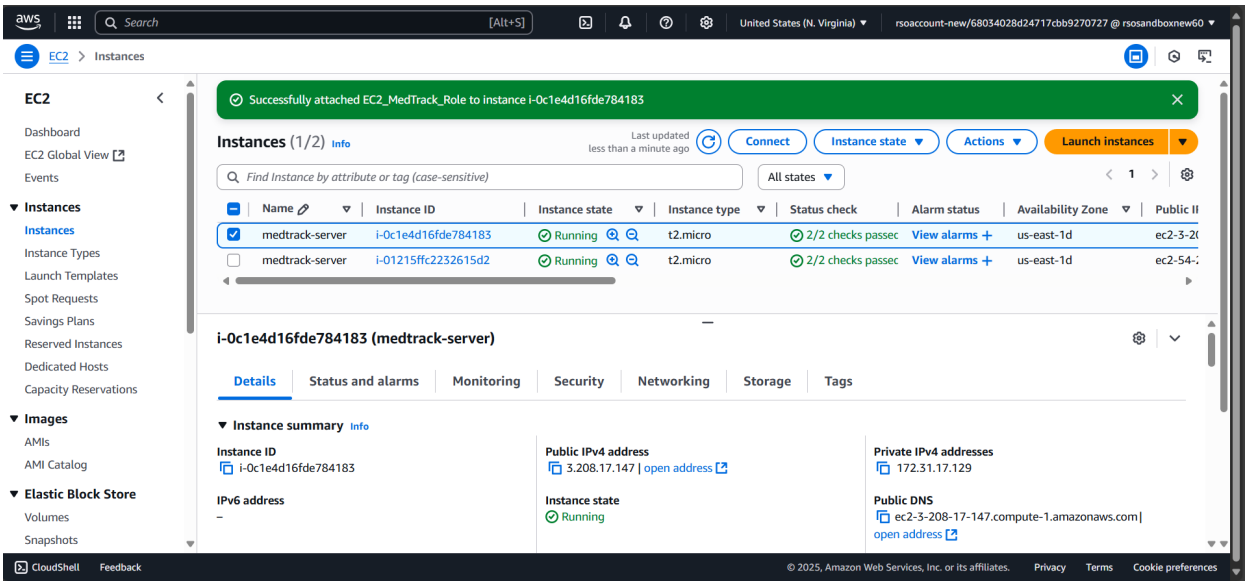
- **Activity 5.2: Attach Policies: Attach the following policiesto the role:**
    - AmazonDynamoDBFullAccess: Allows EC2 to perform read/write operations on DynamoDB.
    - AmazonSNSFullAccess: Grants EC2 the ability to send notifications via SNS.


Search [Alt+S]
Global
rsoaccount-new/68034028d24717cbb9270727 @ rsosandboxnew60
IAM > Roles > Create role
Step 1
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create
Add permissions Info
Permissions policies (2/1063) Info
Choose one or more policies to attach to your new role.
Filter by Type
sns
All types
5 matches
Policy name
Type
Description
AmazonSNSFullAccess
AWS managed
Provides full access to Amazon SNS via...
AmazonSNSReadOnlyAccess
AWS managed
Provides read only access to Amazon S...
AmazonSNSRole
AWS managed
Default policy for Amazon SNS service...
AWSElasticBeanstalkRoleSNS
AWS managed
(Elastic Beanstalk operations role) Allo...
AWSIoTDeviceDefenderPublishFindingsToSN...
AWS managed
Provides messages publish access to S...
Set permissions boundary - optional
Cancel
Previous
Next


Search [Alt+S]
Global
rsoaccount-new/68034028d24717cbb9270727 @ rsosandboxnew60
IAM > Roles > Create role
Step 1
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create
Add permissions Info
Permissions policies (2/1063) Info
Choose one or more policies to attach to your new role.
Filter by Type
dyn
All types
6 matches
Policy name
Type
Description
AmazonDynamoDBFullAccess
AWS managed
Provides full access to Amazon Dynam...
AmazonDynamoDBFullAccess_v2
AWS managed
Provides full access to Amazon Dynam...
AmazonDynamoDBFullAccesswithDataPipeline
AWS managed
This policy is on a deprecation path. Se...
AmazonDynamoDBReadOnlyAccess
AWS managed
Provides read only access to Amazon D...
AWSLambdaDynamoDBExecutionRole
AWS managed
Provides list and read access to Dynam...
AWSLambdaInvocation-DynamoDB
AWS managed
Provides read access to DynamoDB Str...
Set permissions boundary - optional

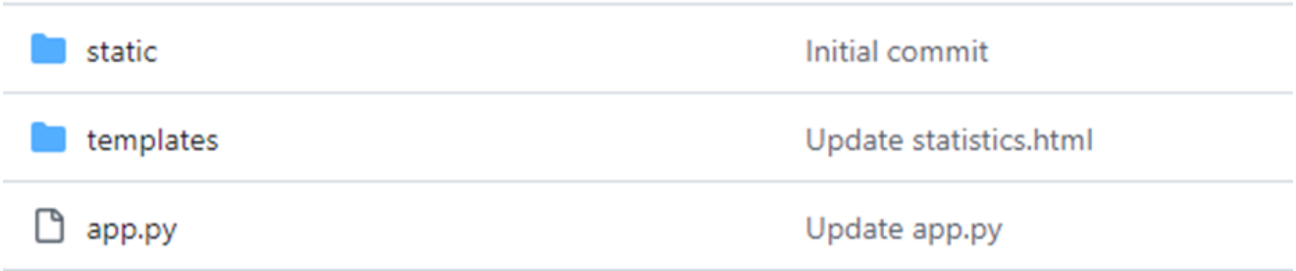

Search [Alt+S]
Global
rsoaccount-new/68034028d24717cbb9270727 @ rsosandboxnew60
IAM > Roles > Create role
Step 1
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create
Name, review, and create
Role details
Role name
Enter a meaningful name to identify this role.
EC2_MedTrack_Role
Maximum 64 characters. Use alphanumeric and '+=,.@-_' characters.
Description
Add a short explanation for this role.
Allows EC2 instances to call AWS services on your behalf.
Maximum 1000 characters. Use letters (A-Z and a-z), numbers (0-9), tabs, new lines, or any of the following characters: _+=,. @-/\[{}]!#$%^*();"'`
Step 1: Select trusted entities
Edit
Trust policy
{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Action": [

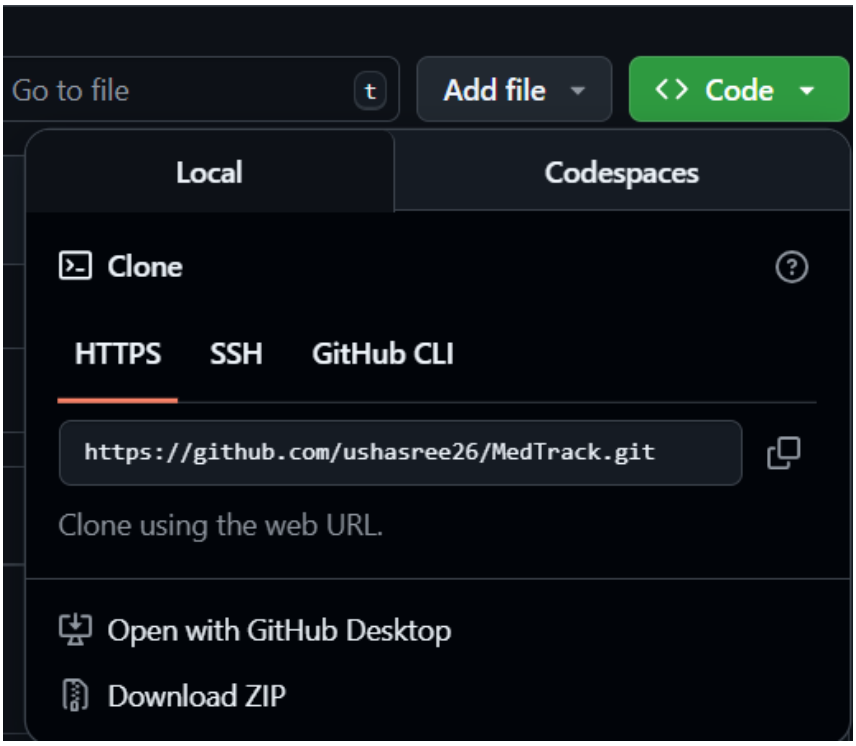




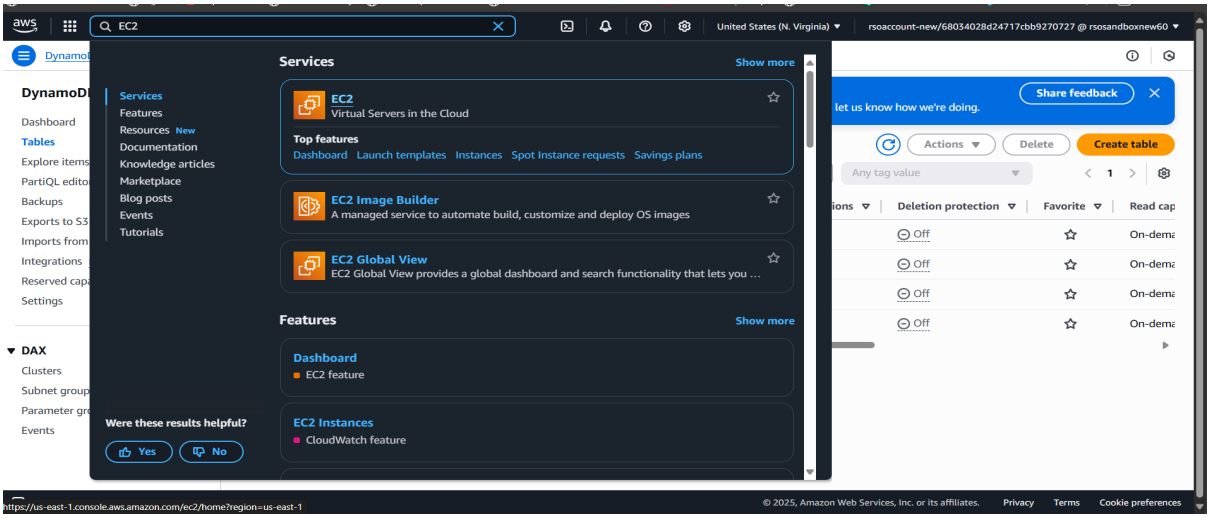
Milestone 6: EC2 InstanceSetup

Note: Load your Flask app and Html files into GitHub repository

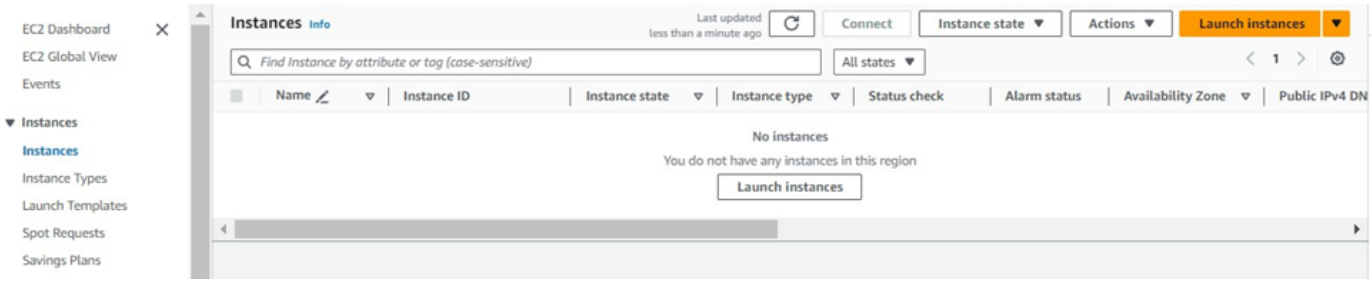


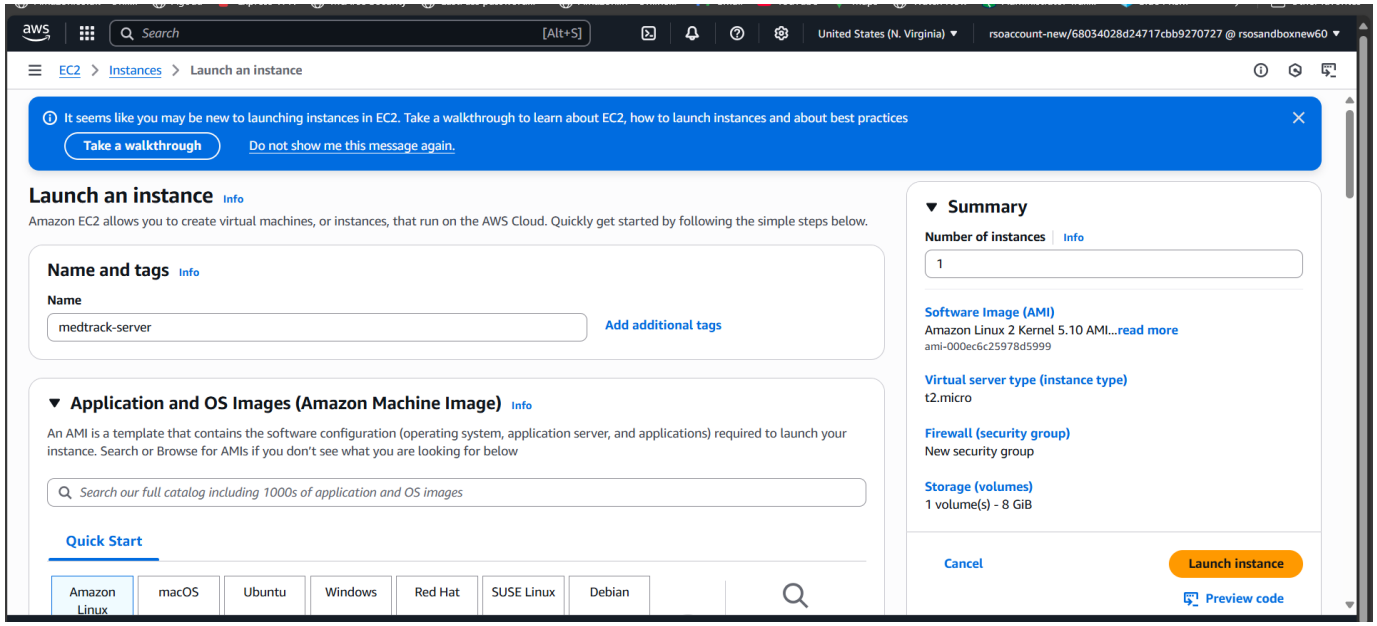


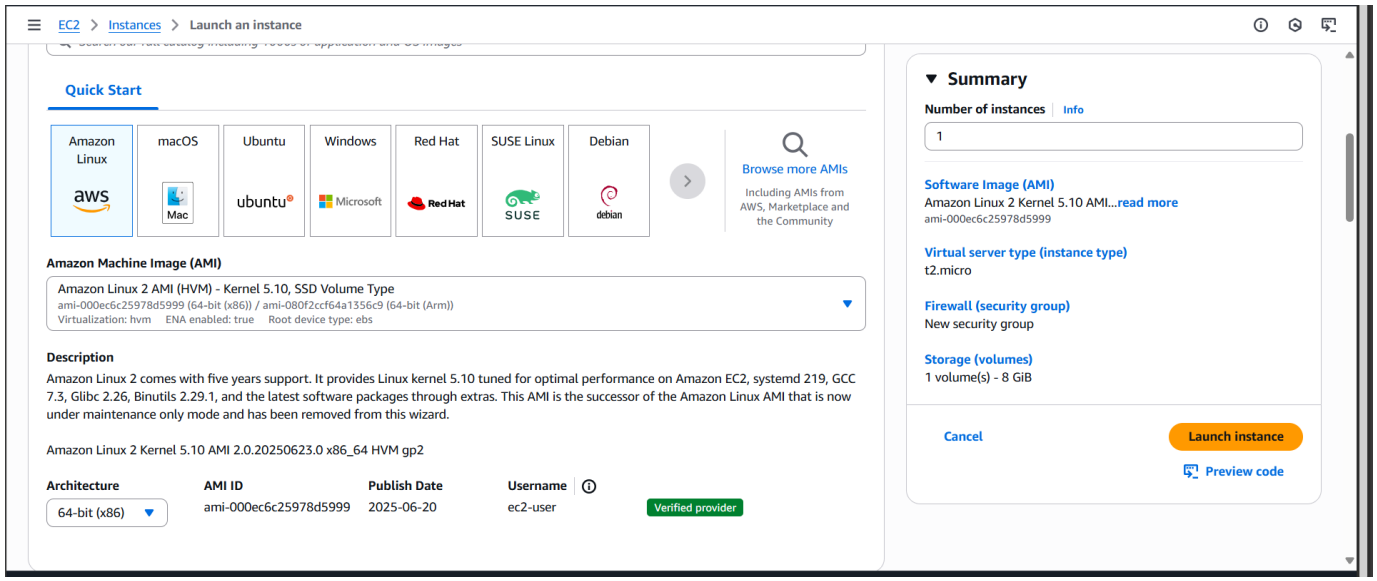
- **Activity 6.1: Launch an EC2 instance to host the Flask application.**
  - **Launch EC2 Instance**
    - In the AWS Console,navigate to EC2 and launcha new instance.
    -



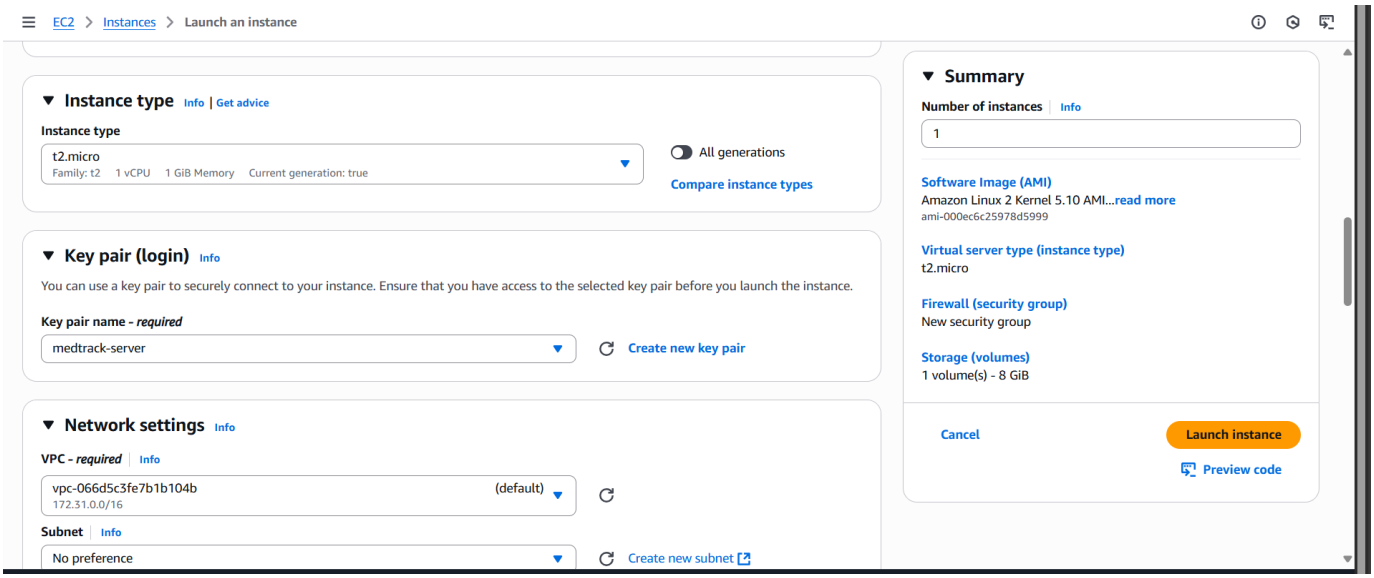
    - Click on Launchinstance to launchEC2 instance



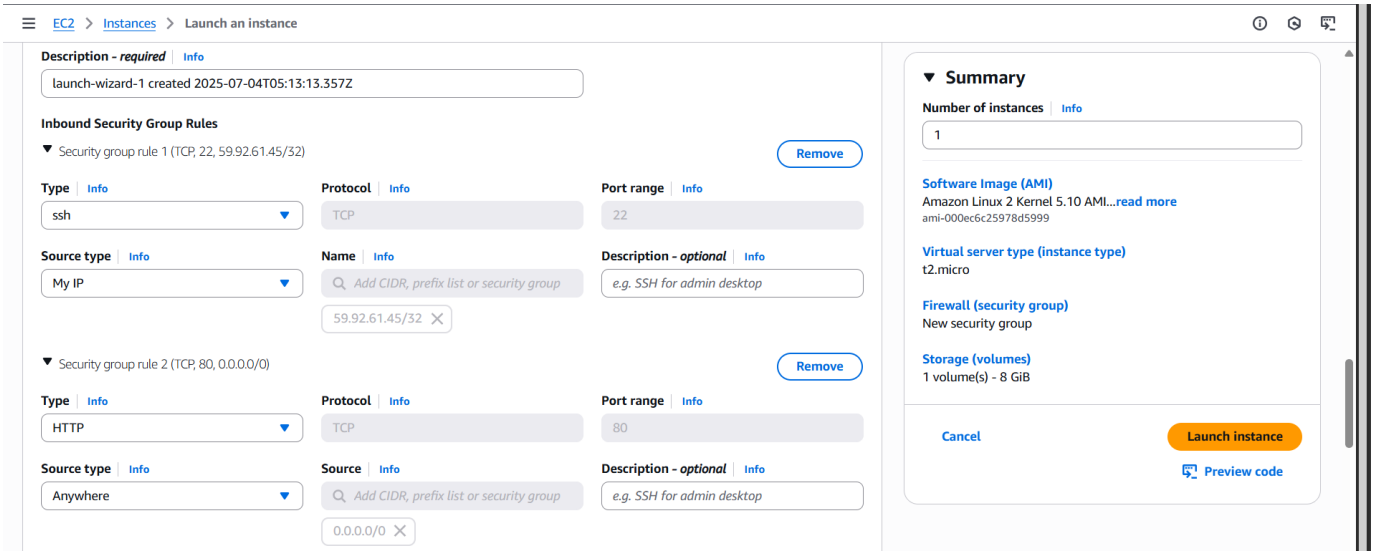




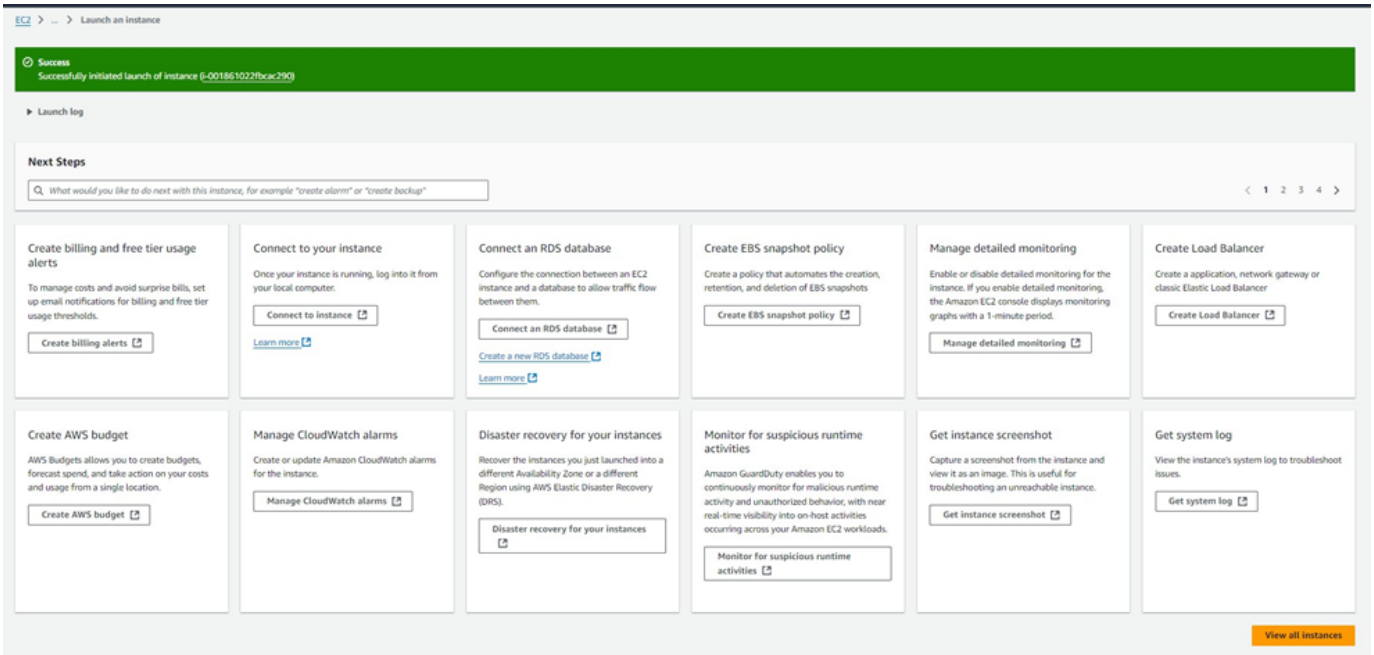
- Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instancetype (free-tier eligible).
- Create and downloadthe key pair for Server access.



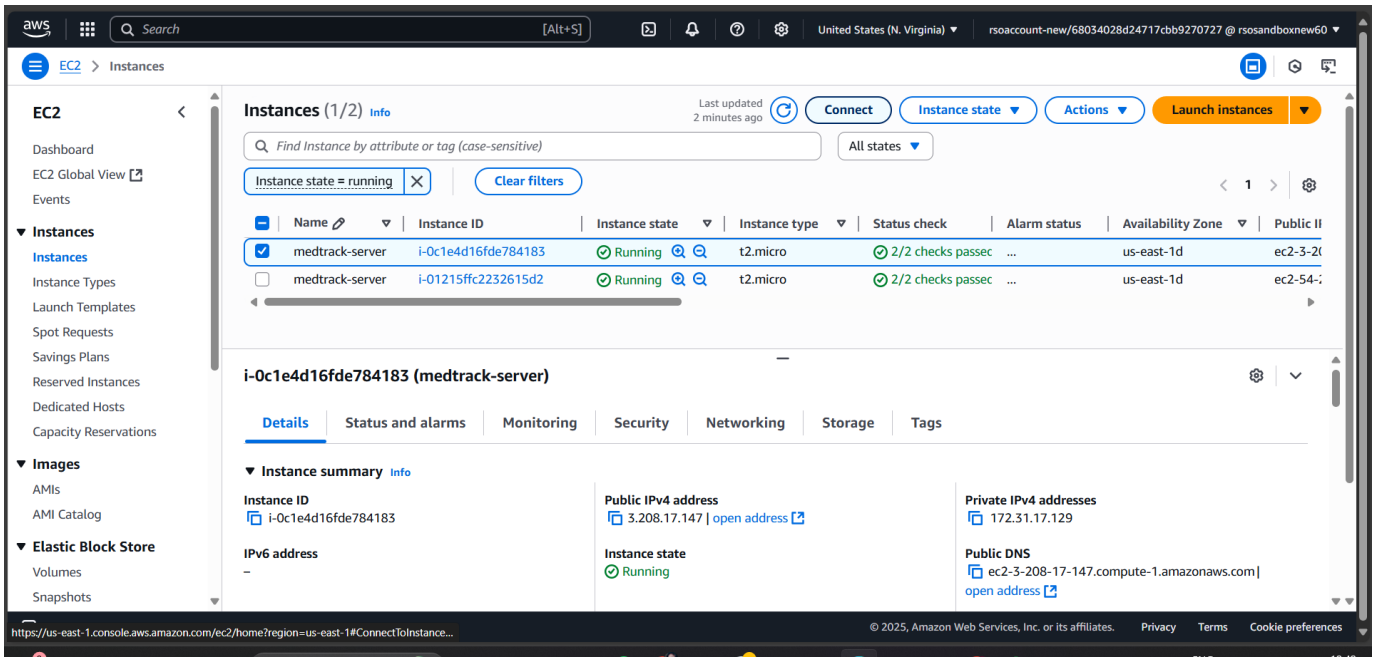
- **Activity 6.2:Configure securitygroups for HTTP,and SSH access.**

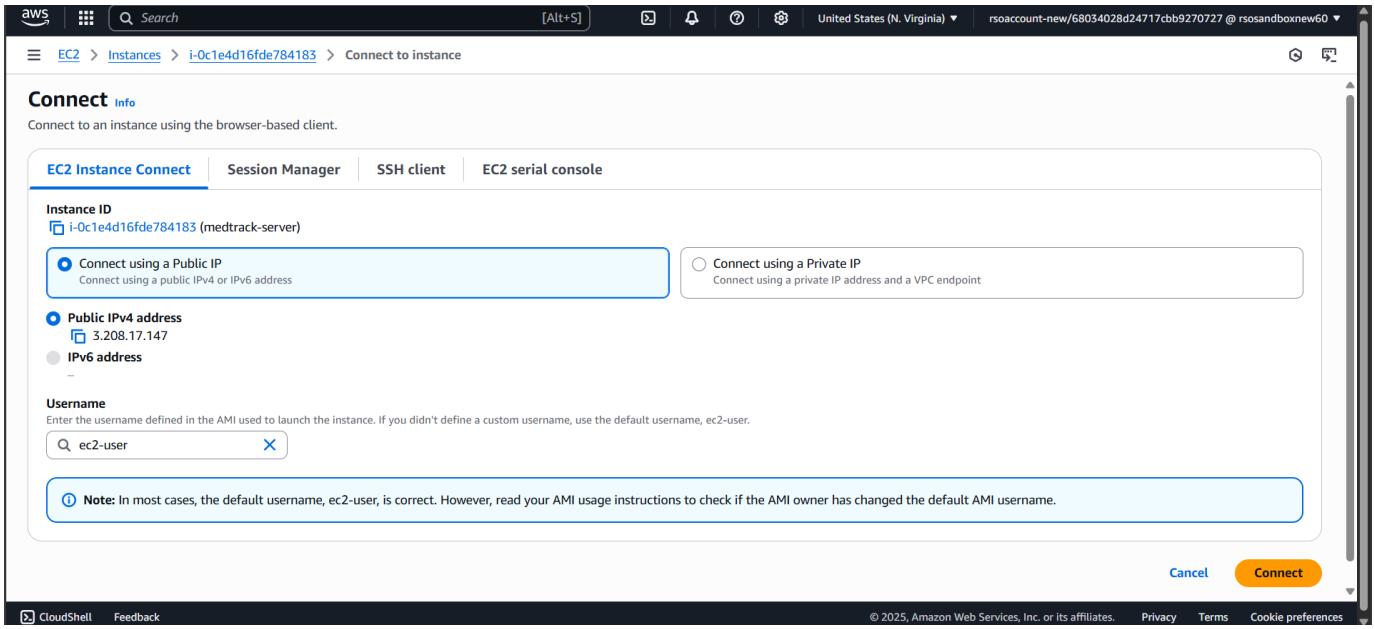


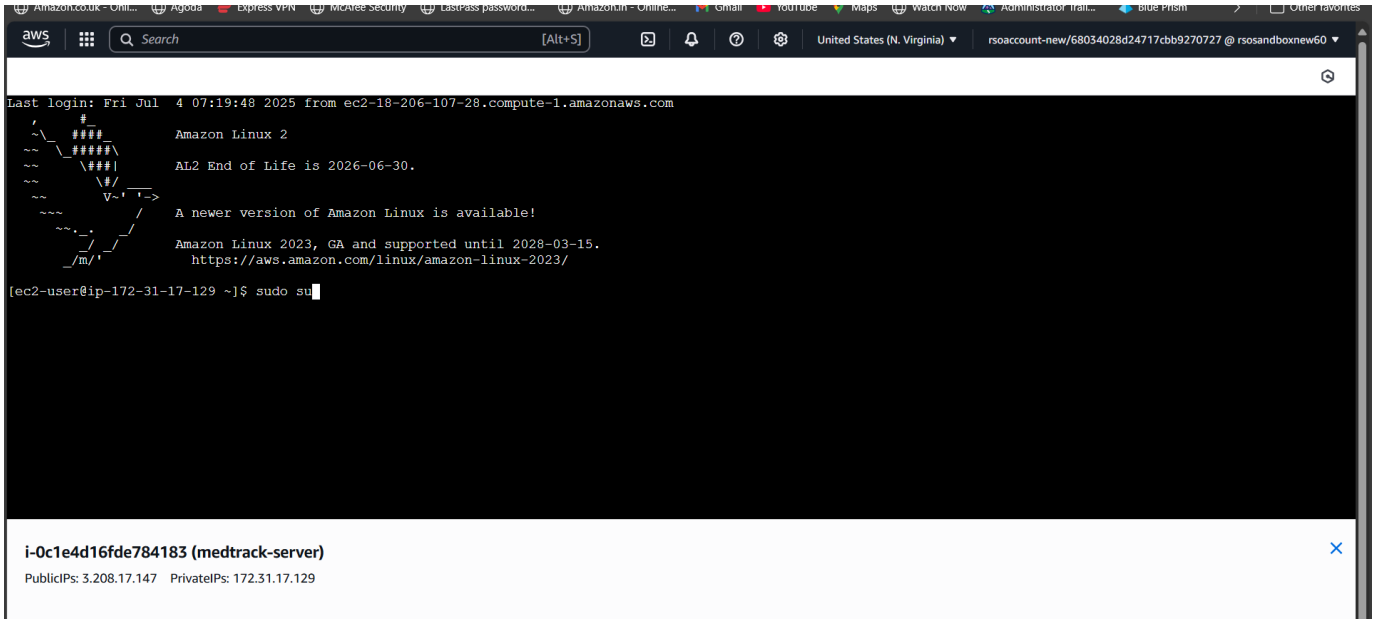
medtrack-server.pem



- To connect to EC2 using **EC2 Instance Connect**, start by ensuring that an **IAM role** is attached to your EC2 instance. You can do this by selecting your instance, clicking on **Actions**, then navigating to **Security** and selecting **Modify IAM Role** to attach the appropriate role. After the IAM role is connected, navigate to the **EC2** section in the **AWS Management Console**. Select the **EC2 instance** you wish to connect to. At the top of the **EC2 Dashboard**, click the **Connect** button. From the connection methods presented, choose **EC2 Instance Connect**. Finally, click **Connect** again, and a new browser-based terminalwill open, allowing you to access your EC2 instance directly from your browser.







Milestone 7: Deployment on EC2

Activity 7.1: Install Softwareon the EC2 Instance

Install Python3, Flask, and Git: On Amazon Linux 2:

sudo yum update-y
sudo yum installpython3
git sudo pip3 install flaskboto3

Verify Installations:

flask --

version git

--version

Activity 7.2:Clone Your Flask Projectfrom GitHub

Clone your projectrepository from GitHubinto the EC2 instance using Git.

Run: ‘git clone : 'https://github.com/ushasree26/MedTrack.git'

- This will download your project to the EC2 instance.

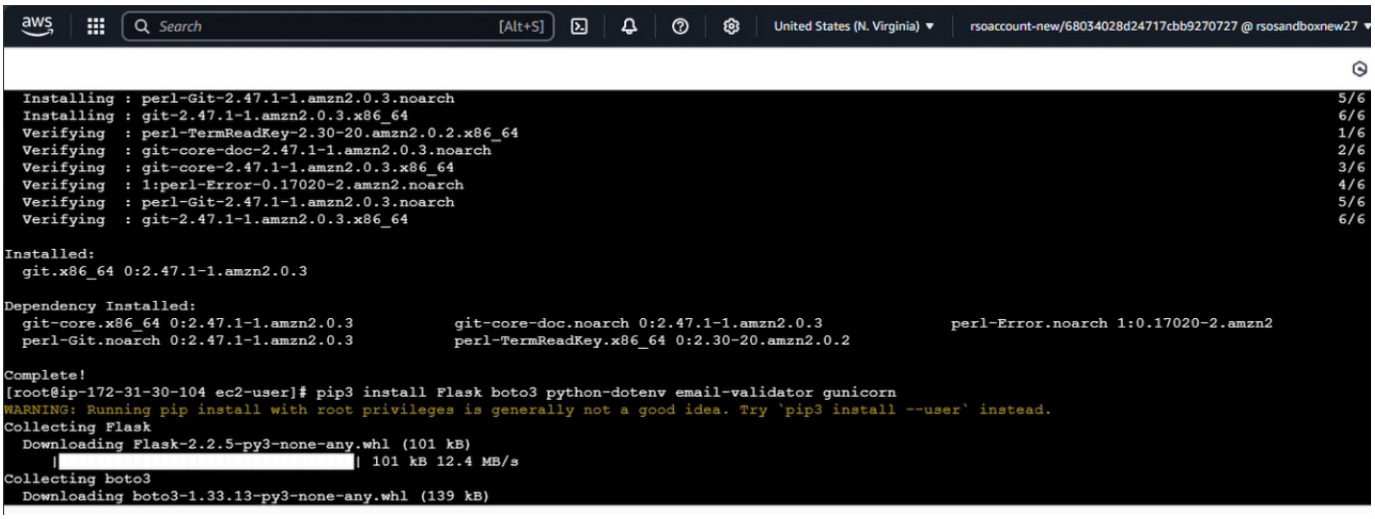
To navigateto the project directory, run the following command:

cd MedTrack

Once inside the projectdirectory, 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=5000



Verify the Flask app is running:

http://3.208.17.147:5000/

- Run the Flask app on the EC2 instance

Access the website through:

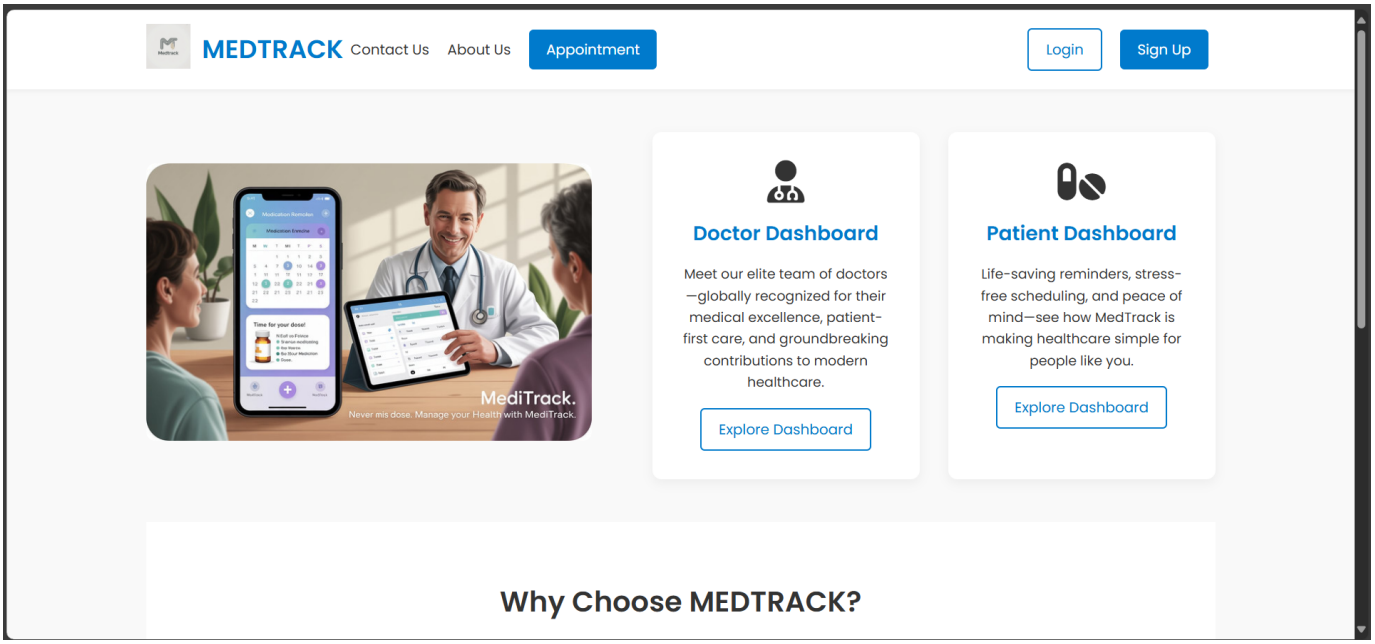
PublicIPs: http://3.208.17.147:5000/

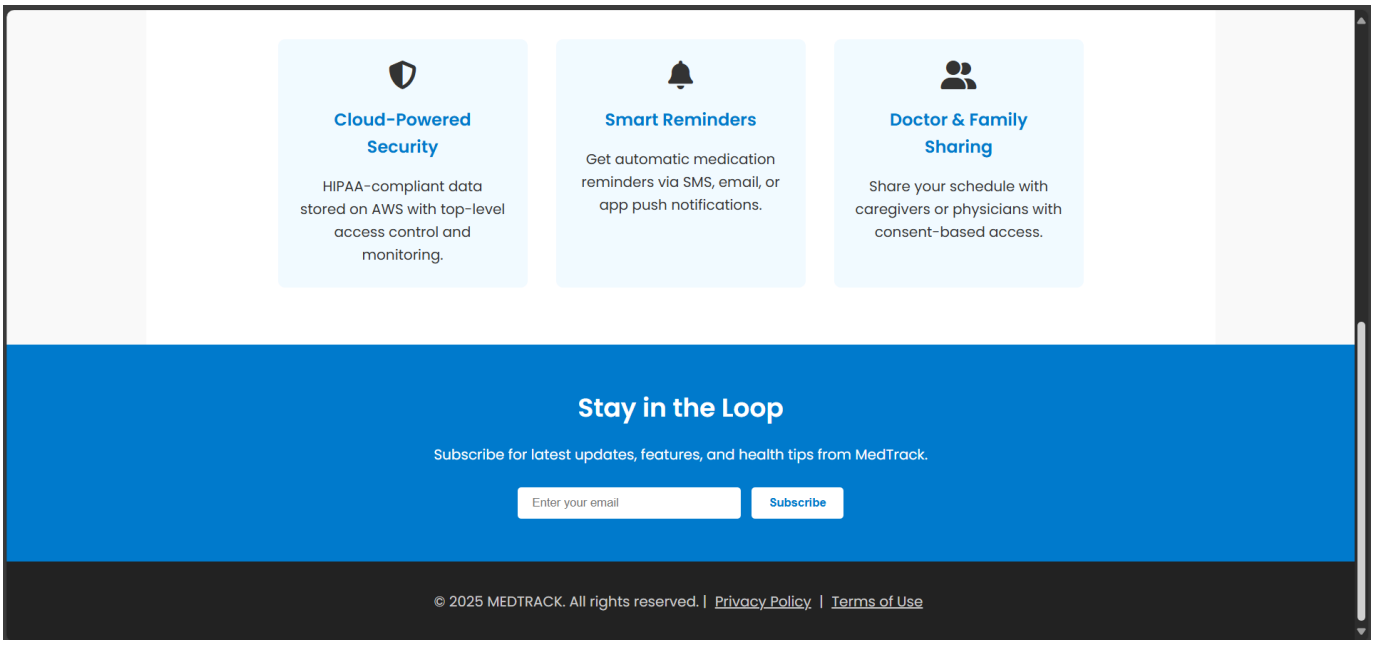
Milestone 8: Testing and Deployment:

● Activity 8.1:

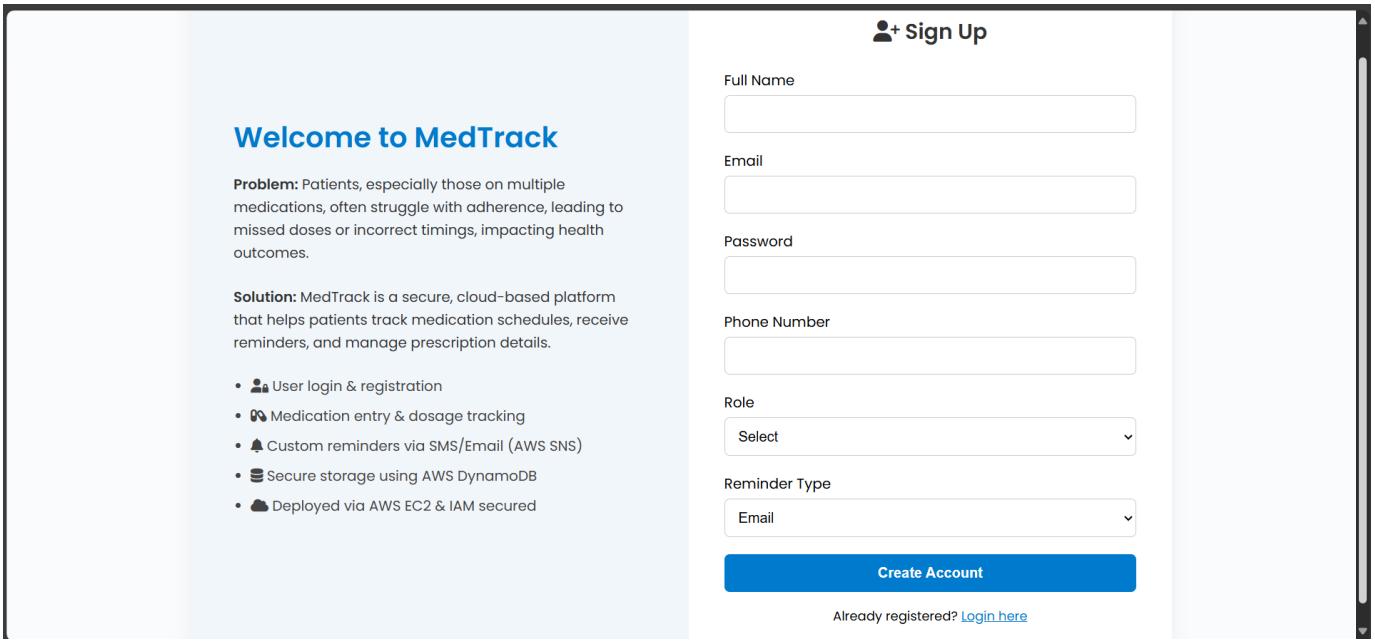
Conduct functional testing to verify user sign-up, login, appointment booking, prescription generation, medication reminders, and SNS notifications.

Index Page:

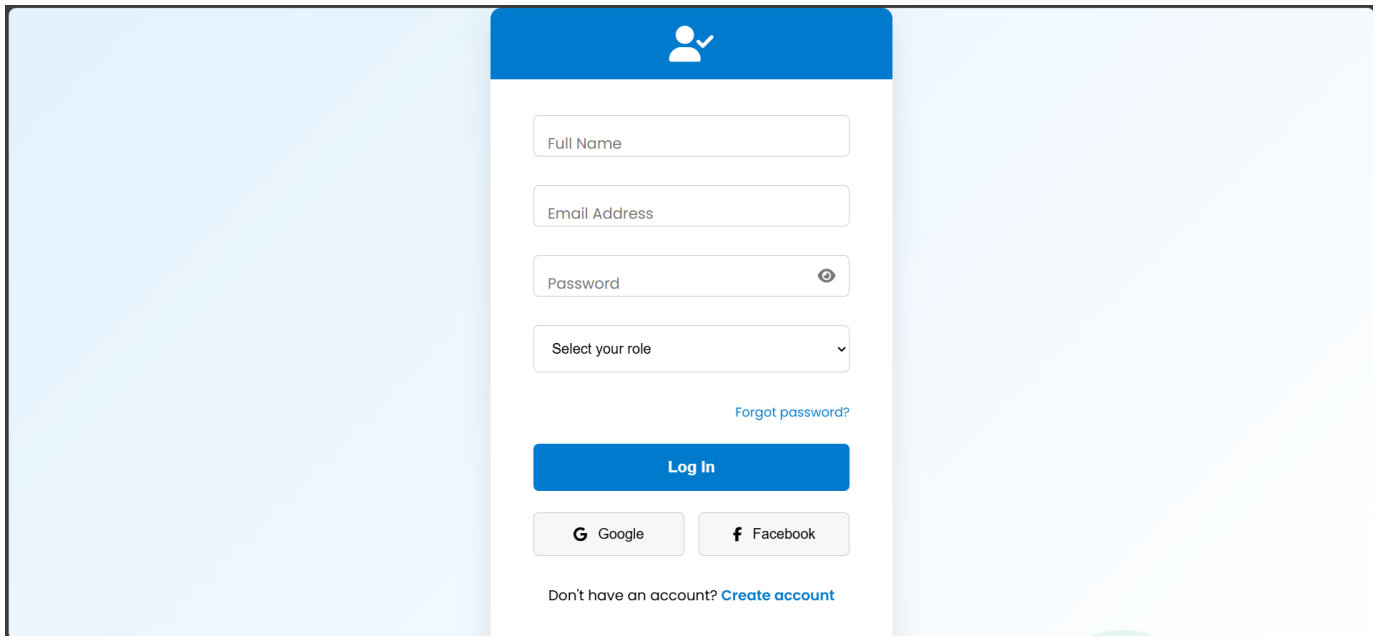



Cloud-Powered Security
HIPAA-compliant data stored on AWS with top-level access control and monitoring.
Smart Reminders
Get automatic medication reminders via SMS, email, or app push notifications.
Doctor & Family Sharing
Share your schedule with caregivers or physicians with consent-based access.
Stay in the Loop
Subscribe for latest updates, features, and health tips from MedTrack.
Enter your email
Subscribe
© 2025 MEDTRACK. All rights reserved. | Privacy Policy | Terms of Use

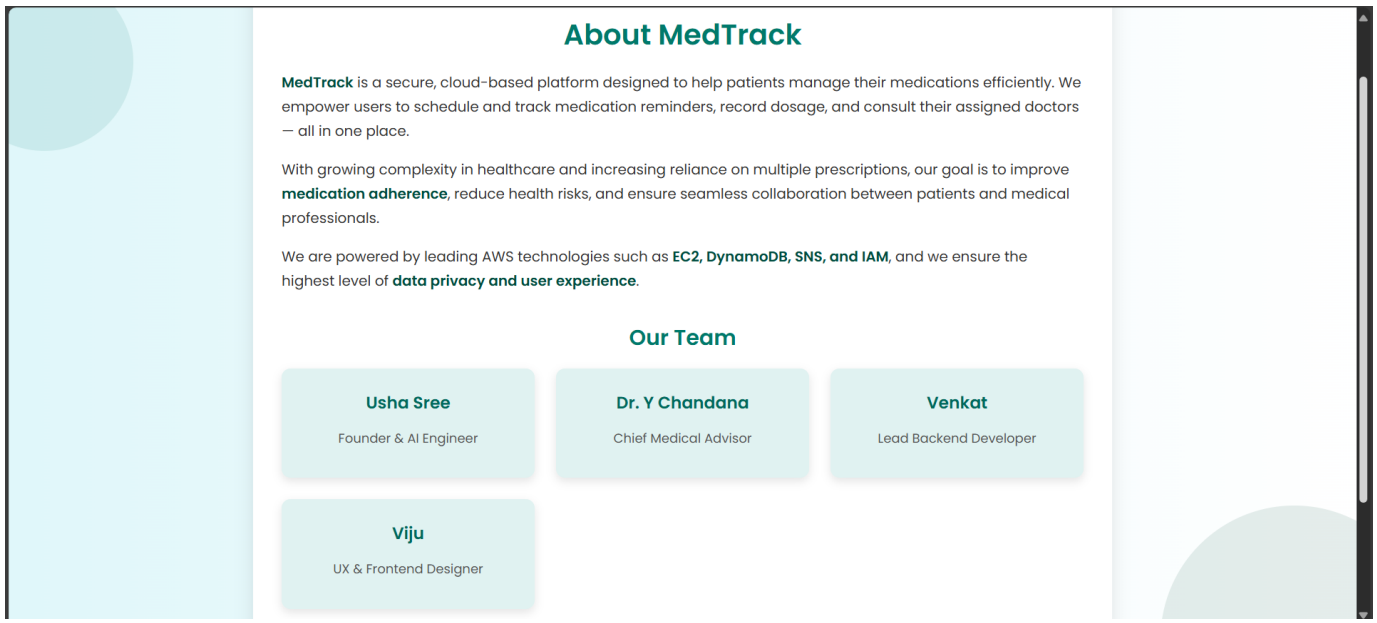
Signup Page:


Welcome to MedTrack
Problem: Patients, especially those on multiple medications, often struggle with adherence, leading to missed doses or incorrect timings, impacting health outcomes.
Solution: MedTrack is a secure, cloud-based platform that helps patients track medication schedules, receive reminders, and manage prescription details.
User login & registration
Medication entry & dosage tracking
Custom reminders via SMS/Email (AWS SNS)
Secure storage using AWS DynamoDB
Deployed via AWS EC2 & IAM secured
Sign Up
Full Name
Email
Password
Phone Number
Role
Select
Reminder Type
Email
Create Account
Already registered? Login here

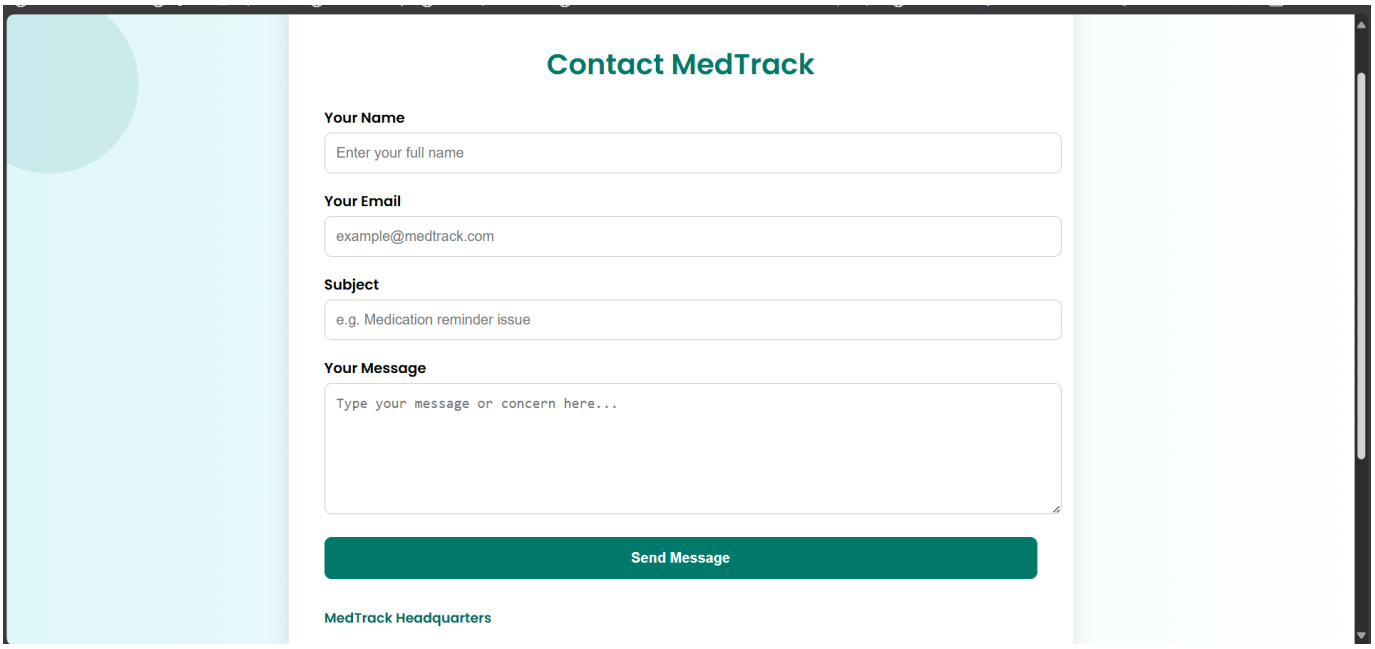
Login Page:



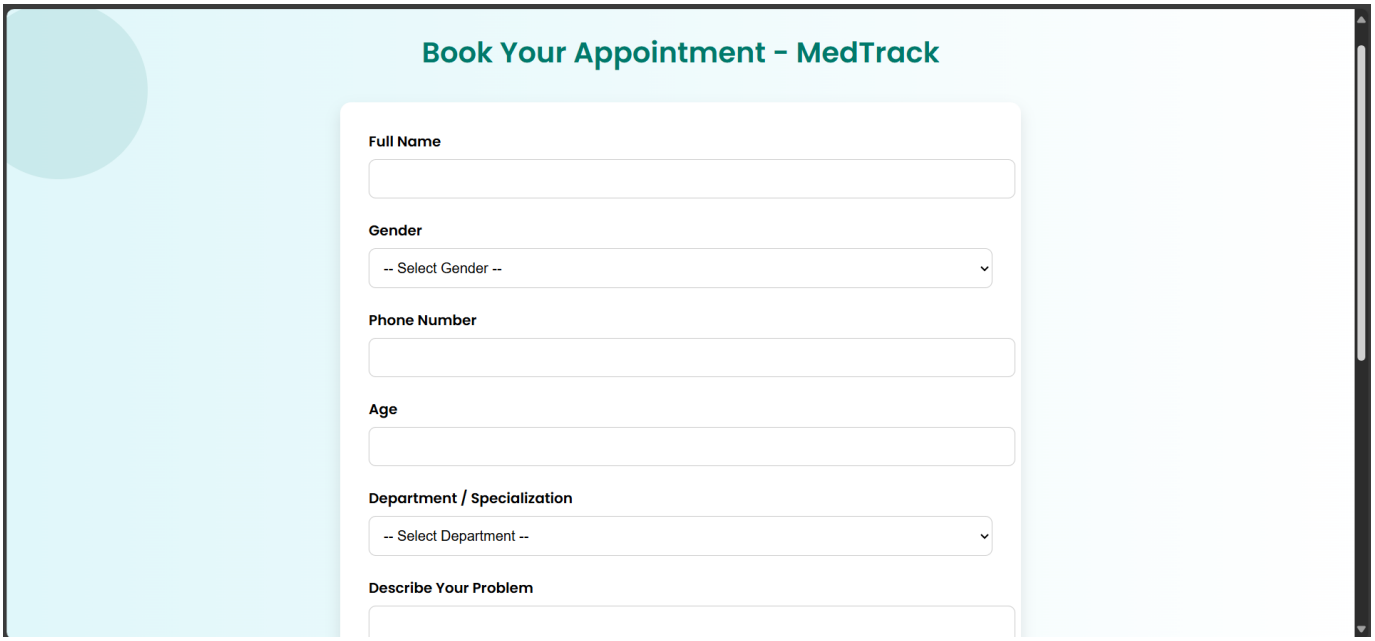
About Us page:

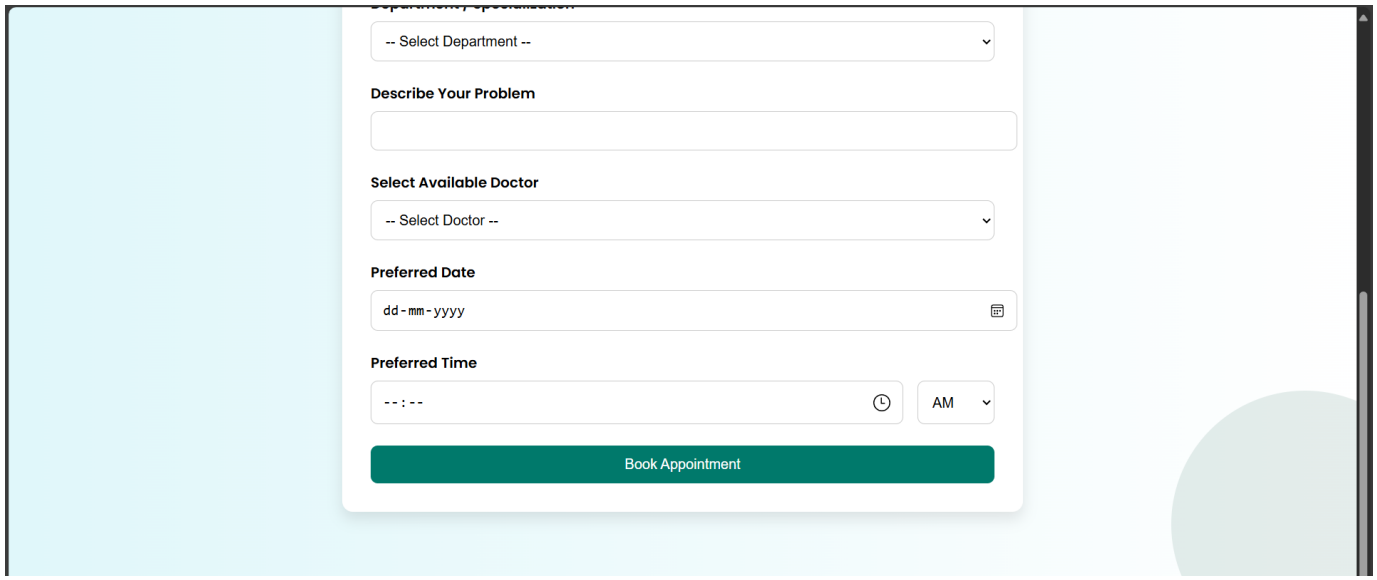


ContactUs Page:

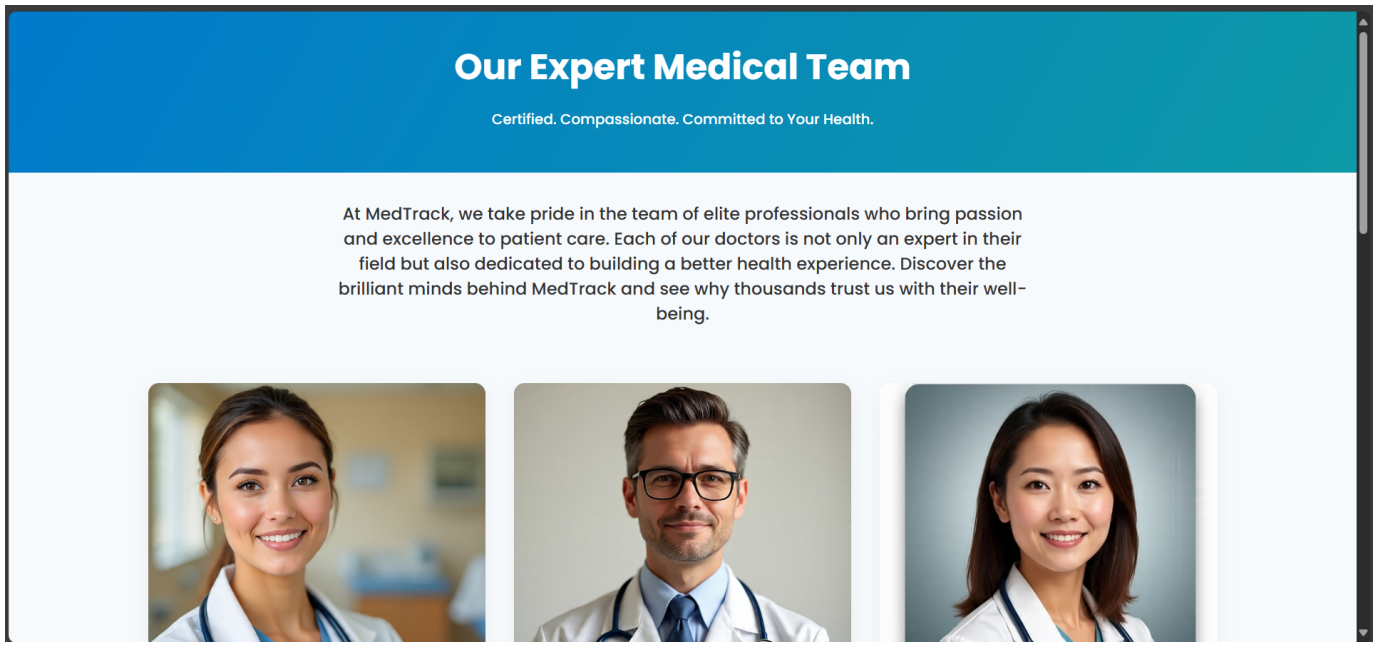


Book Appointment Page:

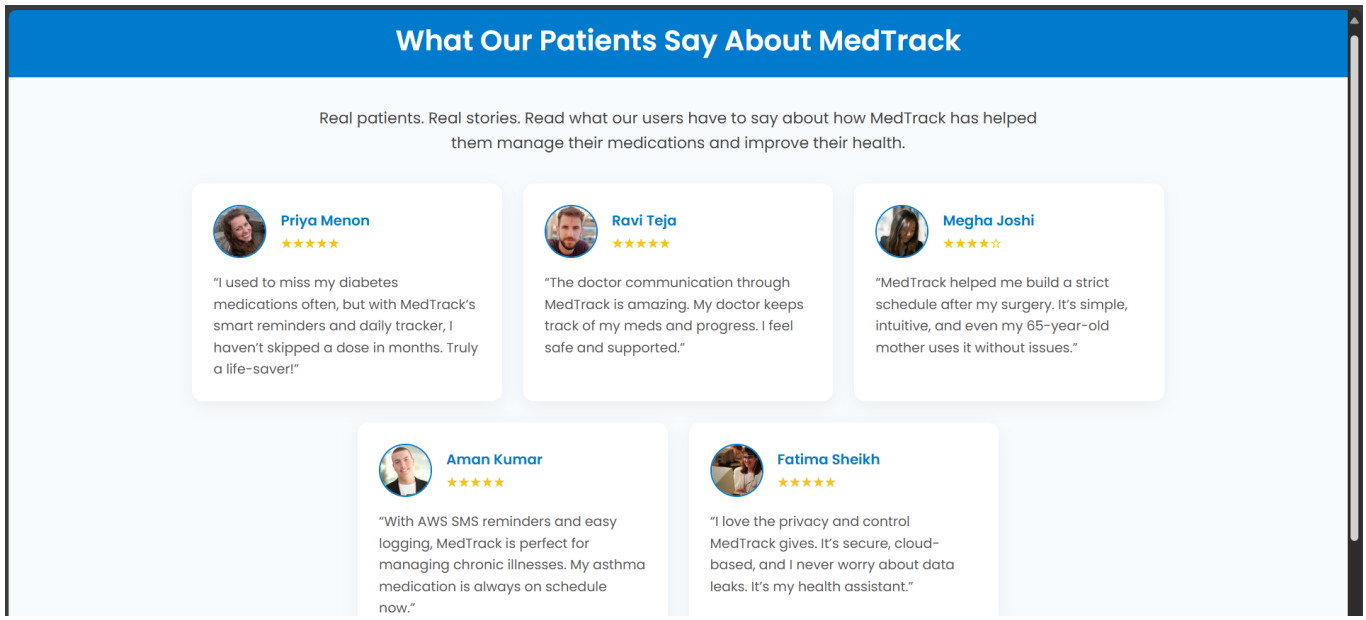




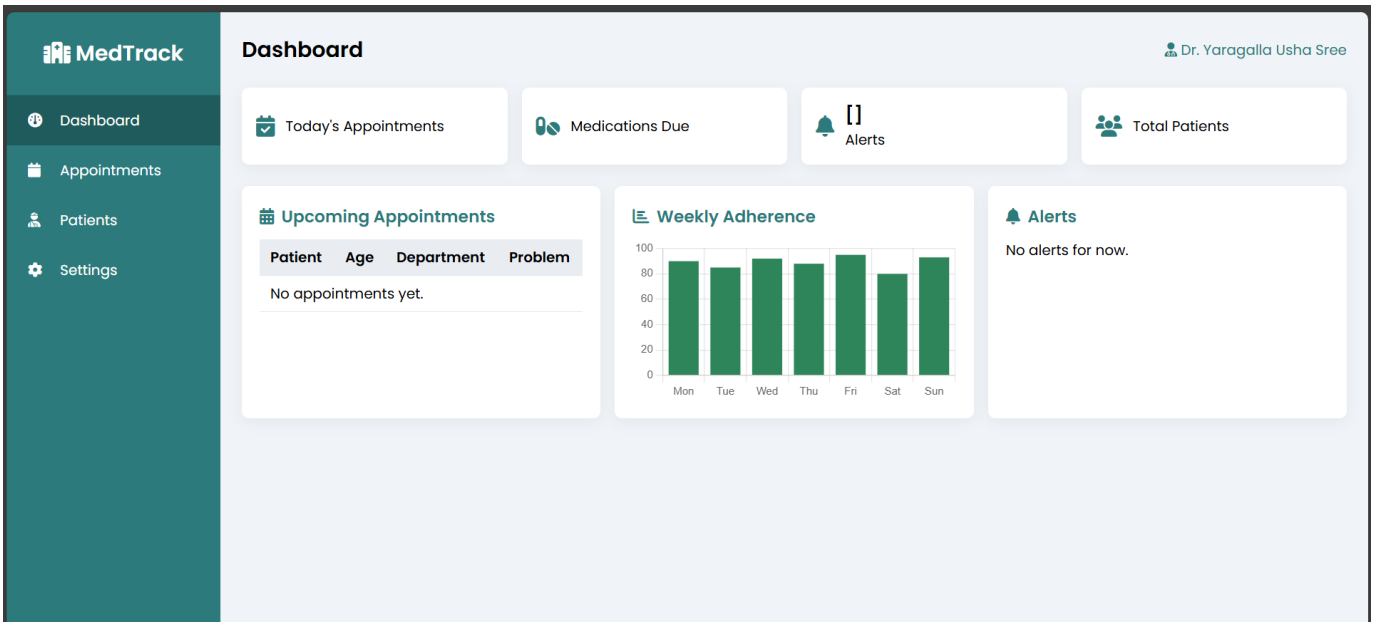
(Doctor)Explore Dashboard Page:



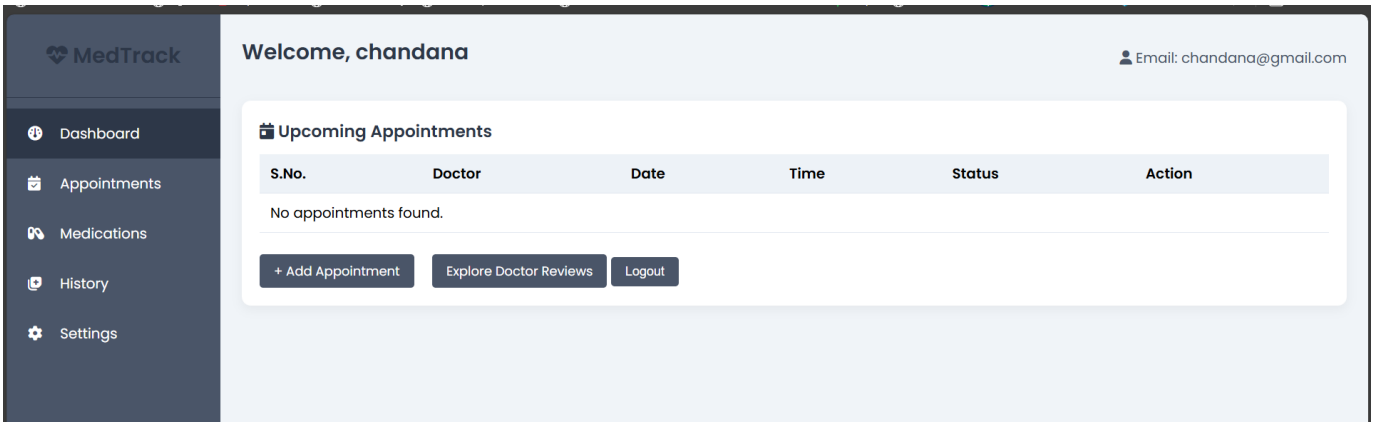
(Patient)Explore Dashboard Page:



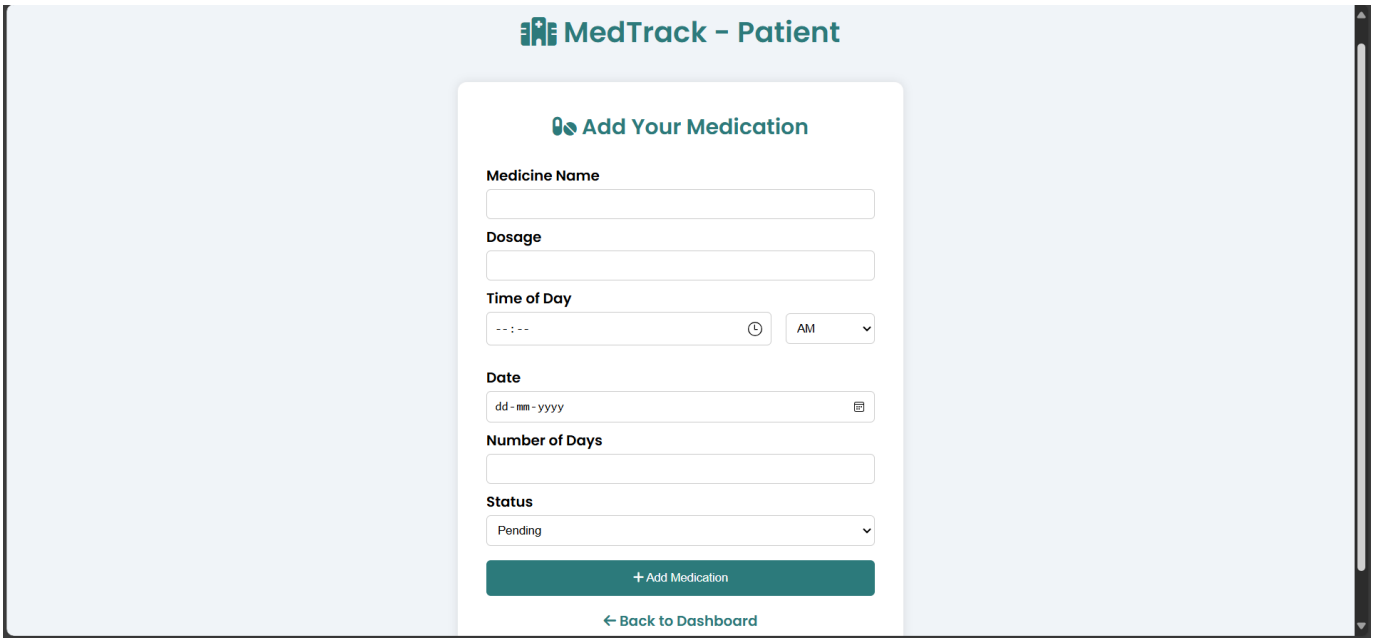
Doctor Dashboard:



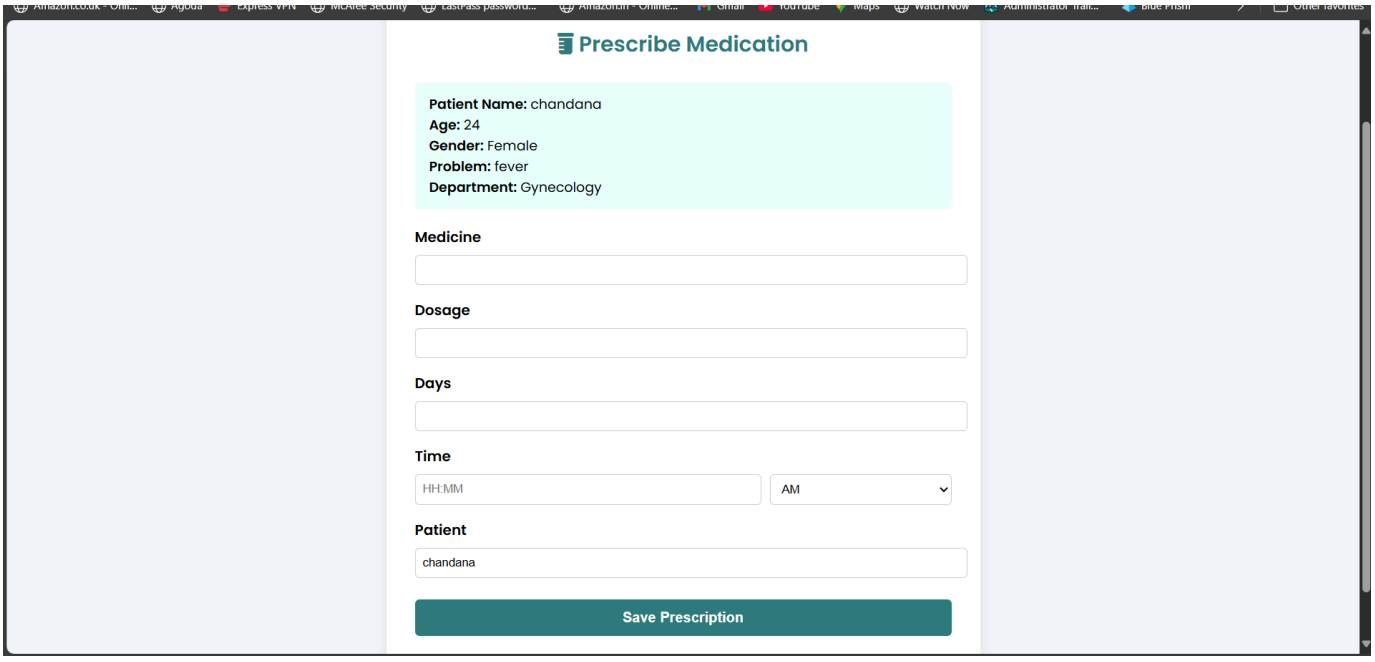
Patient Dashboard:



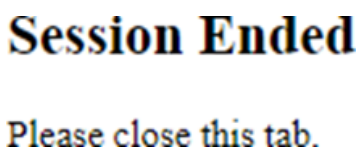
Add Medication Page:



Prescribe Medication:



Exit:



Conclusion:

The **MedTrack Healthcare Assistant** has been successfully developed and deployed using a robust cloud-based architecture. By leveraging **AWS services** such as **EC2 for hosting**, **DynamoDB for data storage**, **SNS for real-time email notifications**, and **IAM for secure access control**, the platform ensures reliable, secure, and scalable access to essential healthcare services for both patients and doctors.

This system addresses common healthcare challenges by allowing **patients to book appointments, receive timely medication reminders**, and track their prescriptions — all from a unified platform. **Doctors can manage appointments, issue prescriptions, and monitor patient adherence**, improving the overall quality of care.

The **cloud-native approach** ensures seamless scalability and high availability, making it adaptable to increasing user demand. The integration of **Flask with AWS services** guarantees efficient backend operations, including secure user authentication and notification handling.

During testing, all core functionalities — from user registration and login to **SNS-powered reminders and real-time medication tracking** — were verified for reliability and accuracy.

In conclusion, **MedTrack offers a modern, cloud-powered solution** for improving patient-doctor engagement, simplifying health management tasks, and enhancing the overall healthcare experience. This project showcases how cloud technologies can effectively address real-world challenges in the medical field.