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.

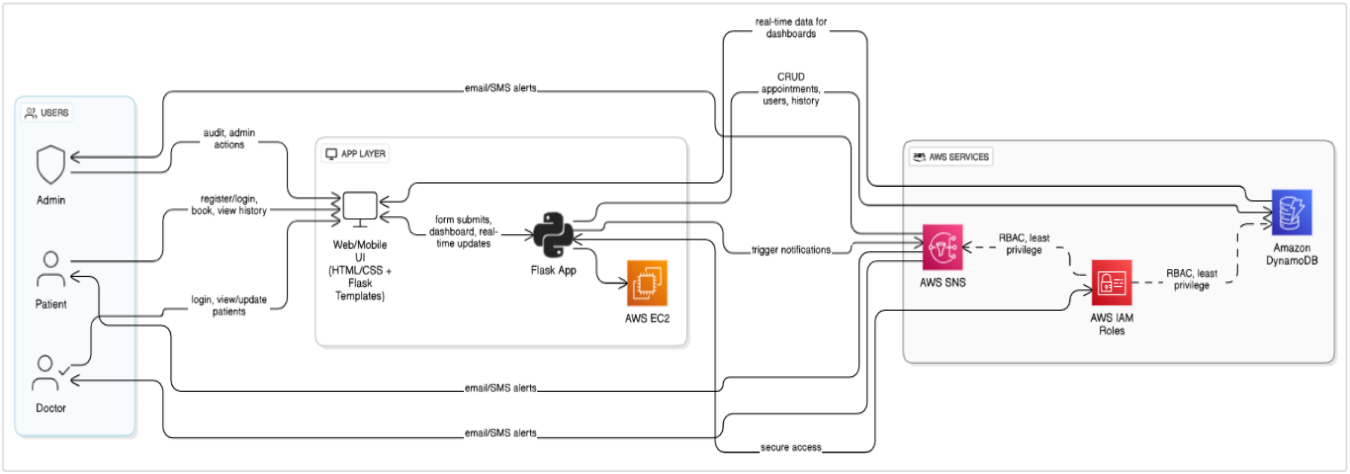
Scenario 2: Secure User Management with IAM

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.

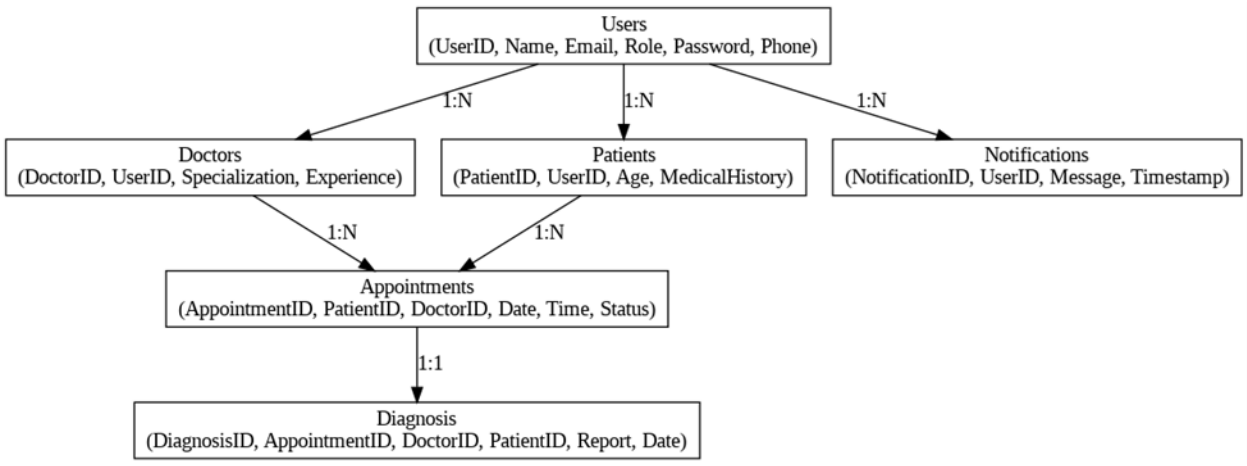
Scenario 3: Easy Access to Medical History and Resources

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:



Entity Relationship (ER)Diagram:



Pre-requisites:

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

Project Workflow:

1. AWS Account Setup and Login

Activity 1.1: Set up an AWS account if not already done**.**

Activity 1.2: Log in to the AWS Management Console

2. DynamoDB Database Creation and Setup

Activity 2.1: Create a DynamoDB Table.

Activity 2.2: Configure Attributes for User Data and Book Requests.

3. SNS Notification Setup

Activity 3.1: Create SNS topics for book request notifications.

Activity 3.2: Subscribe users and library staff to SNS email notifications.

4. Backend Development and Application Setup

Activity 4.1: Develop the Backend Using Flask.

Activity 4.2: Integrate AWS Services Using boto3.

5. IAM Role Setup

Activity 5.1: Create IAM Role

Activity 5.2: Attach Policies

6. EC2 Instance Setup

Activity 6.1: Launch an EC2 instance to host the Flask application.

Activity 6.2: Configure security groups for HTTP, and SSH access.

7. Deployment on EC2

Activity 7.1:Upload Flask Files

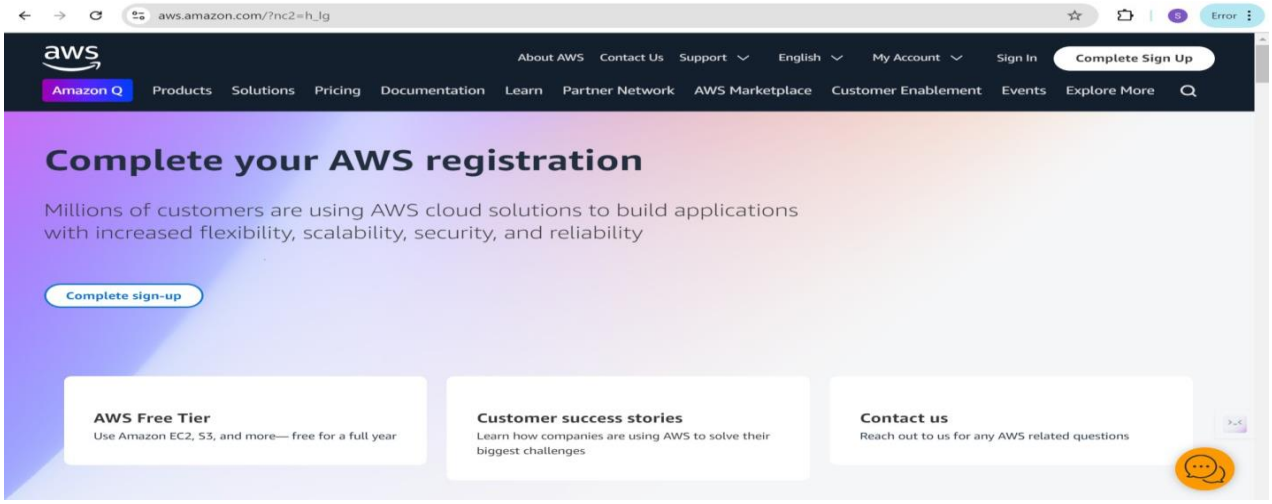
Activity 7.2: Run the Flask App

8. Testing and Deployment

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

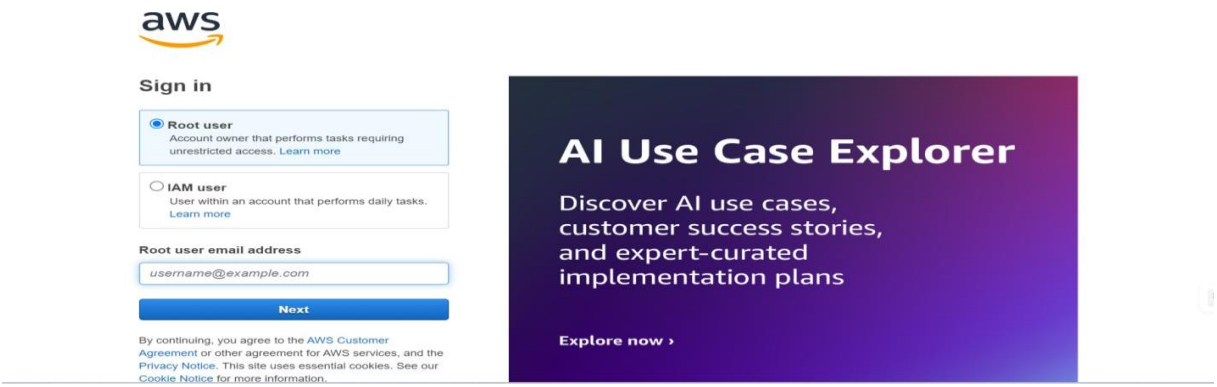
Milestone 1: AWS Account Setup and Login

- **Activity 1.1: Set up an AWS account if not already done.**
  - Sign up for an AWS account and configure billing settings.

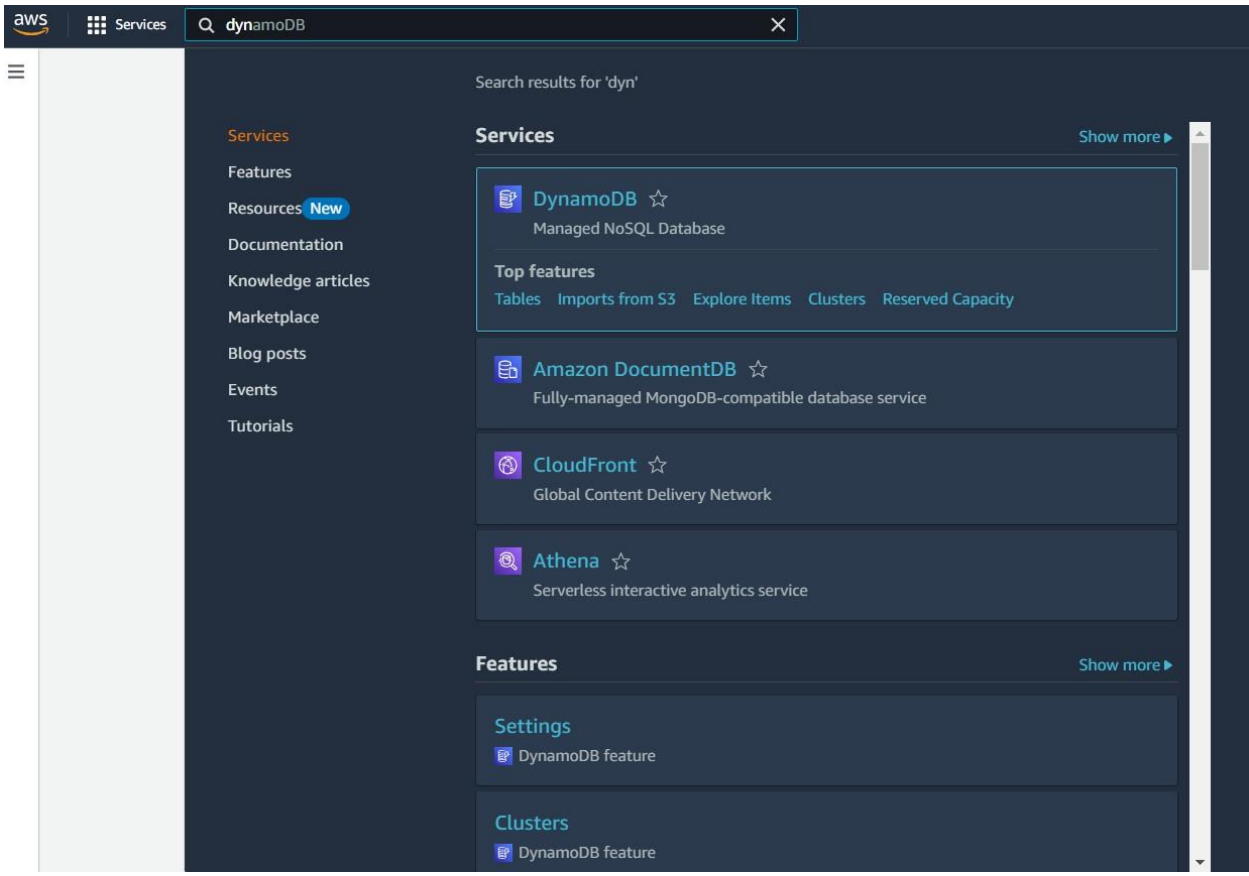


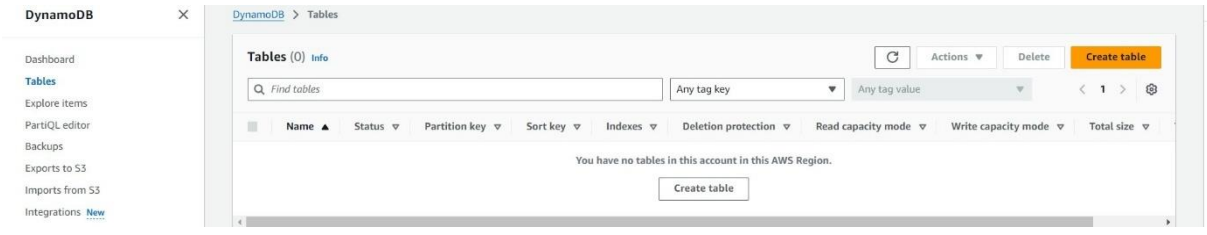
- **Activity 1.2: Log in to the AWS Management Console**

  - After setting up your account, log in to the AWS Management Console.

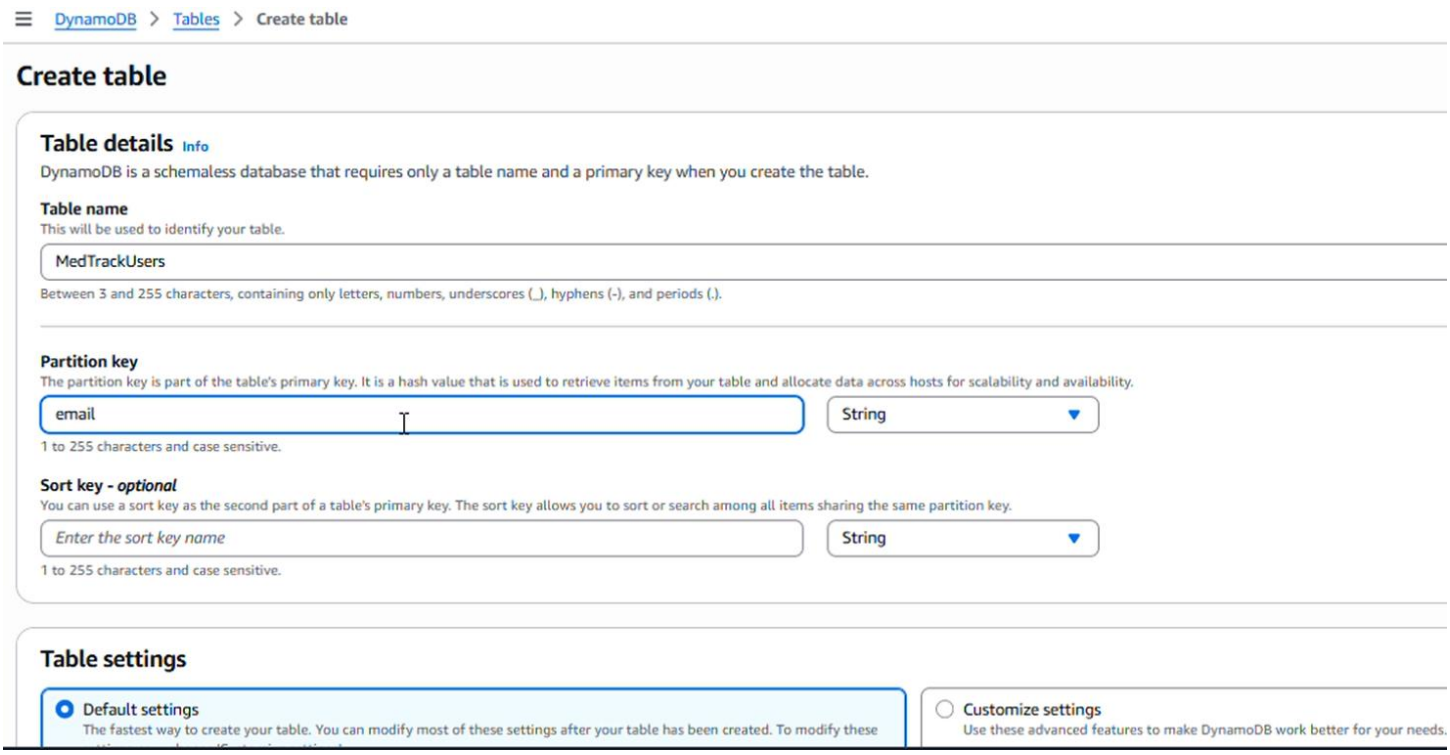


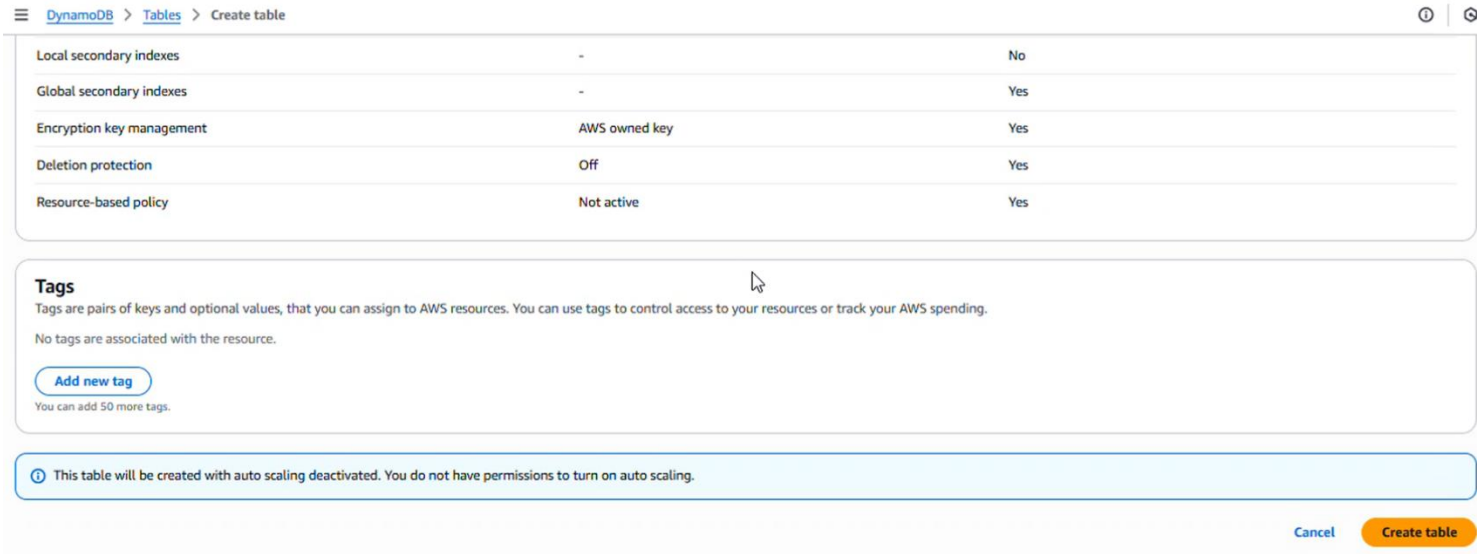
Milestone 2: DynamoDB Database Creation and Setup

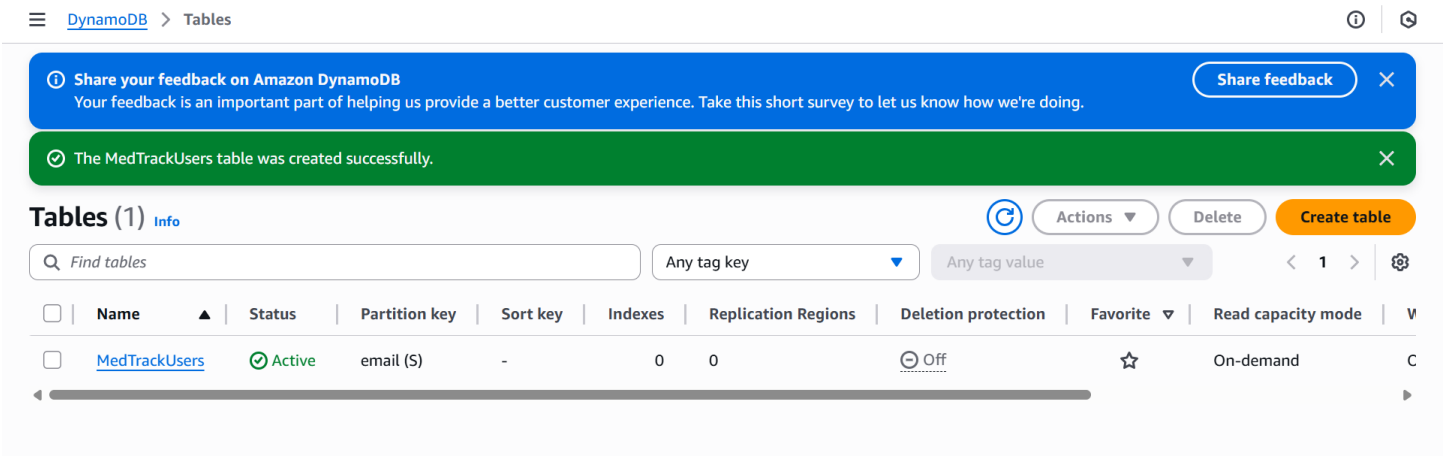
- **Activity 2.1: Navigate to the DynamoDB**
    - In the AWS Console, navigate to DynamoDB and click on create tables.
    - 



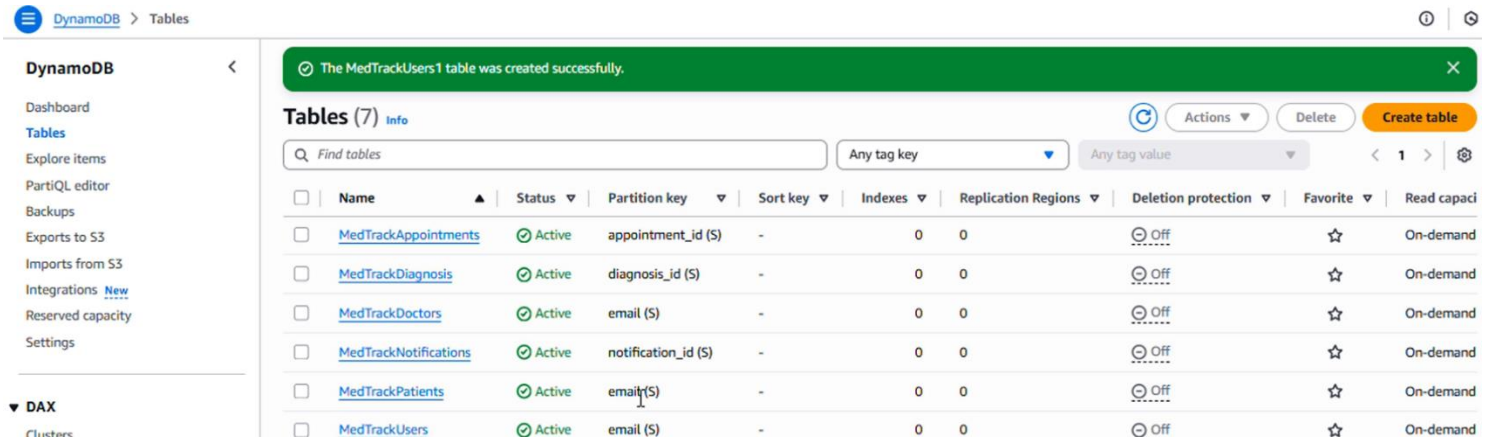
- **Activity 2.2: Create a DynamoDB table for storing data**
    - Create MedTrackUsers table with partition key “Email” with type String and click on create tables.







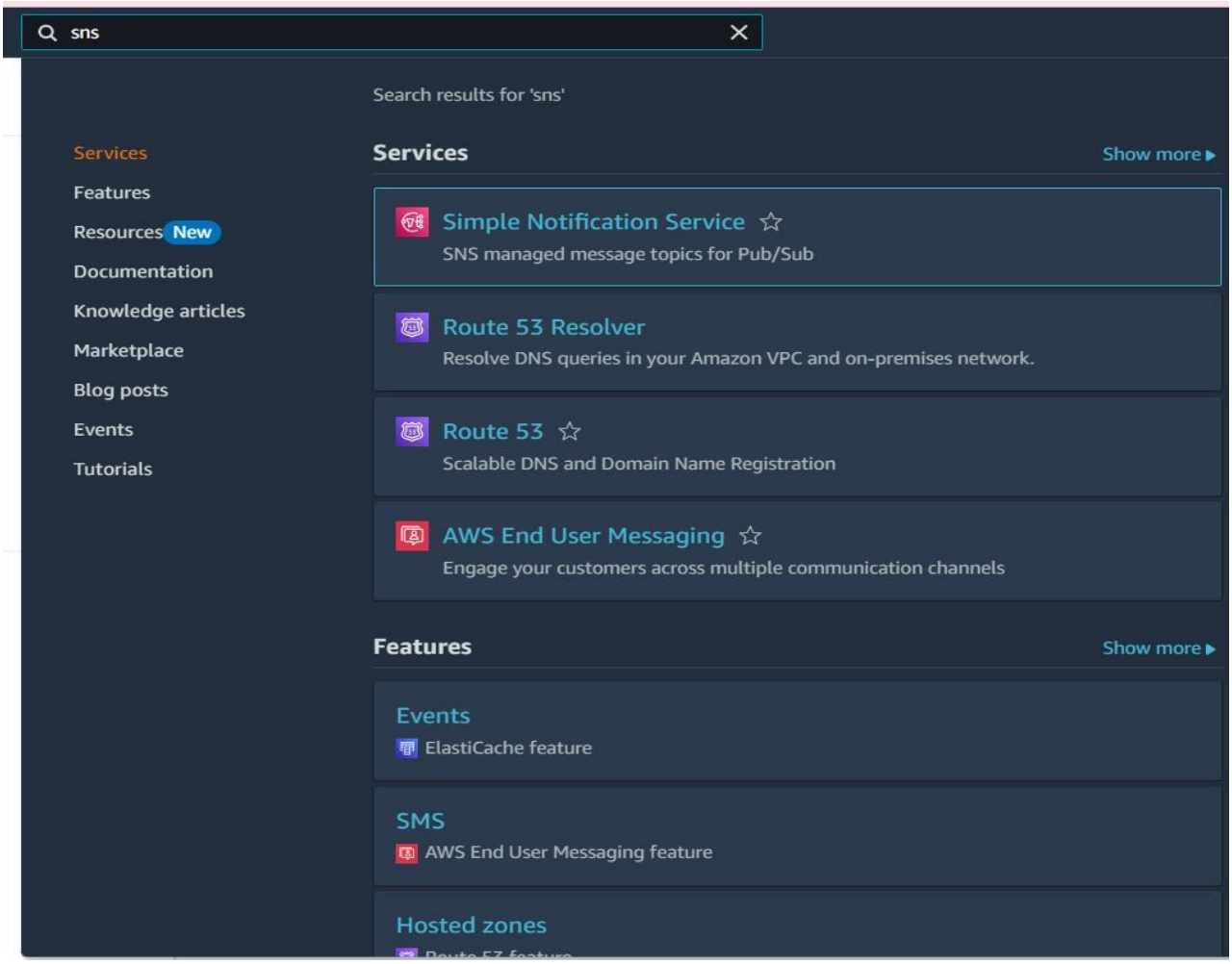
- Follow the same steps to create a remaining table with different partition key

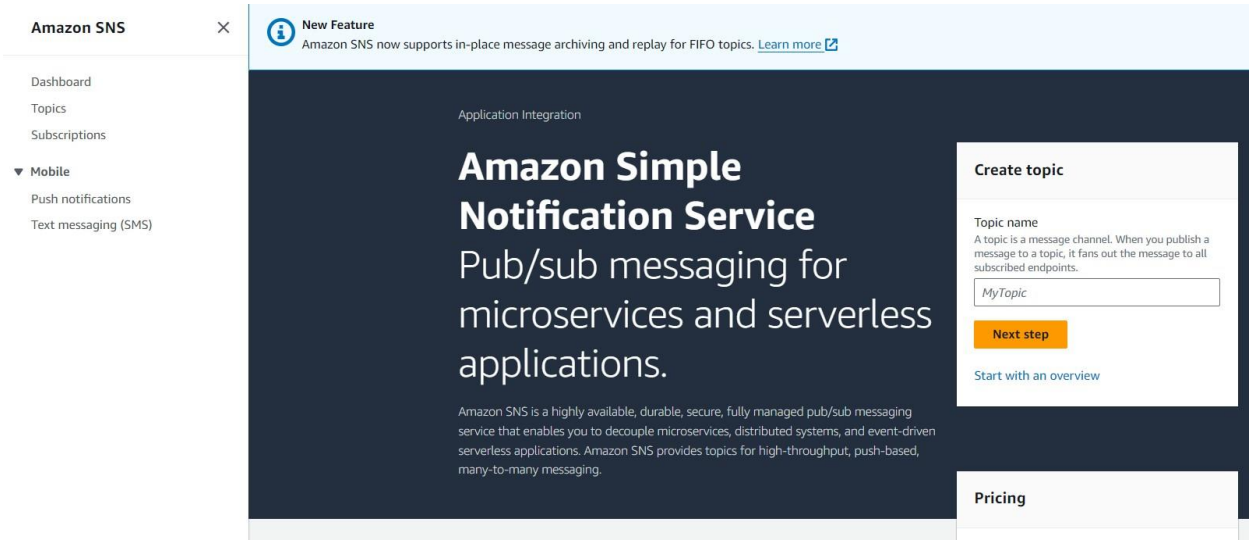


Milestone 3: SNS Notification Setup

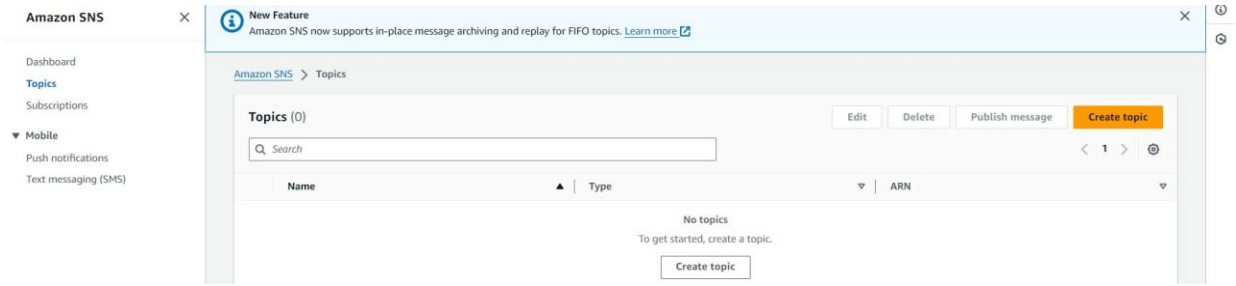
- **Activity 3.1: Create SNS topics to send email notifications to patients about their upcoming appointments.**

- In the AWS Console, search for SNS and navigate to the SNS Dashboard.

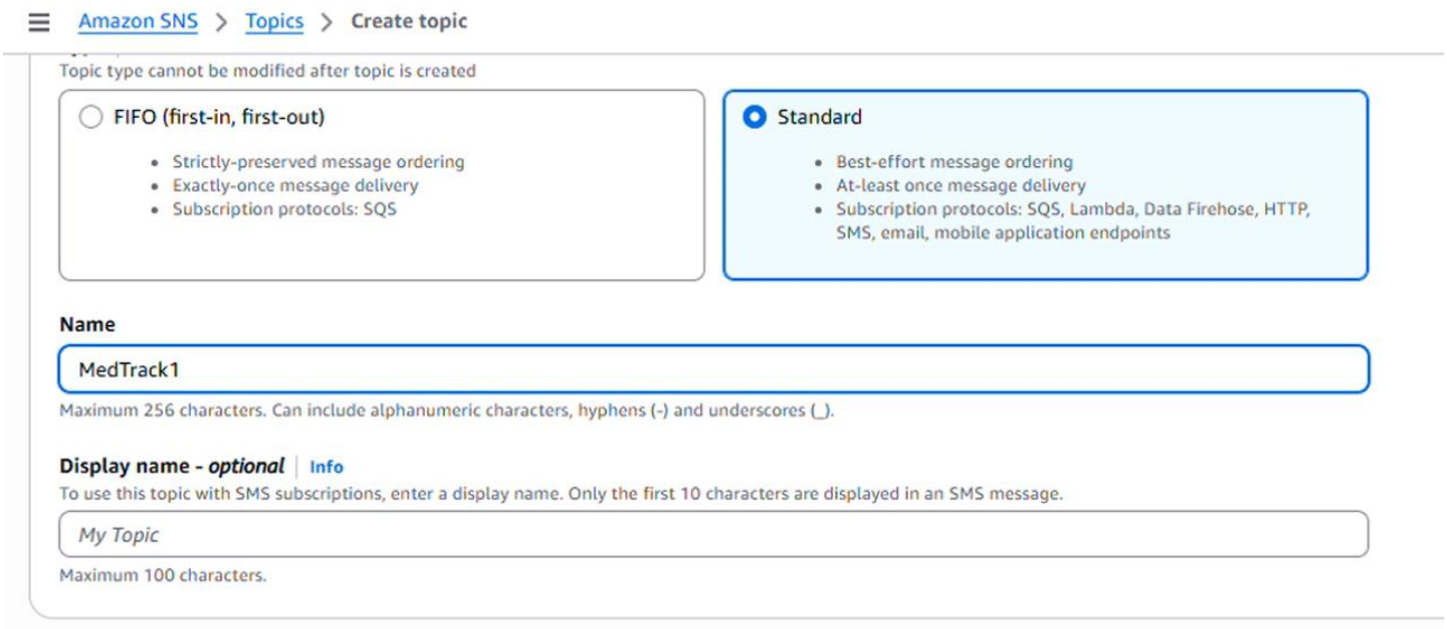

sns
Search results for 'sns'
Services
Features
Resources New
Documentation
Knowledge articles
Marketplace
Blog posts
Events
Tutorials
Services
Show more
Simple Notification Service
SNS managed message topics for Pub/Sub
Route 53 Resolver
Resolve DNS queries in your Amazon VPC and on-premises network.
Route 53
Scalable DNS and Domain Name Registration
AWS End User Messaging
Engage your customers across multiple communication channels
Features
Show more
Events
ElastiCache feature
SMS
AWS End User Messaging feature
Hosted zones

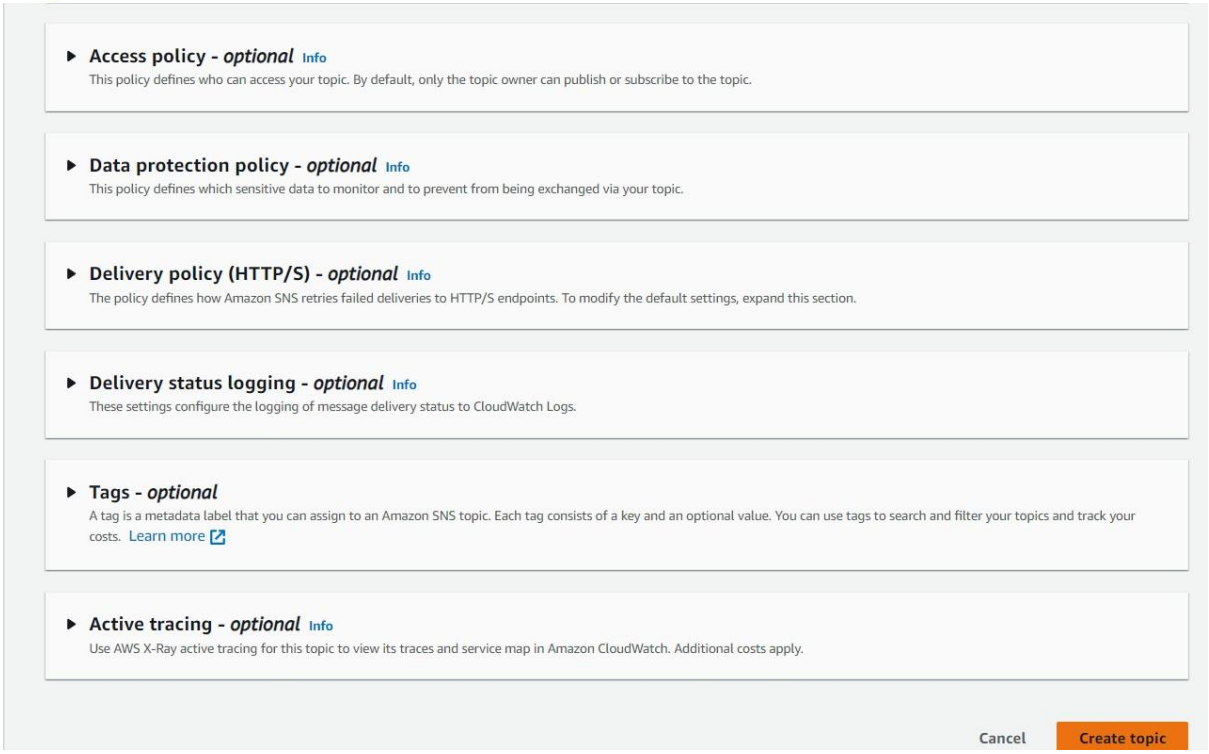

Amazon SNS
Dashboard
Topics
Subscriptions
Mobile
Push notifications
Text messaging (SMS)
New Feature
Amazon SNS now supports in-place message archiving and replay for FIFO topics. Learn more
Application Integration
Amazon Simple Notification Service
Pub/sub messaging for microservices and serverless applications.
Amazon SNS is a highly available, durable, secure, fully managed pub/sub messaging service that enables you to decouple microservices, distributed systems, and event-driven serverless applications. Amazon SNS provides topics for high-throughput, push-based, many-to-many messaging.
Create topic
Topic name
A topic is a message channel. When you publish a message to a topic, it fans out the message to all subscribed endpoints.
MyTopic
Next step
Start with an overview
Pricing

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

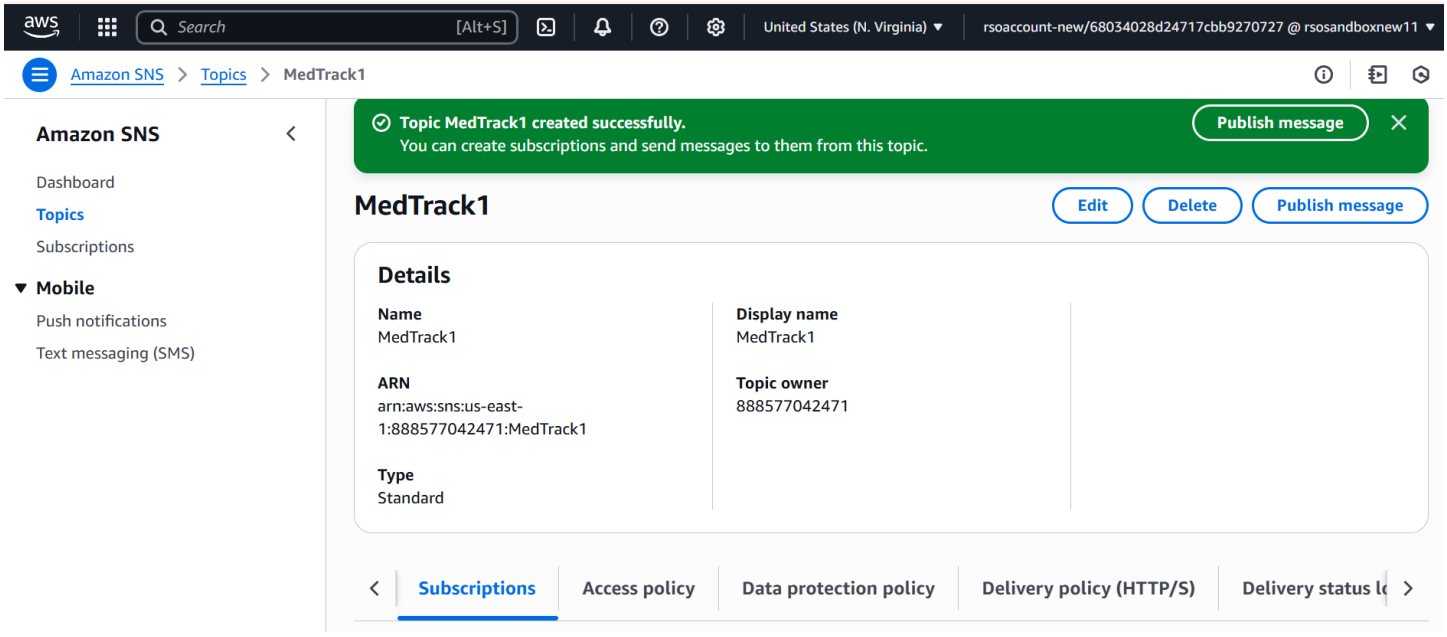


- Choose Standard type for general notification use cases and Click on Create Topic.



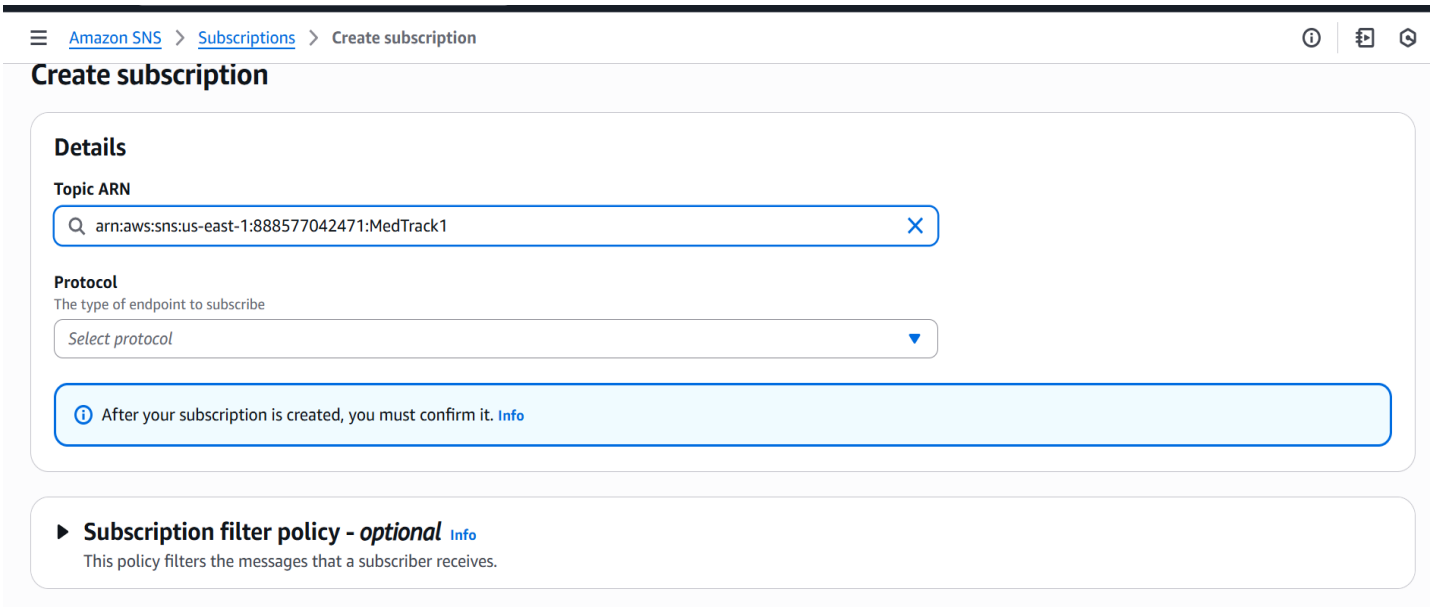


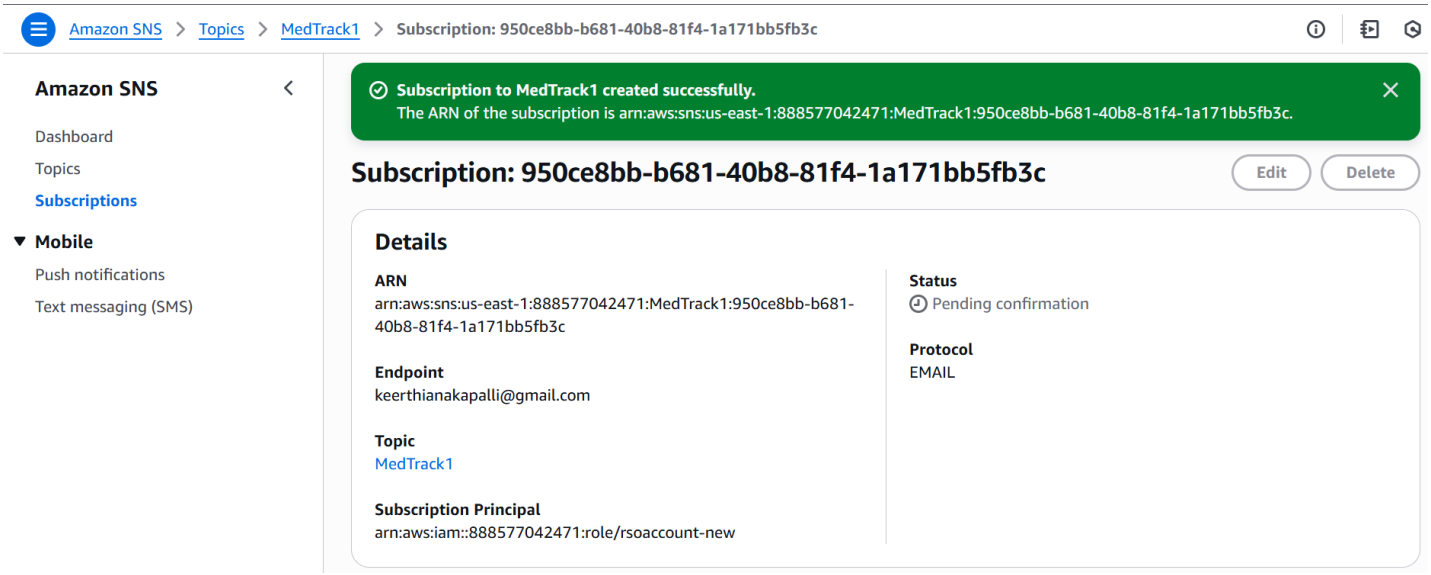
- Configure the SNS topic and note down the **Topic ARN**.



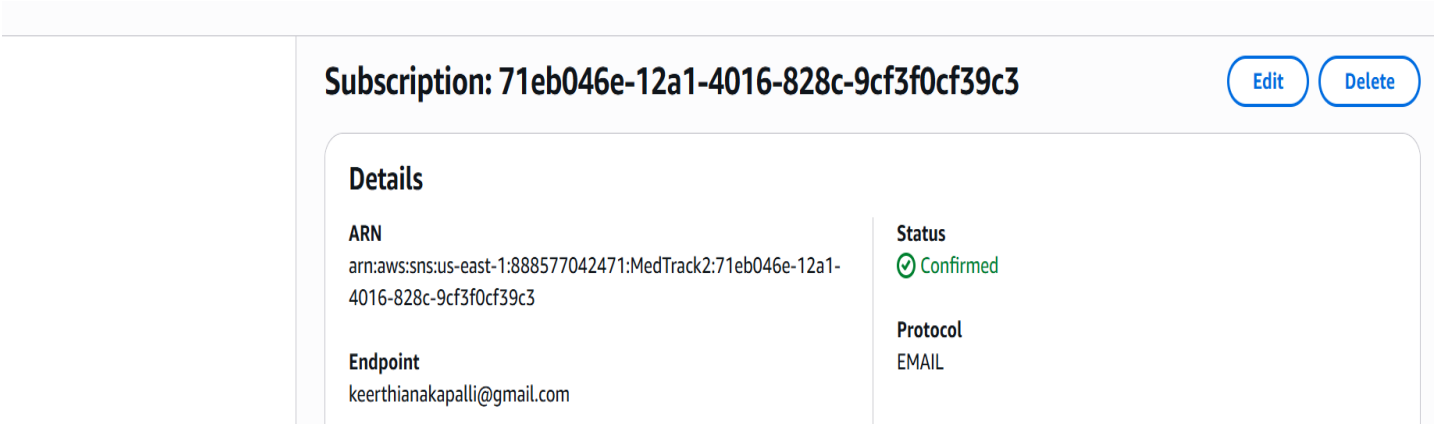
- **Activity 3.2: Subscribe users (or admin staff) to this topic via email. When a book request is made, notifications will be sent to the subscribed emails.**

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

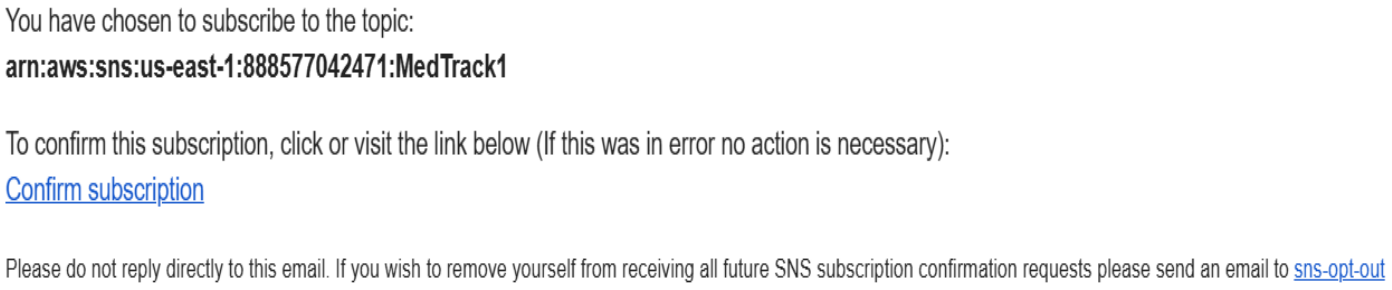




- After subscription request for the mail confirmation



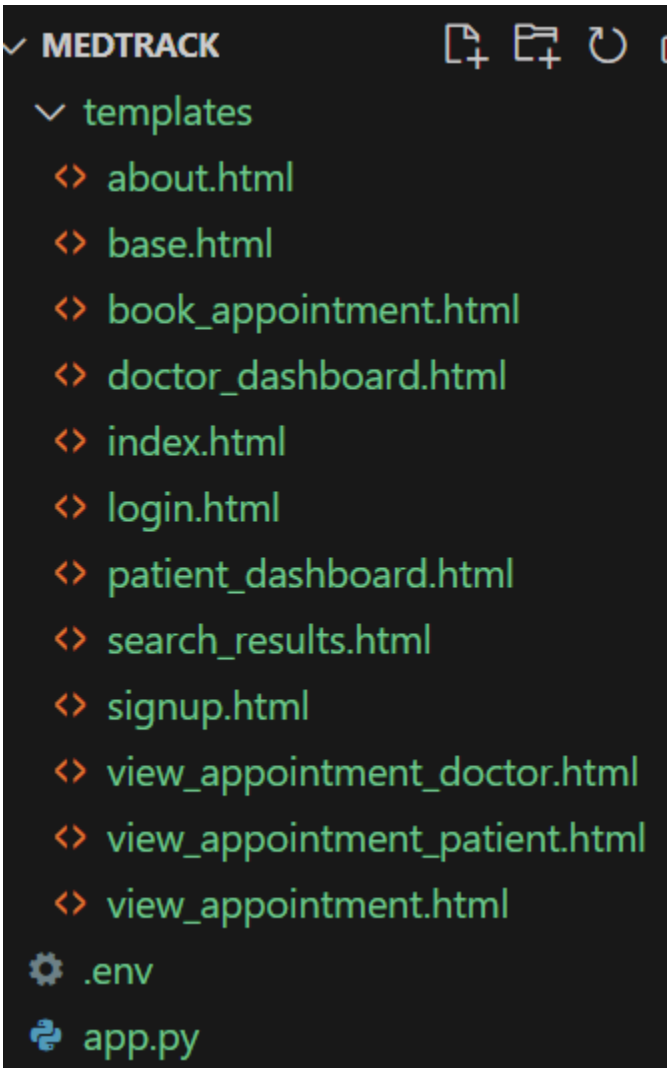
- Navigate to the subscribed Email account and Click on the confirm subscription in the AWS Notification- Subscription Confirmation mail.



- 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**



Description: set up the Med track project with an app.py file, a static/ folder for assets, and a templates/ directory containing all required HTML pages like base, login, signup, patient dashboard, doctor dashboard, book appointment etc.

Description of the code:

- **Flask App Initialization:**

  In the Med Track project, the Flask app is initialized to establish the backend infrastructure, enabling it to handle multiple user interactions such as patient registration, appointment booking, and submission of medical reports. The Flask framework processes incoming requests, communicates with the DynamoDB database for storing user data, and integrates seamlessly with AWS services. Additionally, the routes and APIs are defined to manage different functionalities like secure login, appointment scheduling, and medical history retrieval. This initialization sets up the foundation for smooth, real-time communication between patients and doctors while ensuring the app is scalable and secure.

```
from flask import Flask, request, jsonify, session, render_template, redirect, url_for, flash
from werkzeug.security import generate_password_hash, check_password_hash
from datetime import datetime, timedelta
from dotenv import load_dotenv
import boto3
import logging
import os
import uuid
from functools import wraps
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText


# ------------------------------------
# Load Environment Variables
# ------------------------------------
if not load_dotenv():
    print("Warning: .env file not found. Using default configurations.")

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)
```

Description: Import essential libraries including Flask utilities for routing, session management, and template rendering; Boto3 for interacting with AWS services like DynamoDB and SNS; smtplib and email modules for sending email notifications; uuid for generating unique identifiers; dotenv for loading environment variables; and Werkzeug for secure password hashing and verification.

```
# Flask App Initialization
# -------------------------
app = Flask(__name__)
```

Description: initialize the Flask application instance using Flask(__name__) to start building the web app.

- **Dynamodb Setup:**

```
USERS_TABLE_NAME = os.environ.get('USERS_TABLE_NAME', 'MedTrackUsers')
DOCTORS_TABLE_NAME = os.environ.get('DOCTORS_TABLE_NAME', 'MedTrackDoctors')
PATIENTS_TABLE_NAME = os.environ.get('PATIENTS_TABLE_NAME', 'MedTrackPatients')
APPOINTMENTS_TABLE_NAME = os.environ.get('APPOINTMENTS_TABLE_NAME', 'MedTrackAppointments')
DIAGNOSIS_TABLE_NAME = os.environ.get('DIAGNOSIS_TABLE_NAME', 'MedTrackDiagnosis')
NOTIFICATIONS_TABLE_NAME = os.environ.get('NOTIFICATIONS_TABLE_NAME', 'MedTrackNotifications')
```

Description: Initialize the DynamoDB resource in the ap-south-1 region and set up access to the Users, Doctors, Patients, Appointments, Diagnosis, and Notifications tables for storing and managing user details, doctor information, patient data, appointment bookings, medical diagnoses, and notification records in the MedTrack system

- **SNS Connection**

```
# Email Configuration
SMTP_SERVER = os.environ.get('SMTP_SERVER', 'smtp.gmail.com')
SMTP_PORT = int(os.environ.get('SMTP_PORT', 587))
SENDER_EMAIL = os.environ.get('SENDER_EMAIL')
SENDER_PASSWORD = os.environ.get('SENDER_PASSWORD')
ENABLE_EMAIL = os.environ.get('ENABLE_EMAIL', 'False').lower() == 'true'
```

```
# SNS Configuration
SNS_TOPIC_ARN = os.environ.get('SNS_TOPIC_ARN')
ENABLE_SNS = os.environ.get('ENABLE_SNS', 'False').lower() == 'true'
```

```
def send_email_notification(to_email, subject, body):
    if not ENABLE_EMAIL or not SENDER_EMAIL:
        logger.info(f"Email notification would be sent: {subject}")
        return True

    try:
        msg = MIMEMultipart()
        msg['From'] = SENDER_EMAIL
        msg['To'] = to_email
        msg['Subject'] = subject
        msg.attach(MIMEText(body, 'plain'))

        server = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)
        server.starttls()
        server.login(SENDER_EMAIL, SENDER_PASSWORD)
        server.send_message(msg)
        server.quit()

        logger.info(f"Email sent successfully to {to_email}")
        return True
    except Exception as e:
        logger.error(f"Failed to send email: {e}")
        return False
```

Description: Configure Amazon SNS to send notifications (e.g., for appointments or medication alerts). Paste your stored SNS Topic ARN in the sns_topic_arn variable and specify the region name where the topic is created (e.g., ap-south-1). Set the email service provider in SMTP_SERVER (e.g., Gmail or Outlook), and provide the sender's subscribed email address in SENDER_EMAIL. Generate an app-specific password for the email account and store it in SENDER_PASSWORD to enable secure authentication for sending emails via SMTP.

- **Routes for Web Pages**
  - **Index Route:**

```
# Route: Select role
@app.route('/')
def index():
    return render_template('index.html')
```

Description: define the index route / to automatically redirect users to the index page when they access the base URL.

- **Signup Route:**

```
@app.route('/signup/<role>', methods=['GET', 'POST'])
def signup(role):
    if role not in ('patient', 'doctor'):
        flash('Invalid role selected.', 'danger')
        return redirect(url_for('index'))

    if request.method == 'POST':
        name = request.form.get('name')
        email = request.form.get('email')
        password = request.form.get('password')

        if not name or not email or not password:
            flash('All fields are required.', 'warning')
            return render_template('signup.html', role=role)

        # Check if user already exists
        if dynamodb:
            user_table = get_user_table()
            response = user_table.get_item(Key={'email': email})
            if 'Item' in response:
                flash('User already exists.', 'danger')
                return render_template('signup.html', role=role)
```

```
    user_id = str(uuid.uuid4())
    hashed_password = generate_password_hash(password)
    user_data = {
        'user_id': user_id,
        'name': name,
        'email': email,
        'password_hash': hashed_password,
        'role': role,
        'created_at': datetime.now().isoformat(),
        'is_active': True
    }

    if dynamodb:
        user_table.put_item(Item=user_data)
    else:
        local_db['users'][email] = user_data

    flash('Signup successful! Please log in.', 'success')
    return redirect(url_for('login', role=role))

return render_template('signup.html', role=role)
```

Description: define the /signup/<role> route to handle registration for both patients and doctors. It validates user input, checks if the user already exists in the DynamoDB Users table (or local DB fallback), securely hashes the password using Werkzeug, generates a unique user ID, and stores the new user record along with metadata like creation time and role. On successful signup, it flashes a success message and redirects to the login page.

- **login Route (GET/POST)**:

```python
@app.route('/login/<role>', methods=['GET', 'POST'])
def login(role):
    if role not in ('patient', 'doctor'):
        flash("Invalid role.", "danger")
        return redirect(url_for('index'))

    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')

        if not email or not password:
            flash('Email and password are required.', 'warning')
            return render_template('login.html', role=role)

        # Rate-limiting check
        client_ip = request.remote_addr
        if client_ip in login_attempts:
            attempt_data = login_attempts[client_ip]
            if attempt_data['count'] >= 5:
                if datetime.now() - attempt_data['last_attempt'] < timedelta(minutes=15):
                    flash('Too many login attempts. Try again later.', 'danger')
                    return render_template('login.html', role=role)
                else:
                    login_attempts[client_ip] = {'count': 0, 'last_attempt': datetime.now()}
```

```python
        # Fetch user data
        user_data = None
        if dynamodb:
            user_table = get_user_table()
            try:
                response = user_table.get_item(Key={'email': email})
                if 'Item' in response:
                    user_data = response['Item']
            except Exception as e:
                logger.error(f"Error fetching user from DynamoDB: {e}")
        else:
            user_data = local_db['users'].get(email)

        # Validate user
        if not user_data or user_data.get('role') != role:
            login_attempts.setdefault(client_ip, {'count': 0, 'last_attempt': datetime.now()})
            login_attempts[client_ip]['count'] += 1
            login_attempts[client_ip]['last_attempt'] = datetime.now()
            flash('Invalid email or role.', 'danger')
            return render_template('login.html', role=role)

        if not check_password_hash(user_data.get('password_hash', ''), password):
            login_attempts.setdefault(client_ip, {'count': 0, 'last_attempt': datetime.now()})
            login_attempts[client_ip]['count'] += 1
```

```
            login_attempts[client_ip]['last_attempt'] = datetime.now()
            flash('Incorrect password.', 'danger')
            return render_template('login.html', role=role)
        # Reset rate-limiting counter
    login_attempts.pop(client_ip, None)

    session['user'] = user_data['email']   # This is what your @login_required checks!
    session['user_id'] = user_data['user_id']
    session['email'] = user_data['email']
    session['name'] = user_data['name']
    session['role'] = user_data['role']
    session.permanent = True

    logger.info(f"{role.capitalize()} logged in: {email}")
    flash('Login successful!', 'success')

    # Redirect to appropriate dashboard
    print("Redirecting to:", url_for(f"{role}_dashboard"))

    return redirect(url_for(f"{role}_dashboard"))

return render_template('login.html', role=role)
```

Description: define the /login/<role> route to handle login functionality for both patients and doctors. The route validates the user's email and password, checks for role mismatches, and verifies the password using Werkzeug. A rate-limiting mechanism restricts repeated failed login attempts based on IP address. On successful authentication, the user's session is initialized, login attempts are reset, and the user is redirected to their respective dashboard.

- **Book appointment, patient dashboard and doctor dashboard routes:**

```
@app.route('/patient_dashboard')
@login_required(role='patient')
def patient_dashboard():
    user_email = session['user']
    dashboard_data = get_patient_dashboard_data(user_email)
    doctors = [
        {'email': 'drsmith@example.com', 'name': 'Dr. Smith'},
        {'email': 'drjohnson@example.com', 'name': 'Dr. Johnson'},
        {'email': 'saikiran@gmail.com', 'name': 'Dr. Sai'},
        # ... more doctors ...
    ]
    return render_template('patient_dashboard.html', **dashboard_data, doctors=doctors)

@app.route('/doctor_dashboard')
@login_required(role='doctor')
def doctor_dashboard():
    user_email = session['user']
    dashboard_data = get_doctor_dashboard_data(user_email)
    return render_template('doctor_dashboard.html', **dashboard_data)
```

```python
@app.route('/book_appointment', methods=['GET', 'POST'])
@login_required(role='patient')
def book_appointment():
    if request.method == 'POST':
        doctor_email = request.form['doctor']
        date = request.form['date']
        time = request.form['time']
        title = request.form.get('title', 'Consultation')
        location = request.form.get('location', 'Office 203')
        color = request.form.get('color', '#3498db')

        # Get doctor and patient names from local_db
        doctor_name = local_db['users'].get(doctor_email, {}).get('name', 'Doctor')
        patient_email = session['user']
        patient_name = local_db['users'].get(patient_email, {}).get('name', 'Patient')

        appointment = {
            'patient': patient_email,
            'patient_name': patient_name,
            'doctor': doctor_email,
            'doctor_name': doctor_name,
            'title': title,
            'date': date,
            'time': time,
            'location': location,
            'color': color
        }
        local_db['appointments'].append(appointment)
        flash('Appointment booked successfully!', 'success')
        return redirect(url_for('patient_dashboard'))
    return render_template('book_appointment.html', user_name=session['name'])
```

Description: define the /book-appointment route to allow patients to schedule appointments with doctors by selecting from available options and submitting a form. The appointment details are stored in the Appointments DynamoDB table, and notifications may be triggered upon successful booking. The /patient-dashboard route displays personalized patient data including upcoming appointments, prescriptions, and medical history.The /doctor-dashboard route provides doctors with access to view and manage appointments, patient records, and diagnosis details.

Exit Route:

```
# Exit Page
@app.route('/exit')
def exit_page():
    return render_template('exit.html')
```

Description: define /exit route to render the exit.html page when the user chooses to leave or close the application.

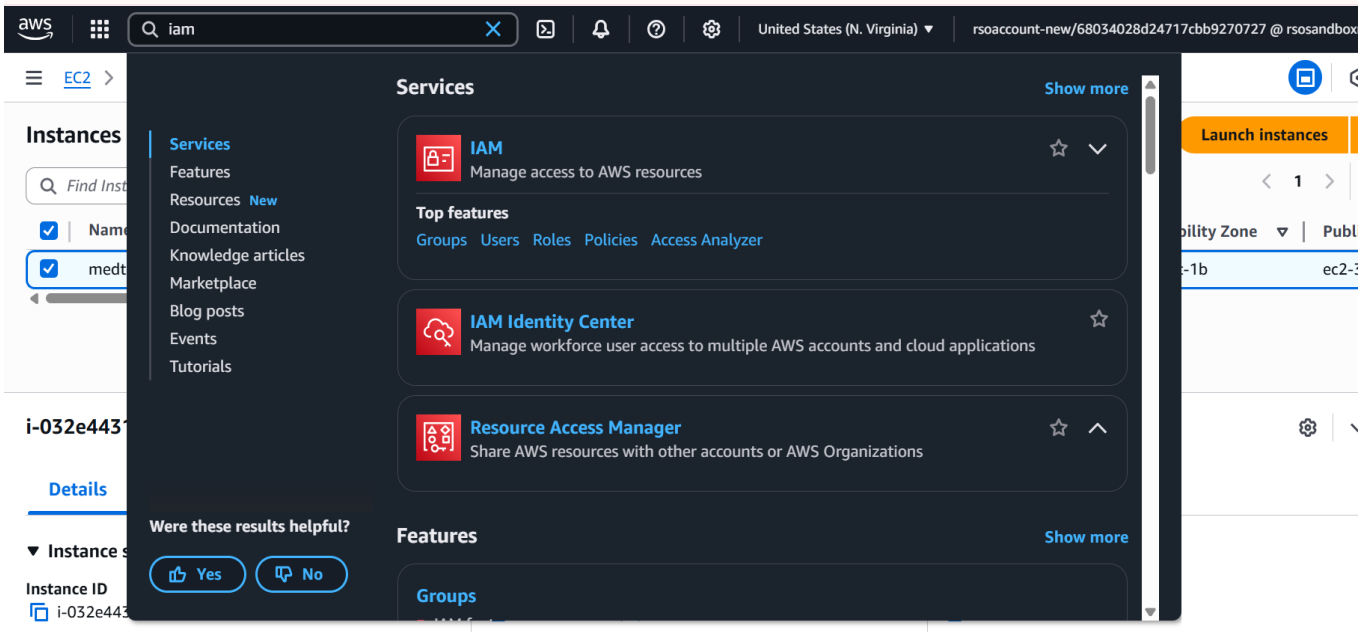
Deployment Code:

```
if __name__ == '__main__':
    app.run(debug=True)
```

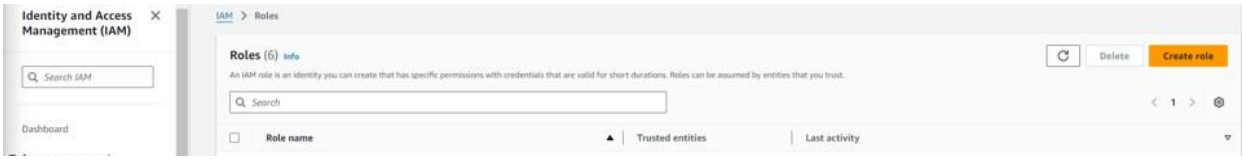
Description: This block checks if the script is being run directly. If so, it starts the Flask development server with debug=True, enabling live reload and detailed error messages for development and testing purposes.

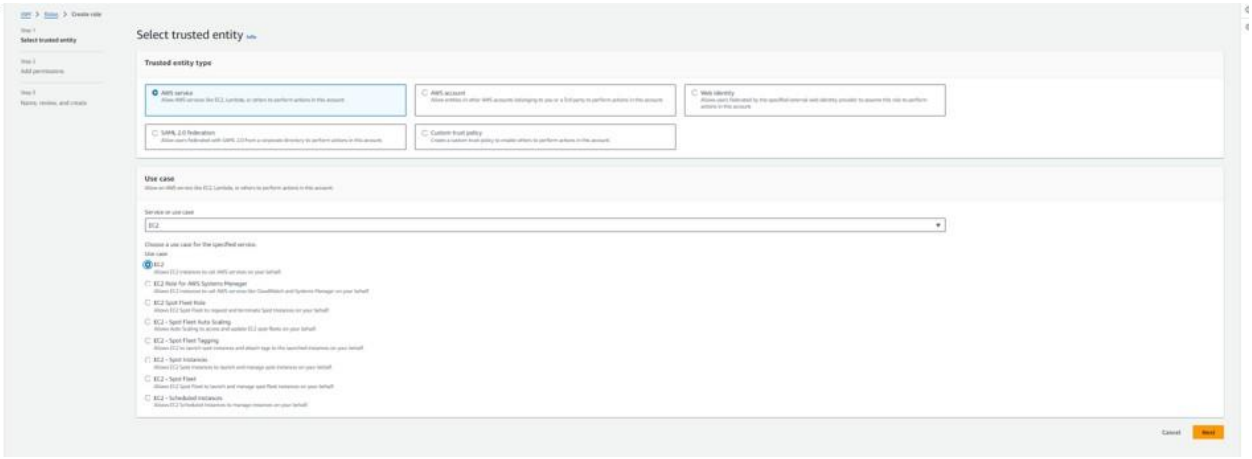
Milestone 5: IAM Role Setup

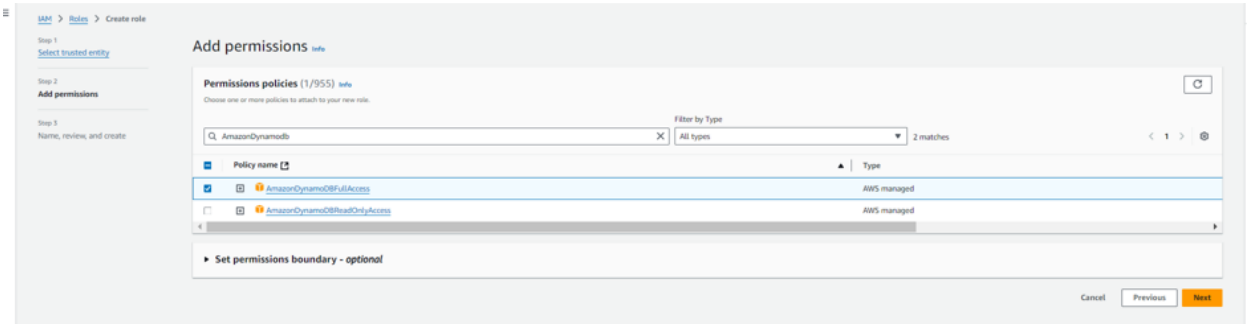
- **Activity 5.1: Create IAM Role.**
    - In the AWS Console, go to IAM and create a new IAM Role for EC2 to interact with DynamoDB and SNS.



-



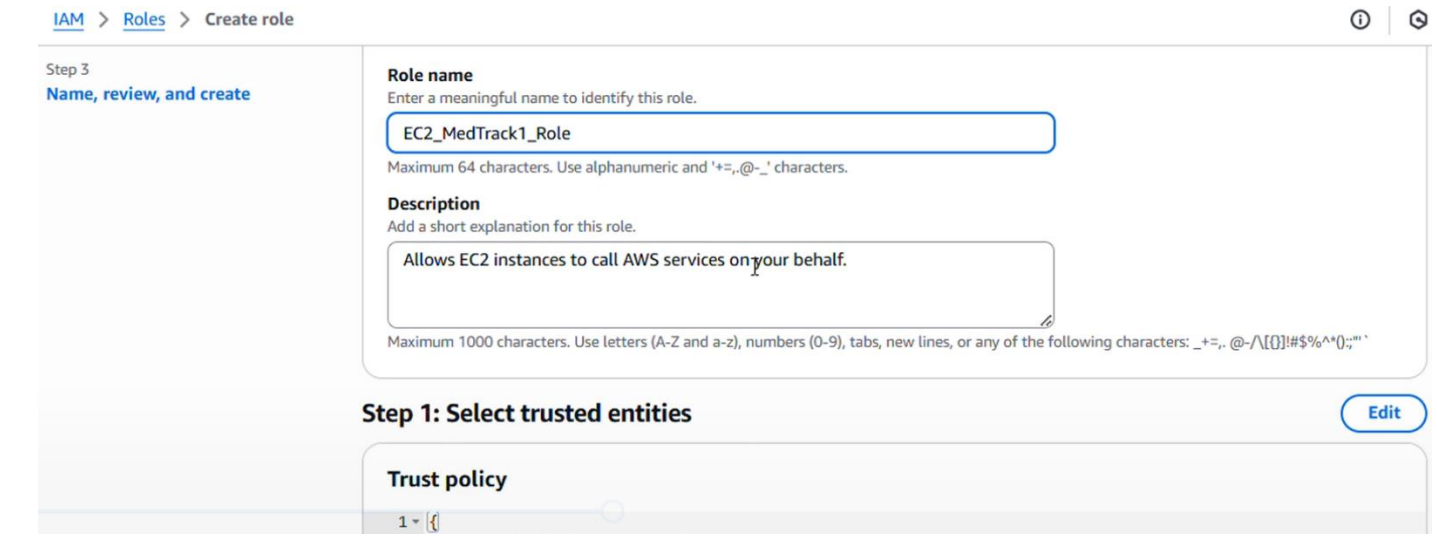


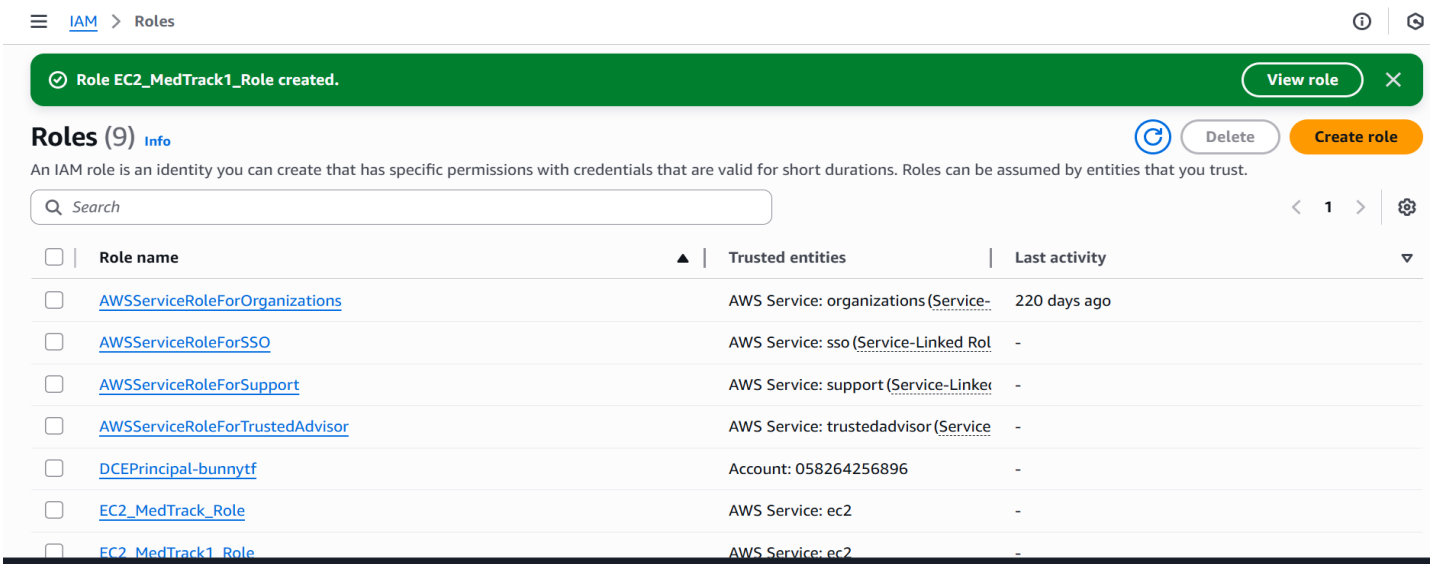


- **Activity 5.2: Attach Policies.**

  Attach the following policies to the role:

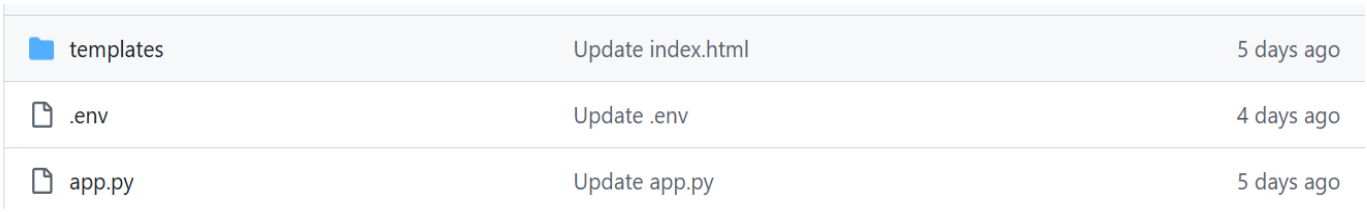
  - AmazonDynamoDBFullAccess: Allows EC2 to perform read/write operations on DynamoDB.
  - AmazonSNSFullAccess: Grants EC2 the ability to send notifications via SNS.

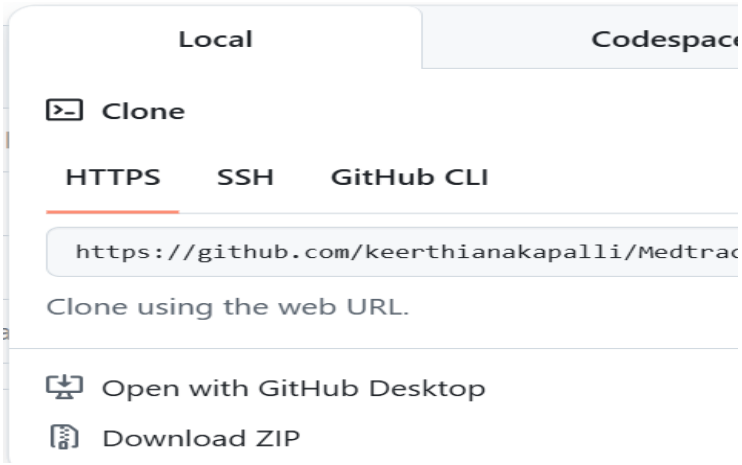




Milestone 6: EC2 Instance Setup

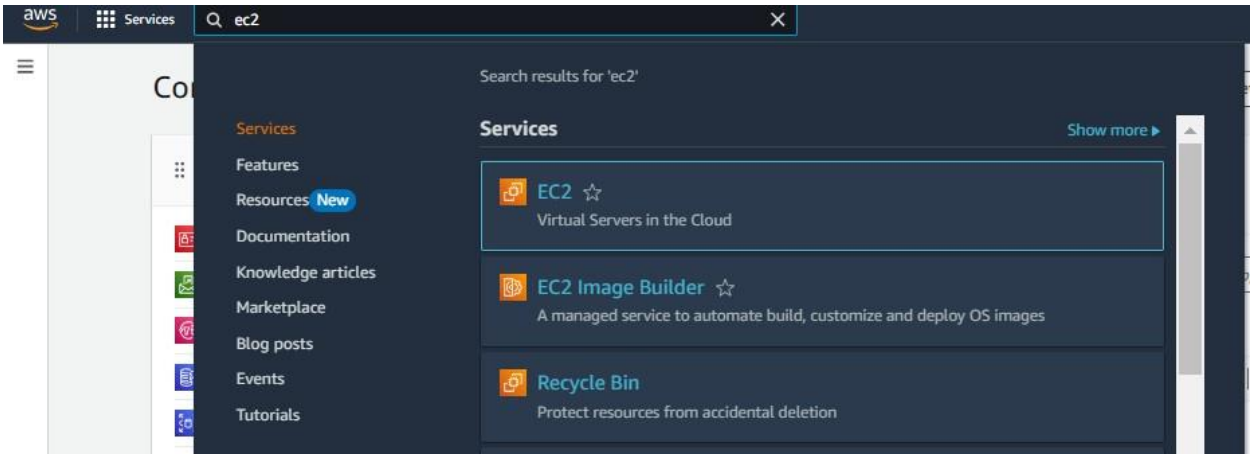
- **Note: Load your Flask app and Html files into GitHub repository.**



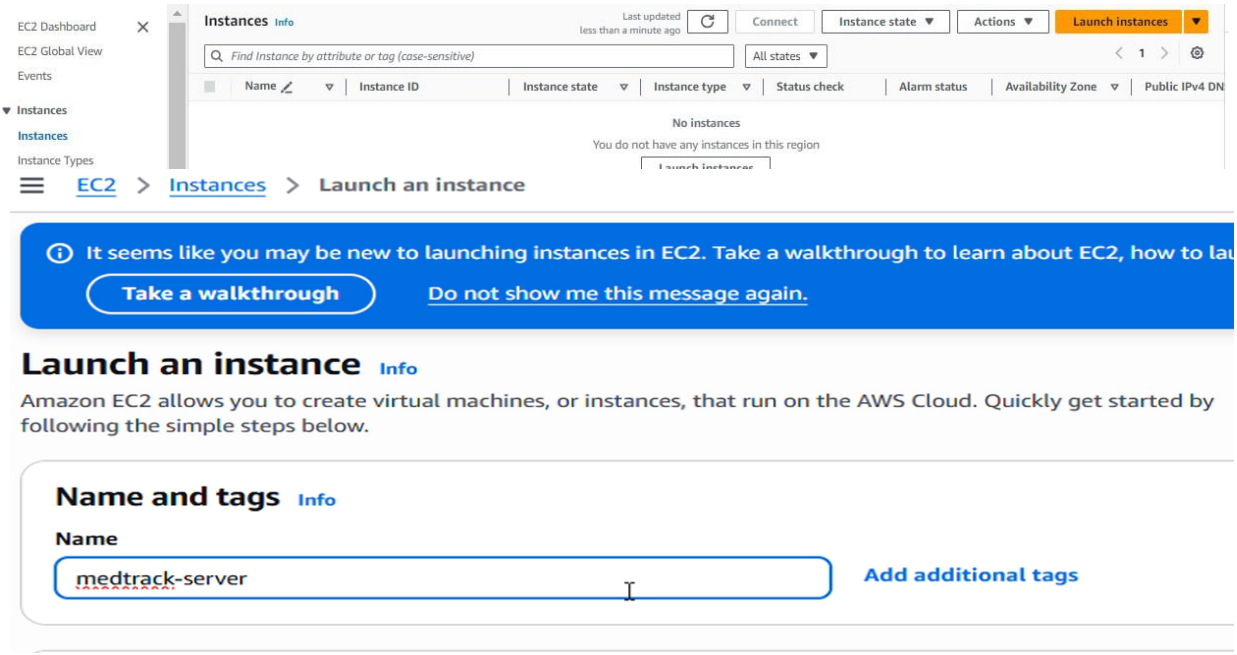


- **Activity 6.1: Launch an EC2 instance to host the Flask application.**

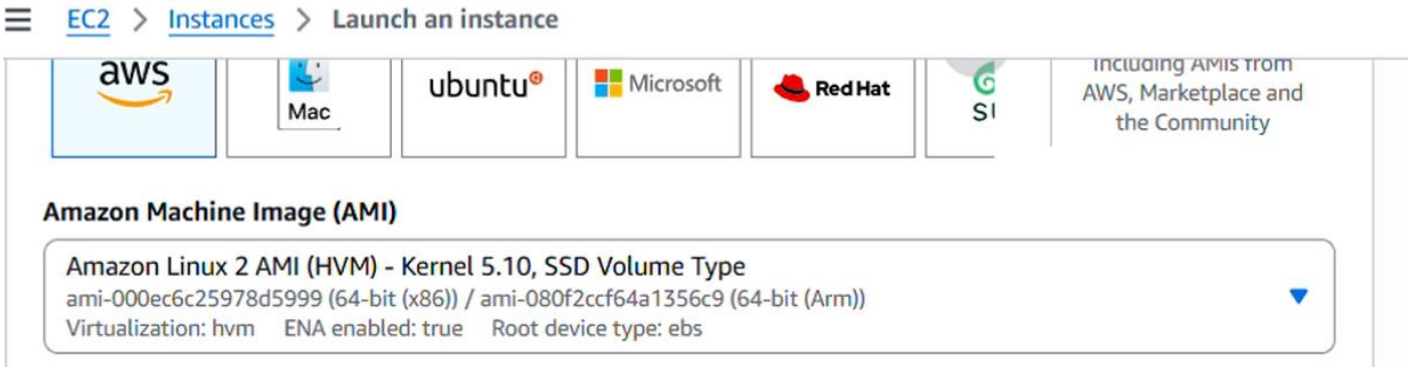
    - **Launch EC2 Instance**
        - In the AWS Console, navigate to EC2 and launch a new instance.

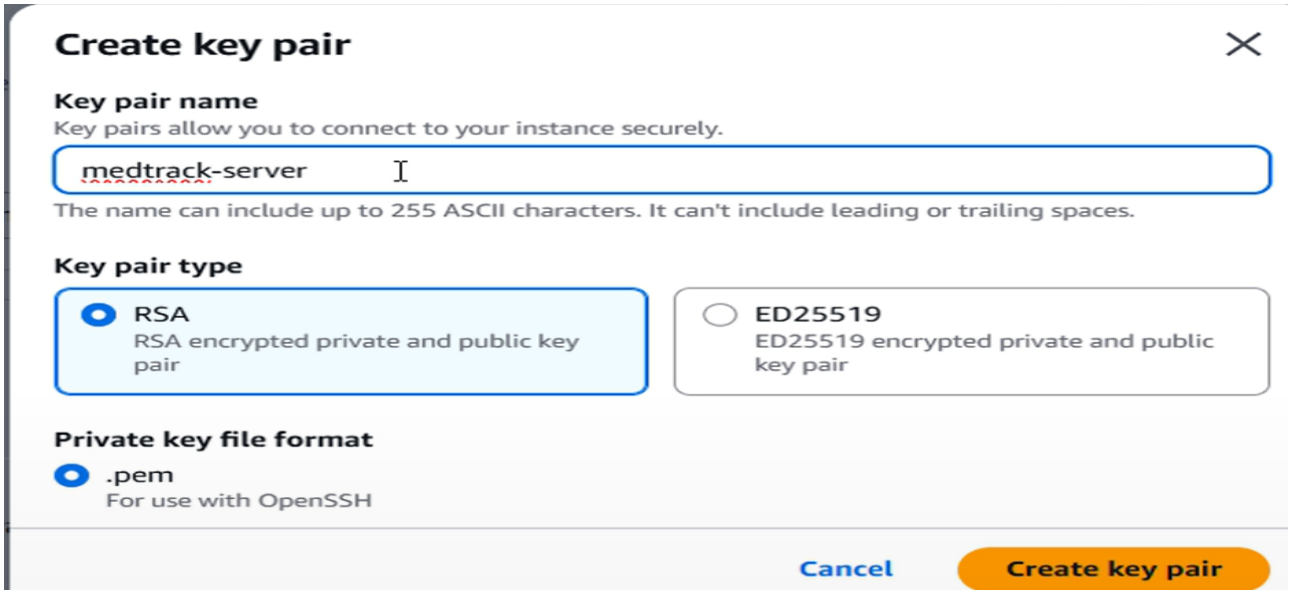


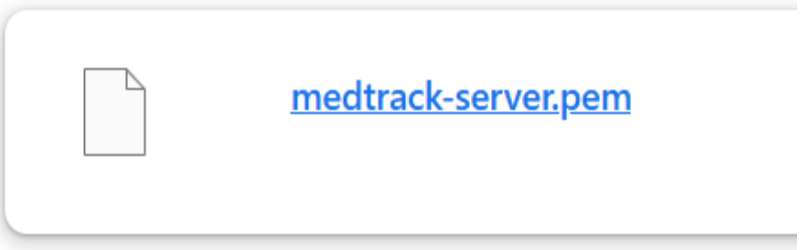
-
-
-
-
-

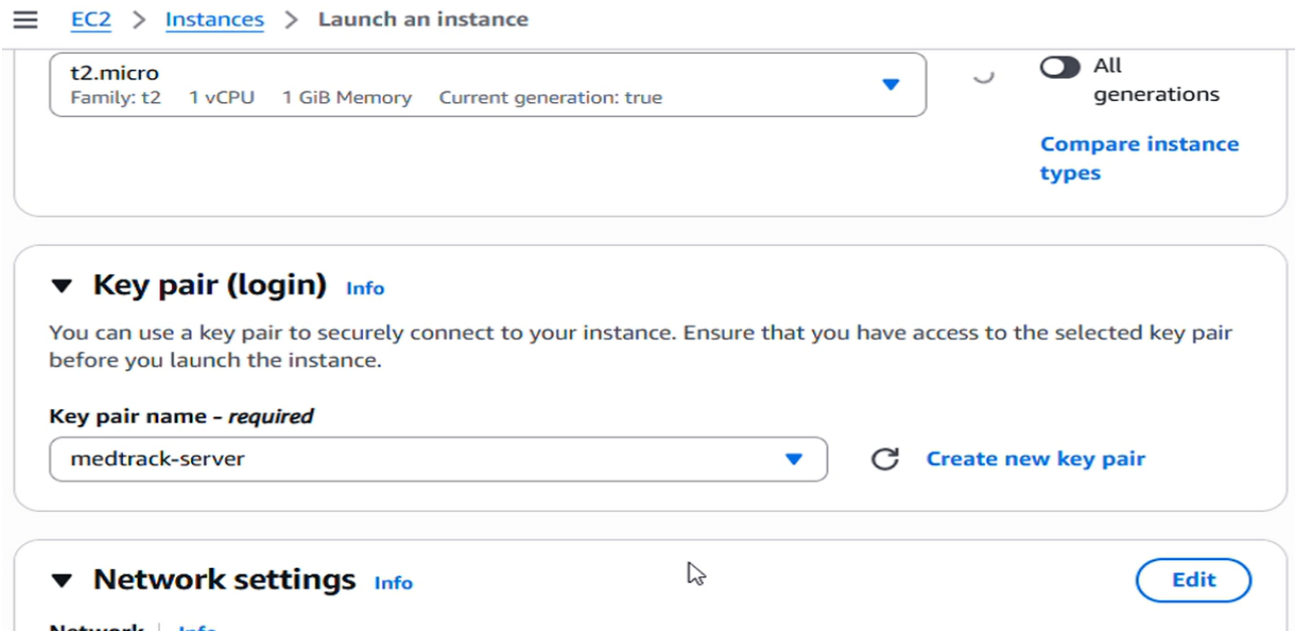


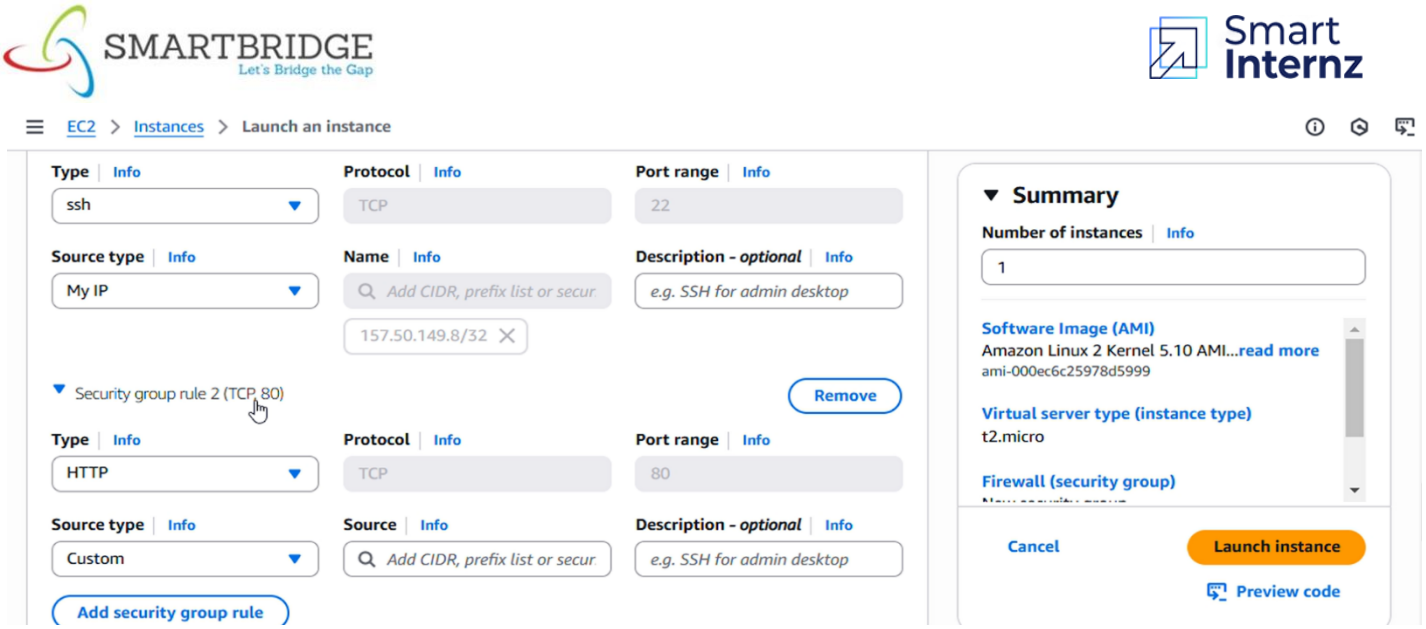
- Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).

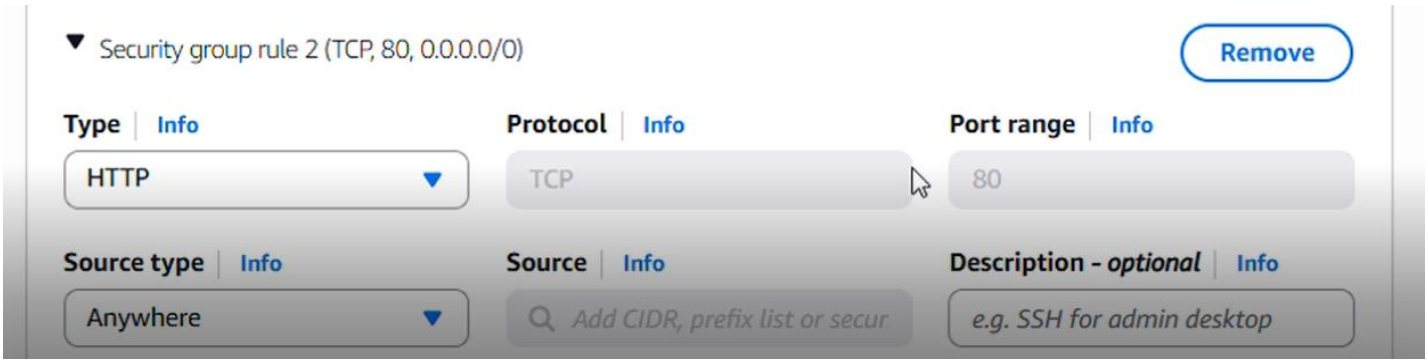


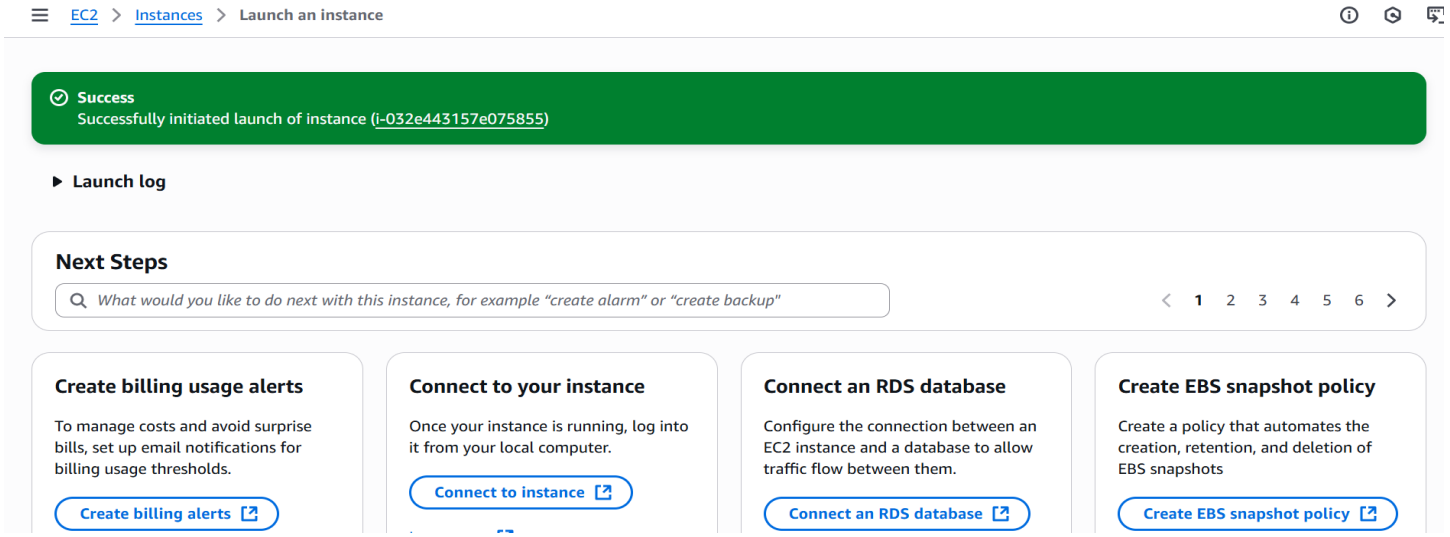

Create key pair
Key pair name
Key pairs allow you to connect to your instance securely.
medtrack-server
The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.
Key pair type
RSA
RSA encrypted private and public key pair
ED25519
ED25519 encrypted private and public key pair
Private key file format
.pem
For use with OpenSSH
Cancel
Create key pair


medtrack-server.pem


EC2 > Instances > Launch an instance
t2.micro
Family: t2 1 vCPU 1 GiB Memory Current generation: true
All generations
Compare instance types
Key pair (login) Info
You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.
Key pair name - required
medtrack-server
Create new key pair
Network settings Info
Edit

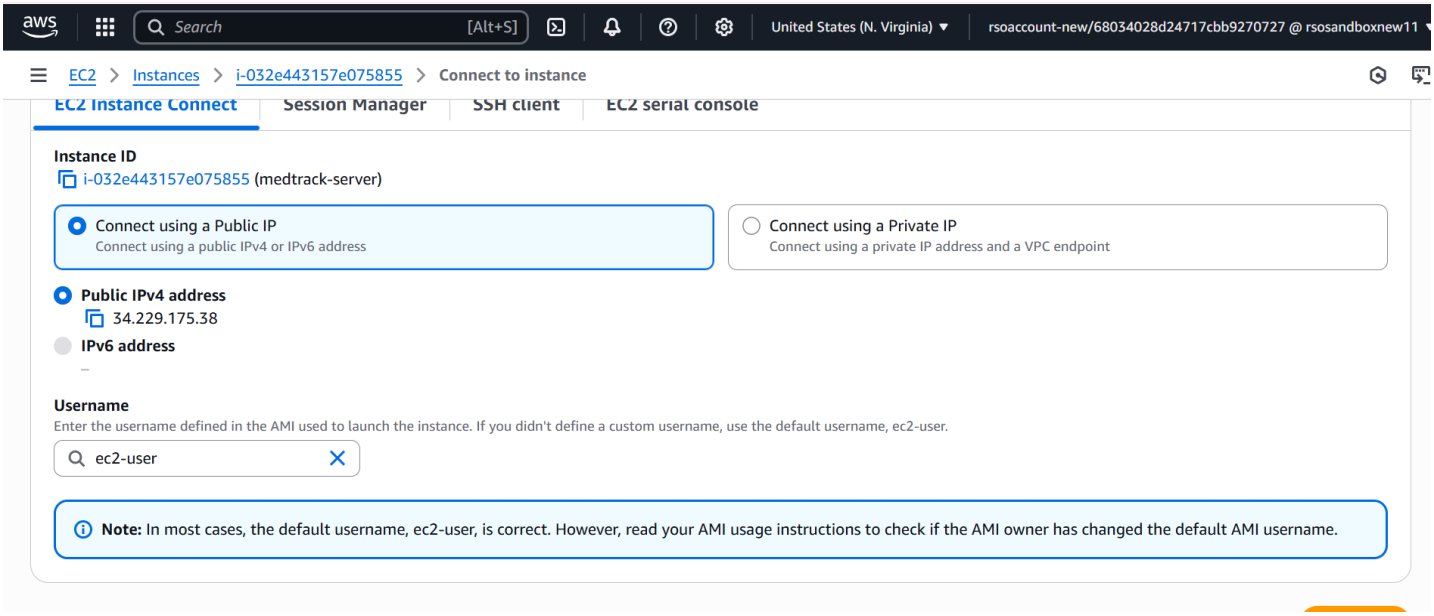

SMARTBRIDGE
Let's Bridge the Gap
Smart Internz
EC2 > Instances > Launch an instance
Type | Info
ssh
Protocol | Info
TCP
Port range | Info
22
Source type | Info
My IP
Name | Info
Add CIDR, prefix list or secur
Description - optional | Info
e.g. SSH for admin desktop
157.50.149.8/32
Security group rule 2 (TCP, 80)
Remove
Type | Info
HTTP
Protocol | Info
TCP
Port range | Info
80
Source type | Info
Custom
Source | Info
Add CIDR, prefix list or secur
Description - optional | Info
e.g. SSH for admin desktop
Add security group rule
Summary
Number of instances | Info
1
Software Image (AMI)
Amazon Linux 2 Kernel 5.10 AMI...read more
ami-000ec6c25978d5999
Virtual server type (instance type)
t2.micro
Firewall (security group)
Cancel
Launch instance
Preview code


Security group rule 2 (TCP, 80, 0.0.0.0/0)
Remove
Type | Info
HTTP
Protocol | Info
TCP
Port range | Info
80
Source type | Info
Anywhere
Source | Info
Add CIDR, prefix list or secur
Description - optional | Info
e.g. SSH for admin desktop


EC2 > Instances > Launch an instance
Success
Successfully initiated launch of instance (i-032e443157e075855)
Launch log
Next Steps
What would you like to do next with this instance, for example "create alarm" or "create backup"
1 2 3 4 5 6
Create billing usage alerts
To manage costs and avoid surprise bills, set up email notifications for billing usage thresholds.
Create billing alerts
Connect to your instance
Once your instance is running, log into it from your local computer.
Connect to instance
Connect an RDS database
Configure the connection between an EC2 instance and a database to allow traffic flow between them.
Connect an RDS database
Create EBS snapshot policy
Create a policy that automates the creation, retention, and deletion of EBS snapshots
Create EBS snapshot policy

- 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 terminal will open, allowing you to access your EC2 instance directly from your browser.

- Now connect the EC2 with the files



Milestone 7: Deployment on EC2

Activity 7.1: 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
```

Activity 7.2:Clone Your Flask Project from GitHub

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

Run: 'git clone https://github.com/your-github-username/your-repository-name.git'

Note: change your-github-username and your-repository-name with your credentials

here: 'git clone https://github.com/keerthianakapalli/Medtrack.git'

- This will download your project to the EC2 instance.

To navigate to the project directory, run the following command:

```
cd Medtrack
```

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=5000
```

```
A newer release of "Amazon Linux" is available.
  Version 2023.6.20241010:
Run "/usr/bin/dnf check-release-update" for full release and version update info
   ,     #_
   ~\_   ####_        Amazon Linux 2023
  ~~  \_#####\
  ~~     \###|
  ~~       \#/ ___   https://aws.amazon.com/linux/amazon-linux-2023
   ~~       V~' '->
    ~~~         /
      ~~._.   _/
         _/ _/
       _/m/'
Last login: Tue Oct 15 04:17:59 2024 from 13.233.177.3
[ec2-user@ip-172-31-3-5 ~]$ git clone https://github.com/AlekhyaPenubakula/InstantLibrary.git
fatal: destination path 'InstantLibrary' already exists and is not an empty directory.
[ec2-user@ip-172-31-3-5 ~]$ cd InstantLibrary
[ec2-user@ip-172-31-3-5 InstantLibrary]$ cd InstantLibrary
[ec2-user@ip-172-31-3-5 InstantLibrary]$ flask run --host=0.0.0.0 --port=80
 * Debug mode: off
Permission denied
[ec2-user@ip-172-31-3-5 InstantLibrary]$ ^C
[ec2-user@ip-172-31-3-5 InstantLibrary]$ ^C
[ec2-user@ip-172-31-3-5 InstantLibrary]$ sudo flask run --host=0.0.0.0 --port=80
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:80
 * Running on http://172.31.3.5:80
Press CTRL+C to quit
^C[ec2-user@ip-172-31-3-5 InstantLibrary]$
[ec2-user@ip-172-31-3-5 InstantLibrary]$ sudo flask run --host=0.0.0.0 --port=80
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:80
 * Running on http://172.31.3.5:80
Press CTRL+C to quit
183.82.125.56 - - [22/Oct/2024 07:42:00] "GET / HTTP/1.1" 302 -
183.82.125.56 - - [22/Oct/2024 07:42:01] "GET /register HTTP/1.1" 200 -
183.82.125.56 - - [22/Oct/2024 07:42:01] "GET /static/images/library3.jpg HTTP/1.1" 200 -
183.82.125.56 - - [22/Oct/2024 07:42:01] "GET /favicon.ico HTTP/1.1" 404 -
183.82.125.56 - - [22/Oct/2024 07:42:16] "GET /login HTTP/1.1" 200 -
183.82.125.56 - - [22/Oct/2024 07:42:16] "GET /static/images/library3.jpg HTTP/1.1" 304 -
183.82.125.56 - - [22/Oct/2024 07:42:21] "POST /login HTTP/1.1" 200 -
183.82.125.56 - - [22/Oct/2024 07:42:24] "GET /login HTTP/1.1" 200 -
183.82.125.56 - - [22/Oct/2024 07:42:27] "POST /login HTTP/1.1" 302 -
183.82.125.56 - - [22/Oct/2024 07:42:28] "GET /home-page HTTP/1.1" 200 -
```

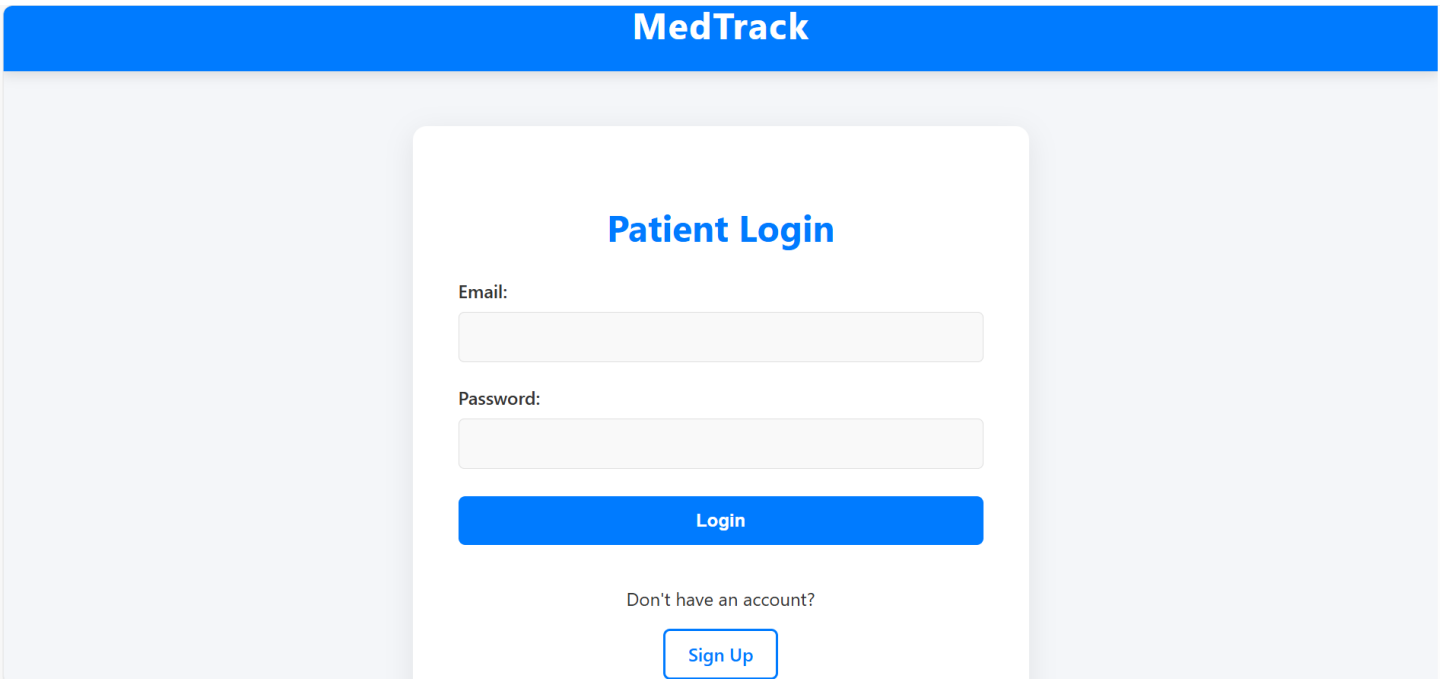
Access the website through:

PublicIPs: https://34.229.175.38/

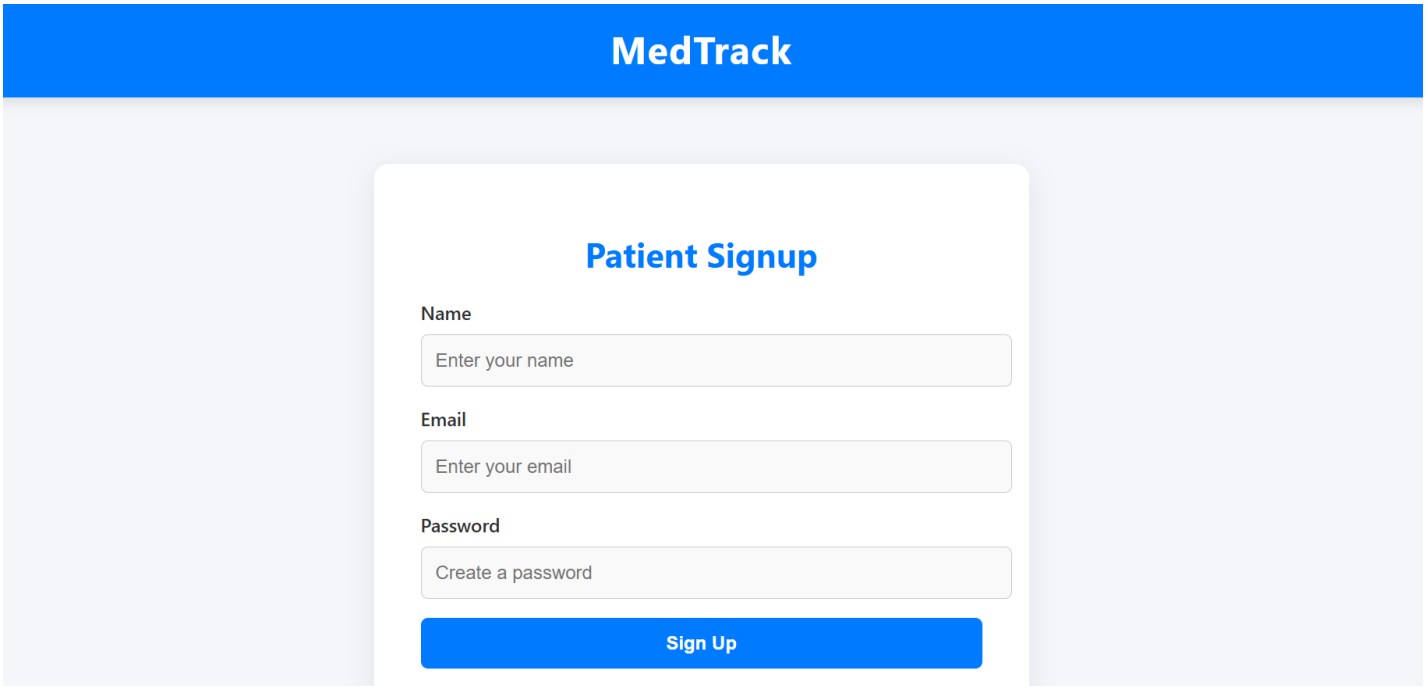
Milestone 8: Testing and Deployment

- **Activity 8.1: Conduct functional testing to verify user registration, login, book requests, and notifications.**

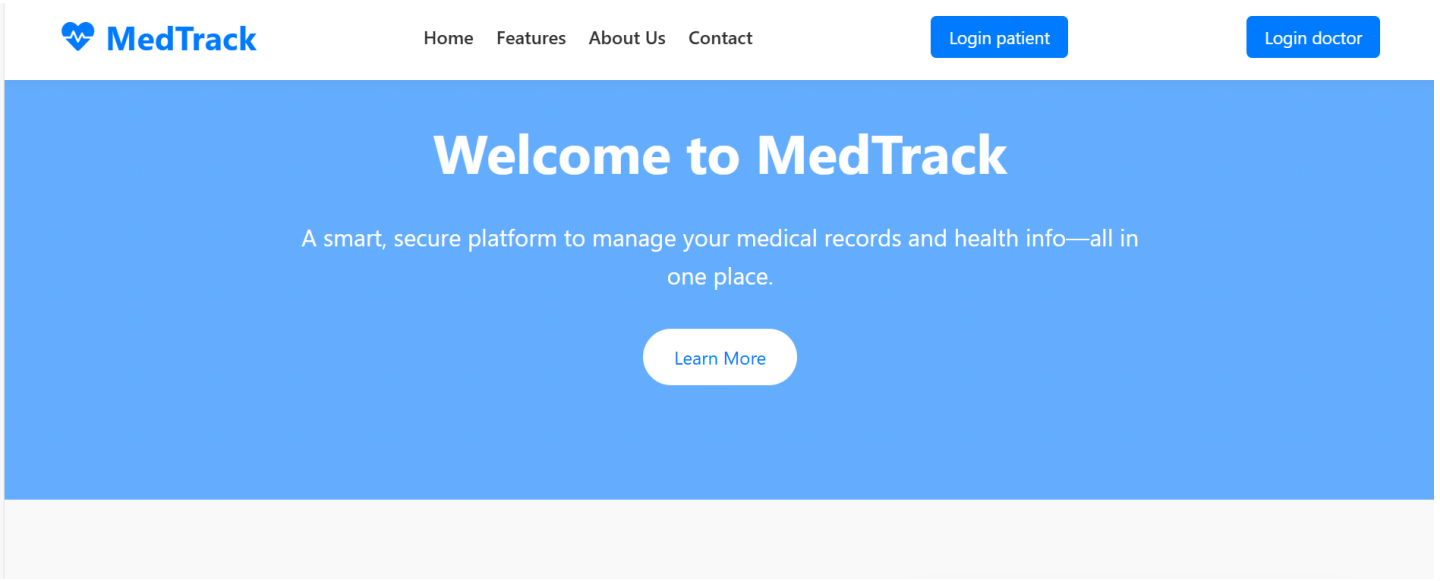
  Login Page:


MedTrack
Patient Login
Email:
Password:
Login
Don't have an account?
Sign Up

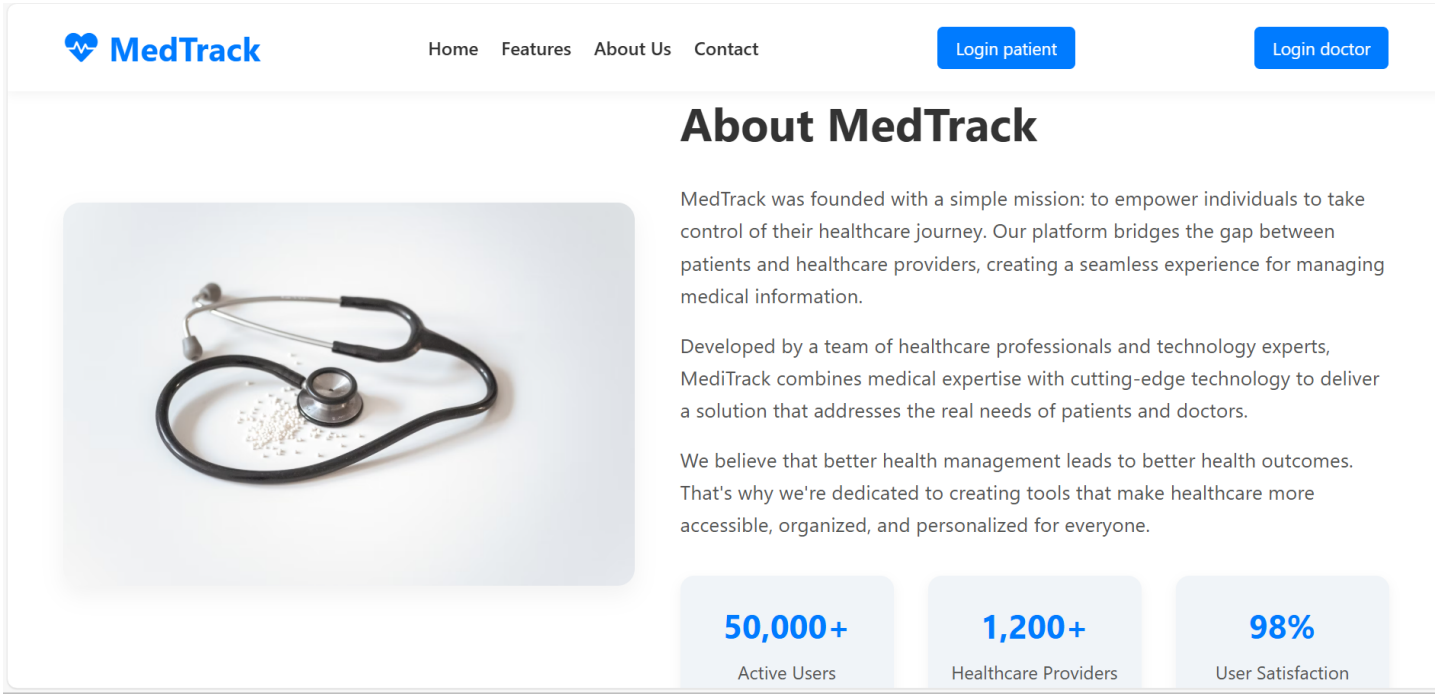
Signup Page:


MedTrack
Patient Signup
Name
Enter your name
Email
Enter your email
Password
Create a password
Sign Up

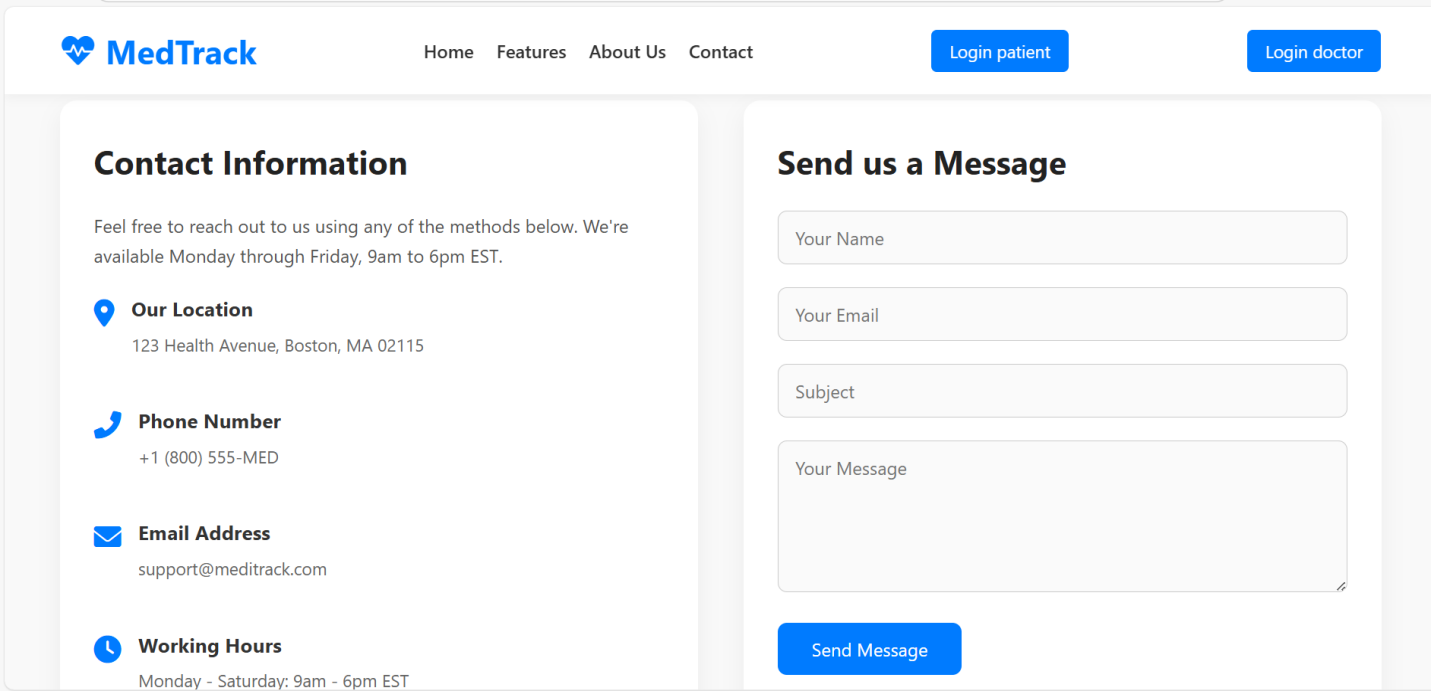
Home page:



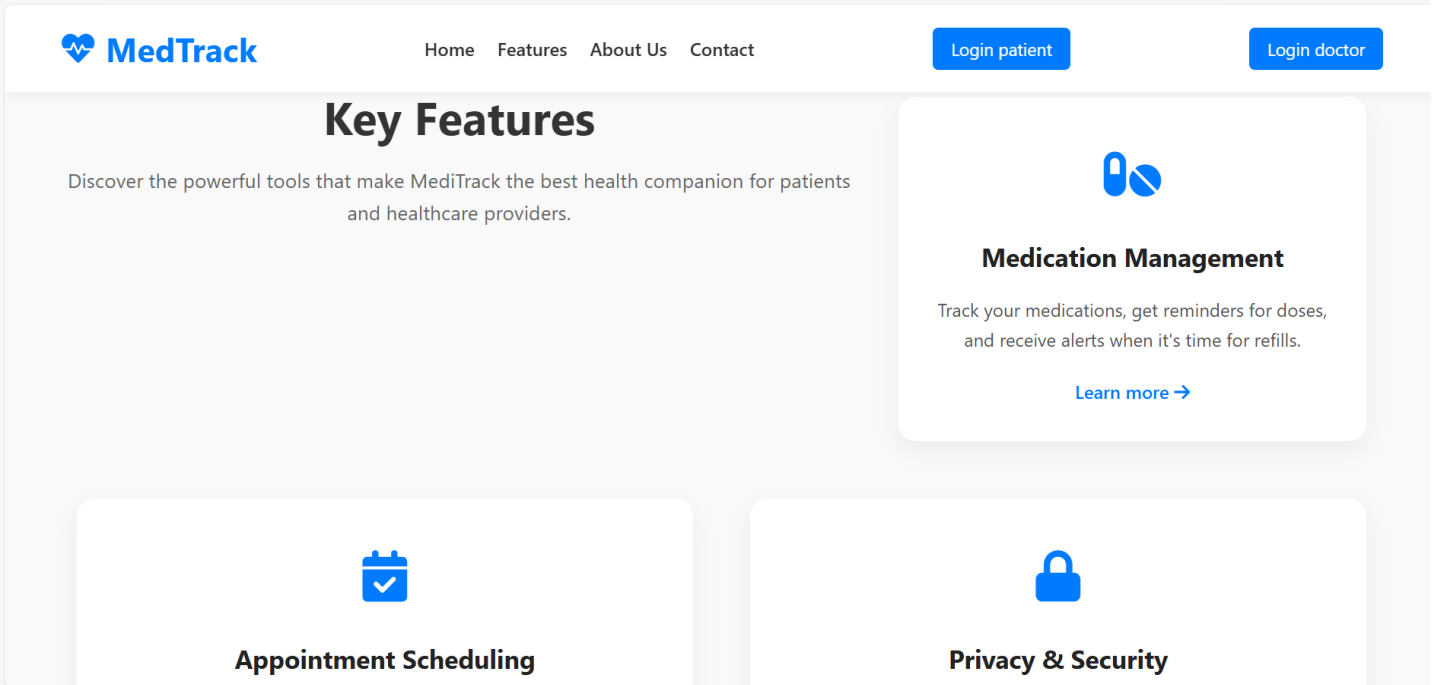
About Us page:



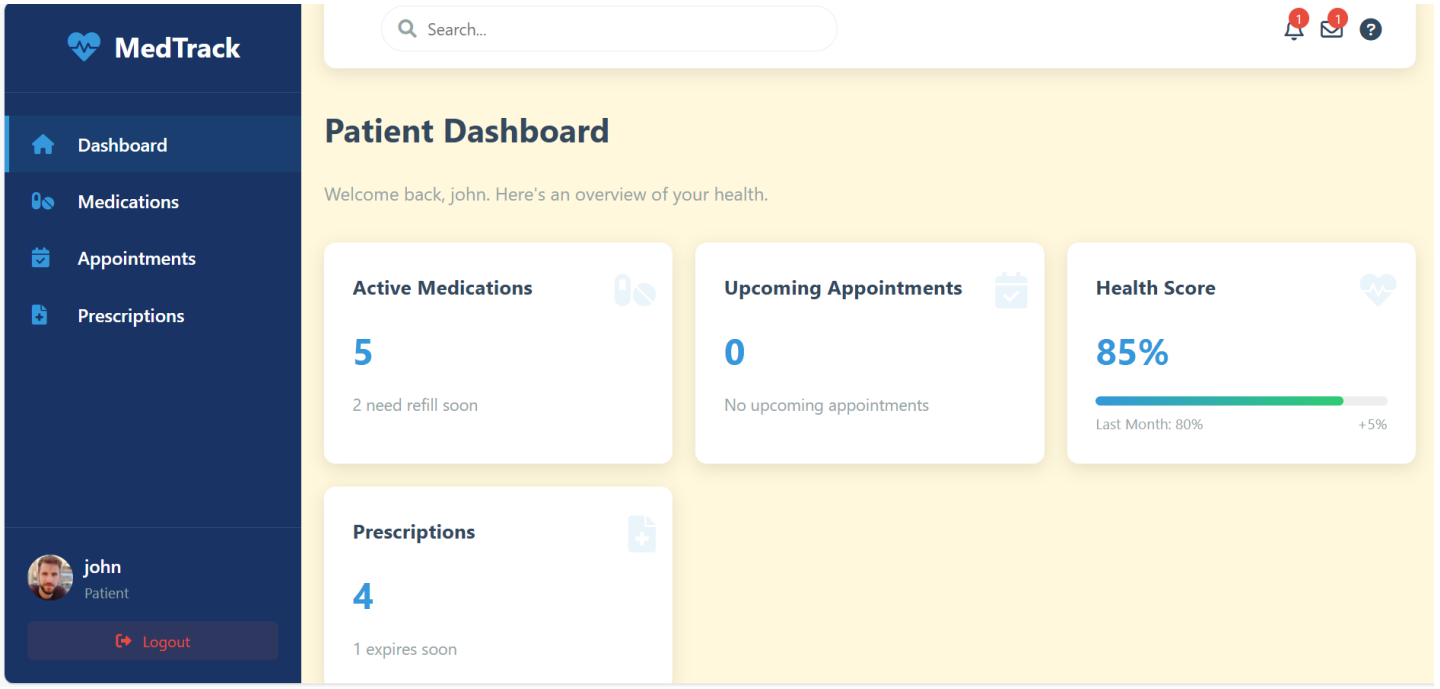
Contact Page:

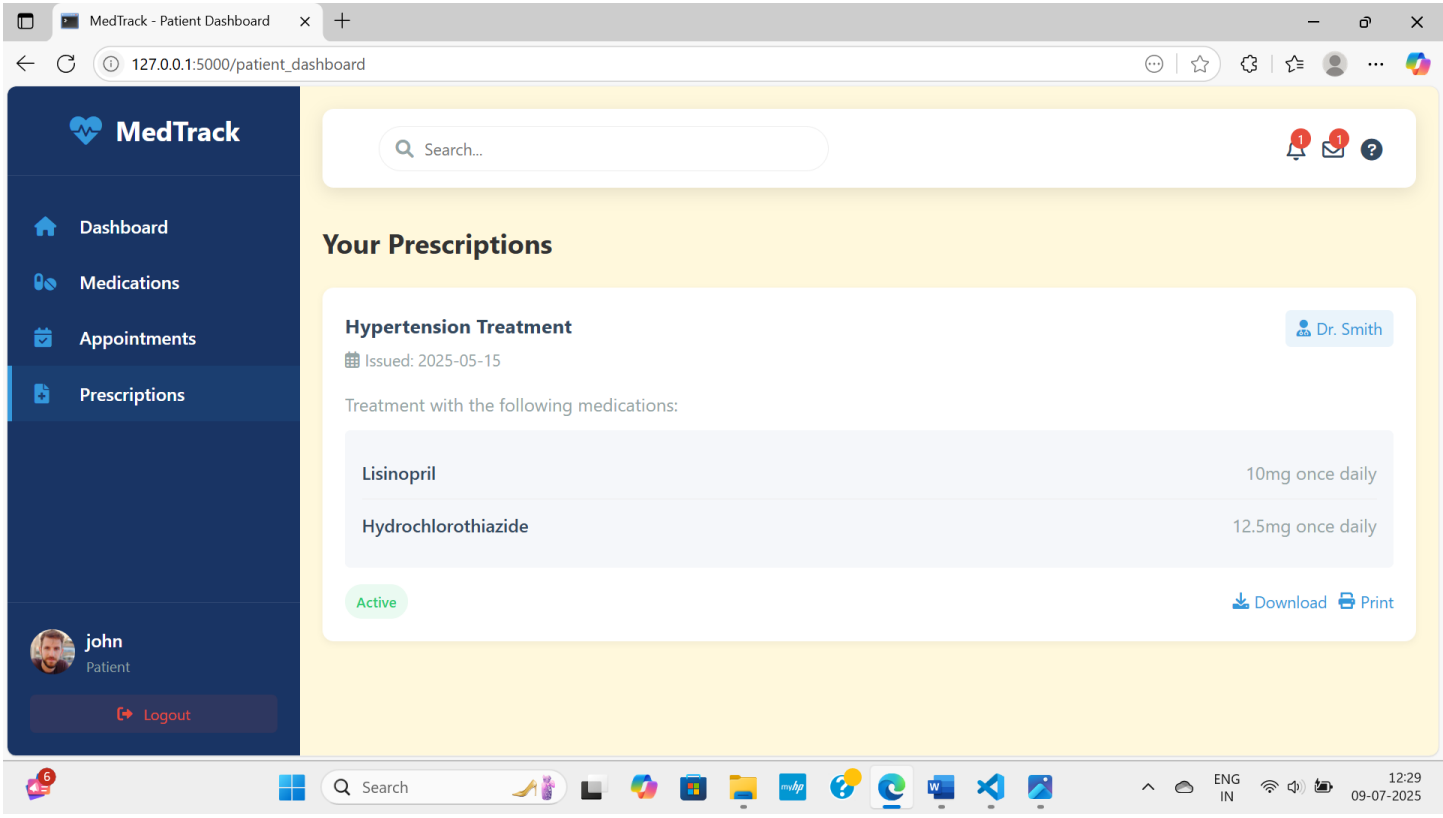


Features page:

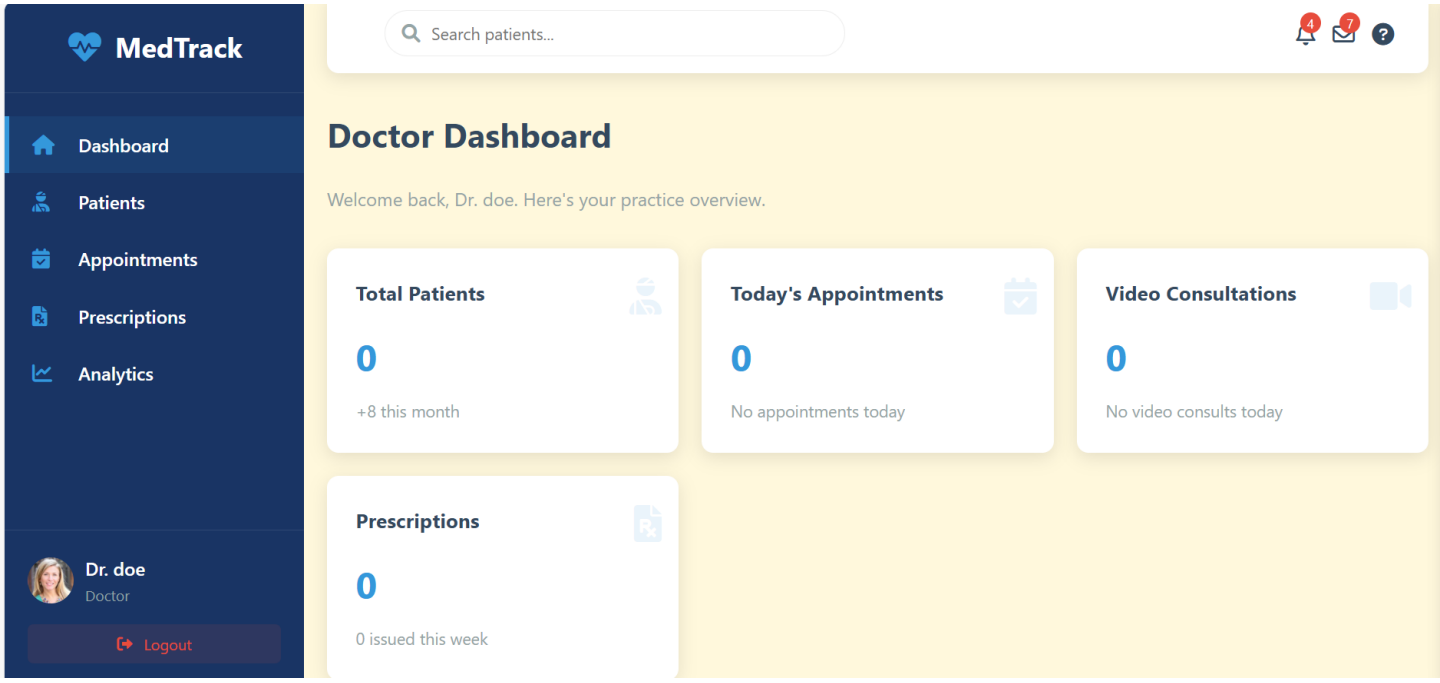


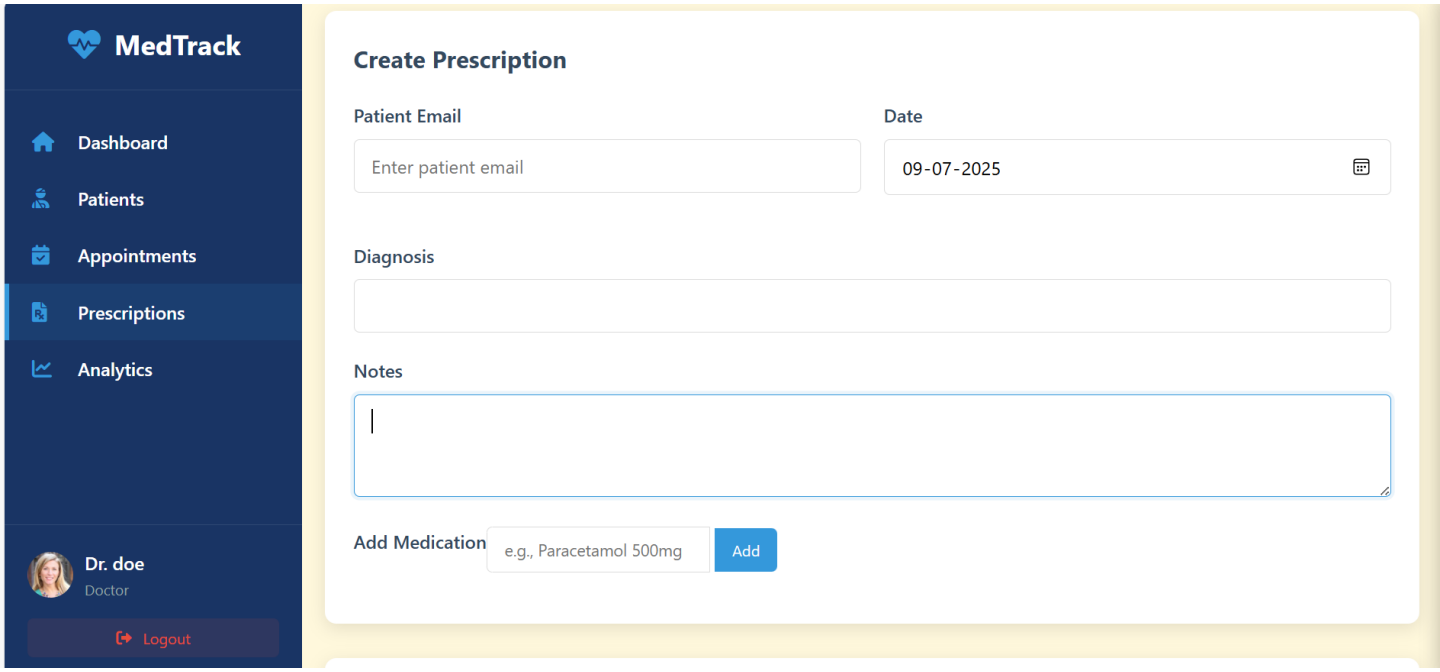
Patient dashboard page:



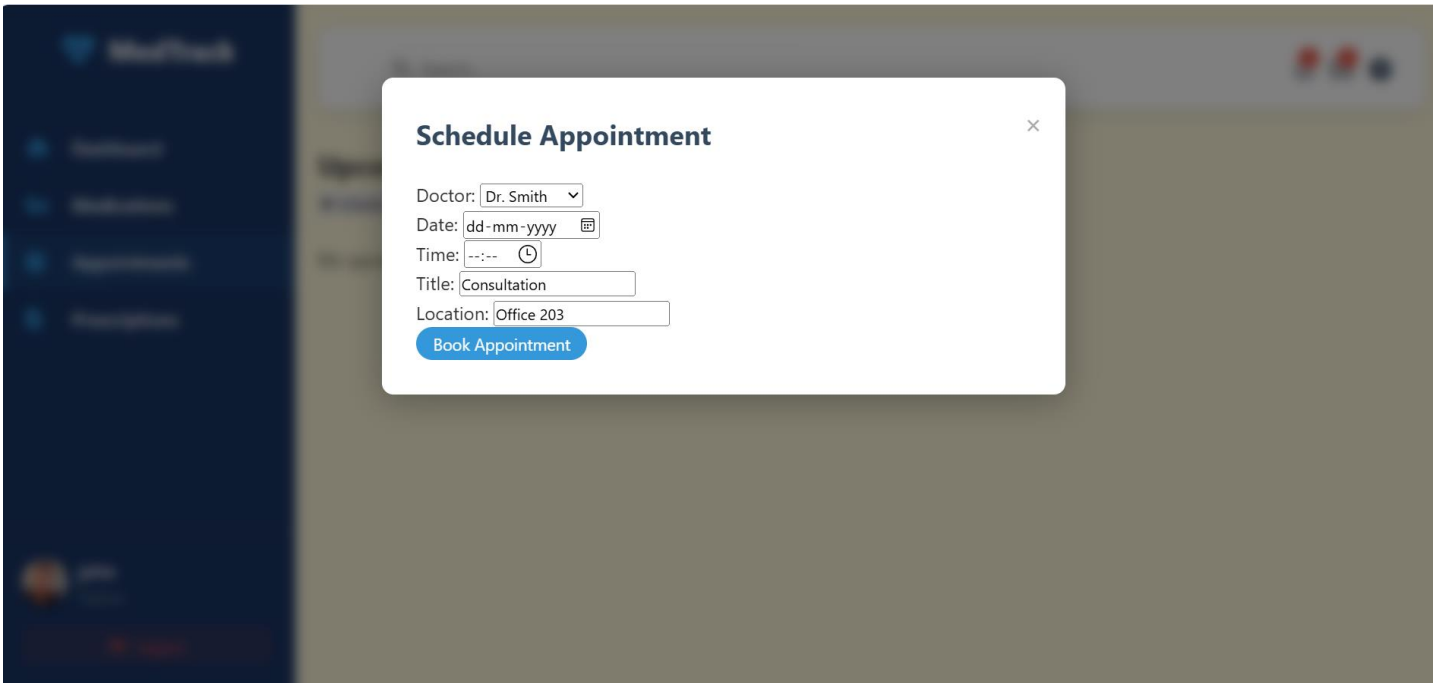


Doctor dashboard page:

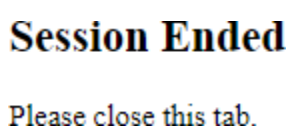




Book Appointment page:



Exit:



Conclusion:

The MedTrack application has been successfully developed and deployed in a local environment to demonstrate core healthcare functionalities such as user registration, login, appointment booking, medication tracking, and patient-doctor interaction. By utilizing Flask for the backend, HTML/CSS for the frontend, and a local mock database for data handling, the system ensures smooth and responsive user interactions during development and testing.

This locally deployed setup allows for effective testing of essential workflows like patient dashboard access, doctor appointments, diagnosis entry, and reminder notifications, without the need for cloud infrastructure. Secure password management, session handling, and form validation have been implemented to ensure a realistic and secure simulation of a healthcare system.

In conclusion, the local deployment of Medtrack serves as a reliable foundation for development, enabling efficient prototyping, testing, and demonstration of all key features. It lays the groundwork for future migration to a scalable cloud-based environment while already providing value as an offline or intranet-based medical record management system.