Homemade pickles & snacks: taste the best

Project Description:

The HomeMade Pickles & Snacks project is a mini e-commerce web application developed using Flask (Python-based web framework), aimed at showcasing and selling home-made pickles and snacks. The application provides users with an interactive shopping experience, allowing them to browse a product catalog, add items to a cart, and place orders using Cash on Delivery (COD).

The project is fully AWS-ready, integrated with key cloud services to ensure scalability, reliability, and real-world deployment capabilities.

Scenario 1: Smooth Shopping Experience During Festival Rush

During festive seasons like Diwali or Sankranti, HomeMade Pickles sees a spike in orders. Thanks to AWS EC2, the website can handle hundreds of users browsing and adding items to their cart at the same time.For example, a customer logs in to the website and selects *Mango Pickle* (500g) and *Banana Chips* (1kg). Flask processes this request, calculates the price based on quantity and weight, and adds it to the user's cart. Even with heavy traffic, the website continues to run smoothly without downtime, offering a seamless shopping experience.

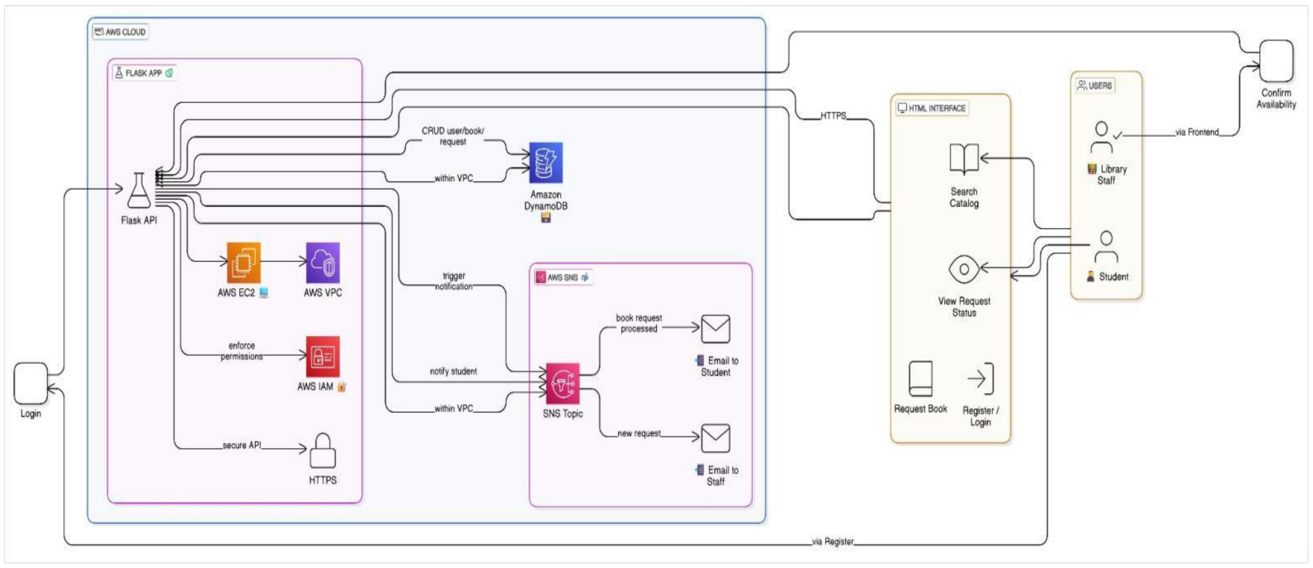
Scenario 2:Order Confirmation with Email Alerts

Once a customer finalizes the order, the application provides instant order confirmation through email. When a user submits their delivery address and clicks "Order Now", Flask handles the backend operations by fetching all cart items, calculating the total, clearing the cart, and triggering an automated email. This email confirmation, sent using AWS Simple Email Service (SES) or Gmail SMTP configured via environment variables, contains the delivery details and payment information. This seamless integration ensures customers receive immediate communication about their purchase, enhancing trust and user experience.

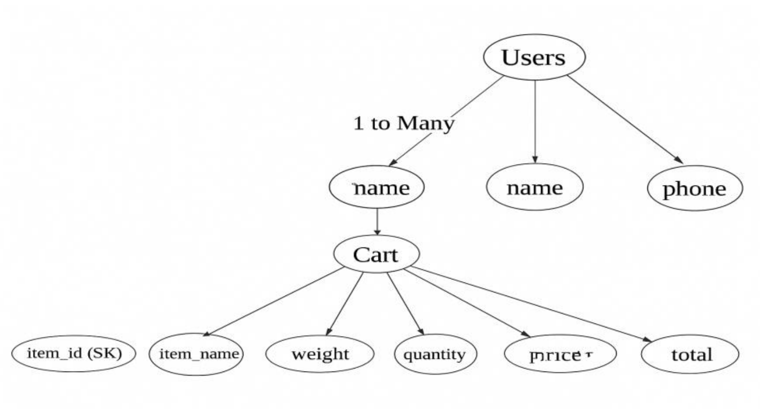
Scenario 3: Customer Feedback and Review Submission

To build community trust and product credibility, HomeMade Pickles allows registered users to submit reviews after trying the products. After enjoying the Lemon Pickle, for instance, a user can log in and submit a review like, “Tangy and delicious! Reminds me of my grandma’s recipe.” Flask validates the user's login session, processes the review form, and stores the review in the AWS DynamoDB Reviews table along with a timestamp. These reviews are later displayed on the reviews page, along with a few dummy testimonials, helping new users gain confidence in the product quality and service reliability.

AWS ARCHITECTURE:



Entity Relationship:



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:

- Users: email (PK), name, phone, password
- Cart: email (PK), item_id (SK), item_name, weight, quantity, price, total, timestamp
- Reviews: email (PK), timestamp (SK), name, message

3. SNS Notification Setup

· **Activity 3.1:** Create an SNS topic for order confirmation.

· **Activity 3.2:** Subscribe user email IDs for notifications upon placing orders.

4.Backend Development and Application Setup

· **Activity 4.1:** Develop the backend using Flask to manage routing, session, and logic.

· **Activity 4.2:** Integrate AWS services (DynamoDB, SNS, SES) using boto3 library.

5.IAM Role Setup:

· **Activity 5.1:** Create an IAM Role (EC2_DynamoDB_SES_Role) to allow EC2 access to AWS services.

· **Activity 5.2:** Attach policies like AmazonDynamoDBFullAccess, AmazonSNSFullAccess, AmazonSESFullAccess.

6.EC2 Instance Setup

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

· **Activity 6.2:** Configure Security Groups to allow inbound traffic on ports:

- **80** (HTTP)
- **22** (SSH)
- **5000** (Flask default, for testing)

7.Deploymet on EC2

· **Activity 7.1:** Upload Flask project files to EC2 using Git.

· **Activity 7.2:** Run the Flask app using python app.py

8.Testing and Deployment

Activity 8.1: Conduct full functional testing:

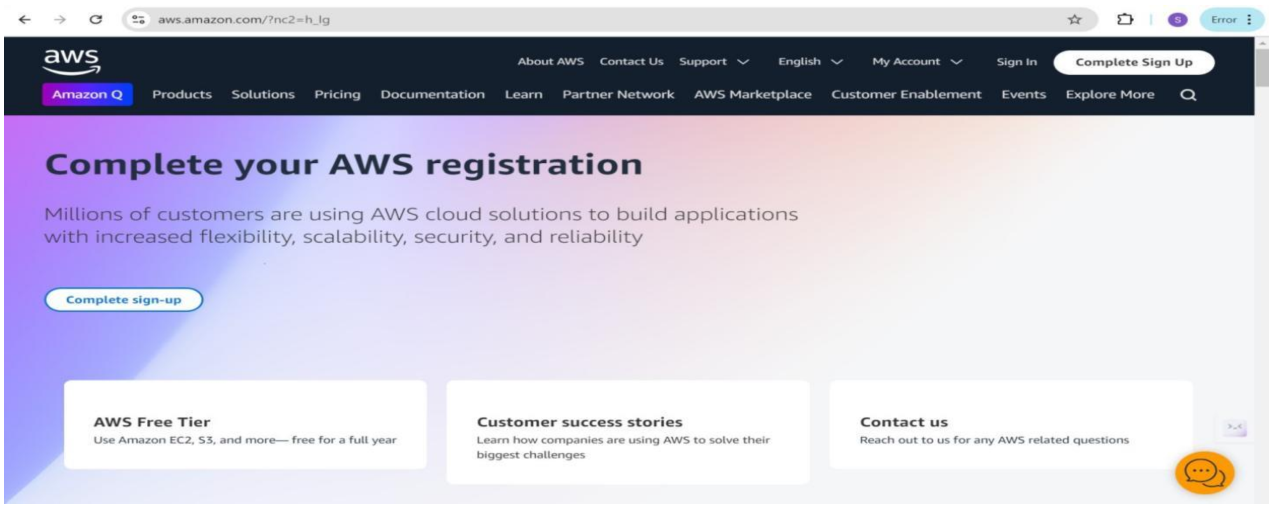
User registration and login

Add to cart and review

Place orders and receive SNS/SES notification.

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.



Complete your AWS registration

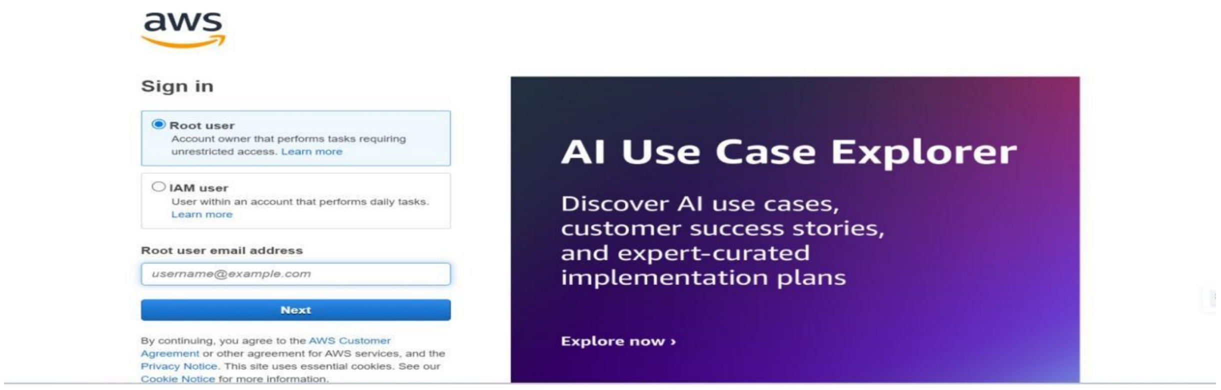
Millions of customers are using AWS cloud solutions to build applications with increased flexibility, scalability, security, and reliability

AWS Free Tier
Use Amazon EC2, S3, and more— free for a full year

Customer success stories
Learn how companies are using AWS to solve their biggest challenges

Contact us
Reach out to us for any AWS related questions

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



Sign in

Root user
Account owner that performs tasks requiring unrestricted access. Learn more

IAM user
User within an account that performs daily tasks. Learn more

Root user email address
username@example.com

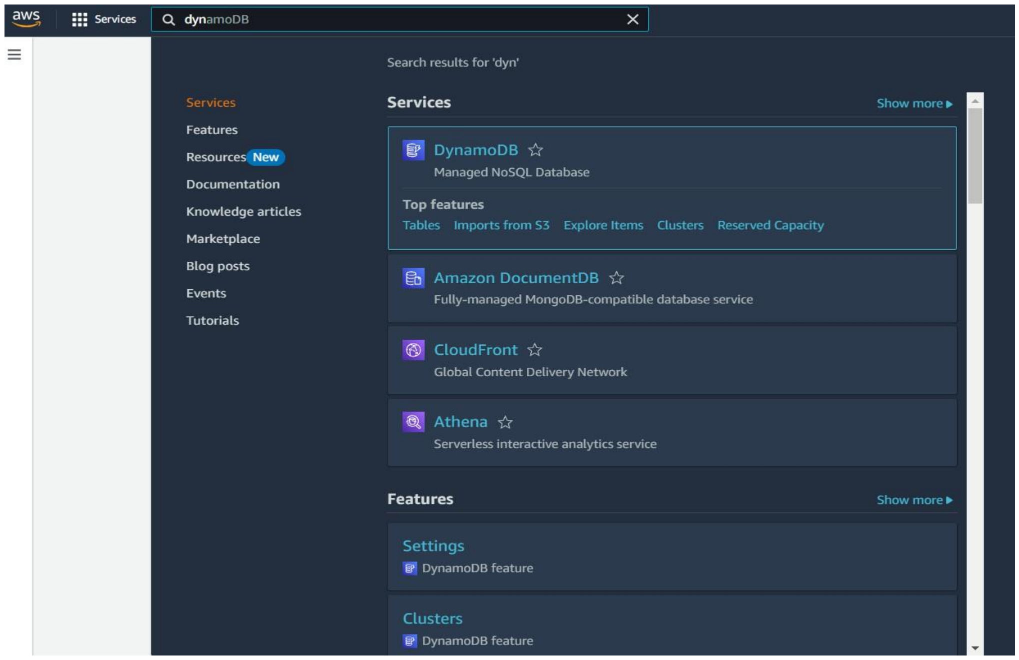
By continuing, you agree to the AWS Customer Agreement or other agreement for AWS services, and the Privacy Notice. This site uses essential cookies. See our Cookie Notice for more information.

AI Use Case Explorer

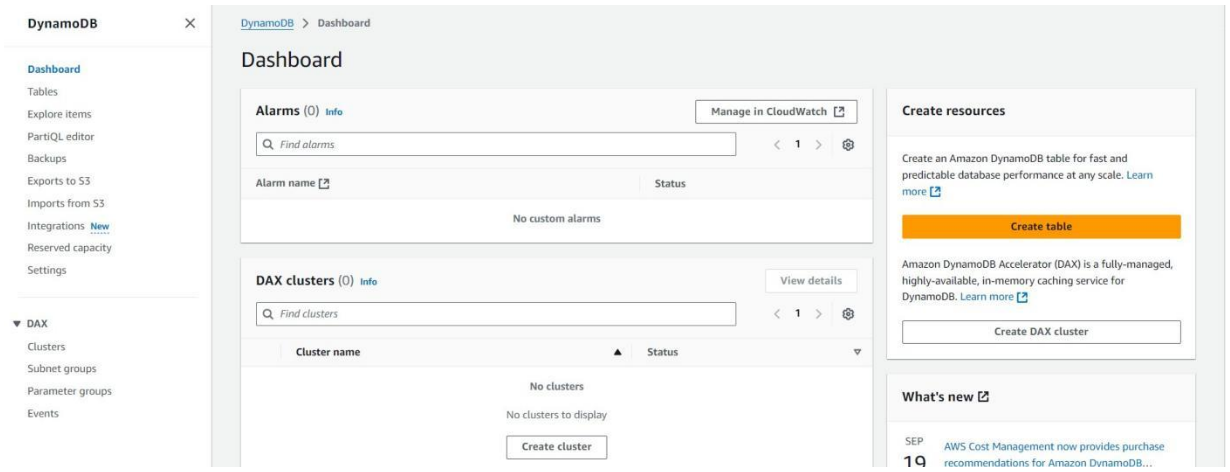
Discover AI use cases, customer success stories, and expert-curated implementation plans

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.

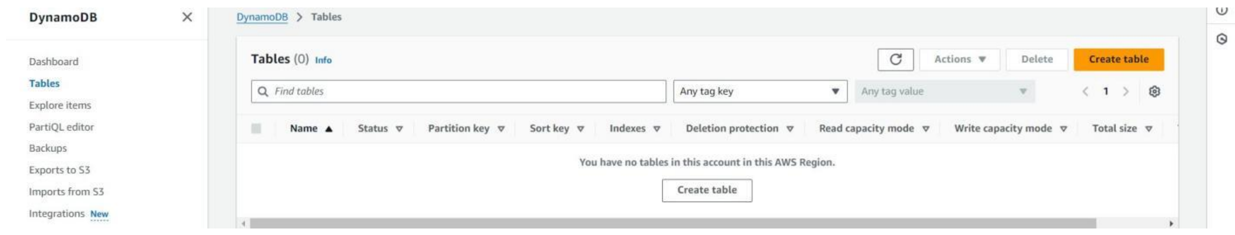


    -



Create an Amazon DynamoDB table for fast and predictable database performance at any scale. Learn more

Amazon DynamoDB Accelerator (DAX) is a fully-managed, highly-available, in-memory caching service for DynamoDB. Learn more


DynamoDB

Dashboard
Tables
Explore items
PartiQL editor
Backups
Exports to S3
Imports from S3
Integrations New

DynamoDB > Tables

Tables (0) Info

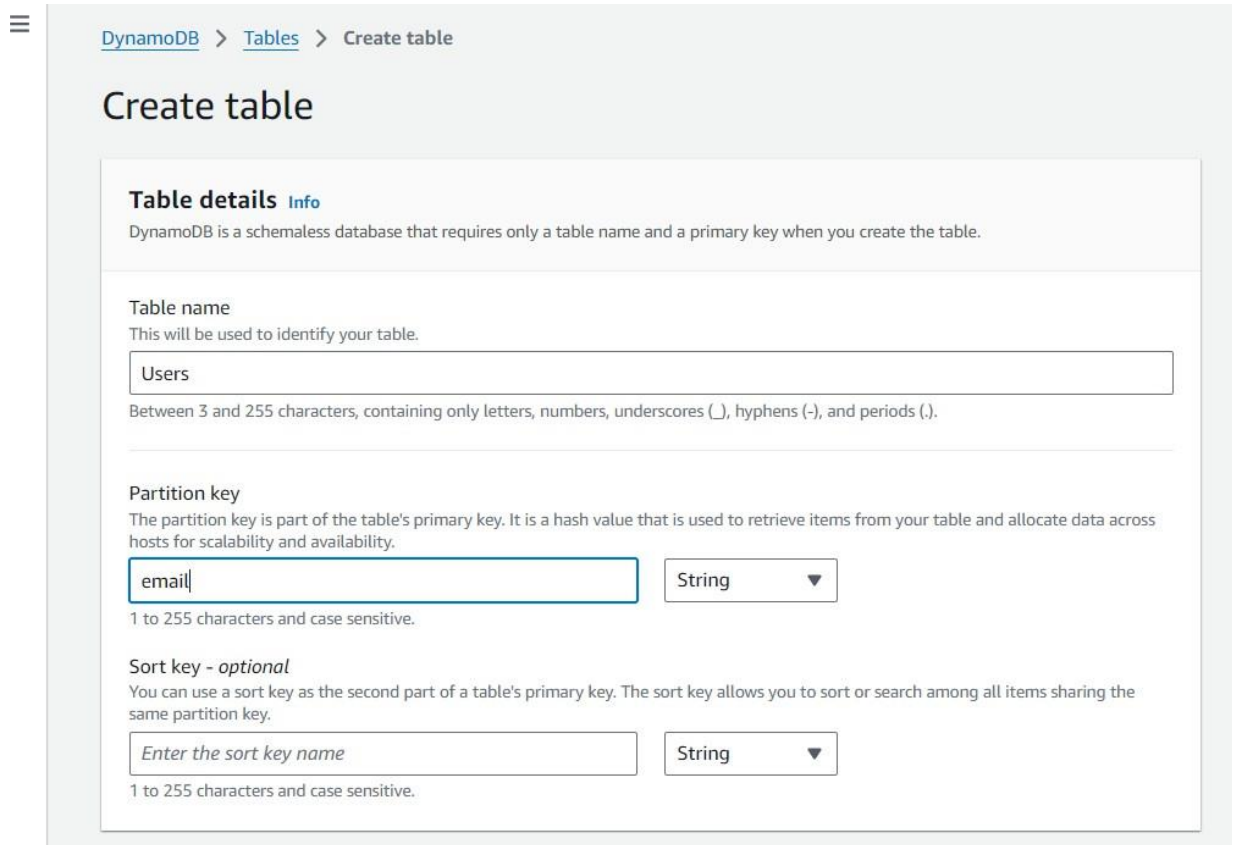
Name	Status	Partition key	Sort key	Indexes	Deletion protection	Read capacity mode	Write capacity mode	Total size

You have no tables in this account in this AWS Region.

Create table

Activity 2.2:Create a DynamoDB table for storing user registration details, cart data, and customer reviews.

Create Users table with partition key “Email” with type String and click on create tables.


DynamoDB > Tables > Create table

Create table

Table details Info

DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name

This will be used to identify your table.

Users

Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).

Partition key

The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.

email | String

1 to 255 characters and case sensitive.

Sort key - *optional*

You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.

Enter the sort key name | String

1 to 255 characters and case sensitive.

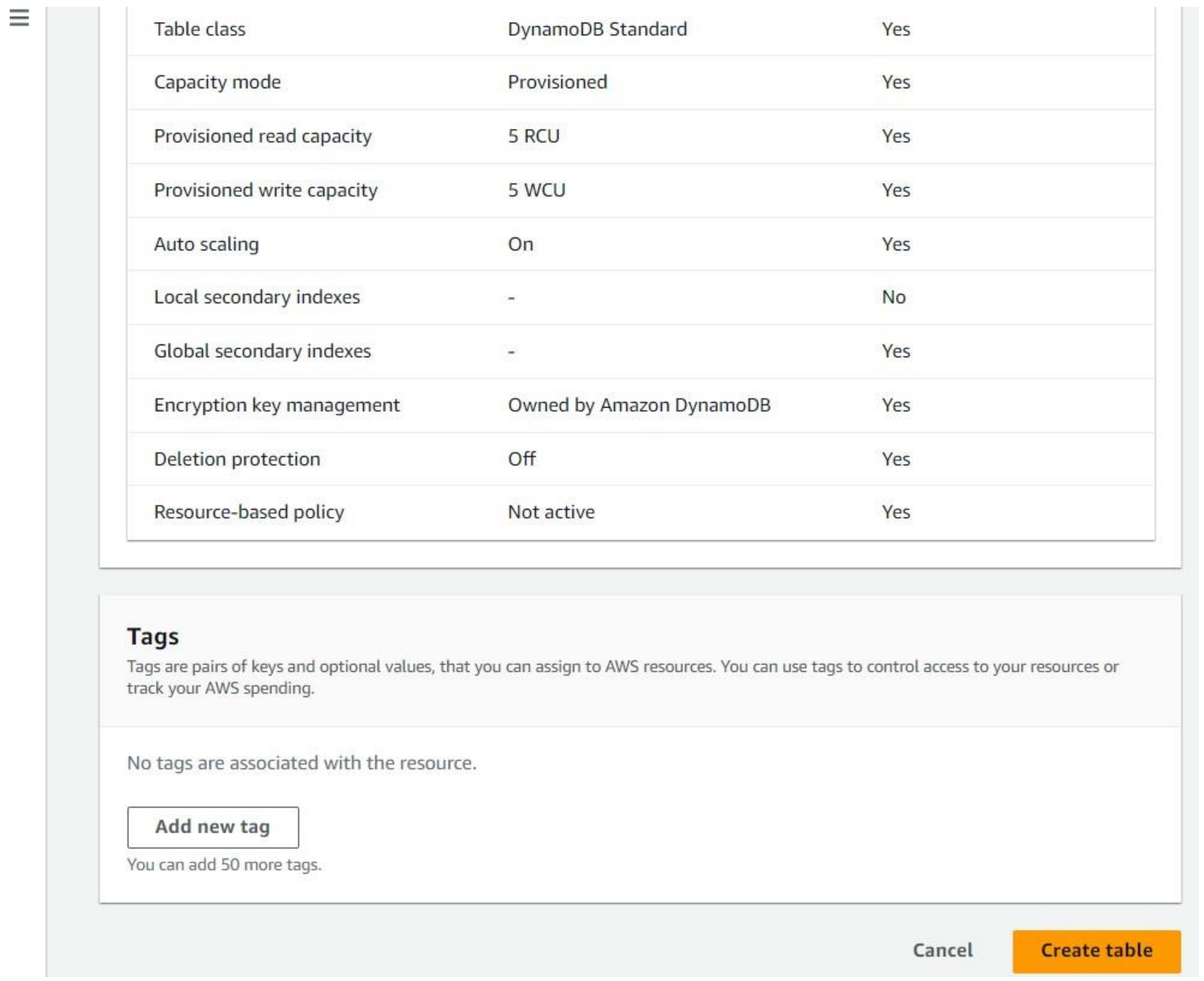


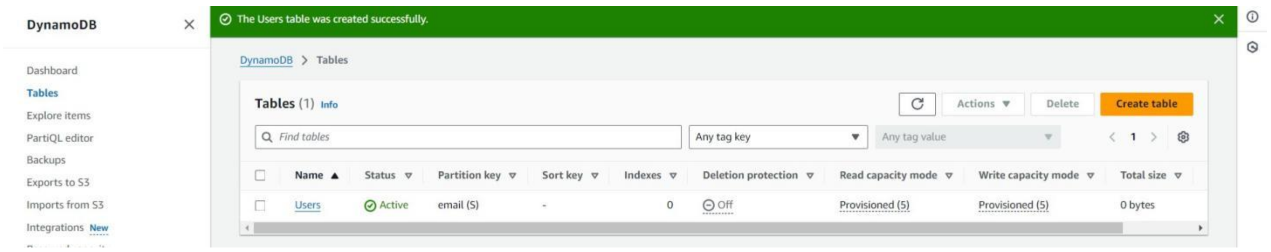
Table class	DynamoDB Standard	Yes
Capacity mode	Provisioned	Yes
Provisioned read capacity	5 RCU	Yes
Provisioned write capacity	5 WCU	Yes
Auto scaling	On	Yes
Local secondary indexes	-	No
Global secondary indexes	-	Yes
Encryption key management	Owned by Amazon DynamoDB	Yes
Deletion protection	Off	Yes
Resource-based 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.

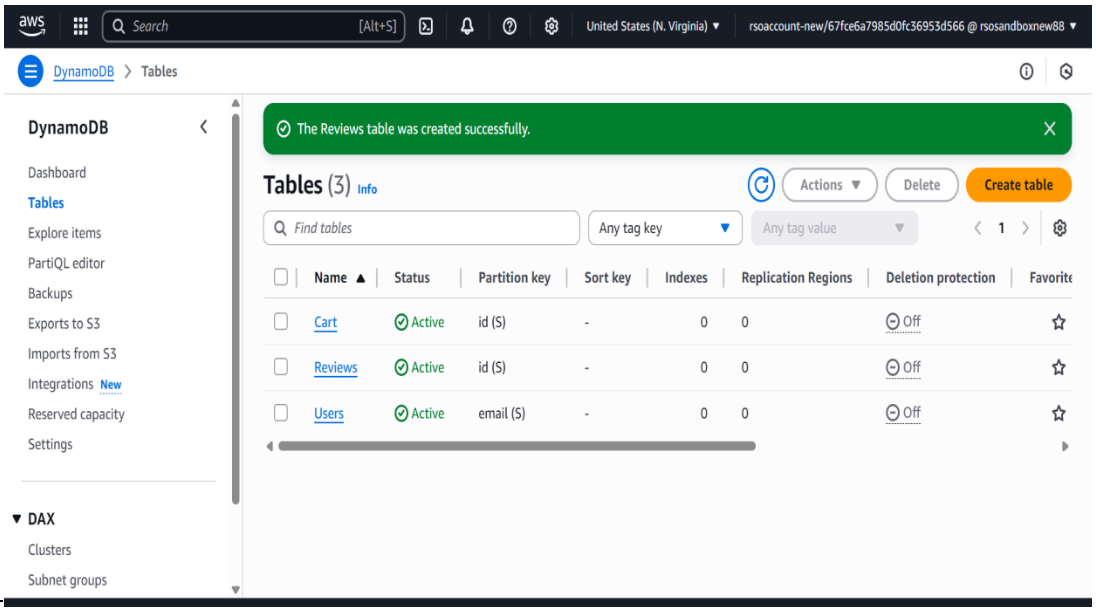
You can add 50 more tags.



The Users table was created successfully.

Name	Status	Partition key	Sort key	Indexes	Deletion protection	Read capacity mode	Write capacity mode	Total size
Users	Active	email (S)	-	0	Off	Provisioned (5)	Provisioned (5)	0 bytes

-
- Follow the same steps to create a reviews table with id as the primary key for customer reviews data and cart table to store the items in the cart.

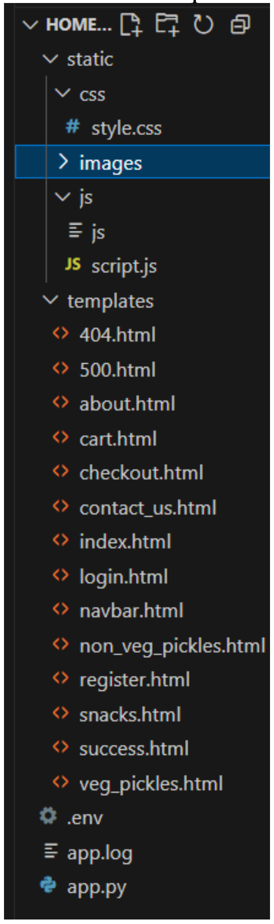

The Reviews table was created successfully.

Tables (3)

Name	Status	Partition key	Sort key	Indexes	Replication Regions	Deletion protection	Favorite
Cart	Active	id (S)	-	0	0	Off	☆
Reviews	Active	id (S)	-	0	0	Off	☆
Users	Active	email (S)	-	0	0	Off	☆

Milestone 4:Backend Development and Application Setup

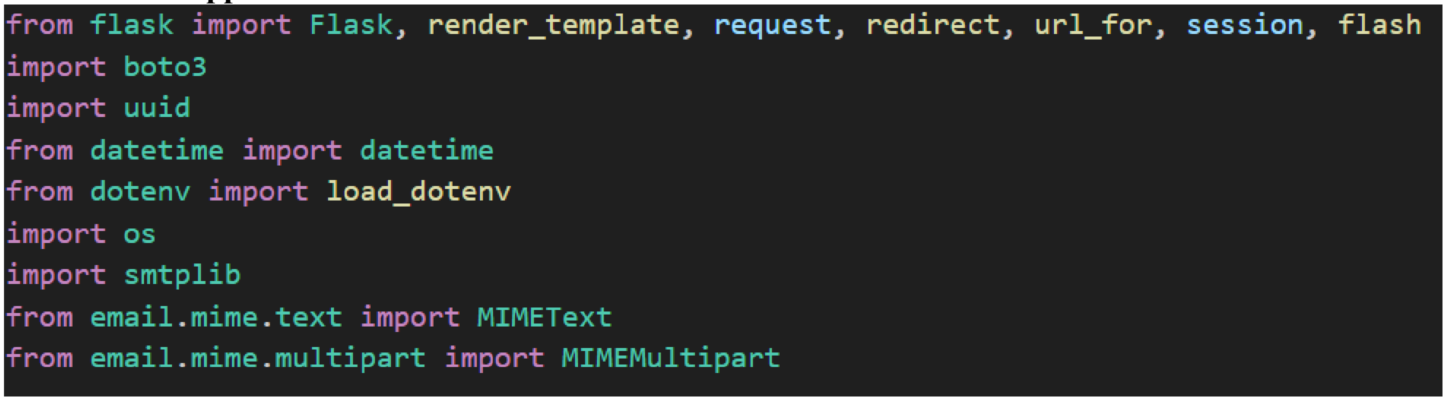
- **Activity 4.1: Develop the backend using Flask**
    - File Explorer Structure


```
HOME...
  static
    css
      style.css
    images
    js
      js
      script.js
  templates
    404.html
    500.html
    about.html
    cart.html
    checkout.html
    contact_us.html
    index.html
    login.html
    navbar.html
    non_veg_pickles.html
    register.html
    snacks.html
    success.html
    veg_pickles.html
  .env
  app.log
  app.py
```

-
    -

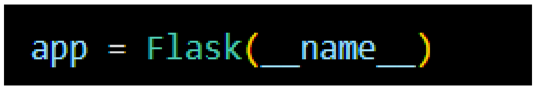
Description: The project directory is organized into key folders and files essential for a Flask-based web application integrated with AWS. The app.py file is the core backend script that handles routing, session management, user authentication, and communication with AWS services like DynamoDB and SNS. The static folder contains subfolders for css (for styling), images (for visual assets like logos or product pictures), and js (for any frontend interactivity using JavaScript). The templates folder holds HTML files rendered by Flask, including home.html (the main page after login), login.html (for user authentication), register.html (for new user sign-up), and welcome.html (the initial landing page shown before login). This structure ensures a clean separation of frontend and backend components, enabling efficient development and deployment of the web application.

- **Flask App Initialization**



```
from flask import Flask, render_template, request, redirect, url_for, session, flash
import boto3
import uuid
from datetime import datetime
from dotenv import load_dotenv
import os
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
```

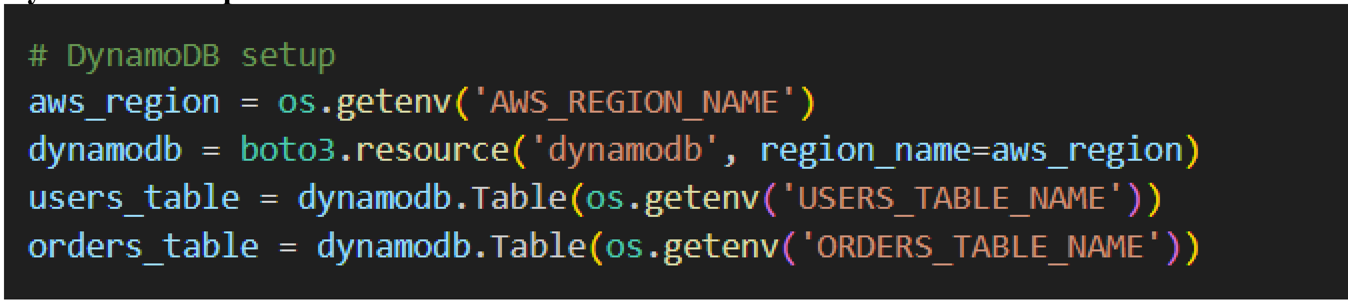
Description: This image displays the import section of your app.py file, which includes all the essential libraries required for your Flask web application integrated with AWS and email services. It starts by importing core Flask modules such as Flask, render_template, request, redirect, url_for, session, and flash for routing, rendering HTML templates, handling form submissions, and managing user sessions. The boto3 library is used to interact with AWS services, particularly DynamoDB and SNS. The uuid module generates unique identifiers for user data and requests, while datetime handles timestamps. dotenv is used to load environment variables securely via the .env file. The os module accesses environment variables and file paths. Finally, smtplib, MIMEText, and MIMEMultipart are imported to facilitate sending email notifications, such as order confirmations, using SMTP. Together, these imports enable your app to handle backend logic, database interactions, and communication functionalities securely and efficiently.



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

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

- **Dynamodb Setup:**



```
# DynamoDB setup
aws_region = os.getenv('AWS_REGION_NAME')
dynamodb = boto3.resource('dynamodb', region_name=aws_region)
users_table = dynamodb.Table(os.getenv('USERS_TABLE_NAME'))
orders_table = dynamodb.Table(os.getenv('ORDERS_TABLE_NAME'))
```

-

Description: This snippet from your app.py file demonstrates how you're connecting to three different DynamoDB tables in the AWS ap-south-1 region using the boto3 library. The line dynamodb = boto3.resource('dynamodb', region_name=region_name) establishes a resource-level connection to DynamoDB. Then, the tables are assigned to variables:

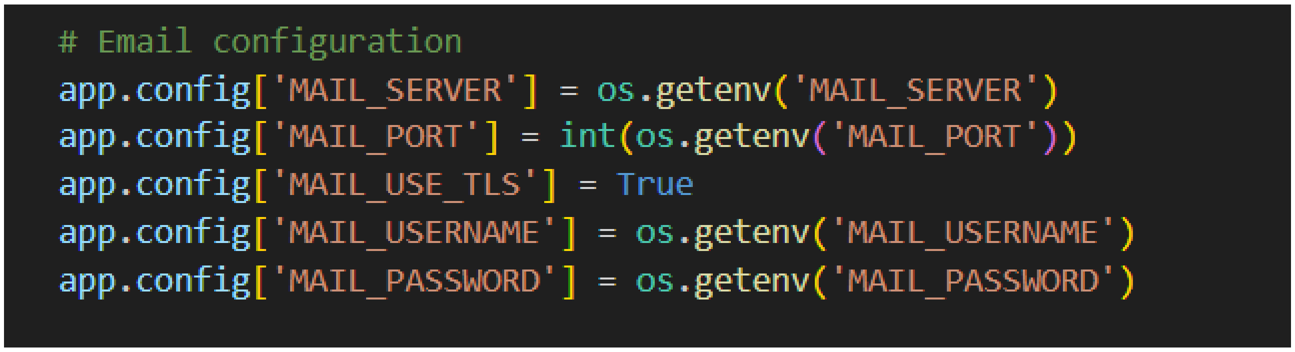
cart_table = dynamodb.Table('Cart'): Handles items added to the cart by users.

reviews_table = dynamodb.Table('Reviews'): Stores customer reviews and feedback.

users_table = dynamodb.Table('Users'): Manages user registration and login data.

This structure ensures modular and clear access to each table, allowing smooth database operations throughout your Flask app.

Email Confirmation:

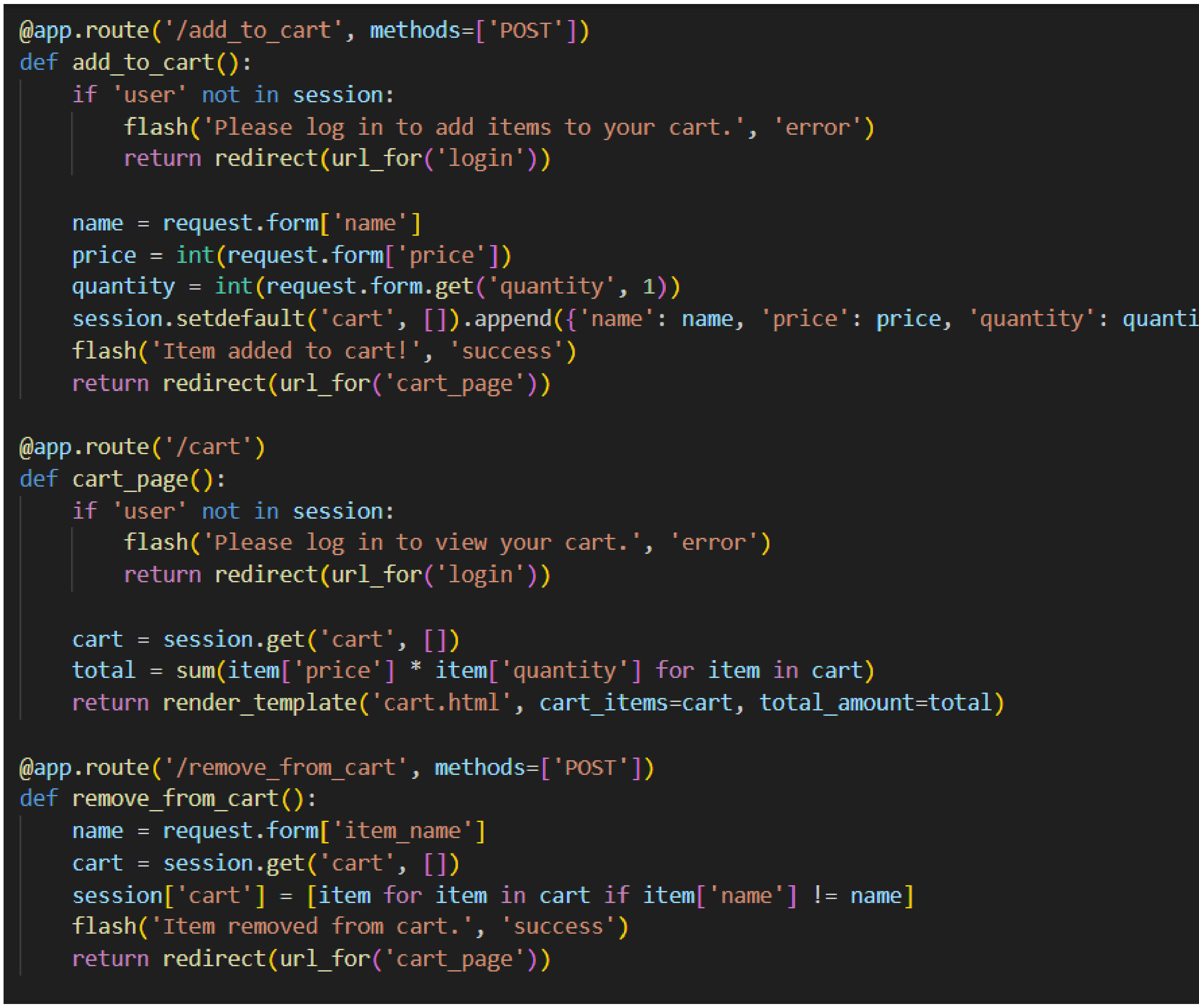


```
# Email configuration
app.config['MAIL_SERVER'] = os.getenv('MAIL_SERVER')
app.config['MAIL_PORT'] = int(os.getenv('MAIL_PORT'))
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = os.getenv('MAIL_USERNAME')
app.config['MAIL_PASSWORD'] = os.getenv('MAIL_PASSWORD')
```

Description: The send_confirmation_email(to_email, address) function is responsible for sending a personalized order confirmation email to users after they successfully place an order on the HomeMade Pickles platform. It retrieves the sender's Gmail credentials securely using environment variables (GMAIL_USER and GMAIL_APP_PASSWORD) to protect sensitive information. The function constructs a message that includes a subject and a plain-text body containing order details and the delivery address, ensuring that customers receive all necessary information. Using Python’s smtplib and email.mime modules, the function creates a secure connection to Gmail’s SMTP server (smtp.gmail.com on port 465) via SSL. After logging in with the provided credentials, it sends the composed email to the recipient. Additionally, the function handles any errors that may occur during the process, providing feedback through exception handling. This ensures a smooth and secure communication channel with the customer, reinforcing trust and enhancing the overall user experience.

- **Routes for Web Pages**

  - Cart route:

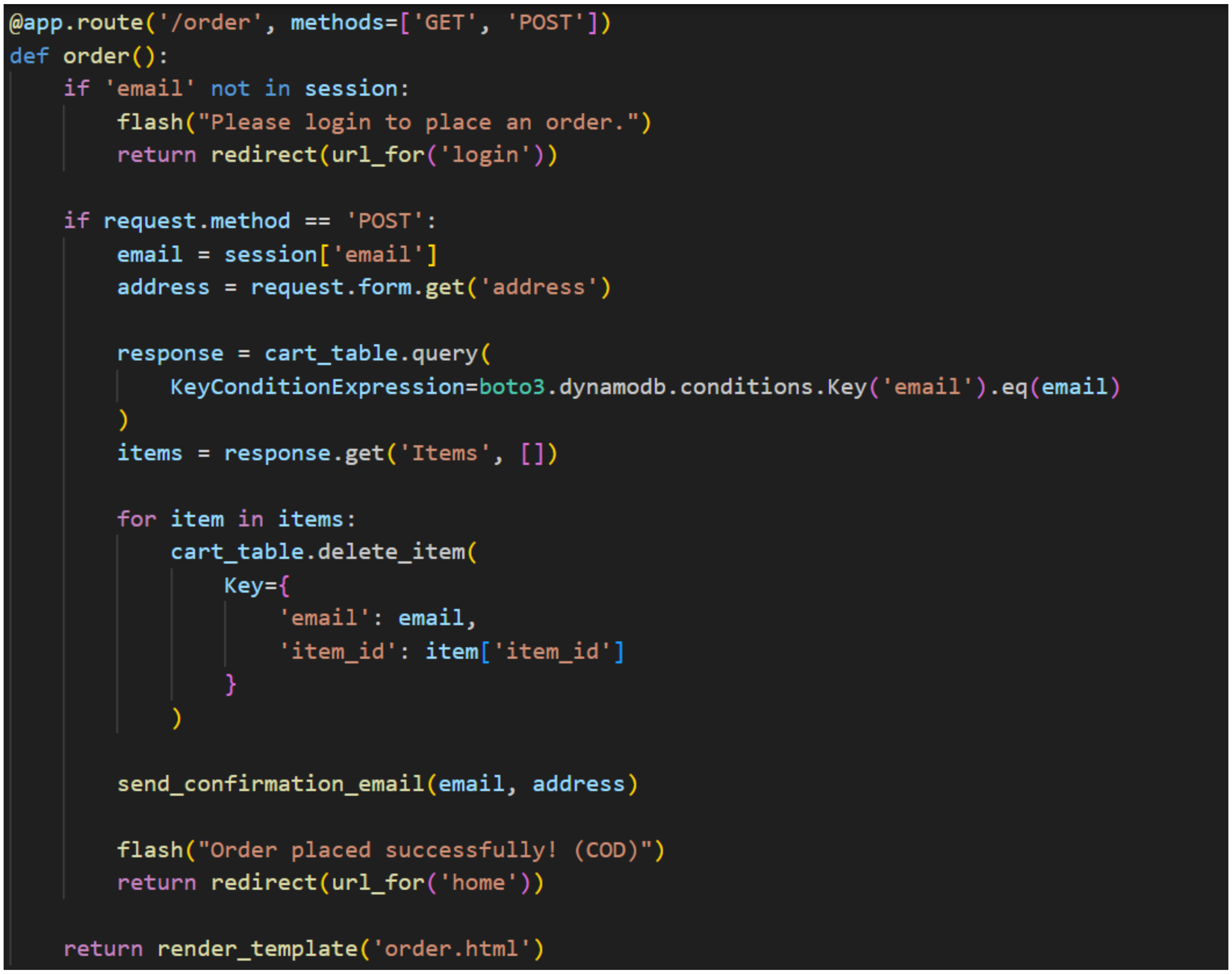


```python
@app.route('/add_to_cart', methods=['POST'])
def add_to_cart():
    if 'user' not in session:
        flash('Please log in to add items to your cart.', 'error')
        return redirect(url_for('login'))

    name = request.form['name']
    price = int(request.form['price'])
    quantity = int(request.form.get('quantity', 1))
    session.setdefault('cart', []).append({'name': name, 'price': price, 'quantity': quanti
    flash('Item added to cart!', 'success')
    return redirect(url_for('cart_page'))

@app.route('/cart')
def cart_page():
    if 'user' not in session:
        flash('Please log in to view your cart.', 'error')
        return redirect(url_for('login'))

    cart = session.get('cart', [])
    total = sum(item['price'] * item['quantity'] for item in cart)
    return render_template('cart.html', cart_items=cart, total_amount=total)

@app.route('/remove_from_cart', methods=['POST'])
def remove_from_cart():
    name = request.form['item_name']
    cart = session.get('cart', [])
    session['cart'] = [item for item in cart if item['name'] != name]
    flash('Item removed from cart.', 'success')
    return redirect(url_for('cart_page'))
```

-

  -

- **Order Route:**

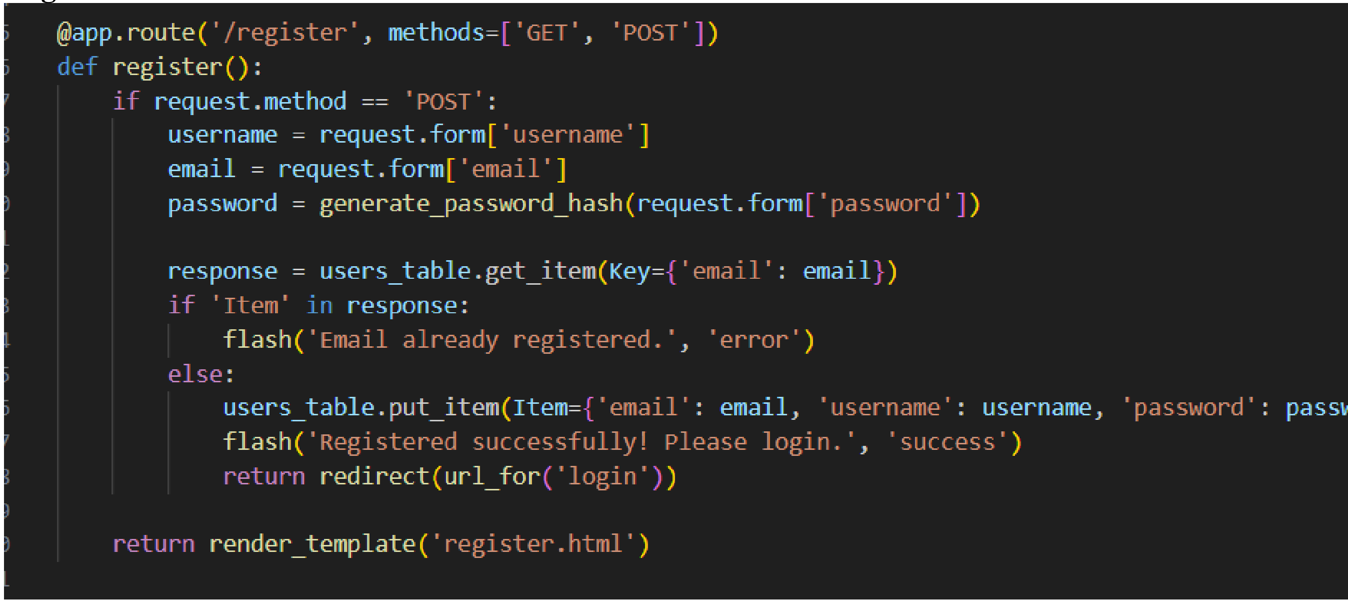


```python
@app.route('/order', methods=['GET', 'POST'])
def order():
    if 'email' not in session:
        flash("Please login to place an order.")
        return redirect(url_for('login'))

    if request.method == 'POST':
        email = session['email']
        address = request.form.get('address')

        response = cart_table.query(
            KeyConditionExpression=boto3.dynamodb.conditions.Key('email').eq(email)
        )
        items = response.get('Items', [])

        for item in items:
            cart_table.delete_item(
                Key={
                    'email': email,
                    'item_id': item['item_id']
                }
            )

        send_confirmation_email(email, address)

        flash("Order placed successfully! (COD)")
        return redirect(url_for('home'))

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

Description: Here's a slightly longer version that's still concise but more detailed:

The order() function is responsible for handling the order placement process. It begins by checking if the user is logged in using session data. If the user is not authenticated, they are redirected to the login page. When the method is POST, the function retrieves the user's email from the session and the entered delivery address from the form. It then queries the Cart table in DynamoDB to fetch all cart items associated with that email. For each item found, it deletes the item from the cart to simulate order processing. After clearing the cart, it calls the send_confirmation_email() function to notify the user with the delivery address and order confirmation. Finally, it displays a flash message indicating successful order placement and redirects the user to the home page.

- **Register Route:**



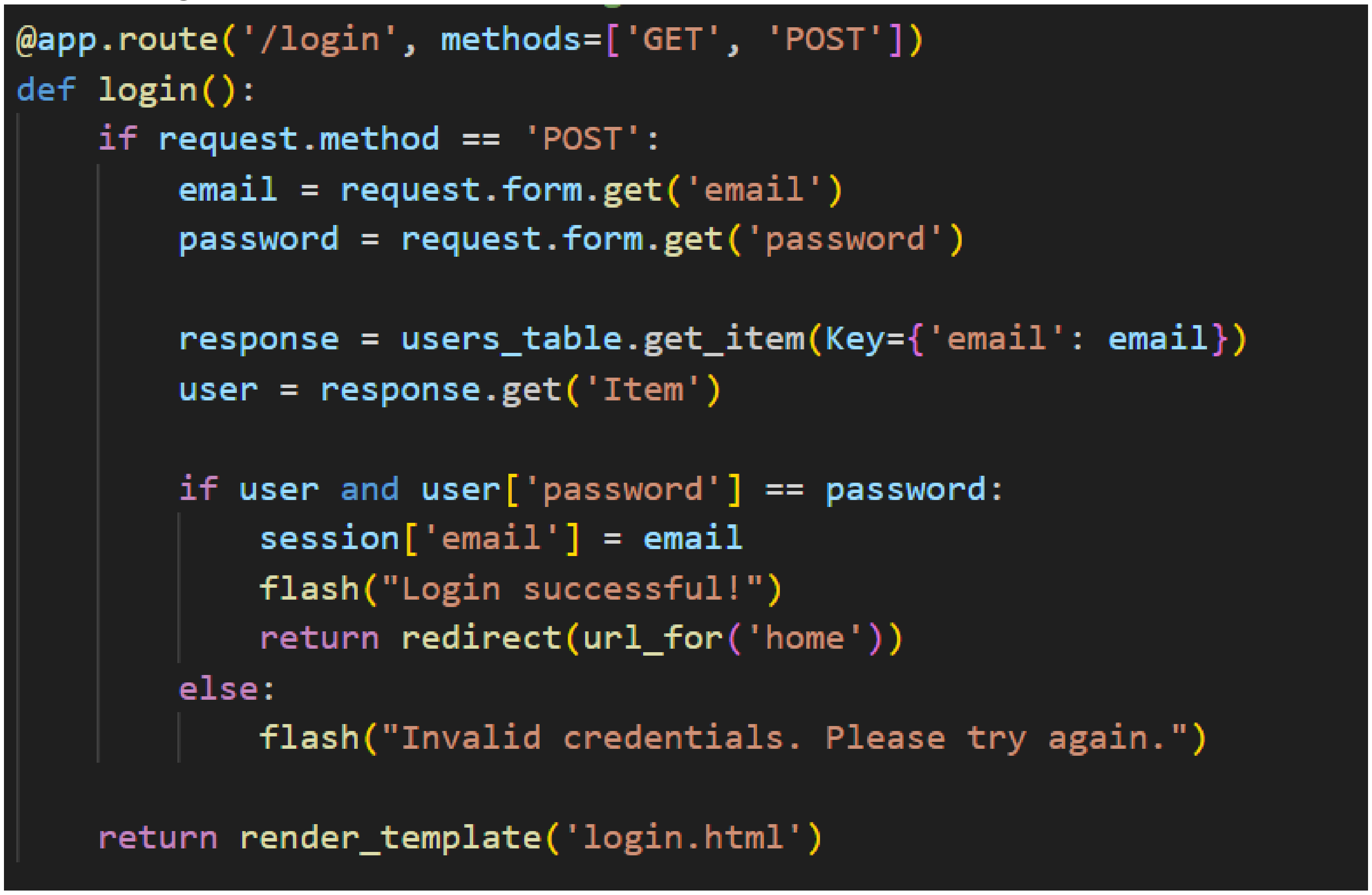
```
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        email = request.form['email']
        password = generate_password_hash(request.form['password'])

        response = users_table.get_item(Key={'email': email})
        if 'Item' in response:
            flash('Email already registered.', 'error')
        else:
            users_table.put_item(Item={'email': email, 'username': username, 'password': passw
            flash('Registered successfully! Please login.', 'success')
            return redirect(url_for('login'))

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

-

car

Description: This Flask Python code handles user registration at the /register route, supporting both GET and POST methods. For POST requests, it extracts 'name', 'email', 'phone', and 'password' from the submitted form, then stores this data as a new item in a users_table (likely a database). Upon successful data storage, it flashes a "Registration successful!" message and redirects the user to the login page; otherwise, for GET requests, it renders the register.html template to display the registration form.

- **Login Routes:**

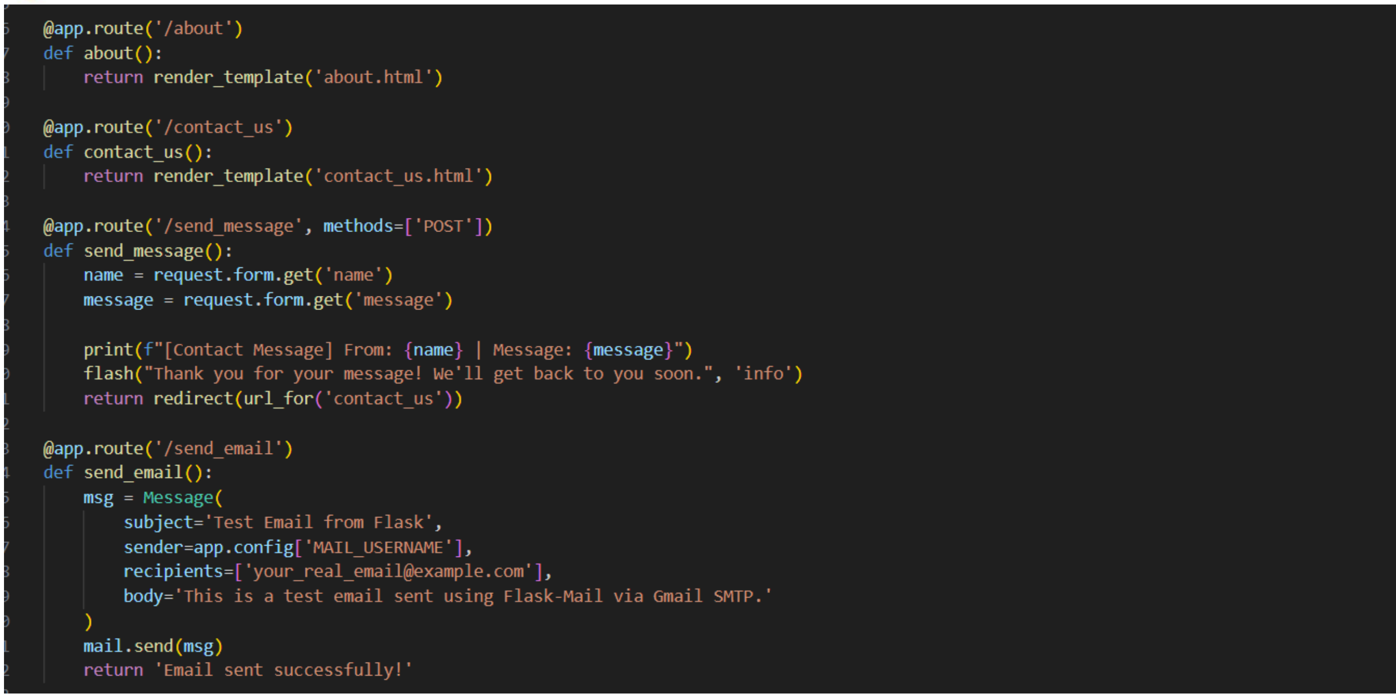


```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')

        response = users_table.get_item(Key={'email': email})
        user = response.get('Item')

        if user and user['password'] == password:
            session['email'] = email
            flash("Login successful!")
            return redirect(url_for('home'))
        else:
            flash("Invalid credentials. Please try again.")

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

-

Description: This Python Flask code defines a /login route that handles user authentication. When a POST request is received, it retrieves the 'email' and 'password' submitted in the form. It then attempts to fetch a user record from the users_table using the provided email as the key. If a user is found and the stored password matches the provided password, the user's email is stored in the session, a "Login successful!" message is flashed, and the user is redirected to the 'home' page. If the credentials do not match, an "Invalid credentials" message is flashed. For GET requests, the code renders the login.html template, which likely displays the login form.

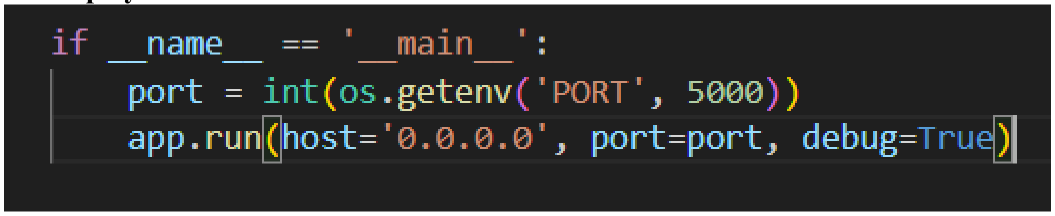
Other Routes:



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

@app.route('/contact_us')
def contact_us():
    return render_template('contact_us.html')

@app.route('/send_message', methods=['POST'])
def send_message():
    name = request.form.get('name')
    message = request.form.get('message')

    print(f"[Contact Message] From: {name} | Message: {message}")
    flash("Thank you for your message! We'll get back to you soon.", 'info')
    return redirect(url_for('contact_us'))

@app.route('/send_email')
def send_email():
    msg = Message(
        subject='Test Email from Flask',
        sender=app.config['MAIL_USERNAME'],
        recipients=['your_real_email@example.com'],
        body='This is a test email sent using Flask-Mail via Gmail SMTP.'
    )
    mail.send(msg)
    return 'Email sent successfully!'
```

Description: This Python Flask code establishes four distinct routes to serve static web pages: the root URL (/) renders welcome.html as the landing page, /home displays home.html for the main content, /contact shows contact.html for contact information, and /about presents about.html for details about the application or organization. These routes collectively define the foundational navigation and content delivery for a Flask-based web application.

Deployment Code:

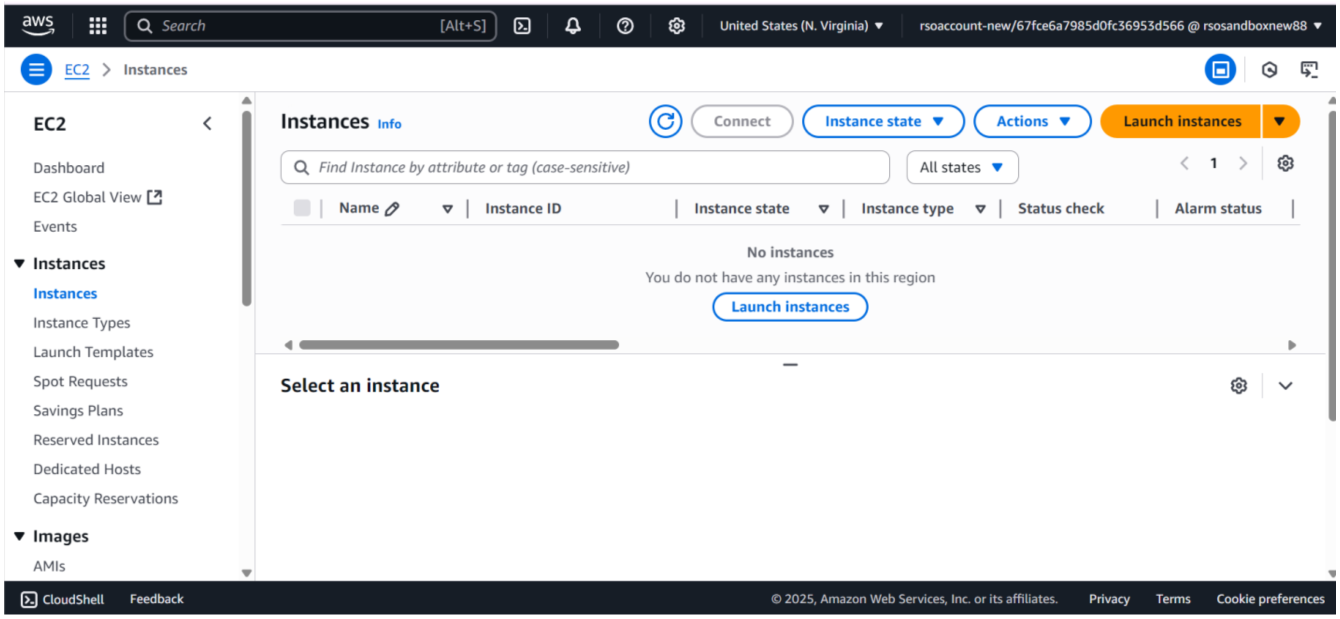


```
if __name__ == '__main__':
    port = int(os.getenv('PORT', 5000))
    app.run(host='0.0.0.0', port=port, debug=True)
```

Description: start the Flask server to listen on all network interfaces (0.0.0.0) at port 5000 with debug mode enabled for development and testing.

Milestone 3:

Create an instance in EC2.
3.1:click on the launch instance button

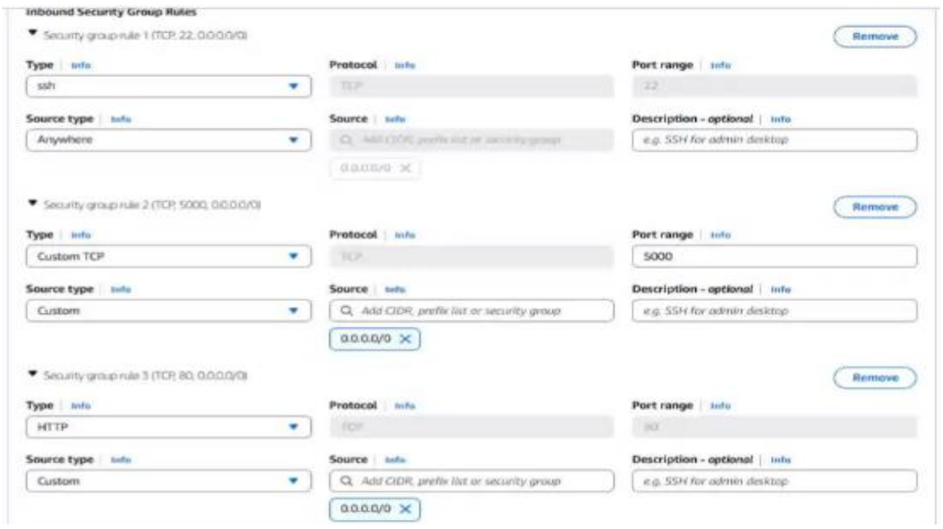


Instances

No instances

You do not have any instances in this region

3.2:



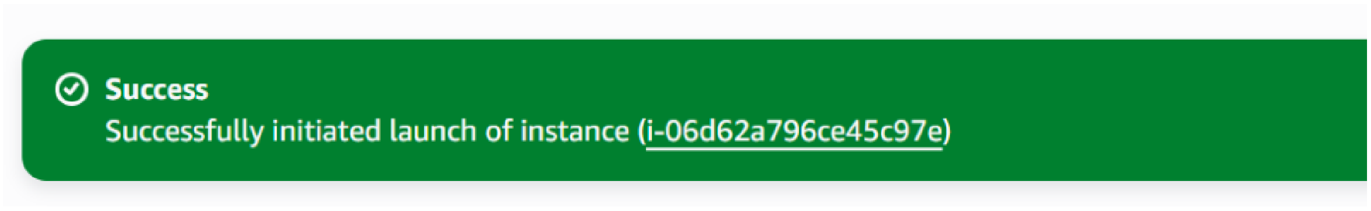
Inbound Security Group Rules

Security group rule 1 (TCP, 22, 0.0.0.0/0)
Type: ssh
Protocol: TCP
Port range: 22
Source type: Anywhere
Source: 0.0.0.0/0
Description - optional: e.g. SSH for admin desktop

Security group rule 2 (TCP, 5000, 0.0.0.0/0)
Type: Custom TCP
Protocol: TCP
Port range: 5000
Source type: Custom
Source: 0.0.0.0/0
Description - optional: e.g. SSH for admin desktop

Security group rule 3 (TCP, 80, 0.0.0.0/0)
Type: HTTP
Protocol: TCP
Port range: 80
Source type: Custom
Source: 0.0.0.0/0
Description - optional: e.g. SSH for admin desktop

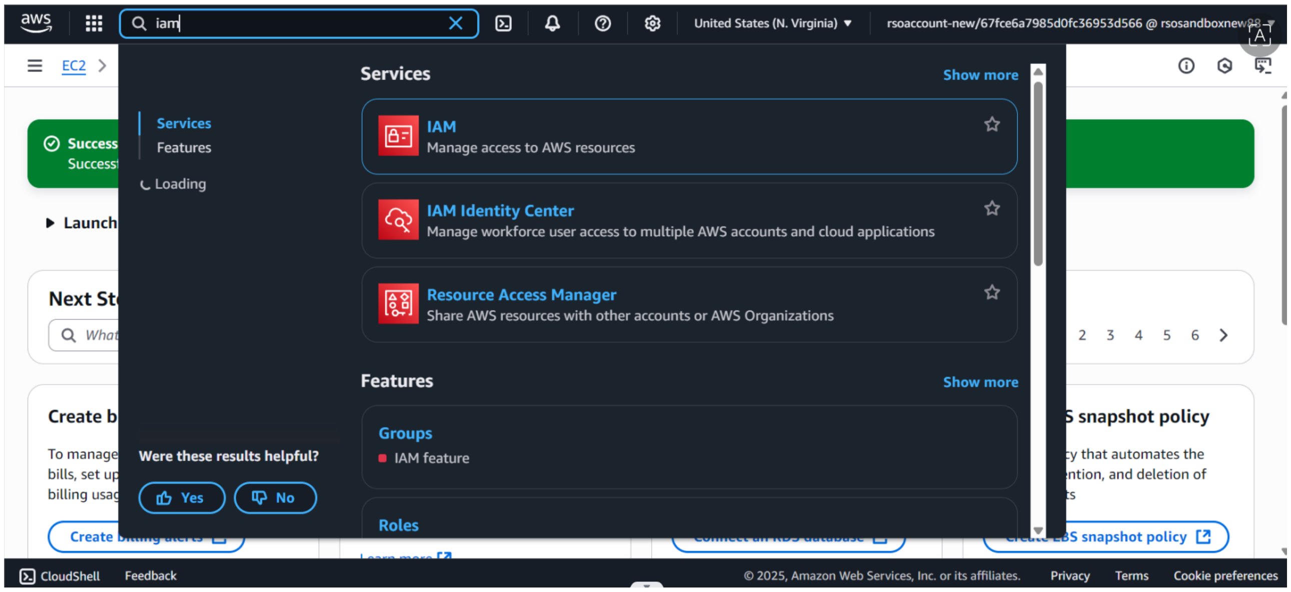
Verification :


Success
Successfully initiated launch of instance (i-06d62a796ce45c97e)

Milestone 4:IAM

Activity 4.1:

Search for IAM in AWS console.



iam
United States (N. Virginia)
rsoaccount-new/67fce6a7985d0fc36953d566 @ rsosandboxnew
Services
Features
Loading
Show more
Were these results helpful?
Yes
No

Services

IAM
Manage access to AWS resources

IAM Identity Center
Manage workforce user access to multiple AWS accounts and cloud applications

Resource Access Manager
Share AWS resources with other accounts or AWS Organizations

Features

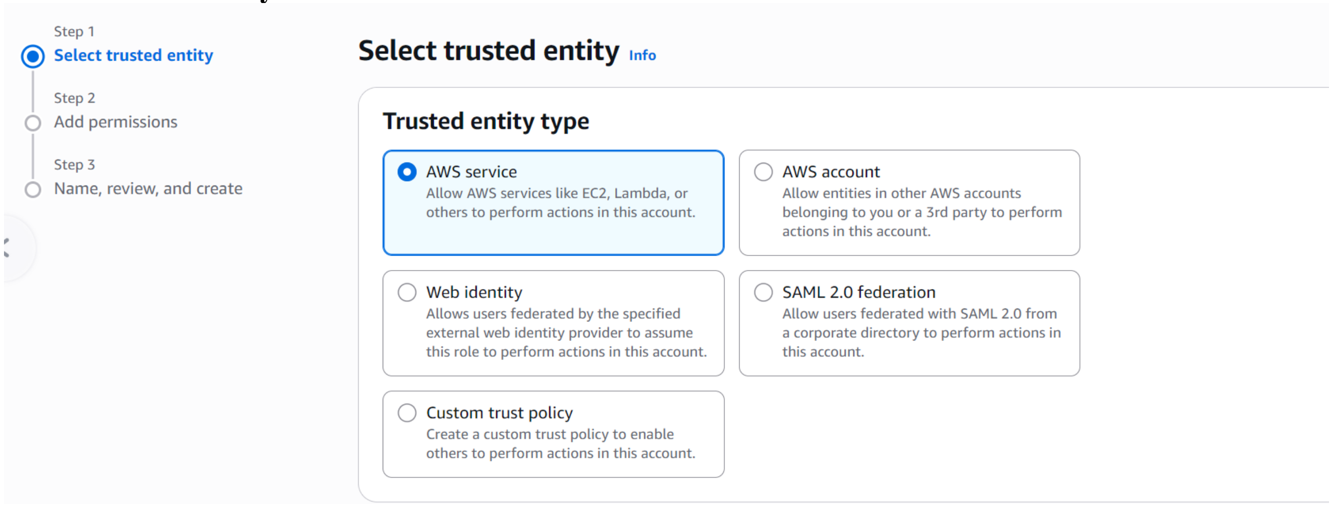
Groups
IAM feature

Roles

CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

Activity 4.2:

Select trusted entity as AWS service.


Step 1
Select trusted entity

Step 2
Add permissions

Step 3
Name, review, and create

Select trusted entity Info

Trusted entity type

[✓ AWS service]
Allow AWS services like EC2, Lambda, or others to perform actions in this account.

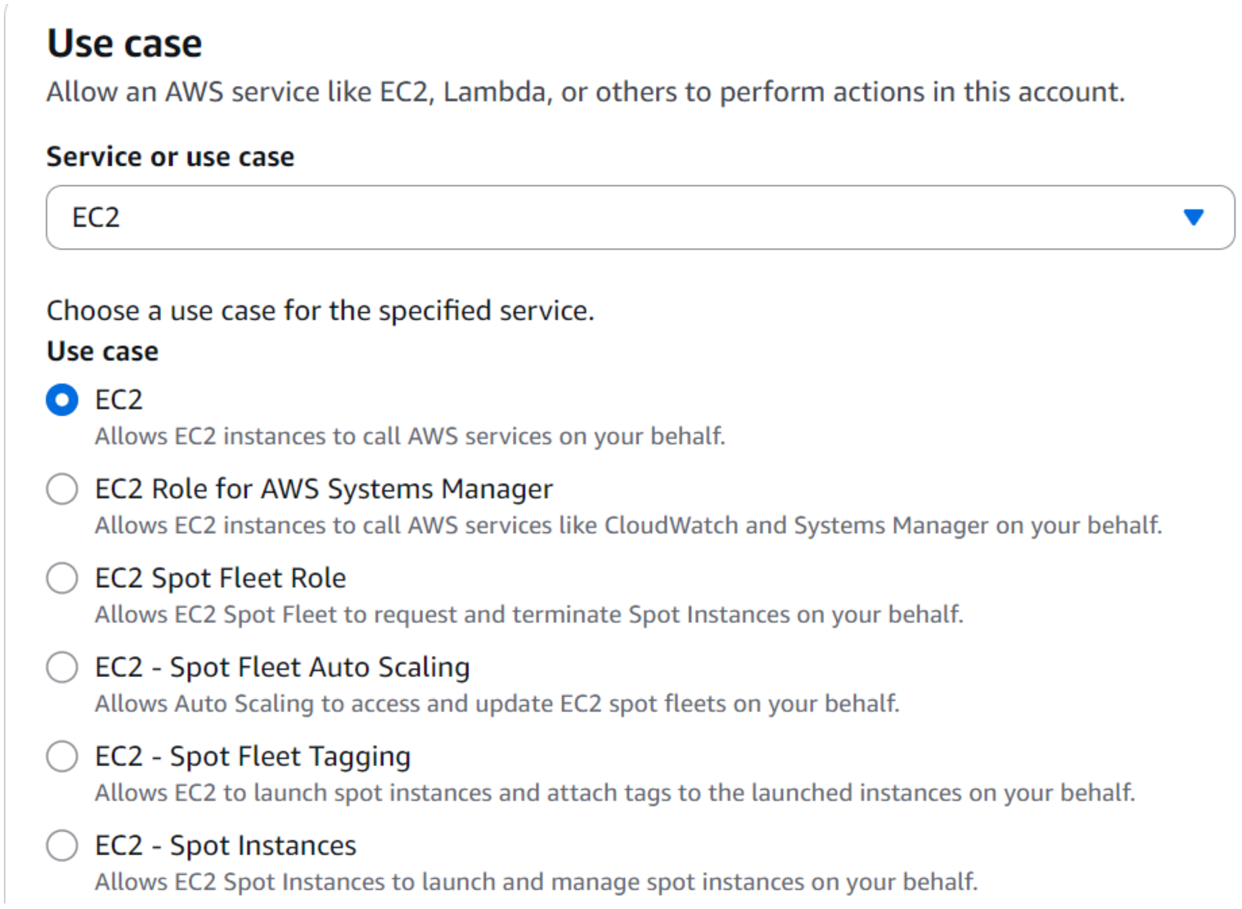
AWS account
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.

Web identity
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.

SAML 2.0 federation
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.

Custom trust policy
Create a custom trust policy to enable others to perform actions in this account.

Activity 4.3:select ec2 as use case



Use case

Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

Service or use case

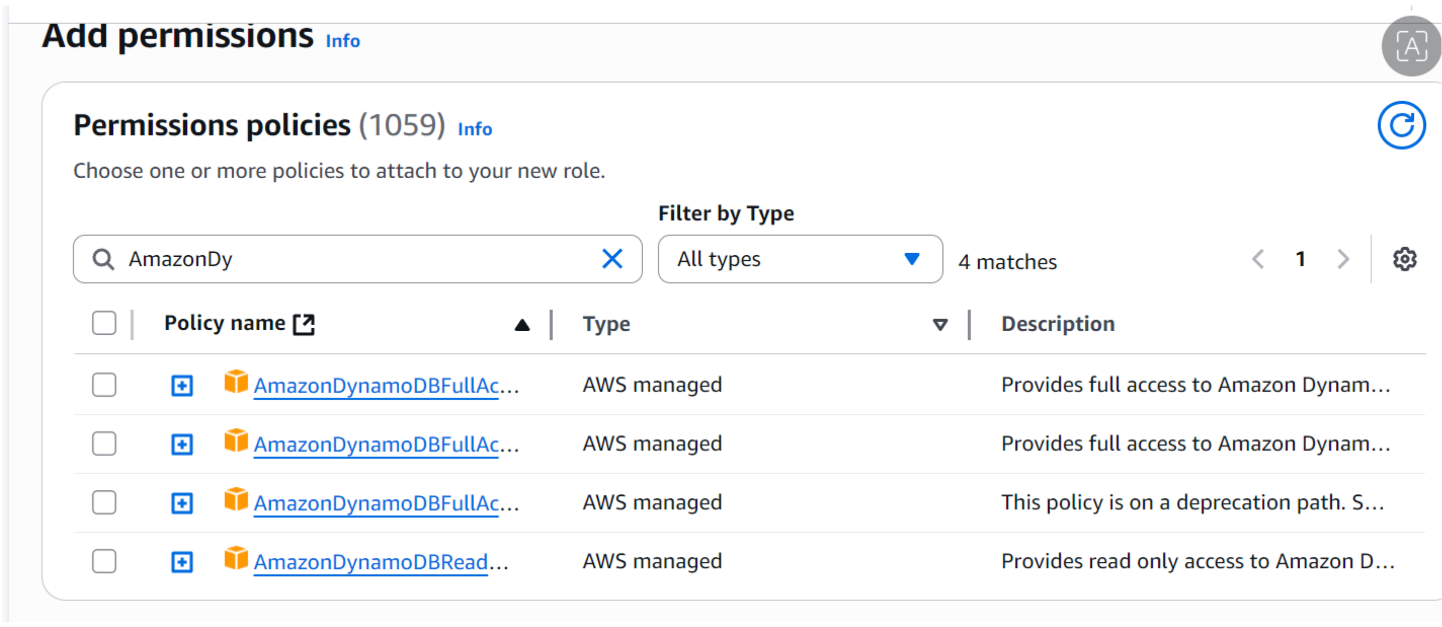
EC2

Choose a use case for the specified service.

Use case

- [✓ EC2] Allows EC2 instances to call AWS services on your behalf.
- EC2 Role for AWS Systems Manager — Allows EC2 instances to call AWS services like CloudWatch and Systems Manager on your behalf.
- EC2 Spot Fleet Role — Allows EC2 Spot Fleet to request and terminate Spot Instances on your behalf.
- EC2 - Spot Fleet Auto Scaling — Allows Auto Scaling to access and update EC2 spot fleets on your behalf.
- EC2 - Spot Fleet Tagging — Allows EC2 to launch spot instances and attach tags to the launched instances on your behalf.
- EC2 - Spot Instances — Allows EC2 Spot Instances to launch and manage spot instances on your behalf.

Activity 4.4:



Add permissions Info

Permissions policies (1059) Info

Choose one or more policies to attach to your new role.

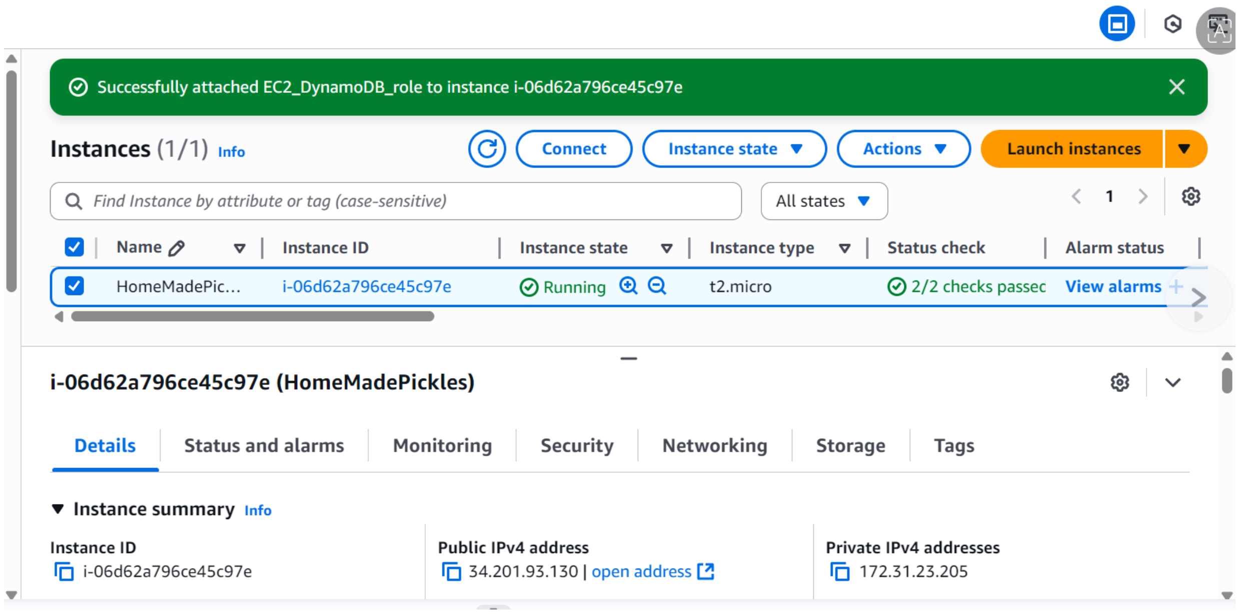
Filter by Type

AmazonDy | All types | 4 matches | 1

Policy name	Type	Description
AmazonDynamoDBFullAc…	AWS managed	Provides full access to Amazon Dynam…
AmazonDynamoDBFullAc…	AWS managed	Provides full access to Amazon Dynam…
AmazonDynamoDBFullAc…	AWS managed	This policy is on a deprecation path. S…
AmazonDynamoDBRead…	AWS managed	Provides read only access to Amazon D…

MileStone 5:

Open aws console, search for ec2 and modify the role to EC2_DynamoDB_role.



Successfully attached EC2_DynamoDB_role to instance i-06d62a796ce45c97e

Instances (1/1) Info

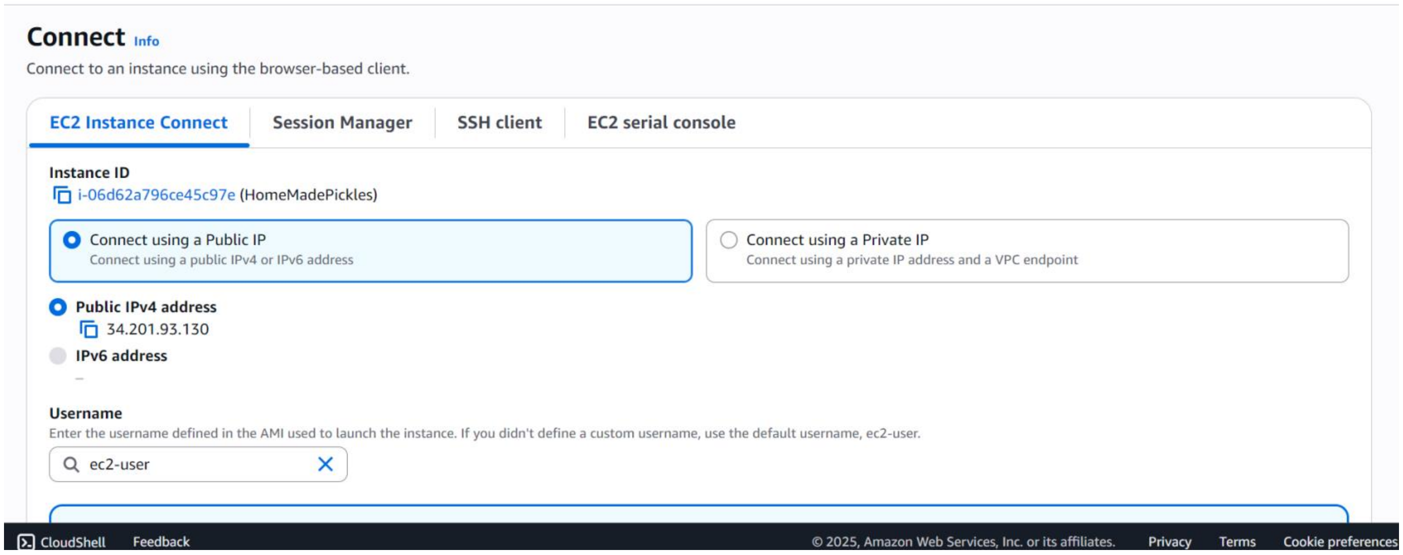
Name	Instance ID	Instance state	Instance type	Status check	Alarm status
HomeMadePic…	i-06d62a796ce45c97e	Running	t2.micro	2/2 checks passec	View alarms

i-06d62a796ce45c97e (HomeMadePickles)

Instance summary Info

Instance ID	Public IPv4 address	Private IPv4 addresses
i-06d62a796ce45c97e	34.201.93.130 | open address	172.31.23.205

MileStone 6: click on connect and open the terminal



Connect Info

Connect to an instance using the browser-based client.

Instance ID

i-06d62a796ce45c97e (HomeMadePickles)

Connect using a Public IP
Connect using a public IPv4 or IPv6 address

Connect using a Private IP
Connect using a private IP address and a VPC endpoint

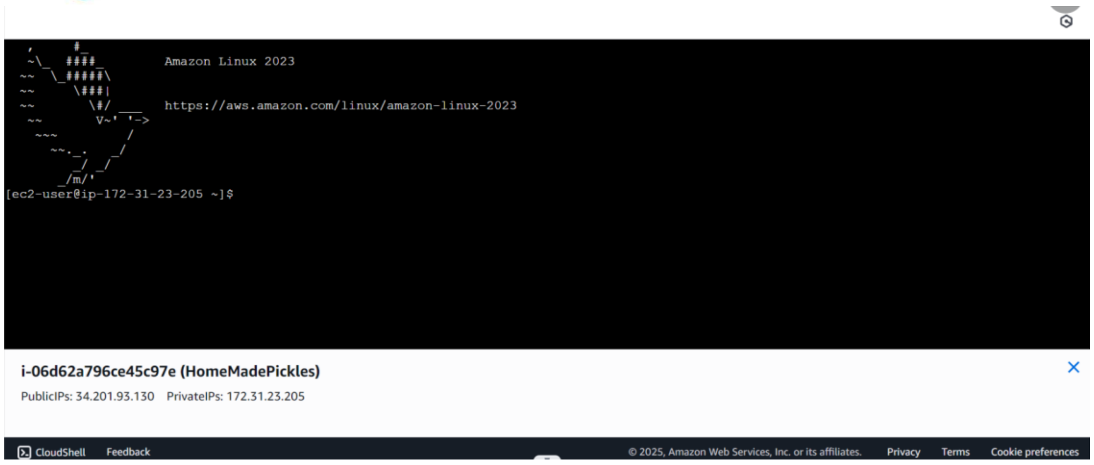
Public IPv4 address
34.201.93.130

IPv6 address
–

Username

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

ec2-user



```
   ,     #_
   ~\_  ####_        Amazon Linux 2023
  ~~  \_#####\
  ~~     \###|
  ~~       \#/ ___   https://aws.amazon.com/linux/amazon-linux-2023
   ~~       V~' '->
    ~~~         /
      ~~._.   _/
         _/ _/
       _/m/'
[ec2-user@ip-172-31-23-205 ~]$
```

i-06d62a796ce45c97e (HomeMadePickles)

PublicIPs: 34.201.93.130 PrivateIPs: 172.31.23.205

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/HarshithaPandranki/smart_interns_project.git

- This will download your project to the EC2 instance.

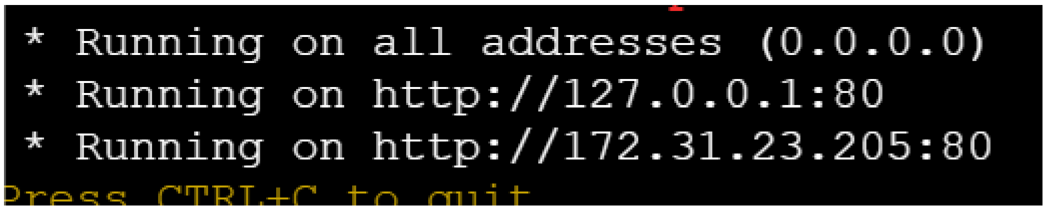
To navigate to the project directory, run the following command:

```
cd smart_interns_project.git
```

Once inside the project directory, configure and run the Flask application by executing the following command with elevated privileges:

Run the Flask Application

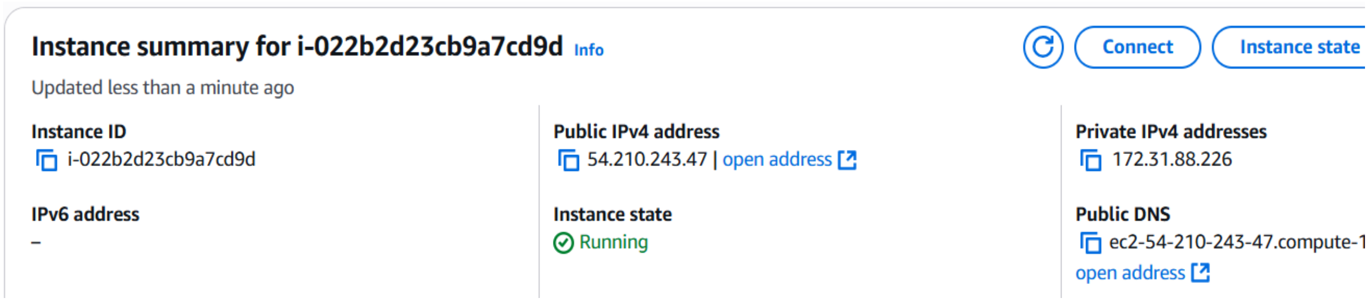
```
sudo flask run --host=0.0.0.0 --port=5000
```



```
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:80
* Running on http://172.31.23.205:80
Press CTRL+C to quit
```

Verify the Flask app is running: http://your-ec2-public-ip

- ○ Run the Flask app on the EC2 instance
- ○



Instance summary for i-022b2d23cb9a7cd9d Info

Updated less than a minute ago

Instance ID
i-022b2d23cb9a7cd9d

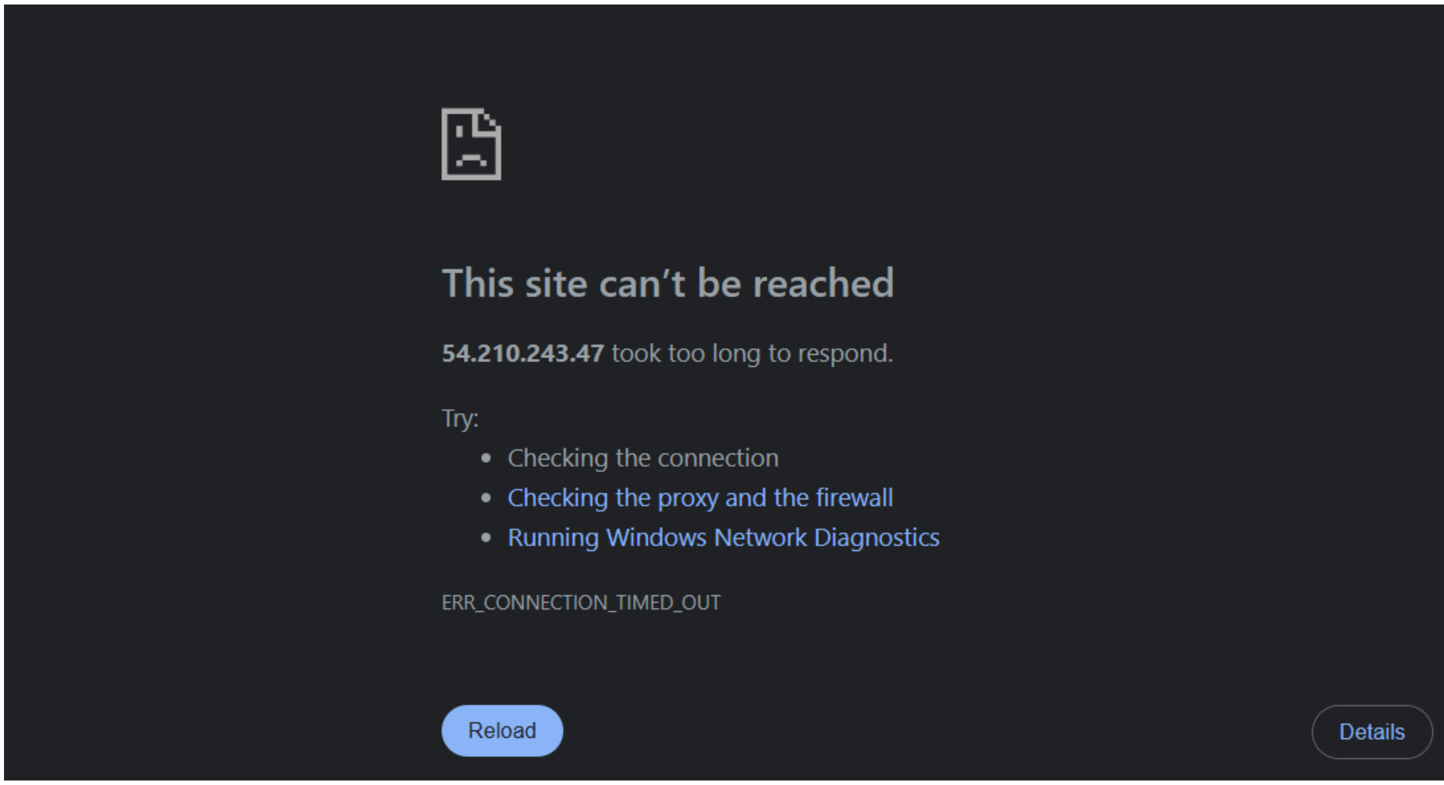
Public IPv4 address
54.210.243.47 | open address

Private IPv4 addresses
172.31.88.226

IPv6 address
–

Instance state
Running

Public DNS
ec2-54-210-243-47.compute-1
open address



This site can't be reached

54.210.243.47 took too long to respond.

Try:

- Checking the connection
- Checking the proxy and the firewall
- Running Windows Network Diagnostics

ERR_CONNECTION_TIMED_OUT

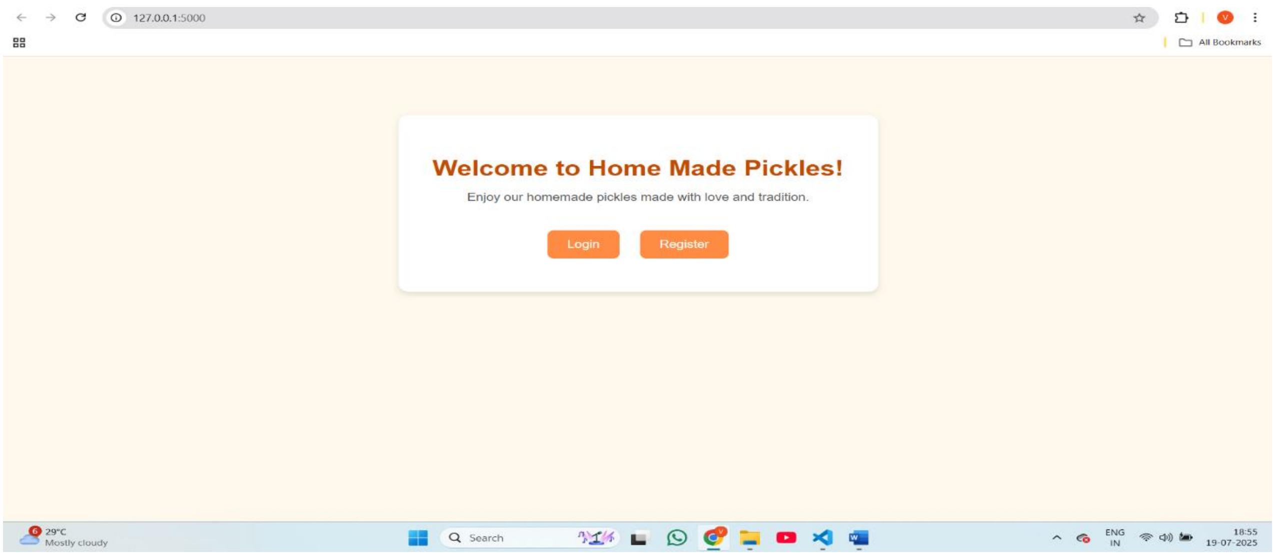
Access the website through(locally deployed):
http://127.0.0.1:5000

Milestone 8: Testing and Deployment

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

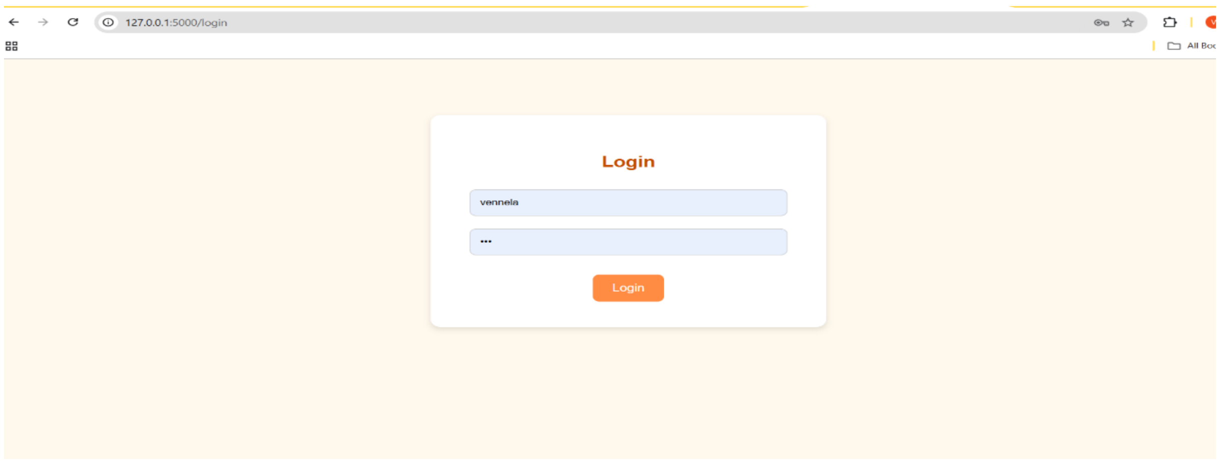
Login Page:

Register Page:

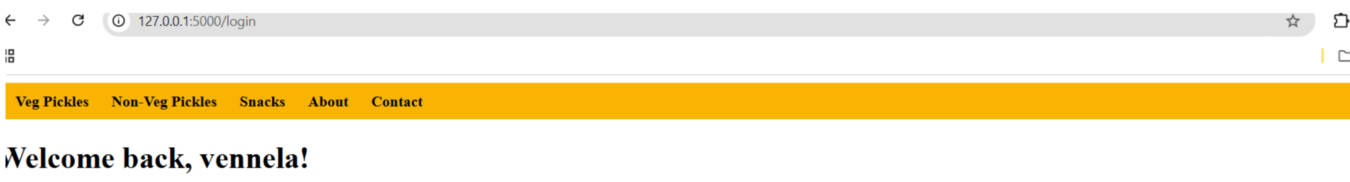


Welcome to Home Made Pickles!

Enjoy our homemade pickles made with love and tradition.

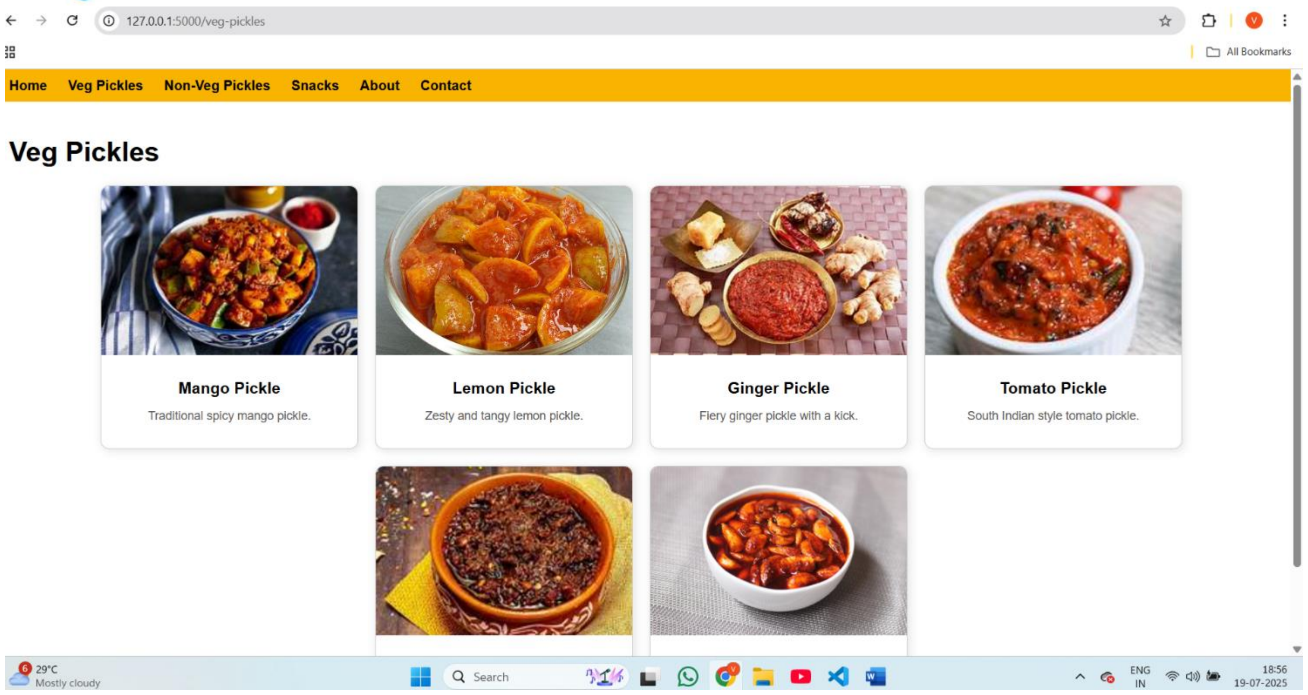


Welcome page :



Welcome back, vennela!

Home page:



127.0.0.1:5000/veg-pickles

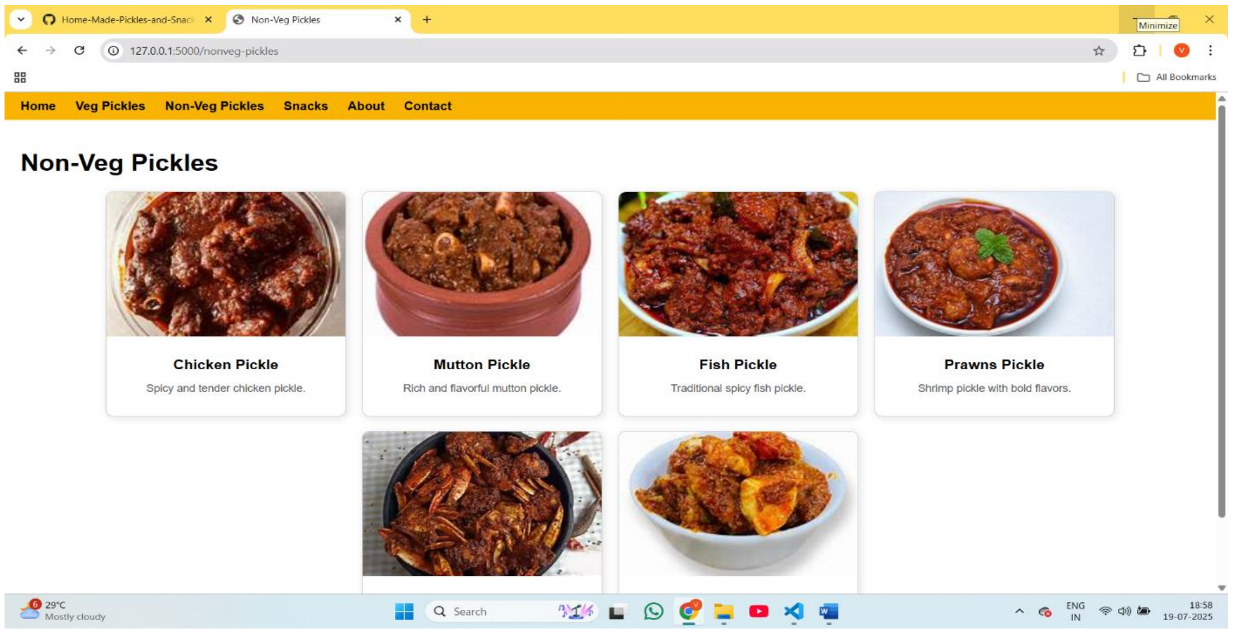
Veg Pickles

Mango Pickle
Traditional spicy mango pickle.

Lemon Pickle
Zesty and tangy lemon pickle.

Ginger Pickle
Fiery ginger pickle with a kick.

Tomato Pickle
South Indian style tomato pickle.



127.0.0.1:5000/nonveg-pickles

Non-Veg Pickles

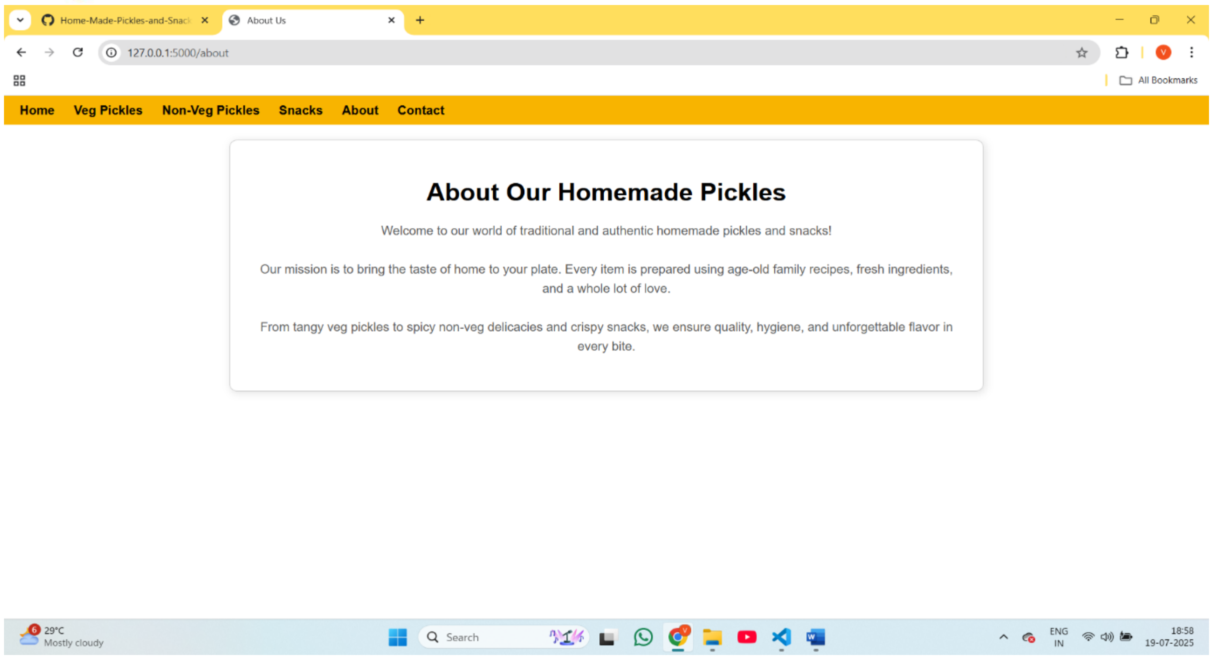
Chicken Pickle
Spicy and tender chicken pickle.

Mutton Pickle
Rich and flavorful mutton pickle.

Fish Pickle
Traditional spicy fish pickle.

Prawns Pickle
Shrimp pickle with bold flavors.

About Us page:



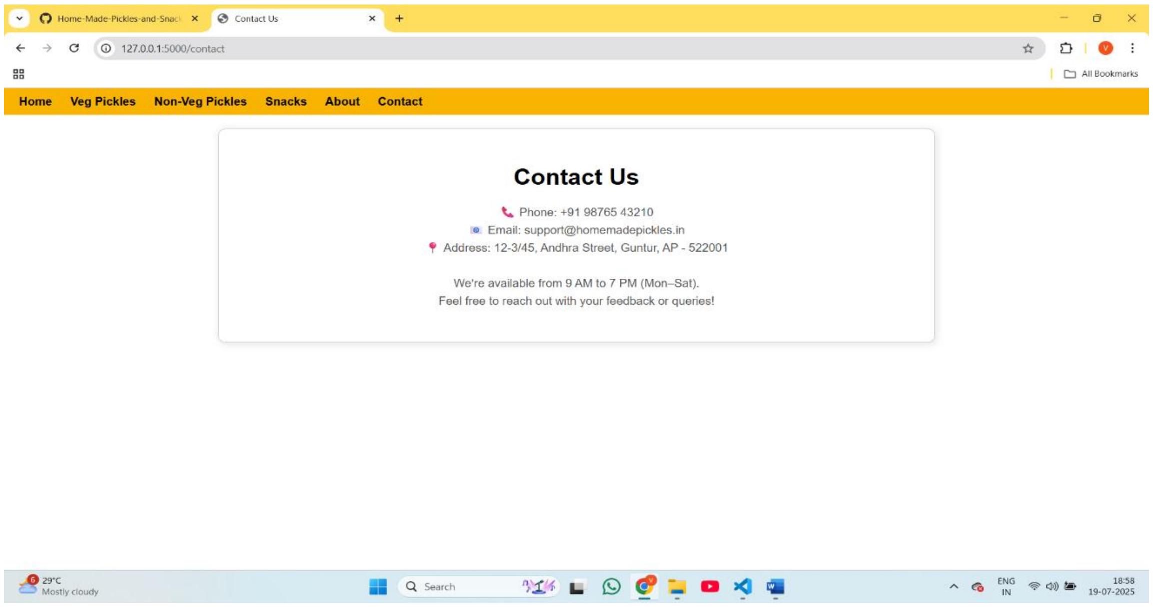
About Our Homemade Pickles

Welcome to our world of traditional and authentic homemade pickles and snacks!

Our mission is to bring the taste of home to your plate. Every item is prepared using age-old family recipes, fresh ingredients, and a whole lot of love.

From tangy veg pickles to spicy non-veg delicacies and crispy snacks, we ensure quality, hygiene, and unforgettable flavor in every bite.

Contact Page:



Contact Us

📞 Phone: +91 98765 43210
📧 Email: support@homemadepickles.in
📍 Address: 12-3/45, Andhra Street, Guntur, AP - 522001

We're available from 9 AM to 7 PM (Mon–Sat).
Feel free to reach out with your feedback or queries!

Conclusion:
Homemade pickles and snacks are more than just delicious treats—they represent tradition, health, and the essence of authentic, handcrafted food. By using natural ingredients and traditional recipes passed down through generations, homemade products bring the warmth of home into every bite. They cater to the growing demand for preservative-free, nutritious, and flavourful alternatives to mass-produced snacks.
Our Homemade Pickles and Snacks platform is built to share this rich culinary heritage with everyone, offering a simple, secure, and delightful online shopping experience. Whether it's spicy veg pickles, tangy non-veg varieties, or crunchy snacks, each product reflects care, authenticity, and a love for wholesome, homemade food.