E-Commerce site for HomeMade Pickles & Snacks: Taste the Best

Project Description:

Home Made Pickles & Snacks — Taste the Best is a cloud-based culinary platform revolutionizing access to authentic, handcrafted pickles and snacks. Addressing the growing demand for preservative-free, traditional recipes, this initiative combines artisanal craftsmanship with cutting-edge technology to deliver farm-fresh flavors directly to consumers. Built on Flask for backend efficiency and hosted on AWS EC2 for scalable performance, the platform offers seamless browsing, ordering, and subscription management. DynamoDB ensures real-time inventory tracking and personalized user experiences, while fostering sustainability through partnerships with local farmers and eco-friendly packaging. From tangy regional pickles to wholesome snacks, every product celebrates heritage recipes, nutritional integrity, and convenience—proving that tradition and innovation can coexist deliciously. "Preserving Traditions, One Jar at a Time."

Scenarios:

Scenario 1: Scalable Order Management for High Demand

A cloud-based system ensures seamless order processing during peak user activity. For instance, during a promotional event, hundreds of users simultaneously access the platform to place orders. The backend efficiently processes requests, updates inventory in real-time, and manages user sessions. The cloud infrastructure handles traffic spikes without performance degradation, ensuring smooth transactions and minimizing wait times.

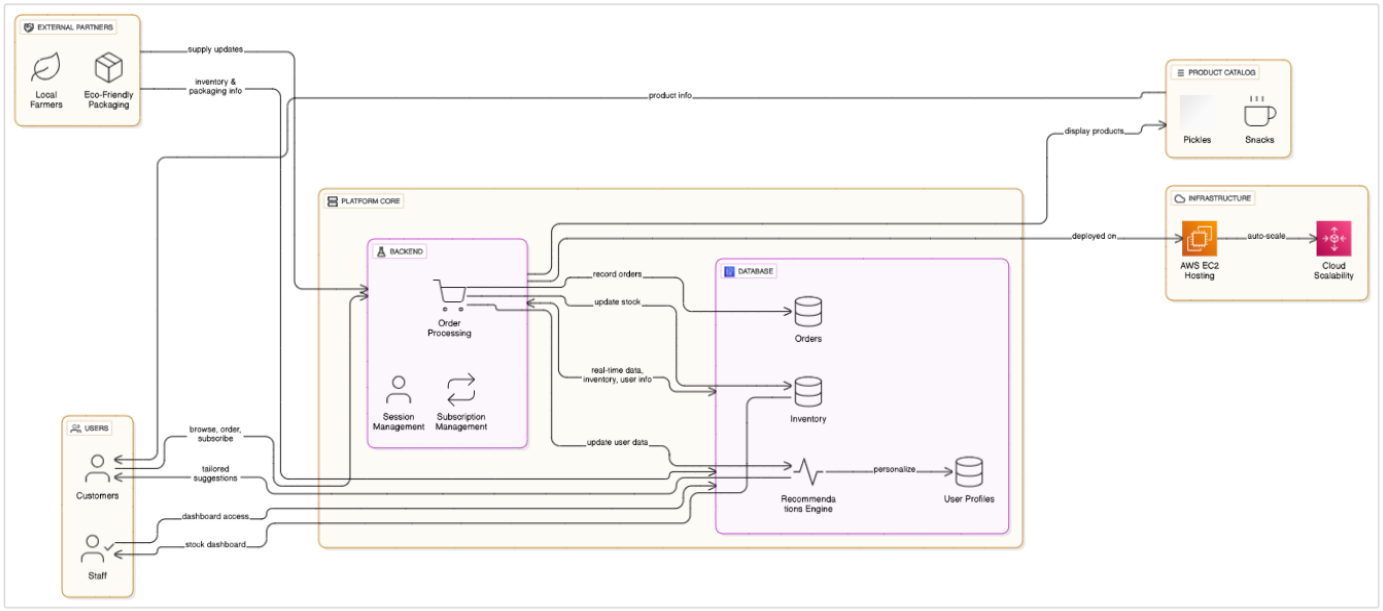
Scenario 2: Real-Time Inventory Tracking and Updates

When a customer places an order for a product, the system instantly updates stock levels and records transaction details. For example, a user purchases an item, triggering automatic inventory deduction and order confirmation. Staff members receive updated dashboards to monitor stock availability and fulfillment progress, ensuring timely restocking and minimizing overselling risks.

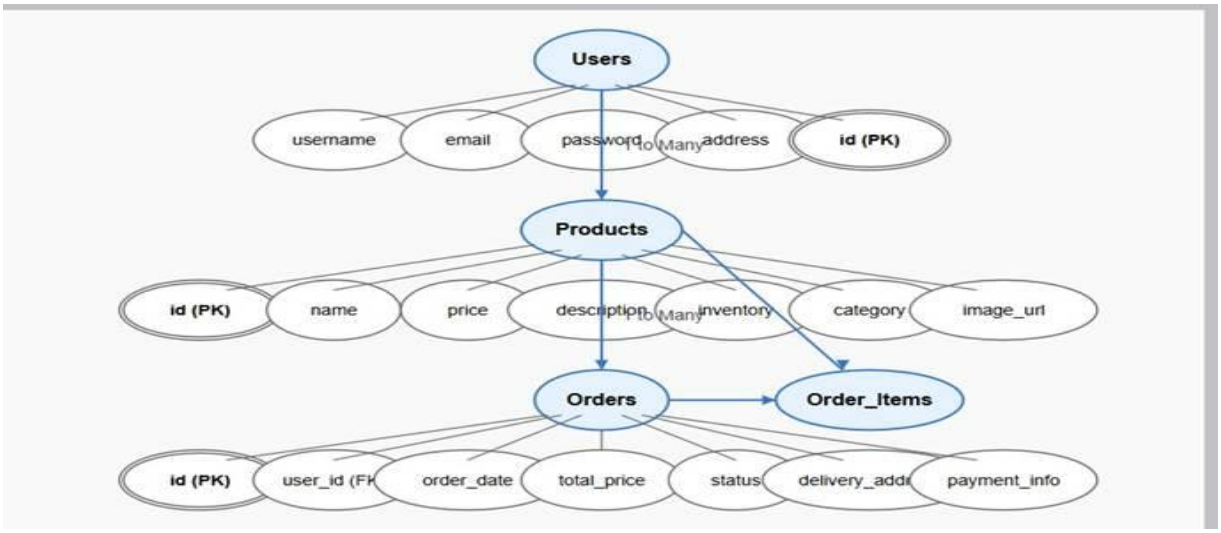
Scenario 3: Personalized User Experience and Recommendations

The platform leverages user behavior data to enhance engagement. A returning customer, for instance, views tailored recommendations based on past purchases and browsing history. The system dynamically adjusts suggestions in real-time, while maintaining fast response rates even during high traffic, creating a frictionless and intuitive shopping experience.

AWS ARCHITECTURE



Entity Relationship (ER)Diagram:



Pre-requisites:

1. **AWS Account Setup**: https://docs.aws.amazon.com/accounts/latest/reference/getting-started.html
2. **Understanding IAM**: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html
3. **Amazon EC2 Basics**: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

4. **DynamoDB Basics**: https://docs.aws.amazon.com/amazondynamodb/Introduction.html
5. **Git Documentation: https://git-scm.com/doc**
6. **VS Code Installation**: https://code.visualstudio.com/download

Project WorkFlow :

Activity 1: Backend Development and Application Setup

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

Activity 2: AWS Account Setup and Login

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

Activity 3: DynamoDB Database Creation and Setup

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

Activity 4: SNS Notification Setup

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

Activity 5: IAM Role Setup

- **Create IAM Role**
- **Attach Policies**

Activity 6: EC2 Instance Setup

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

Activity 7: Deployment on EC2

- **Upload Flask Files**

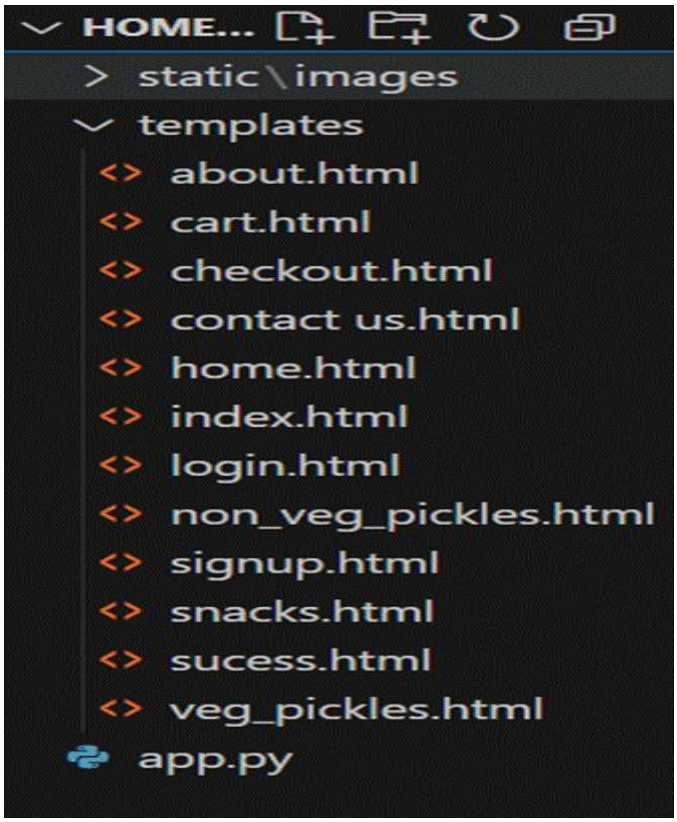
- **Run the Flask App**

Activity 8: Testing and Deployment

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

Milestone 1 : Web Application Development and Setup

- **Activity 1.1: Develop the Backend code Using Flask.**
  - File Explorer Structure



Description of the code :

Flask App Initialization:

```
from flask import Flask, render_template, request, redirect, url_for, session
from werkzeug.security import generate_password_hash, check_password_hash
import boto3
from datetime import datetime
import json,uuid
```

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

- Use boto3 to connect to DynamoDB for handling user registration, Order details database operations and also mention region_name where Dynamodb tables are created.

```
dynamodb = boto3.resource('dynamodb', region_name='ap-south-1')  # e.g., 'us-east-1'
users_table = dynamodb.Table('Users')
orders_table = dynamodb.Table('Orders')
```

```
products = {
    'non_veg_pickles': [
        {'id': 1, 'name': 'Chicken Pickle', 'weights': {'250': 600, '500': 1200, '1000': 1800}},
        {'id': 2, 'name': 'Fish Pickle', 'weights': {'250': 200, '500': 400, '1000': 800}},
        {'id': 3, 'name': 'Gongura Mutton', 'weights': {'250': 400, '500': 800, '1000': 1600}},
        {'id': 4, 'name': 'Mutton Pickle', 'weights': {'250': 400, '500': 800, '1000': 1600}},
        {'id': 5, 'name': 'Gongura Prawns', 'weights': {'250': 600, '500': 1200, '1000': 1800}},
        {'id': 6, 'name': 'Chicken Pickle (Gongura)', 'weights': {'250': 350, '500': 700, '1000': 1050}}
    ],
    'veg_pickles': [
        {'id': 7, 'name': 'Traditional Mango Pickle', 'weights': {'250': 150, '500': 280, '1000': 500}},
        {'id': 8, 'name': 'Zesty Lemon Pickle', 'weights': {'250': 120, '500': 220, '1000': 400}},
        {'id': 9, 'name': 'Tomato Pickle', 'weights': {'250': 130, '500': 240, '1000': 450}},
        {'id': 10, 'name': 'Kakarakaya Pickle', 'weights': {'250': 130, '500': 240, '1000': 450}},
        {'id': 11, 'name': 'Chintakaya Pickle', 'weights': {'250': 130, '500': 240, '1000': 450}},
        {'id': 12, 'name': 'Spicy Pandu Mirchi', 'weights': {'250': 130, '500': 240, '1000': 450}}
    ],  # Add your veg pickle products here
    'snacks': [
        {'id': 7, 'name': 'Banana Chips', 'weights': {'250': 300, '500': 600, '1000': 800}},
        {'id': 8, 'name': 'Crispy Aam-Papad', 'weights': {'250': 150, '500': 300, '1000': 600}},
        {'id': 9, 'name': 'Crispy Chekka Pakodi', 'weights': {'250': 50, '500': 100, '1000': 200}},
        {'id': 10, 'name': 'Boondhi Acchu', 'weights': {'250': 300, '500': 600, '1000': 900}},
        {'id': 11, 'name': 'Chekkalu', 'weights': {'250': 350, '500': 700, '1000': 1000}},
        {'id': 12, 'name': 'Ragi Laddu', 'weights': {'250': 350, '500': 700, '1000': 1000}},
        {'id': 13, 'name': 'Dry Fruit Laddu', 'weights': {'250': 500, '500': 1000, '1000': 1500}},
        {'id': 14, 'name': 'Kara Boondi', 'weights': {'250': 250, '500': 500, '1000': 750}},
        {'id': 15, 'name': 'Gavvalu', 'weights': {'250': 250, '500': 500, '1000': 750}},
        {'id': 16, 'name': 'Kaju Chikki', 'weights': {'250': 250, '500': 500, '1000': 750}},
```

- Routes for Web Pages
- Login Route (GET/POST):Verifies user credentials, increments login count, and redirects to the dashboard on success.

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

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

        try:
            # Fetch user from DynamoDB
            response = users_table.get_item(Key={'username': username})

            if 'Item' not in response:
                return render_template('login.html', error='User not found')

            user = response['Item']

            # Ensure password field exists in the DB
            if 'password' not in user:
                return render_template('login.html', error='Password not found in database')

            # Verify password
            if check_password_hash(user['password'], password):
                session['logged_in'] = True
                session['username'] = username
                session.setdefault('cart', [])  # Initialize cart if not set
```

- SignUp route: Collecting registration data, hashes the password, and stores user details in the database.

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

        try:
            # Check if username exists
            response = users_table.get_item(Key={'username': username})
            if 'Item' in response:
                return render_template('signup.html', error='Username already exists')

            # Hash password before storing
            hashed_password = generate_password_hash(password)

            # Store new user in DynamoDB
            users_table.put_item(
                Item={
                    'username': username,
                    'email': email,
                    'password': hashed_password  # Store hashed password
                }
            )

            return redirect(url_for('login'))

        except Exception as e:
            app.logger.error(f"Signup error: {str(e)}")
```

```
        except Exception as e:
            app.logger.error(f"Signup error: {str(e)}")
            return render_template('signup.html', error='Registration failed. Please try again.')

    return render_template('signup.html')

@app.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('login'))
```

- Logout route:The user can Logout so that the user can get back to the Login Page

```
@app.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('login'))
```

- Home Route: Home page contains the routing for different categories which are Veg_pickles,Non_Veg_pickles,Snacks.

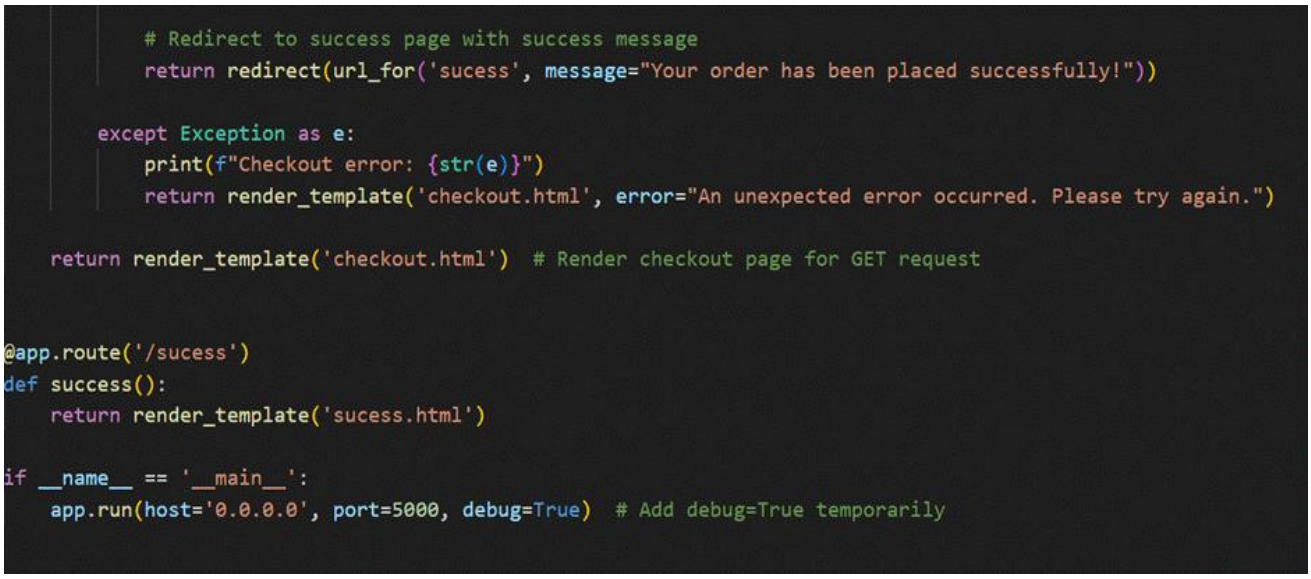
```
@app.route('/home')
def home():
    if not session.get('logged_in'):
        return redirect(url_for('login'))
    return render_template('home.html')

@app.route('/non_veg_pickles')
def non_veg_pickles():
    if not session.get('logged_in'):
        return redirect(url_for('login'))

    return render_template('non_veg_pickles.html', products=products['non_veg_pickles'])

@app.route('/veg_pickles')
def veg_pickles():
    if not session.get('logged_in'):
        return redirect(url_for('login'))

    # Simply pass all products without filtering
    return render_template('veg_pickles.html', products=products['veg_pickles'])

@app.route('/snacks')
def snacks():
    if not session.get('logged_in'):
        return redirect(url_for('login'))

    return render_template('snacks.html', products=products['snacks'])
```

Check out Route:

```python
@app.route('/checkout', methods=['GET', 'POST'])
def checkout():
    if not session.get('logged_in'):
        return redirect(url_for('login'))

    error_message = None  # Variable to hold error messages

    if request.method == 'POST':
        try:
            # Extract form data safely
            name = request.form.get('name', '').strip()
            address = request.form.get('address', '').strip()
            phone = request.form.get('phone', '').strip()
            payment_method = request.form.get('payment', '').strip()

            # Validate inputs
            if not all([name, address, phone, payment_method]):
                return render_template('checkout.html', error="All fields are required.")

            if not phone.isdigit() or len(phone) != 10:
                return render_template('checkout.html', error="Phone number must be exactly 10 digits.")

            # Get cart data from hidden inputs
            cart_data = request.form.get('cart_data', '[]')
            total_amount = request.form.get('total_amount', '0')
```

Restart Visual Studio Code to apply the late

```python
        try:
            cart_items = json.loads(cart_data)
            total_amount = float(total_amount)
        except (json.JSONDecodeError, ValueError):
            return render_template('checkout.html', error="Invalid cart data format.")

        # Ensure cart is not empty
        if not cart_items:
            return render_template('checkout.html', error="Your cart is empty.")

        # Store order in DynamoDB
        try:
            orders_table.put_item(
                Item={
                    'order_id': str(uuid.uuid4()),
                    'username': session.get('username', 'Guest'),
                    'name': name,
                    'address': address,
                    'phone': phone,
                    'items': cart_items,
                    'total_amount': total_amount,
                    'payment_method': payment_method,
                    'timestamp': datetime.now().isoformat()
                }
            )
        except Exception as db_error:
            print(f"DynamoDB Error: {db_error}")
            return render_template('checkout.html', error="Failed to save order. Please try again later.")
```



```
            # Redirect to success page with success message
            return redirect(url_for('sucess', message="Your order has been placed successfully!"))

        except Exception as e:
            print(f"Checkout error: {str(e)}")
            return render_template('checkout.html', error="An unexpected error occurred. Please try again.")

    return render_template('checkout.html')  # Render checkout page for GET request


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

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)  # Add debug=True temporarily
```

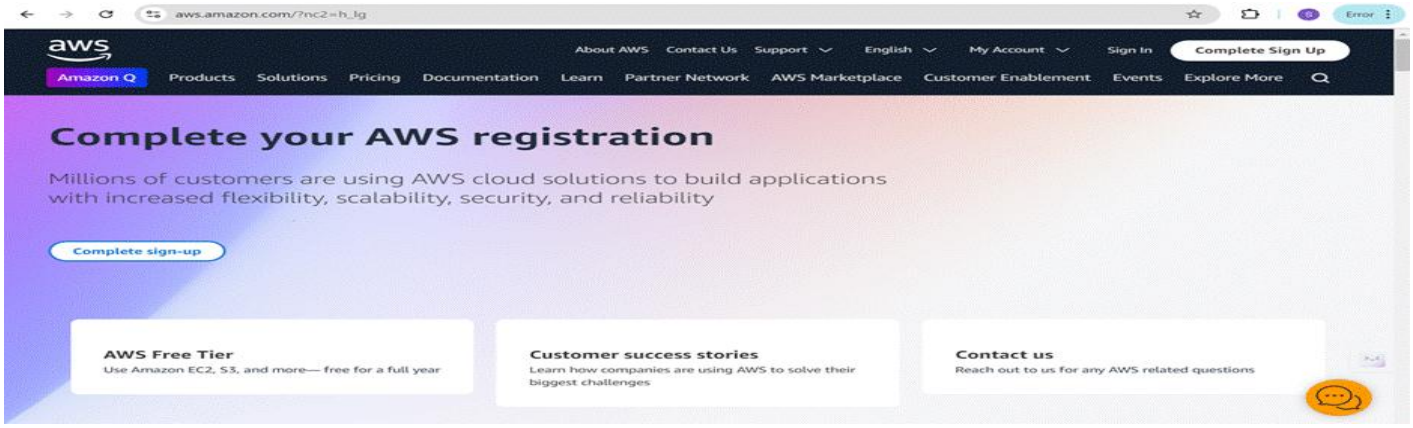
AWS - Local Deployment Sample Code –

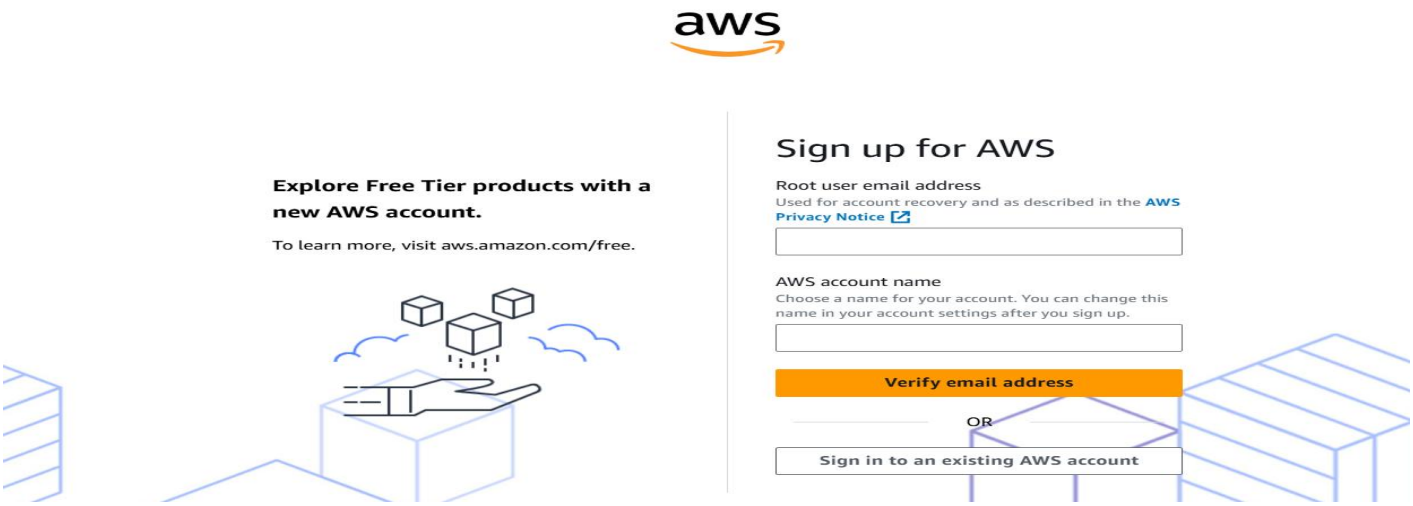
https://docs.google.com/document/d/1sFF7tJ6IgWtRbawWoA4W3PkkxEFrSJZhKzULgLsjxo/edit?usp=sharing

Milestone 2: AWS Account Setup and Login

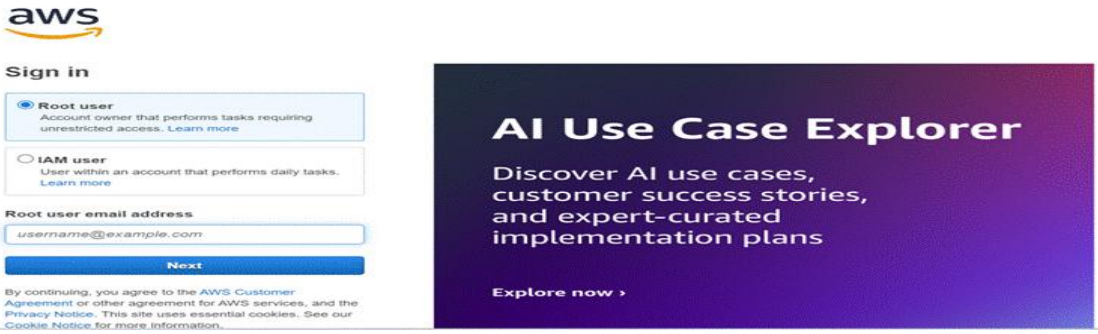
- Go to the AWS website (https://aws.amazon.com/).
- Click on the "Create an AWS Account" button.
- Follow the prompts to enter your email address and choose a password.
- Provide the required account information, including your name, address, and phone number.
- Enter your payment information. (Note: While AWS offers a free tier, a credit card or debit card is required for verification.)
- Complete the identity verification process.

- Choose a support plan (the basic plan is free and sufficient for starting).
- Once verified, you can sign in to your new AWS accounts.



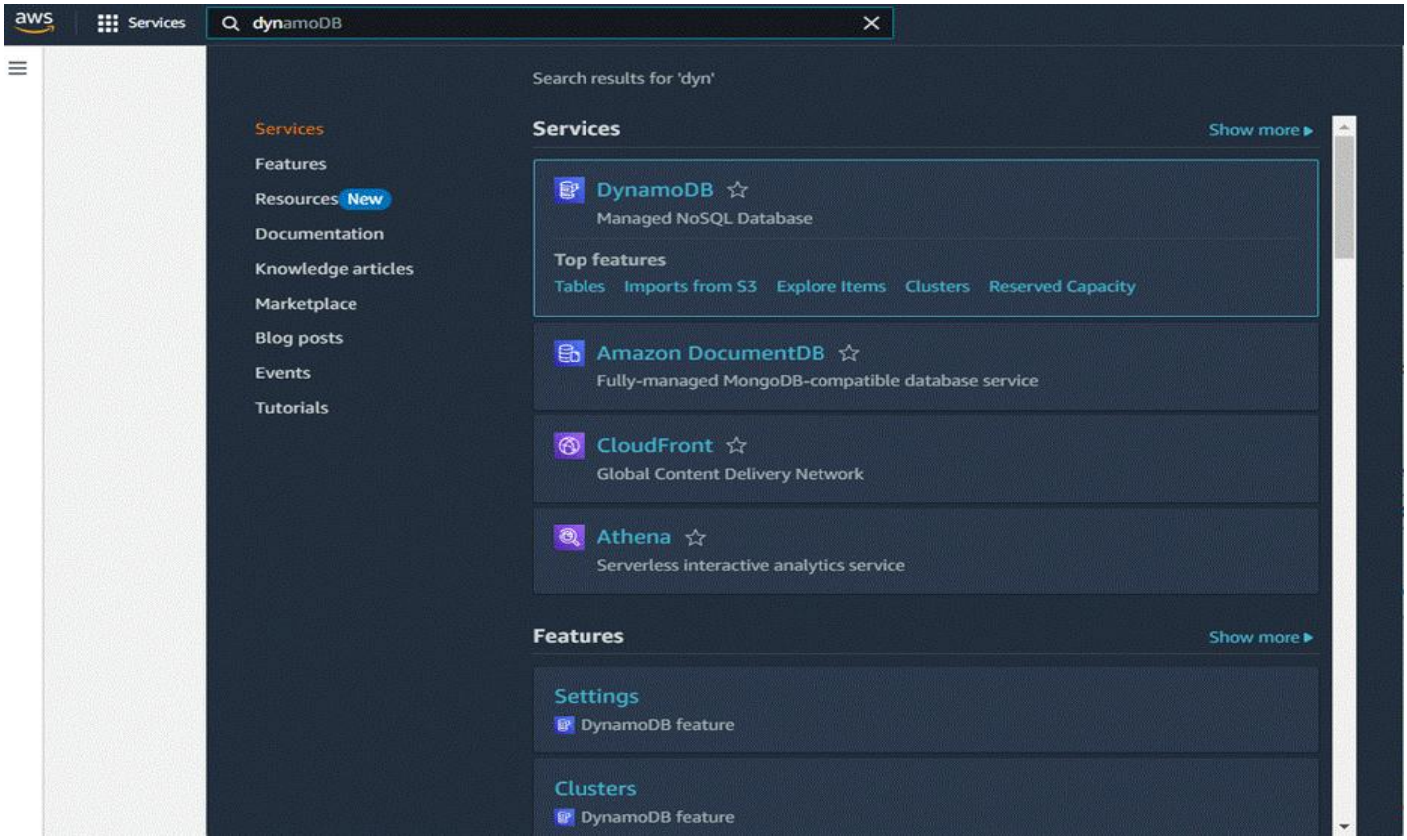


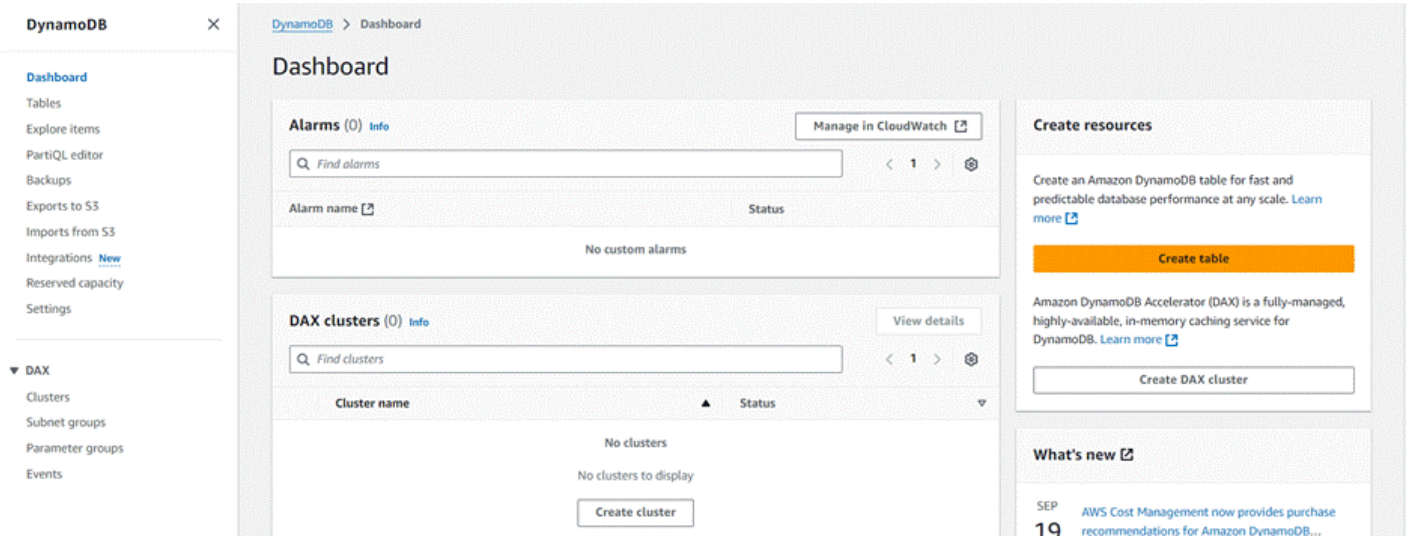
- Log in to the AWS Management Console
- After setting up your account, log in to the AWS Management Console.

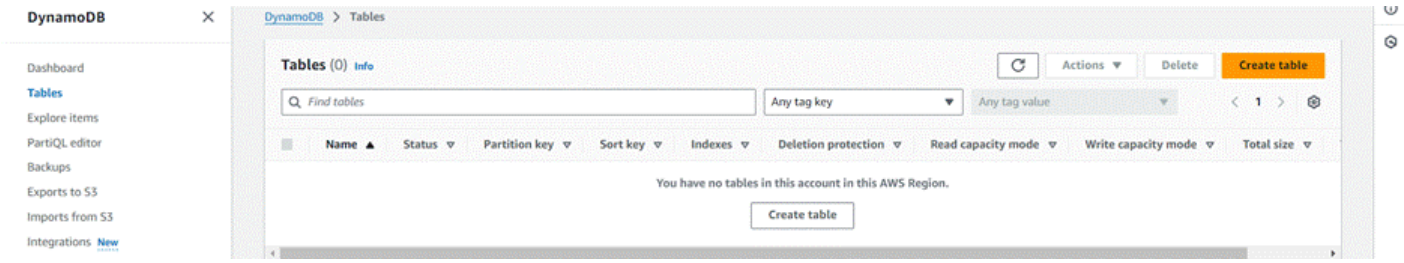


Milestone 3 : DynamoDB Database Creation and Setup

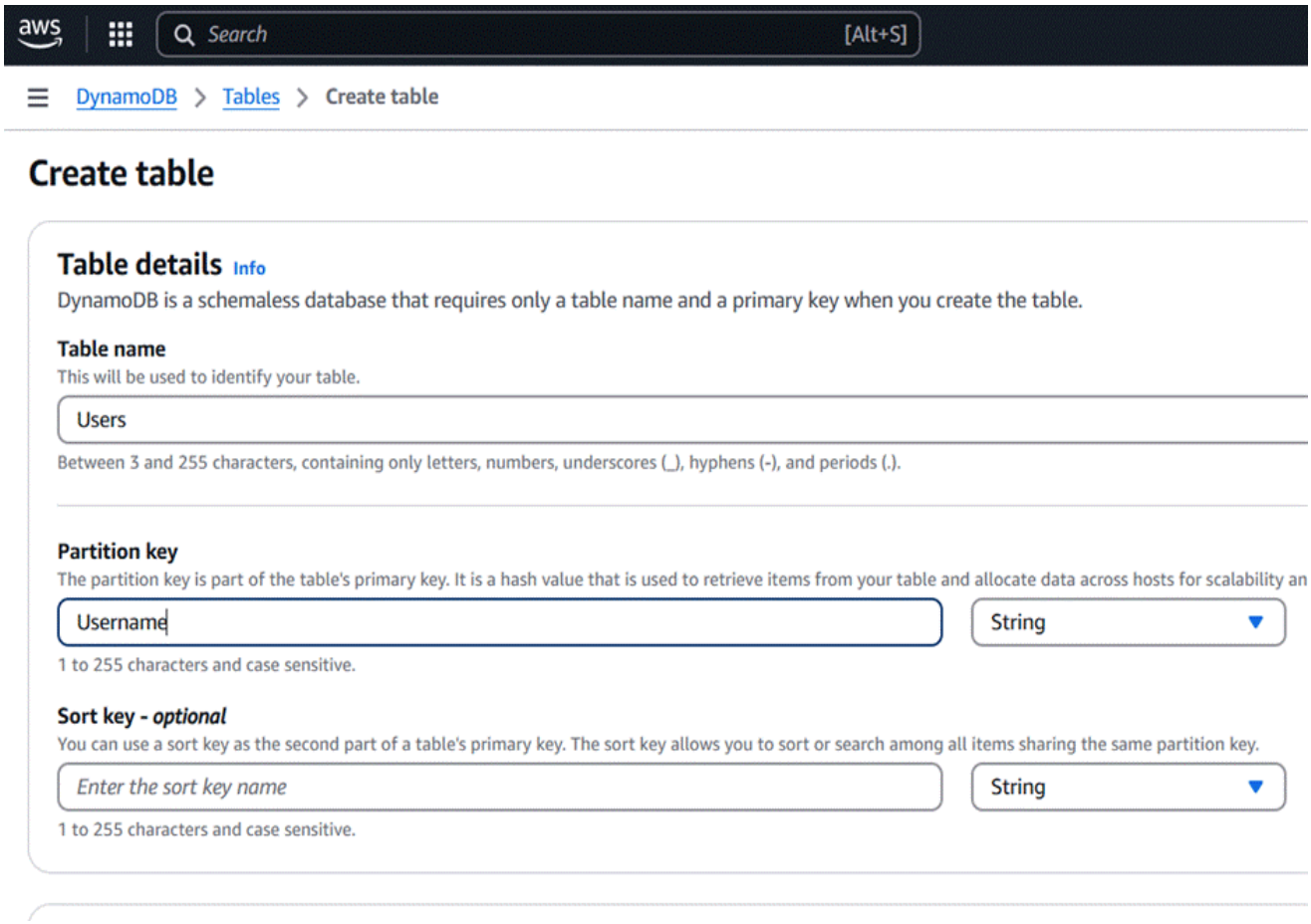
- **Navigate to the DynamoDB**
    - In the AWS Console, navigate to DynamoDB and click on create tables.







- **Create a DynamoDB table for storing registration details and book requests.**
    - Create Users table with partition key “Username” with type String and click on create tables.



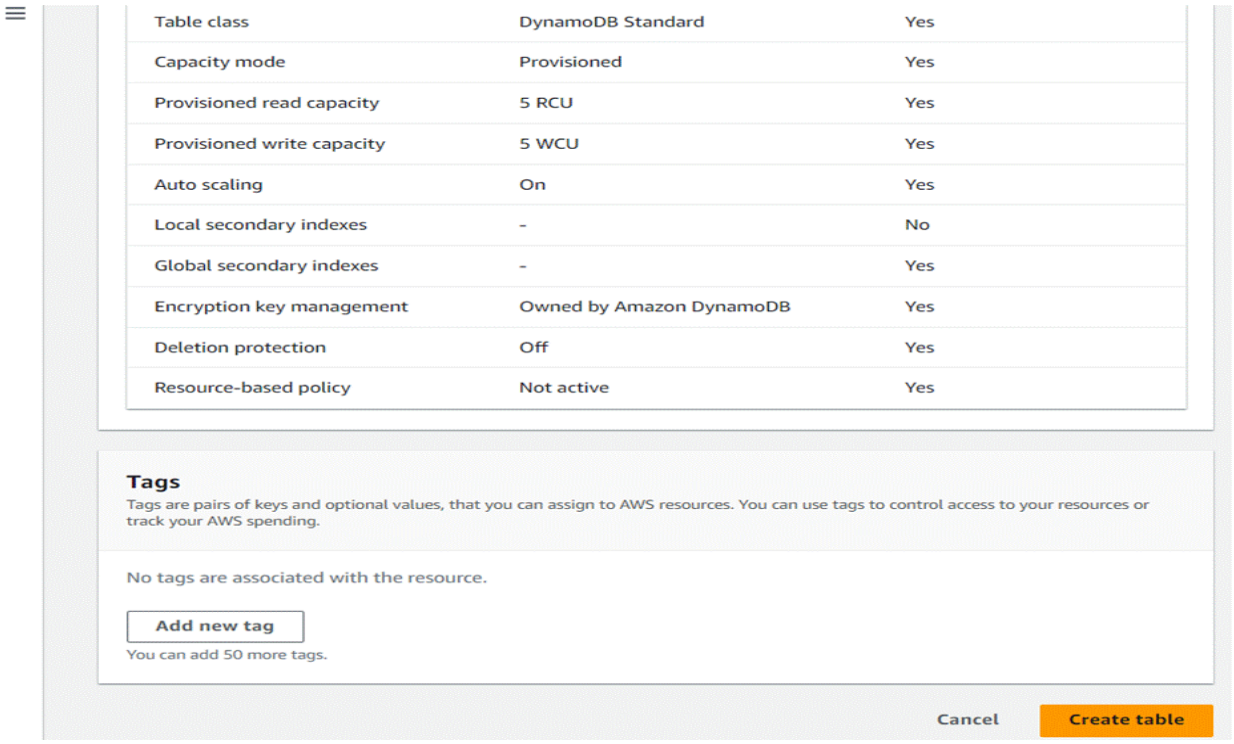
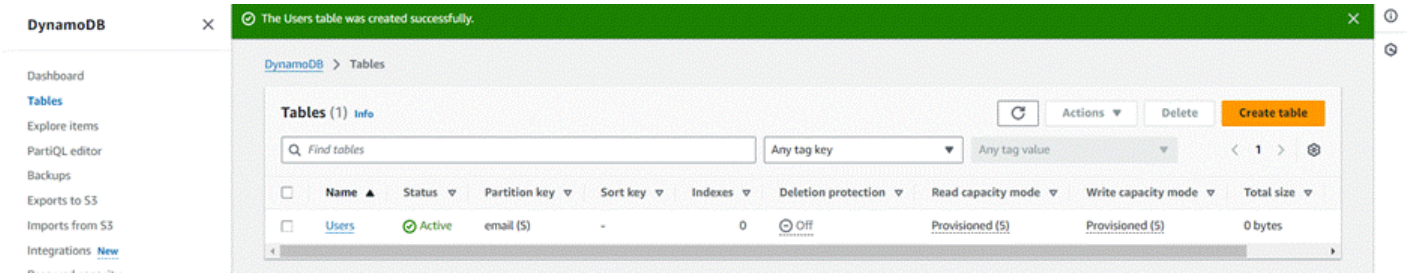
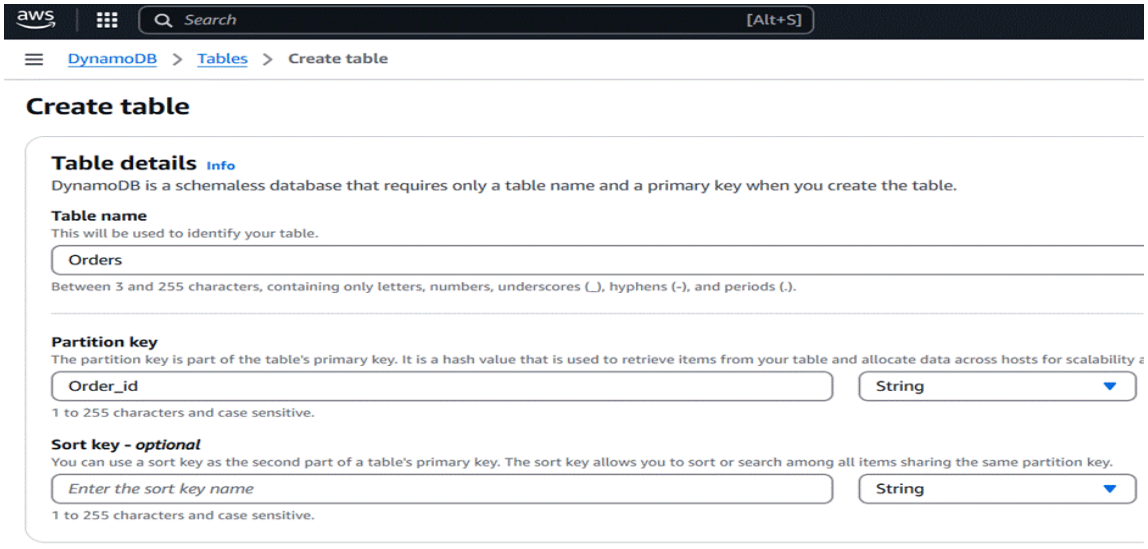
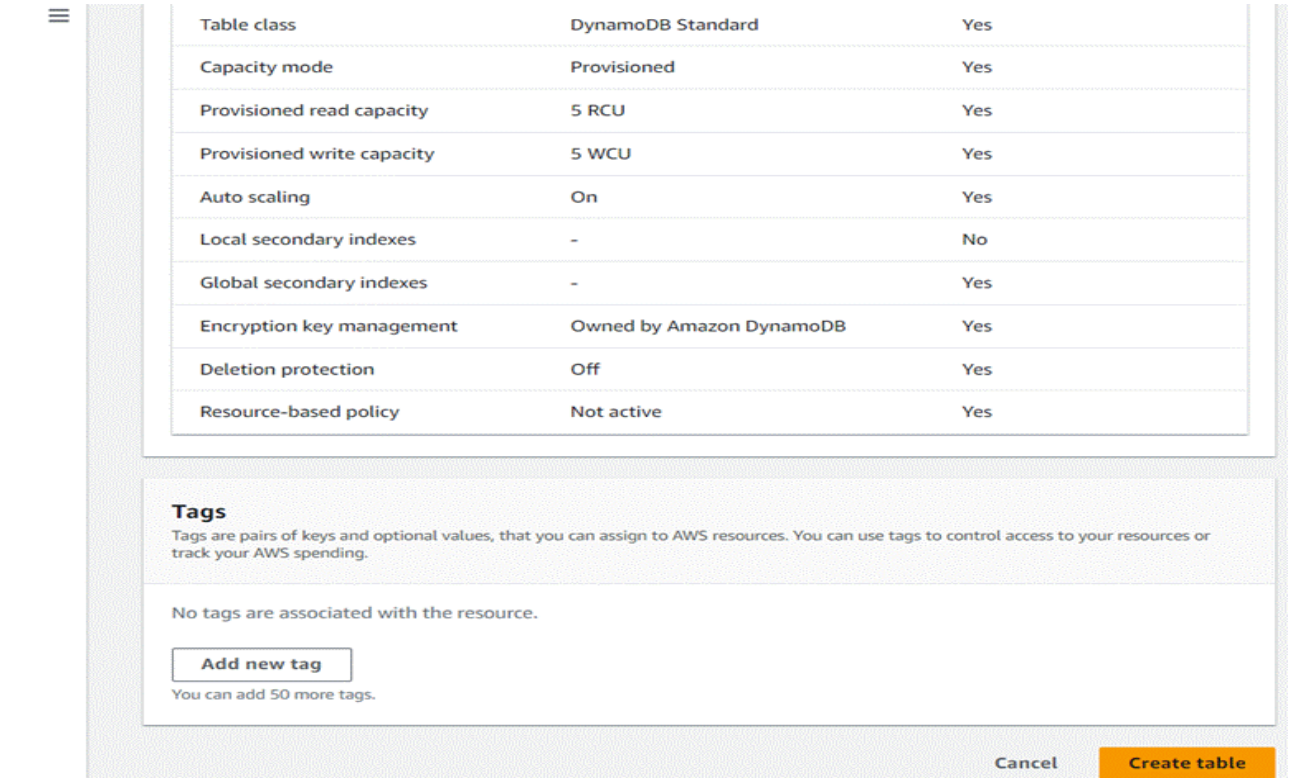


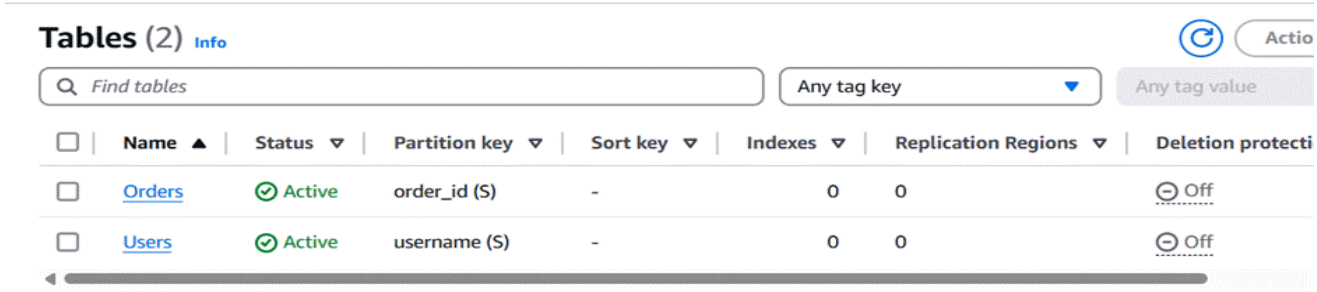
Table class	DynamoDB Standard	Yes
Capacity mode	Provisioned	Yes
Provisioned read capacity	5 RCU	Yes
Provisioned write capacity	5 WCU	Yes
Auto scaling	On	Yes
Local secondary indexes	-	No
Global secondary indexes	-	Yes
Encryption key management	Owned by Amazon DynamoDB	Yes
Deletion protection	Off	Yes
Resource-based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



○ Follow the same steps to create a requests table with Email as the primary key for book requests data.

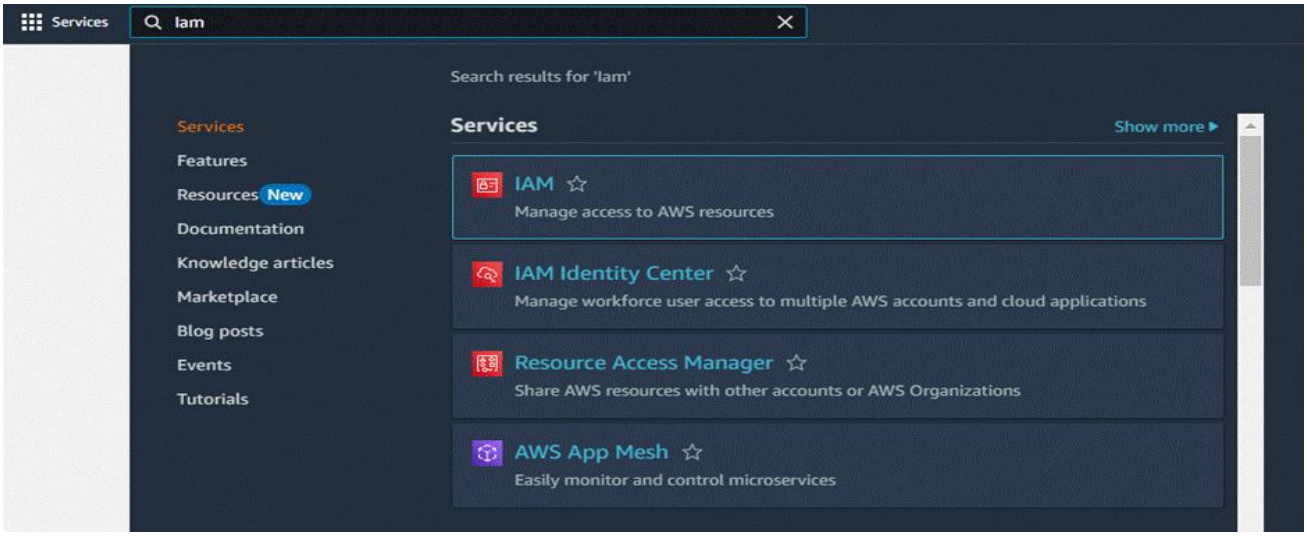


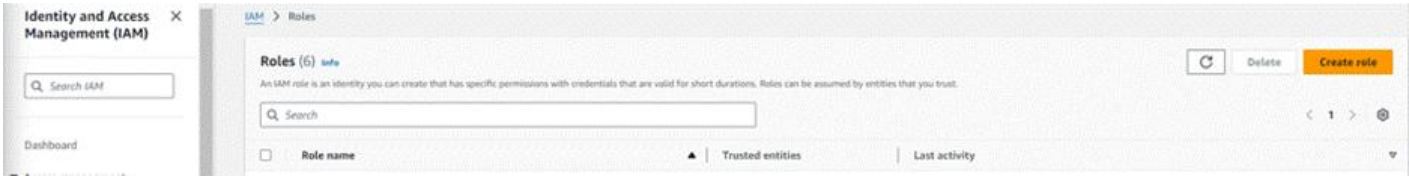


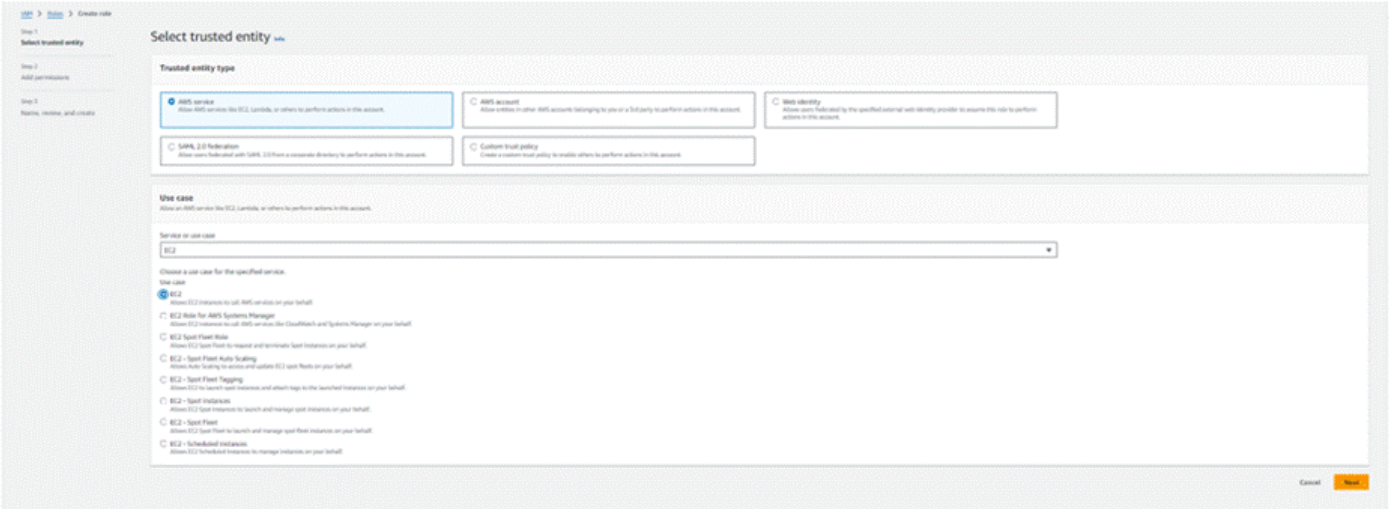


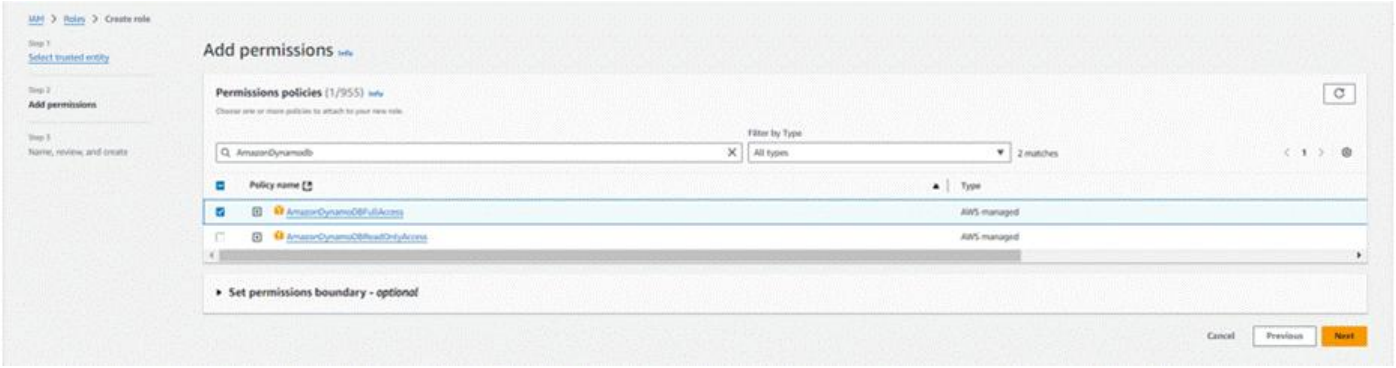
Milestone 4: IAM Role SetUp

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

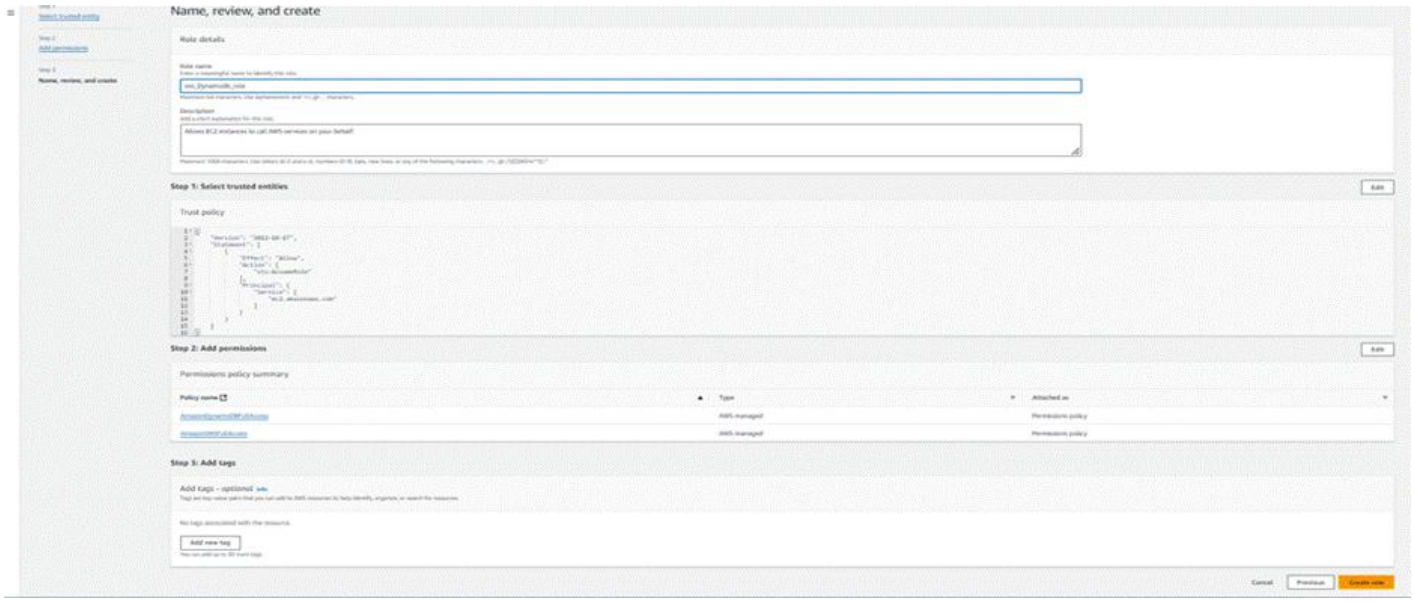

Services
Iam
Search results for 'Iam'
Services
Features
Resources New
Documentation
Knowledge articles
Marketplace
Blog posts
Events
Tutorials
Services
Show more
IAM
Manage access to AWS resources
IAM Identity Center
Manage workforce user access to multiple AWS accounts and cloud applications
Resource Access Manager
Share AWS resources with other accounts or AWS Organizations
AWS App Mesh
Easily monitor and control microservices

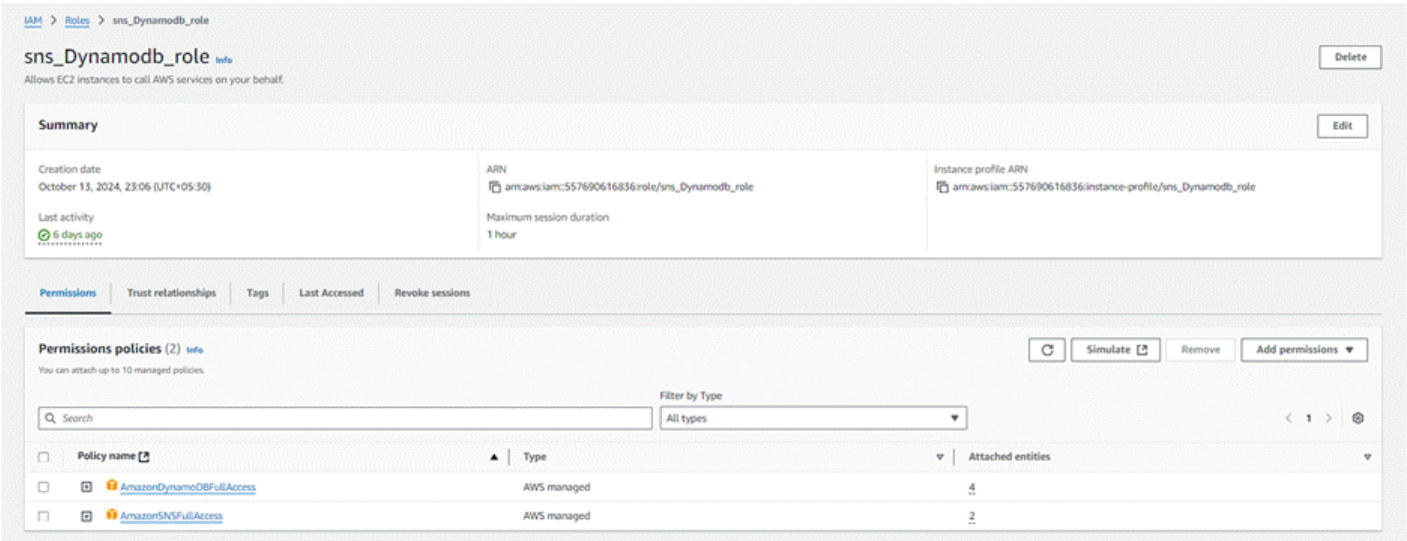

Identity and Access Management (IAM)
Search IAM
Dashboard
IAM > Roles
Roles (6) Info
An IAM role is an identity you can create that has specific permissions with credentials that are valid for short durations. Roles can be assumed by entities that you trust.
Delete
Create role
Search
Role name
Trusted entities
Last activity


Select trusted entity
Trusted entity type
Use case
Cancel
Next


IAM > Roles > Create role
Select trusted entity
Add permissions
Name, review, and create
Add permissions
Permissions policies (1/955)
Filter by Type
AmazonDynamodb
All types
2 matches
Policy name
Type
AmazonDynamoDBFullAccess
AWS managed
AmazonDynamoDBReadOnlyAccess
AWS managed
Set permissions boundary - optional
Cancel
Previous
Next

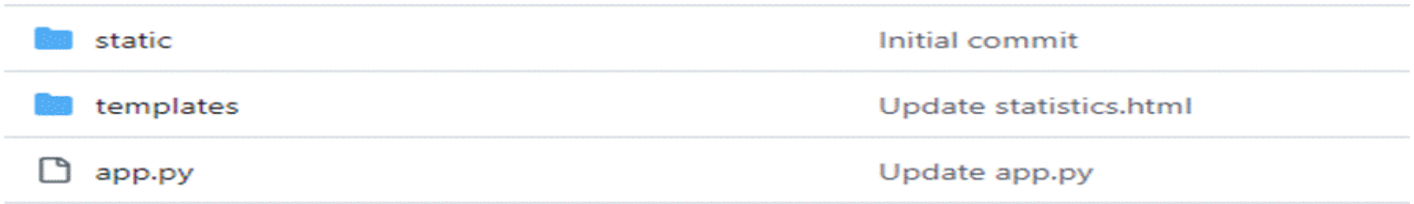
- **Attach Policies**
  - AmazonDynamoDBFullAccess: Allows EC2 to perform read/write operations on DynamoDB.





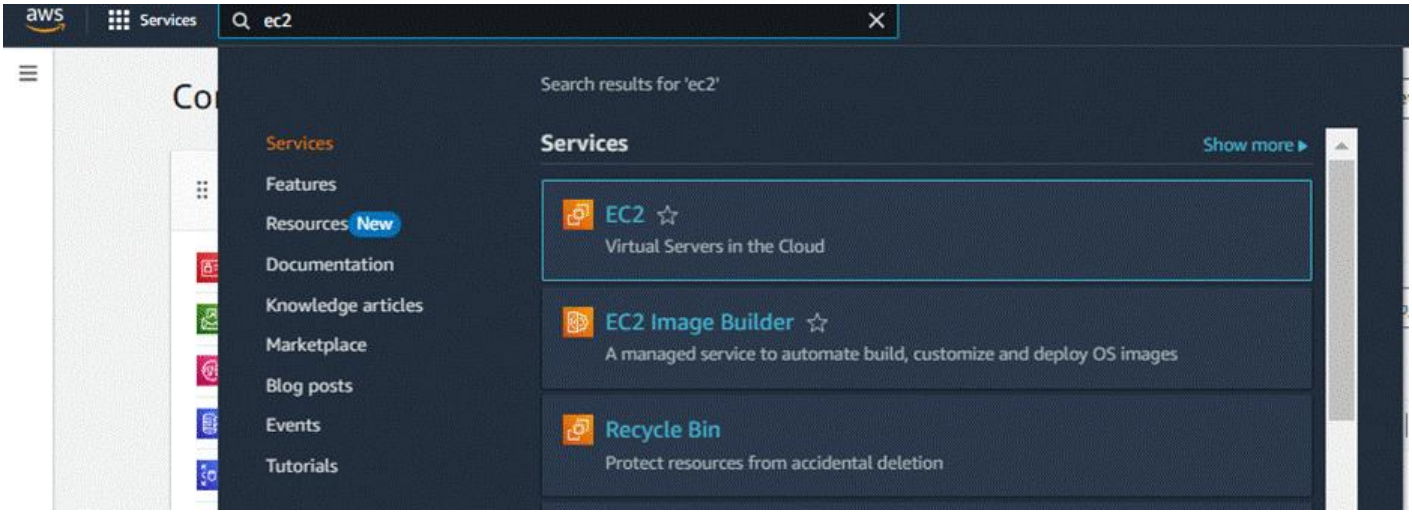
Milestone 5: EC2 Instance Set Up

Load your Flask app and Html files into GitHub repository.

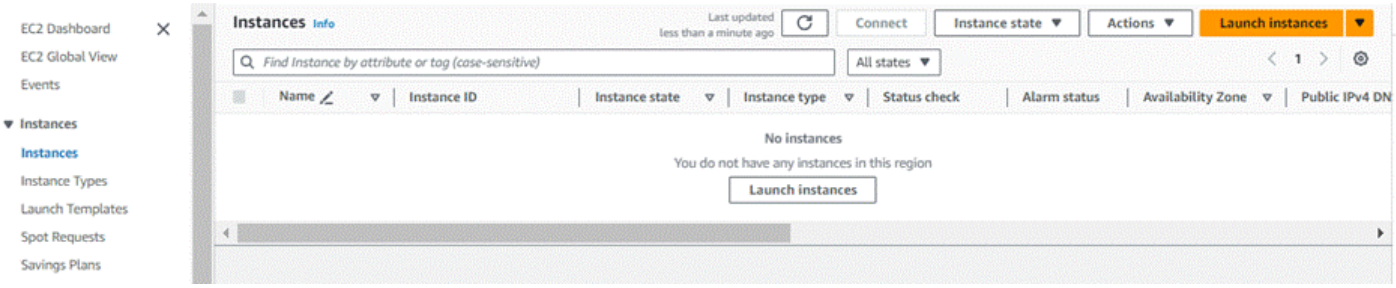


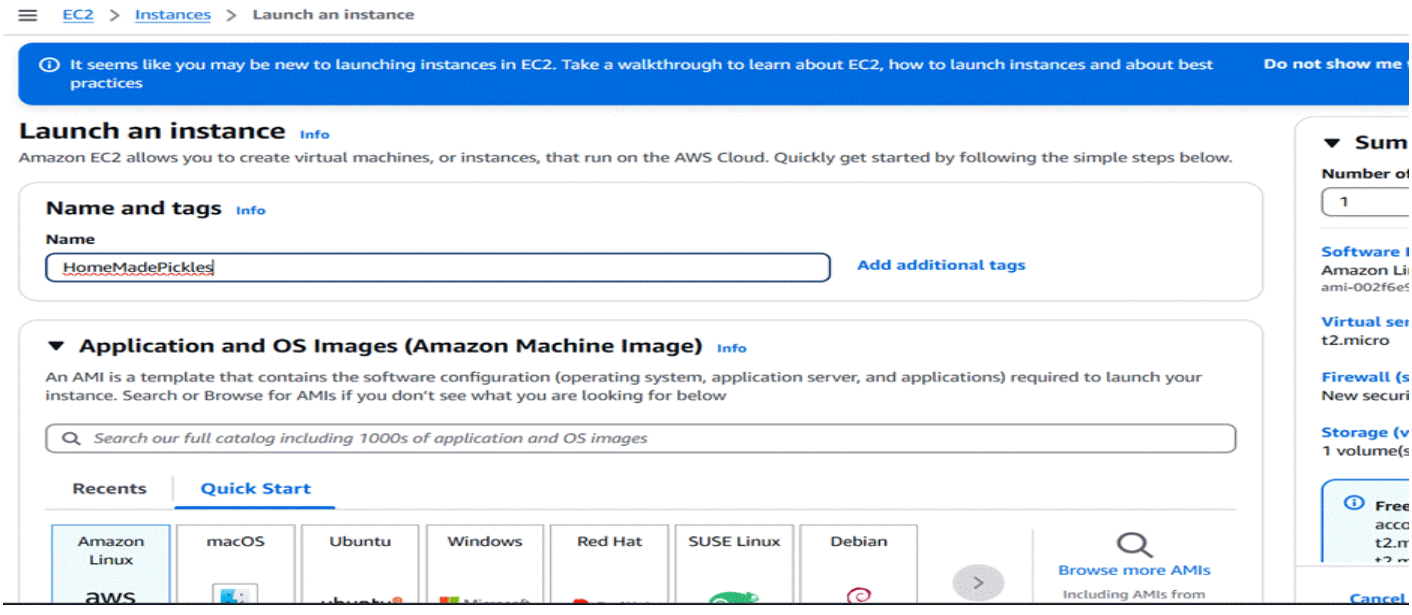
- **Launch an EC2 instance to host the Flask**
    - Launch EC2 Instance.

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

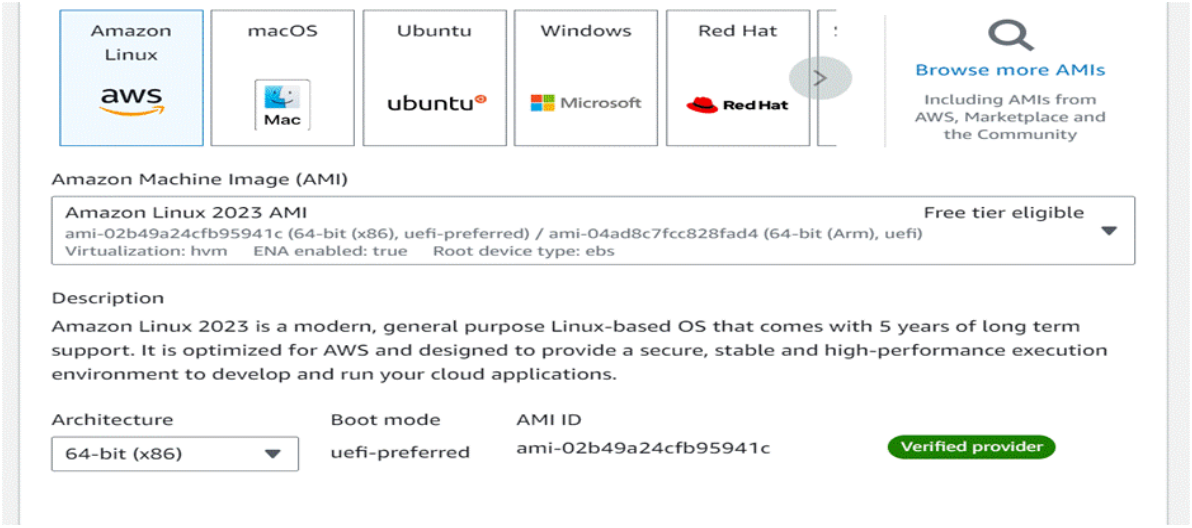


Click on Launch instance to launch EC2 instance

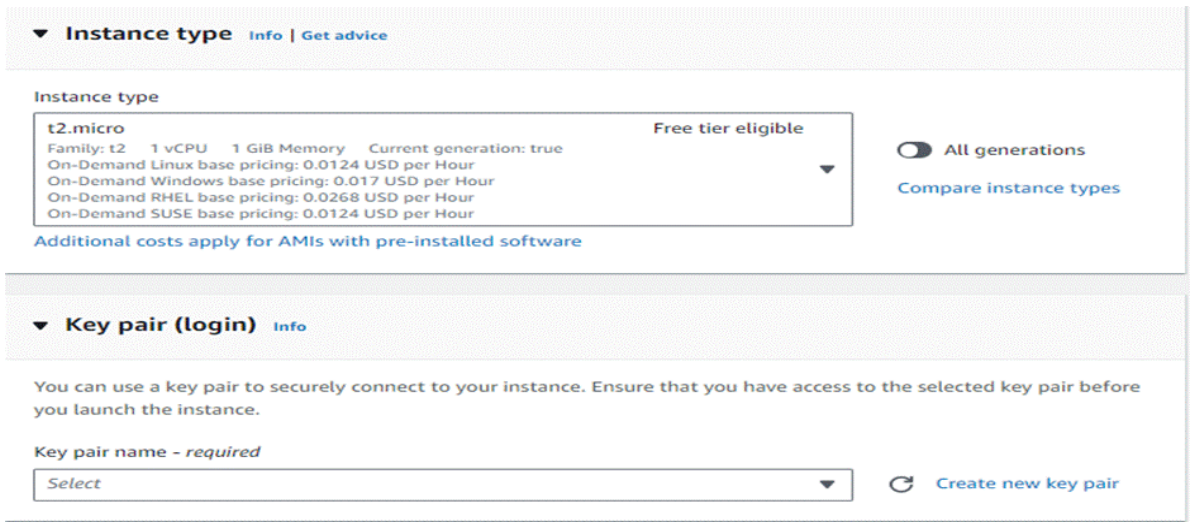


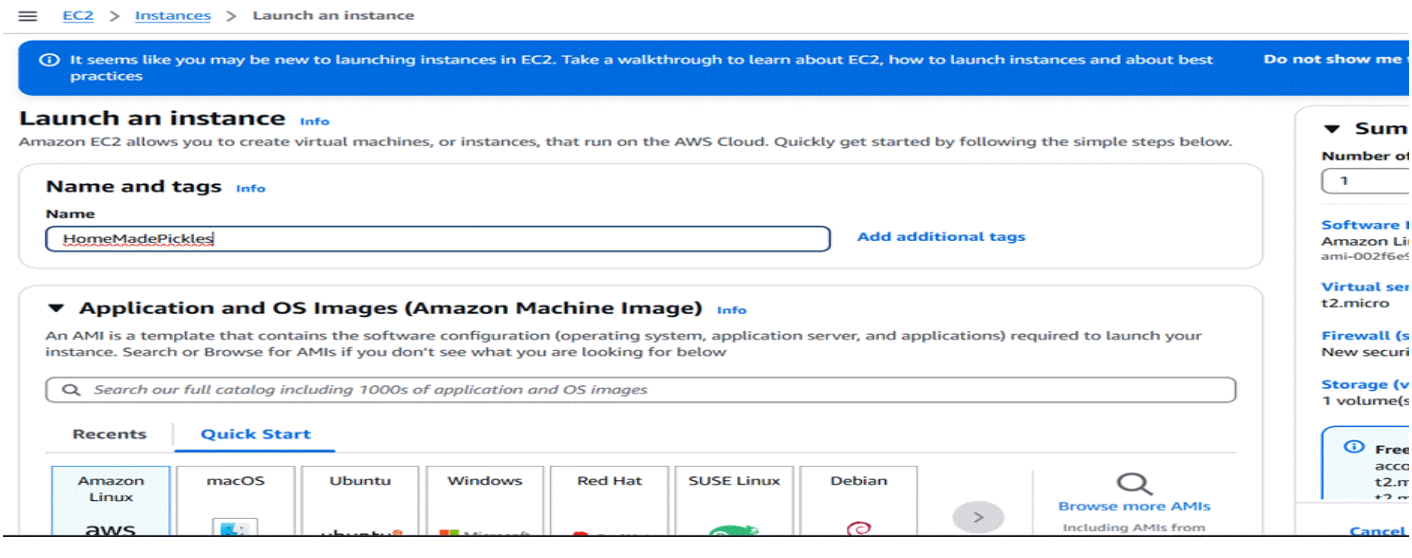


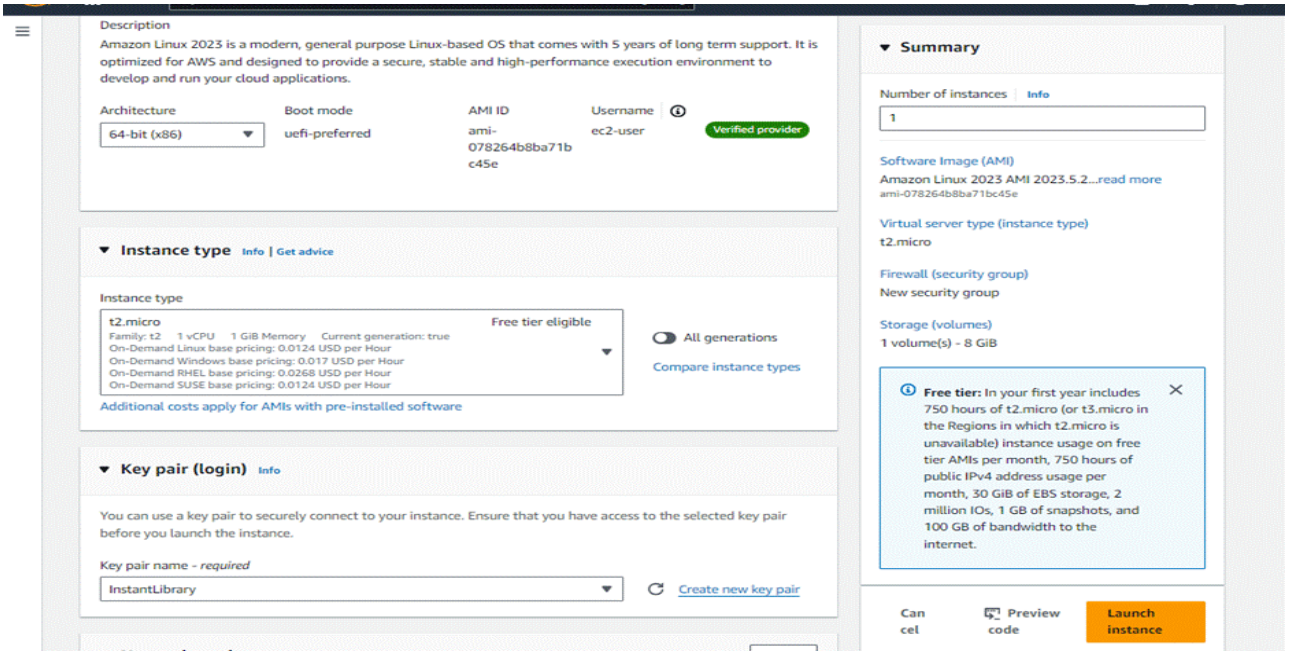
Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).



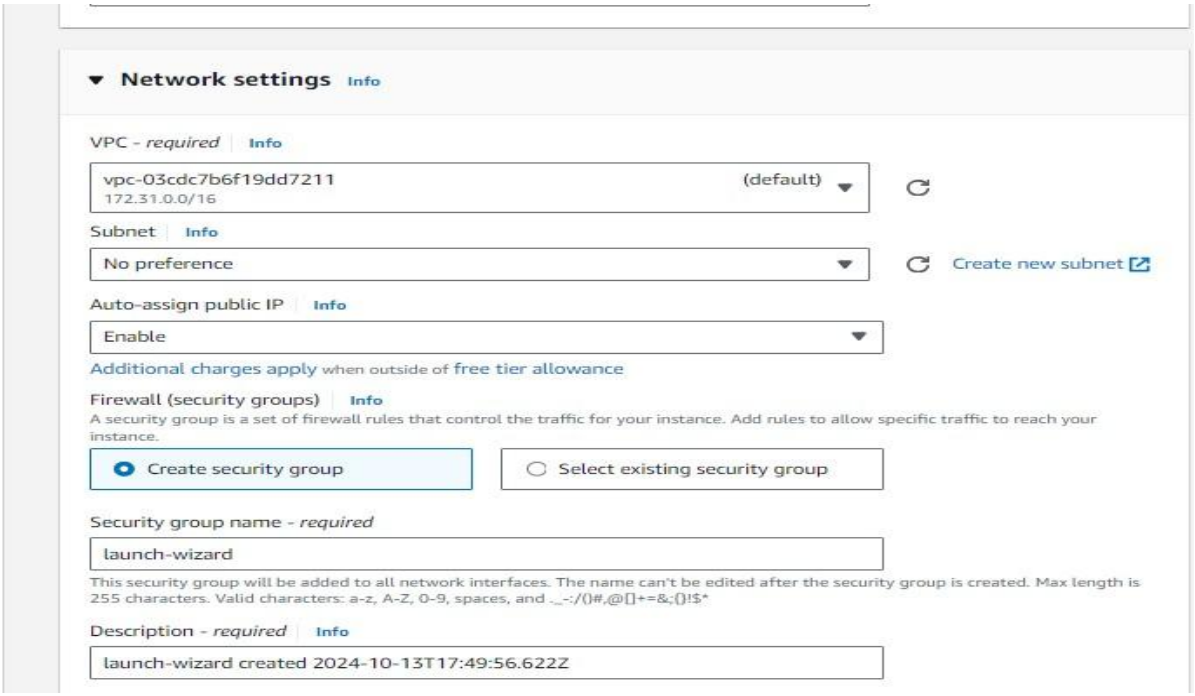
Create and download the key pair for Server access**.**

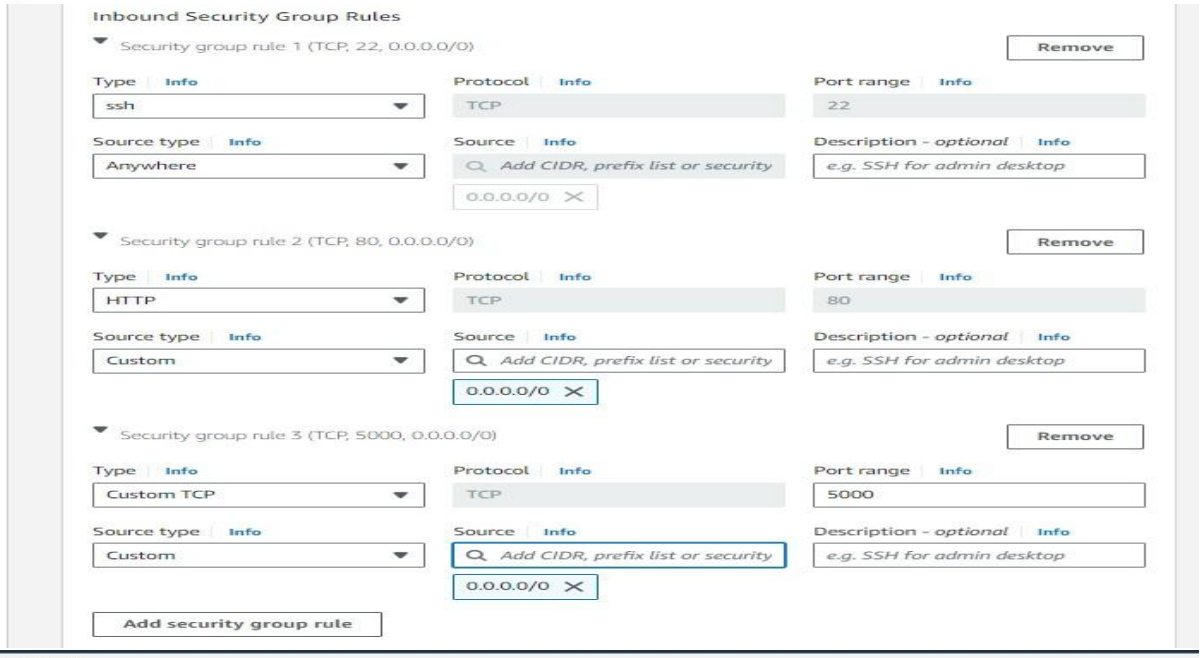


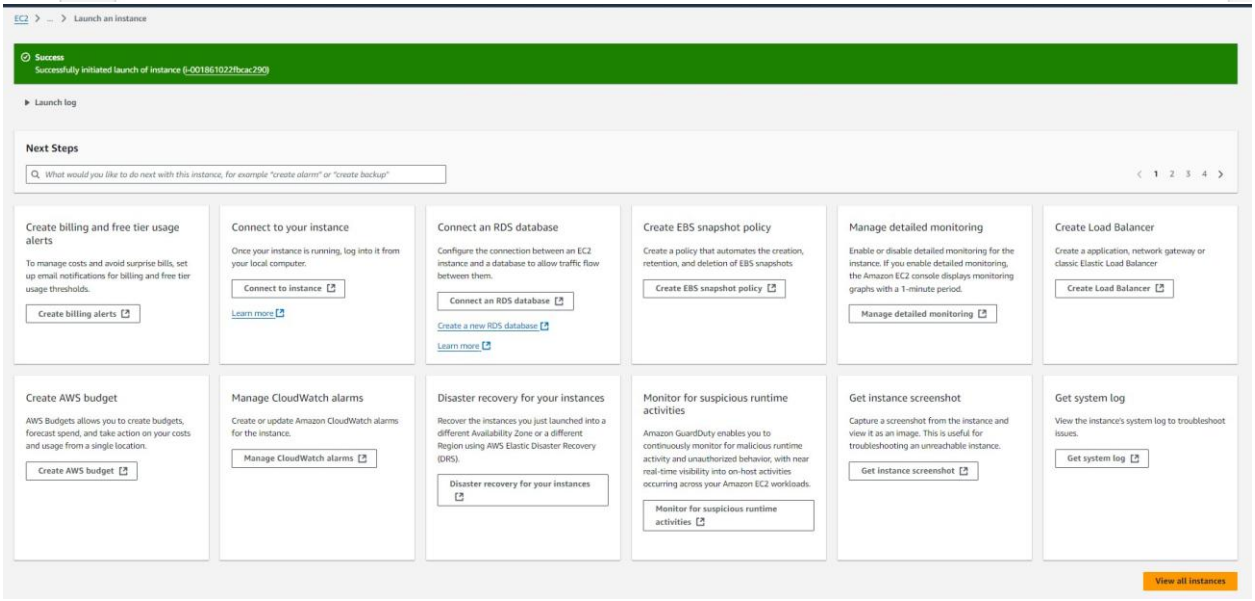




- **Configure security groups for HTTP, and SSH access.**

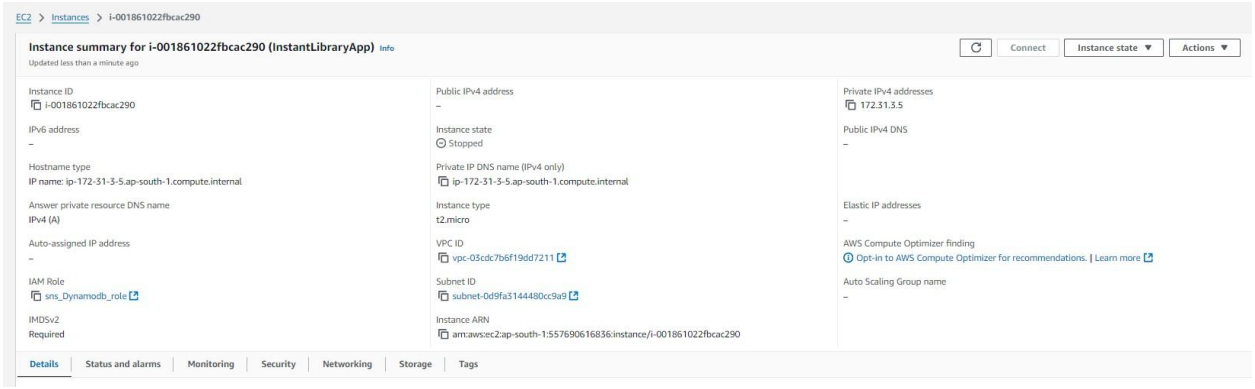


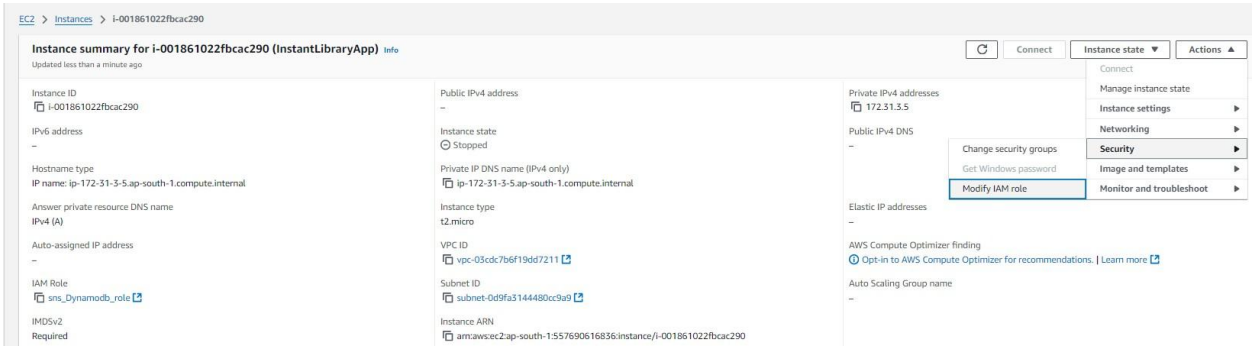


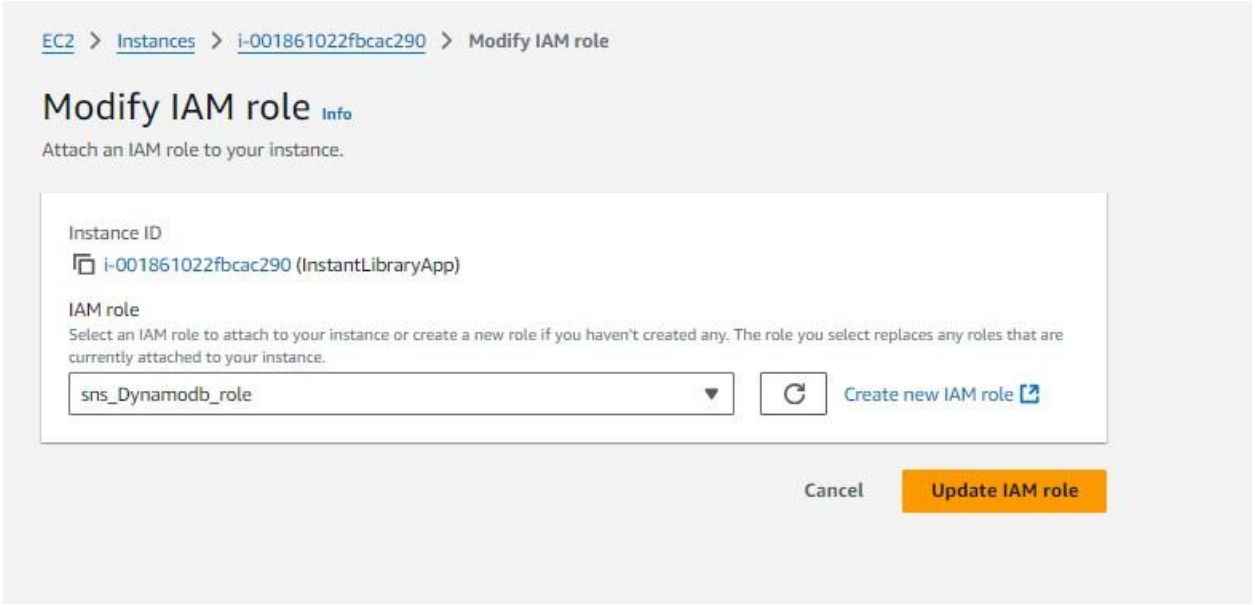


- To connect to EC2 using **EC2 Instance Connect**, start by ensuring that an **IAM role** is attached to your EC2 instance. You can do this by selecting your instance, clicking on **Actions**, then navigating to **Security** and selecting **Modify IAM Role** to attach the appropriate role. After the 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.

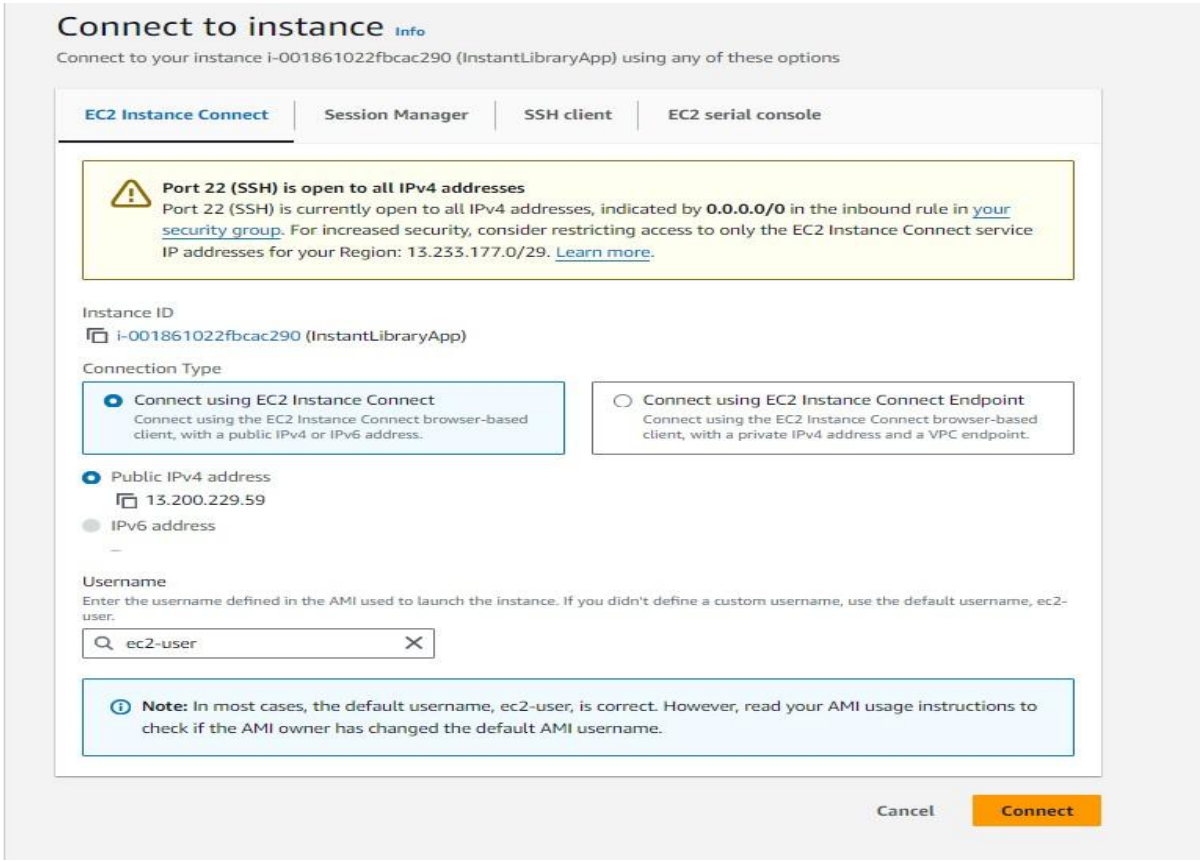

Instances (1/2) Info
Last updated less than a minute ago
Connect
Instance state
Actions
Launch instances
Find Instance by attribute or tag (case-sensitive)
All states
Name
Instance ID
Instance state
Instance type
Status check
Alarm status
Availability Zone
Public IPv4 DNS
Public IPv4 ...
Elastic IP
IPv6 IPs
Monitoring
InstantLibrary...
i-001861022fbcac290
Stopped
t2.micro
View alarms
ap-south-1b
disabled

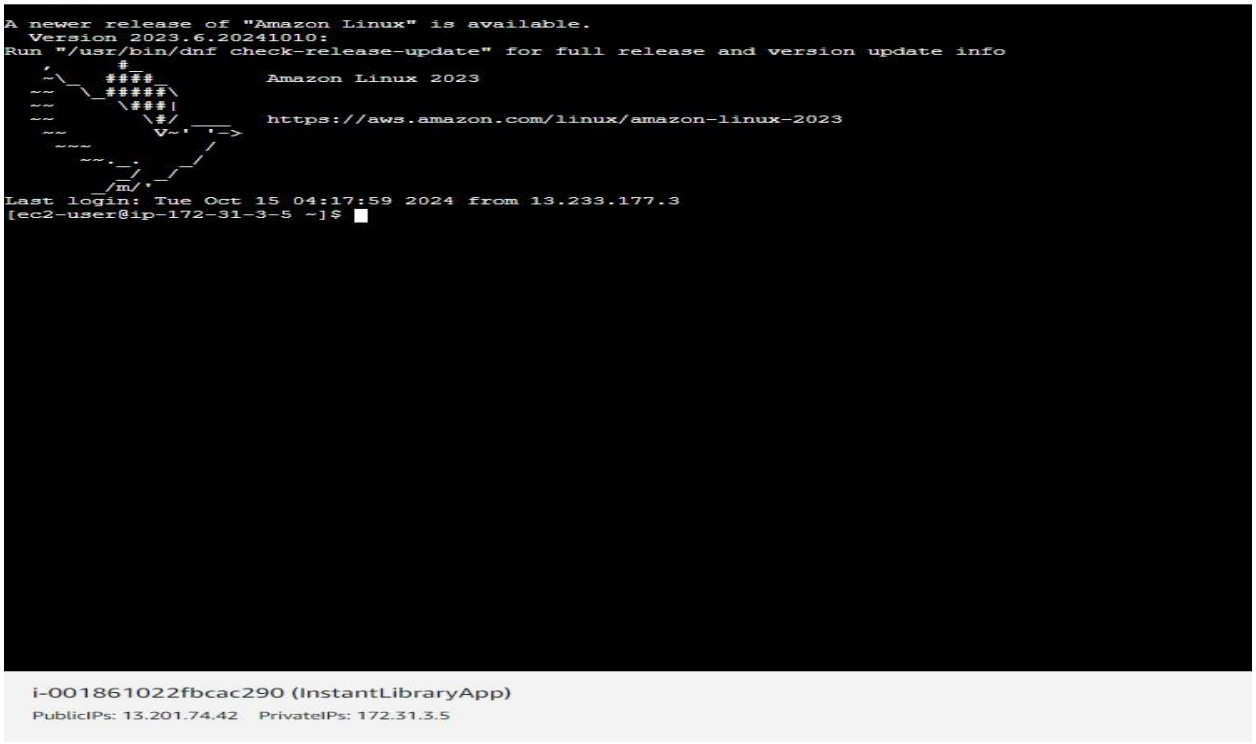

EC2 > Instances > i-001861022fbcac290
Instance summary for i-001861022fbcac290 (InstantLibraryApp) Info
Updated less than a minute ago
Connect
Instance state
Actions
Instance ID
i-001861022fbcac290
IPv6 address
Hostname type
IP name: ip-172-31-3-5.ap-south-1.compute.internal
Answer private resource DNS name
IPv4 (A)
Auto-assigned IP address
IAM Role
sns_Dynamodb_role
IMDSv2
Required
Public IPv4 address
Instance state
Stopped
Private IP DNS name (IPv4 only)
ip-172-31-3-5.ap-south-1.compute.internal
Instance type
t2.micro
VPC ID
vpc-03cdc7b6f19dd7211
Subnet ID
subnet-0d9fa3144480cc9a9
Instance ARN
arn:aws:ec2:ap-south-1:557690616836:instance/i-001861022fbcac290
Private IPv4 addresses
172.31.3.5
Public IPv4 DNS
Elastic IP addresses
AWS Compute Optimizer finding
Opt-in to AWS Compute Optimizer for recommendations. | Learn more
Auto Scaling Group name
Details
Status and alarms
Monitoring
Security
Networking
Storage
Tags


EC2 > Instances > i-001861022fbcac290
Instance summary for i-001861022fbcac290 (InstantLibraryApp) Info
Updated less than a minute ago
Connect
Instance state
Actions
Connect
Manage instance state
Instance settings
Networking
Security
Image and templates
Monitor and troubleshoot
Change security groups
Get Windows password
Modify IAM role
Instance ID
i-001861022fbcac290
IPv6 address
Hostname type
IP name: ip-172-31-3-5.ap-south-1.compute.internal
Answer private resource DNS name
IPv4 (A)
Auto-assigned IP address
IAM Role
sns_Dynamodb_role
IMDSv2
Required
Public IPv4 address
Instance state
Stopped
Private IP DNS name (IPv4 only)
ip-172-31-3-5.ap-south-1.compute.internal
Instance type
t2.micro
VPC ID
vpc-03cdc7b6f19dd7211
Subnet ID
subnet-0d9fa3144480cc9a9
Instance ARN
arn:aws:ec2:ap-south-1:557690616836:instance/i-001861022fbcac290
Private IPv4 addresses
172.31.3.5
Public IPv4 DNS
Elastic IP addresses
AWS Compute Optimizer finding
Opt-in to AWS Compute Optimizer for recommendations. | Learn more
Auto Scaling Group name


EC2 > Instances > i-001861022fbcac290 > Modify IAM role
Modify IAM role Info
Attach an IAM role to your instance.
Instance ID
i-001861022fbcac290 (InstantLibraryApp)
IAM role
Select an IAM role to attach to your instance or create a new role if you haven't created any. The role you select replaces any roles that are currently attached to your instance.
sns_Dynamodb_role
Create new IAM role
Cancel
Update IAM role

- Now connect the EC2 with the files





Milestone 6: Deployment on EC2

Install Software on the EC2 Instance

Install Python3, Flask, and Git:

On Amazon Linux 2:

sudo yum update -y

sudo yum install python3 git

sudo pip3 install flask boto3

Verify Installations:

flask --version

git --version

Clone Your 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/AlekhyaPenubakula/InstantLibrary.git'

- This will download your project to the EC2 instance.

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

- cd Homemadepicklesandsnacks

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

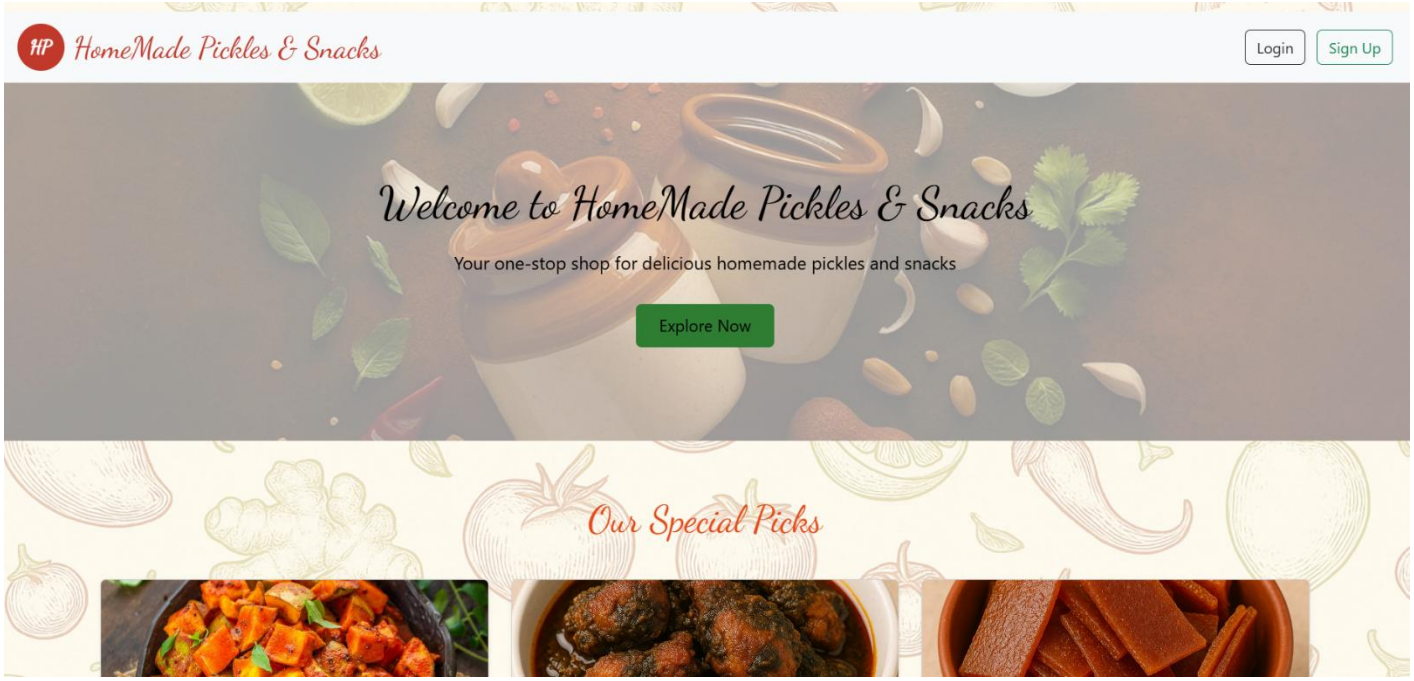
Run the Flask Application

sudo flask run --host=0.0.0.0 --port=80

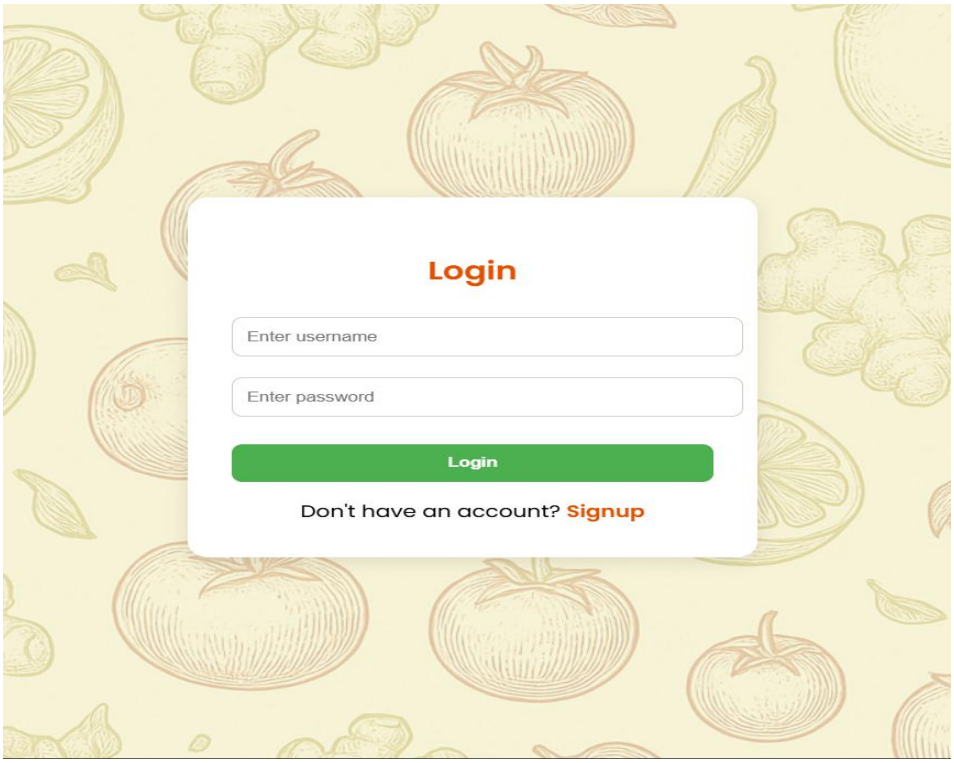
```
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

C:\Users\srava\OneDrive\Desktop\pr1>python app.py
 * Tip: There are .env files present. Install python-dotenv to use them.
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Tip: There are .env files present. Install python-dotenv to use them.
 * Debugger is active!
 * Debugger PIN: 517-909-220
```

Milestone 7: Testing and Deployment

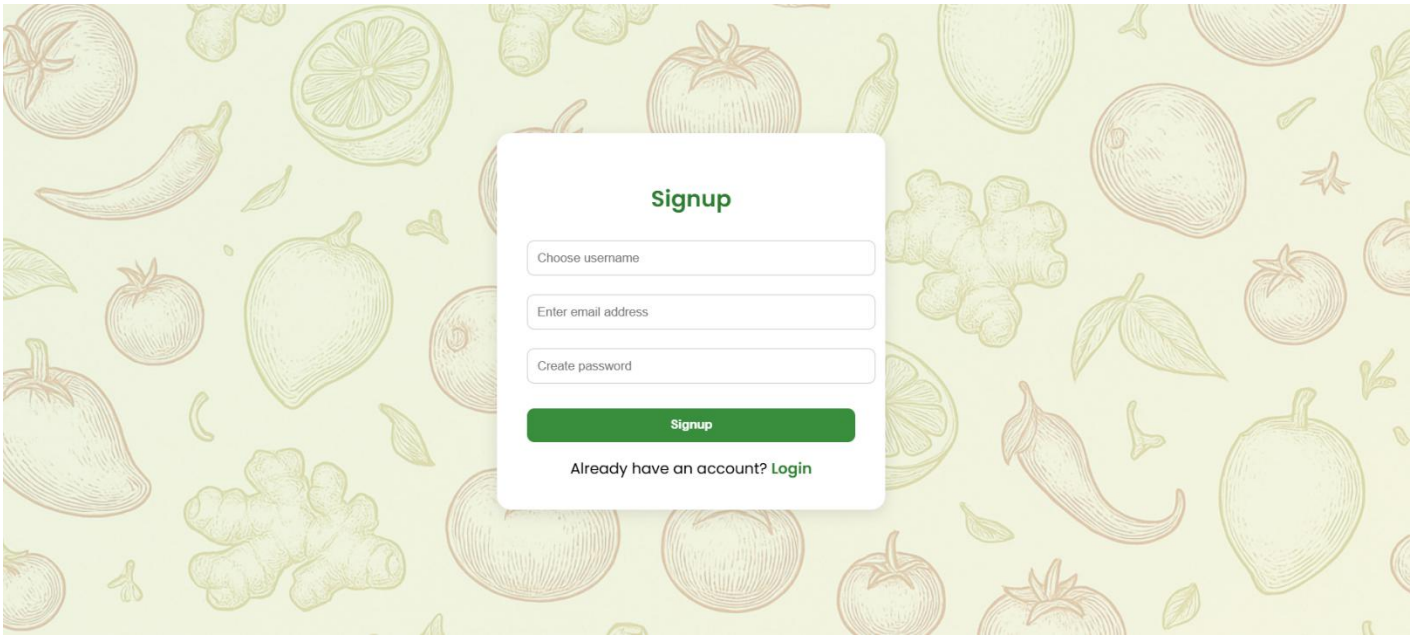
Welcome Page:



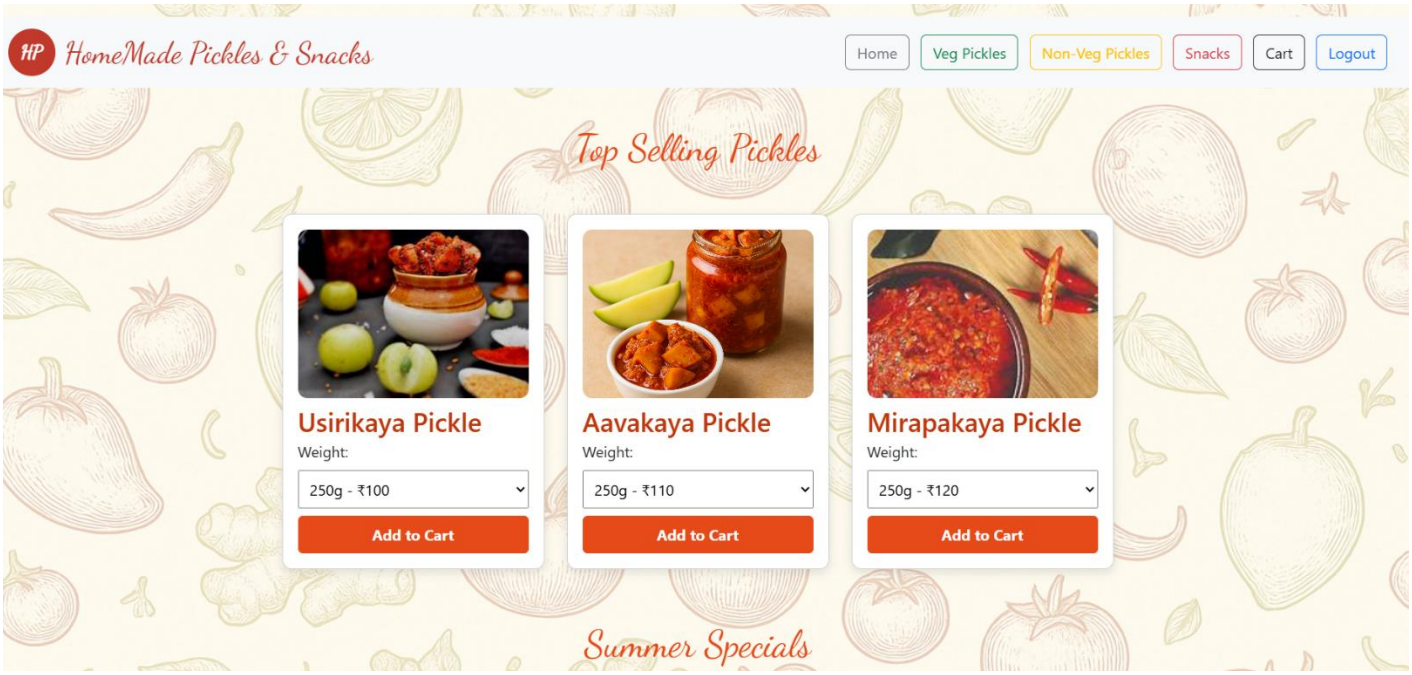
Login Page:



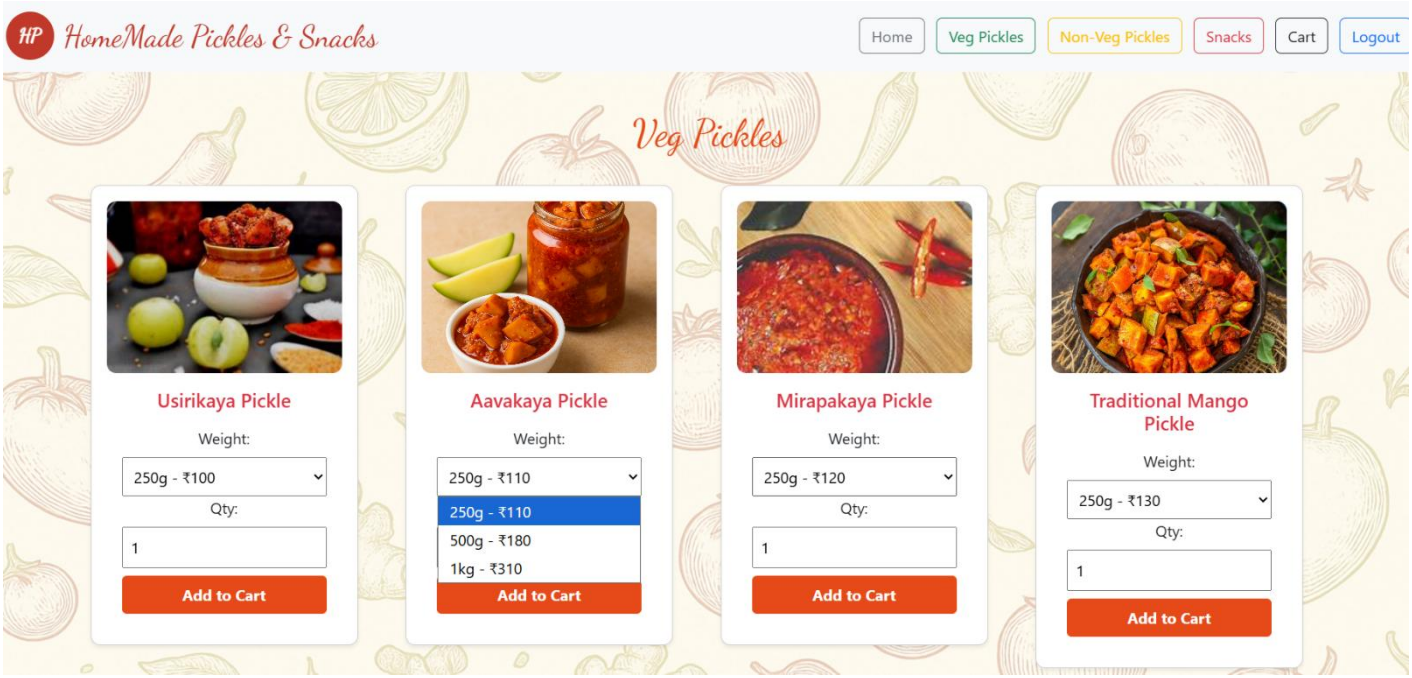
Sign in Page:



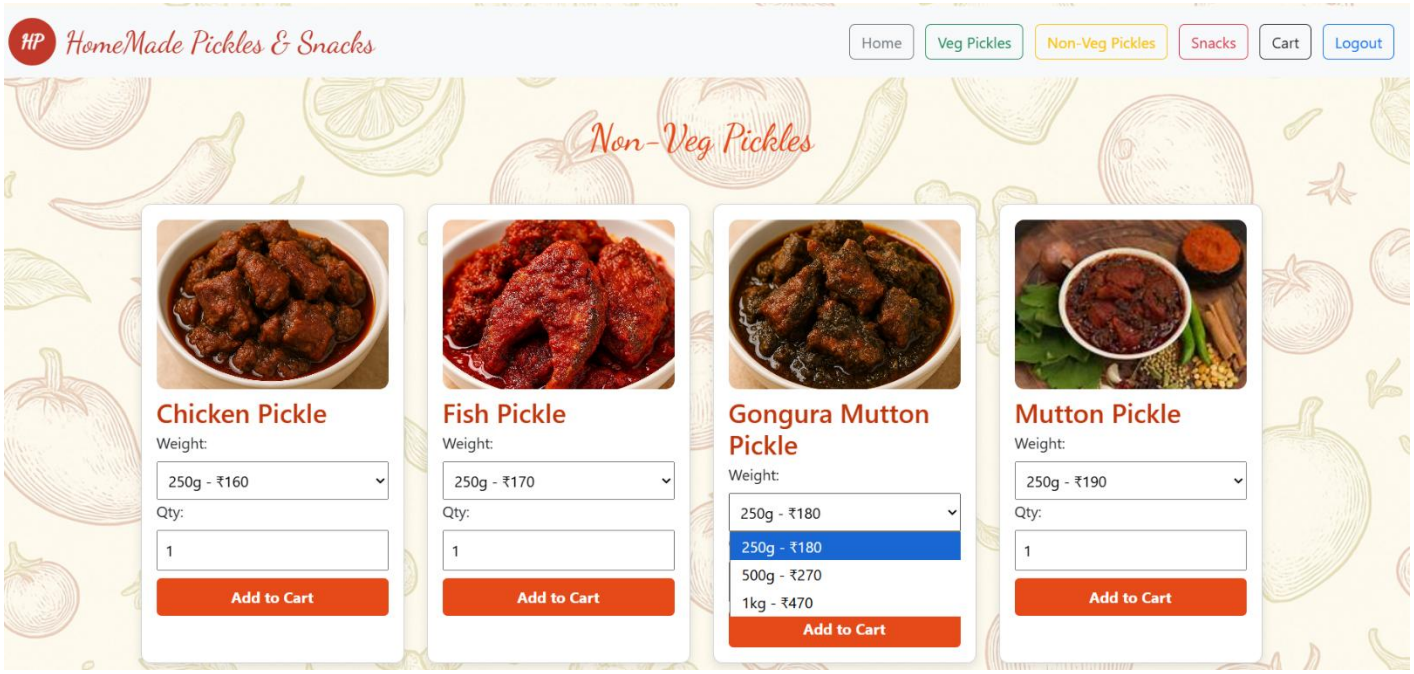
Home Page:



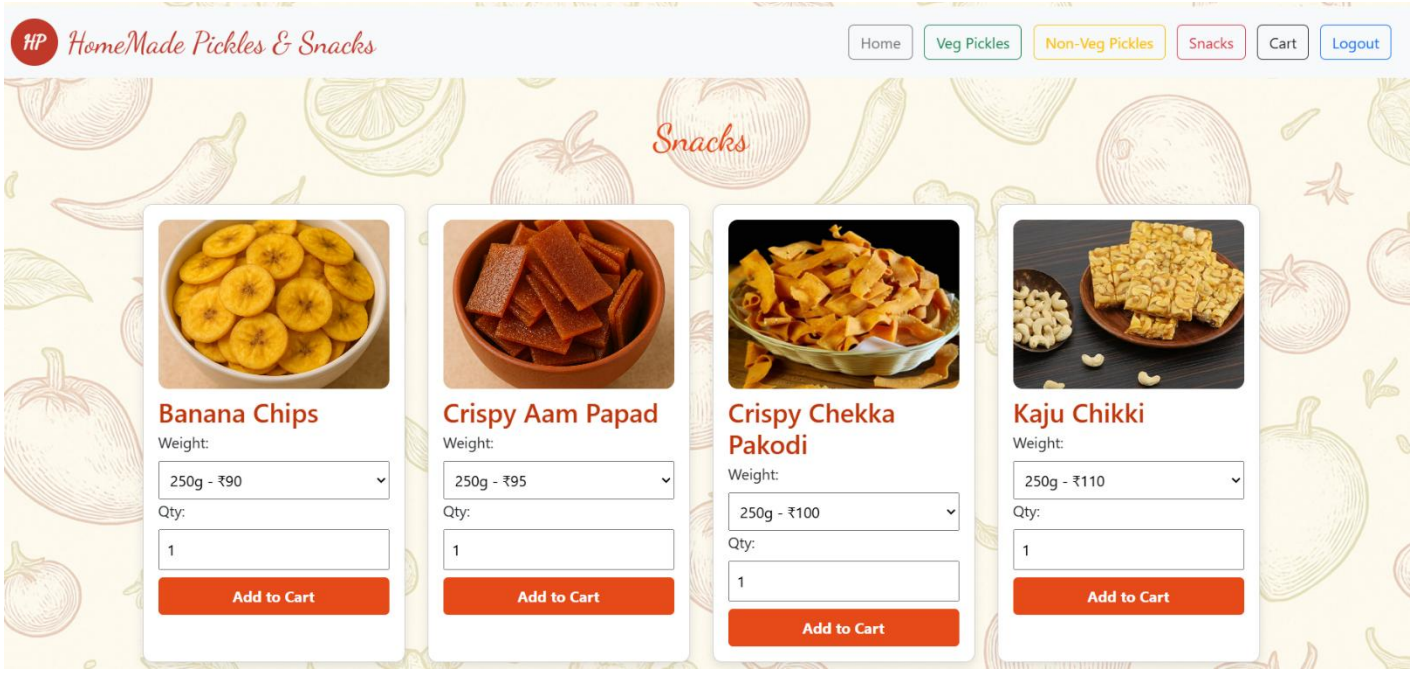
Veg Pickles Page:



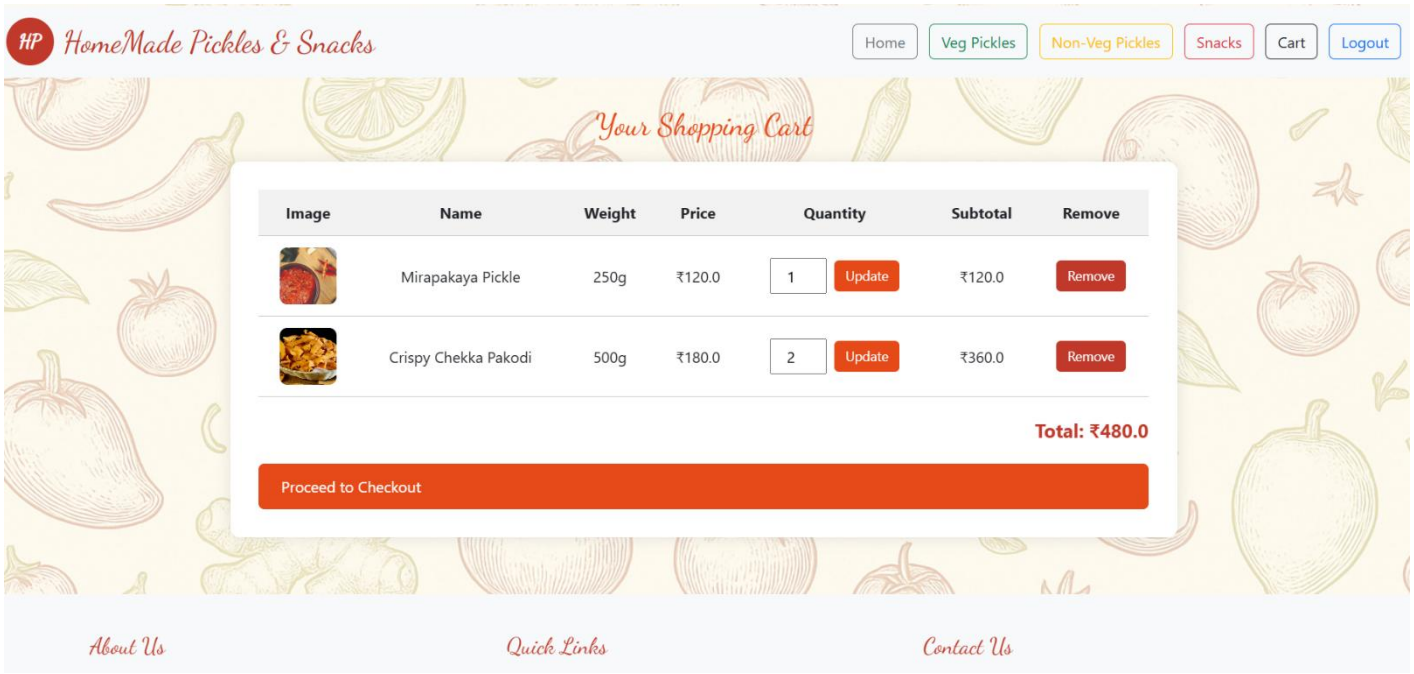
Non Veg Pickles Page:



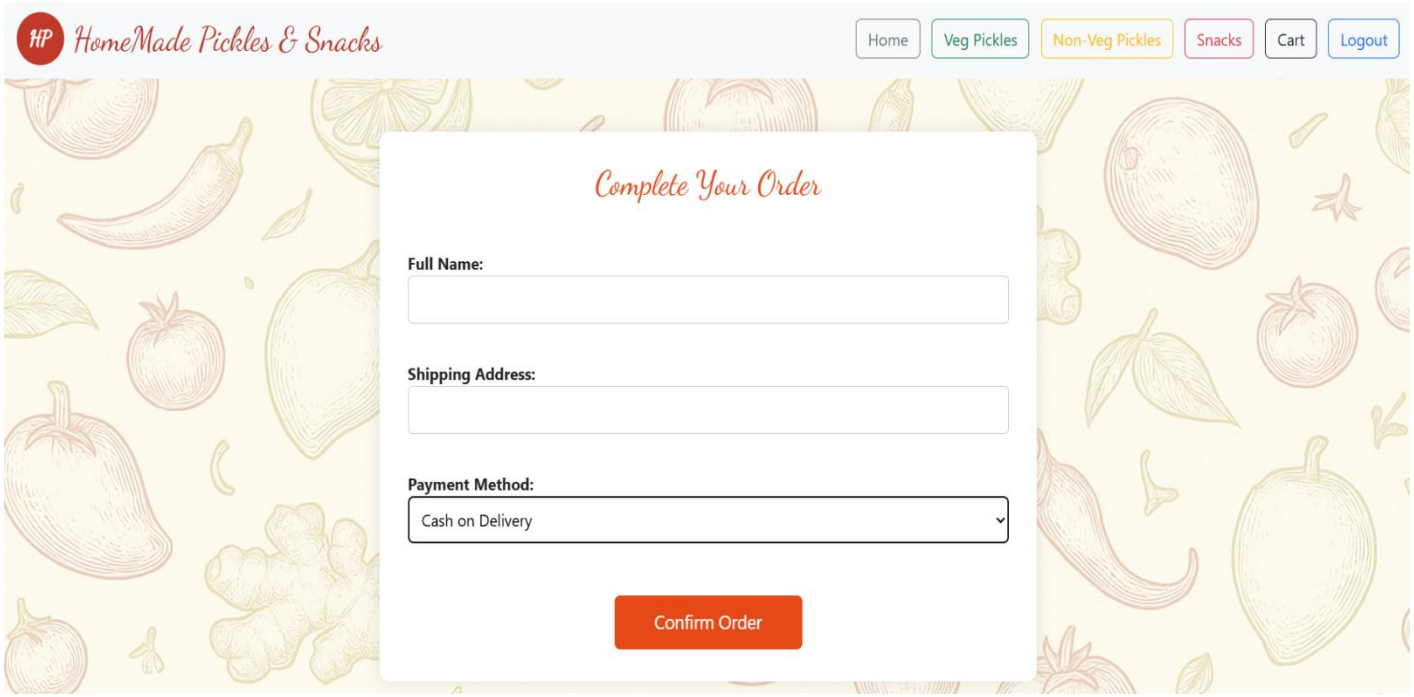
Snacks Page:



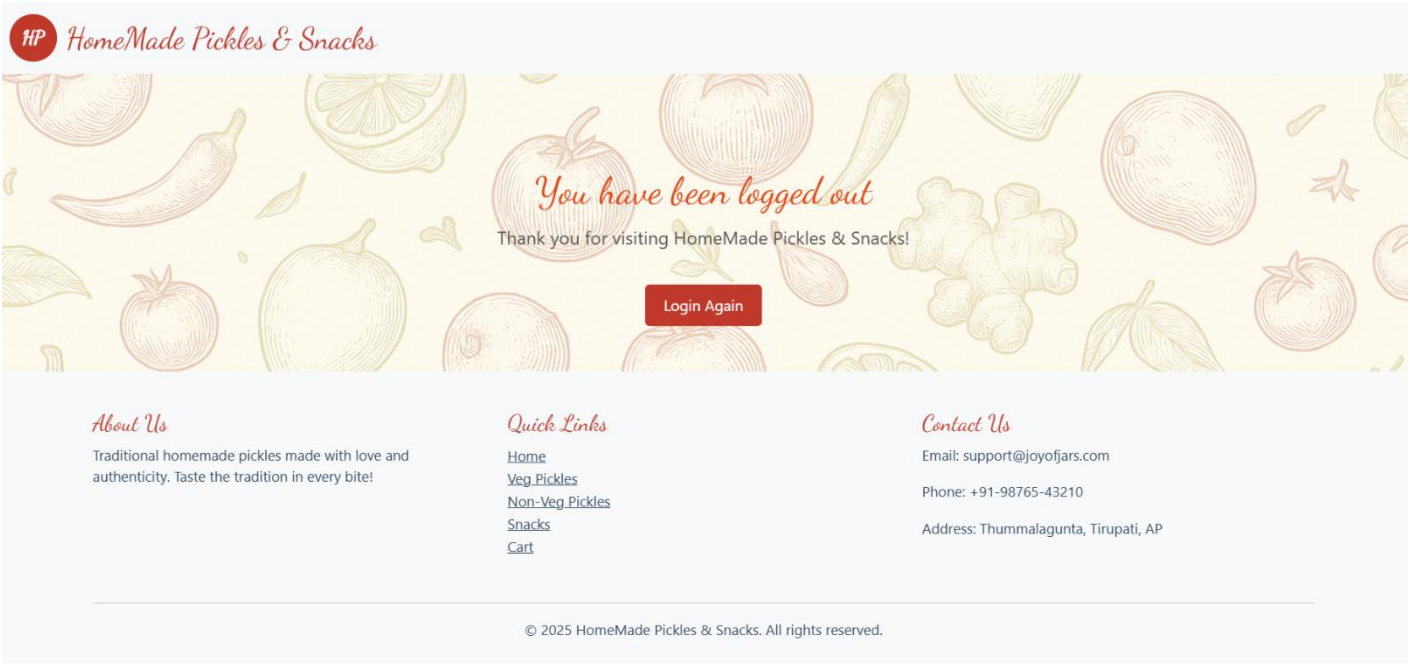
Cart Page:



Checkout Page:



Log Out Page:



Conclusion:

The Homemade Pickles and Snacks platform has been meticulously crafted to deliver a seamless and delightful experience for food enthusiasts seeking authentic, handcrafted flavors. By leveraging modern web technologies such as Flask for backend logic, secure user authentication, and dynamic cart management, the platform ensures a user-friendly interface for browsing, customizing, and ordering artisanal pickles and snacks.

The integration of cloud-ready architecture (e.g., AWS for future scalability) and robust session management allows the platform to handle high traffic efficiently while maintaining real-time updates for orders and inventory. Features like weight-based pricing, category-specific searches, and instant checkout streamline the shopping process, empowering customers to explore a diverse range of traditional and innovative recipes with ease.

This project addresses the growing demand for homemade, preservative-free food products by bridging the gap between small-scale producers and discerning customers. The platform's intuitive design and secure payment workflows enhance trust and convenience, while backend tools enable effortless inventory tracking and order fulfillment for administrators.

By combining time-honored recipes with modern e-commerce capabilities, this website not only preserves culinary heritage but also adapts to the digital age, ensuring that every jar of pickle or snack reaches customers with the same care and quality as a homemade meal. As the platform

evolves, it stands ready to scale, introduce new product lines, and foster a community of food lovers united by a passion for authentic flavors.

In essence, this project redefines the way homemade delicacies are shared and enjoyed, offering a flavorful bridge between tradition and technology.