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.

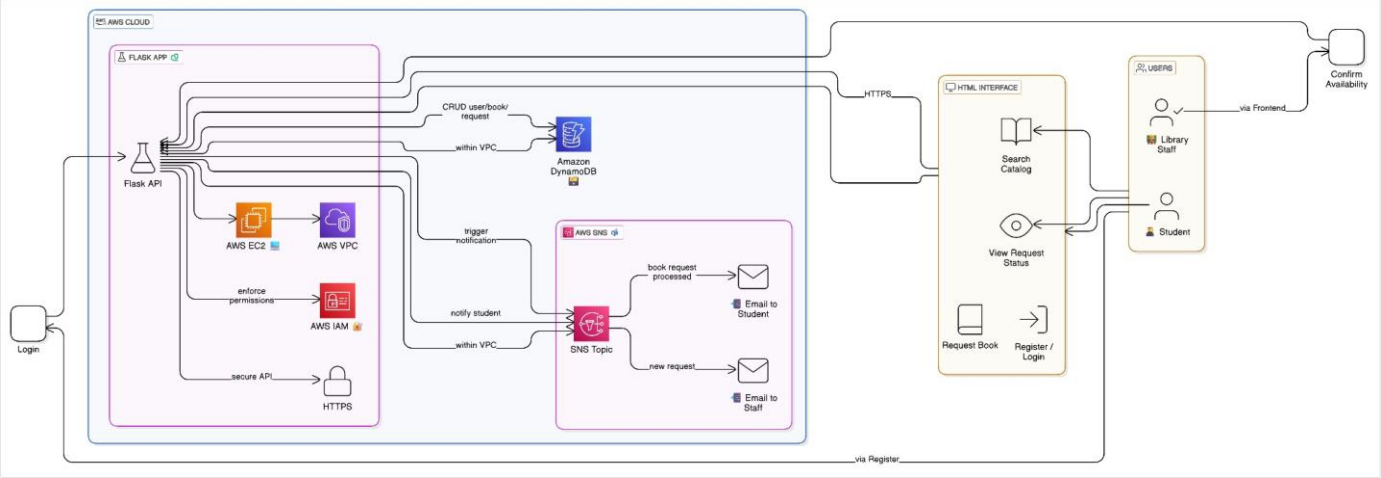
Scenario 2: Real-Time Inventory Tracking and Updates

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.

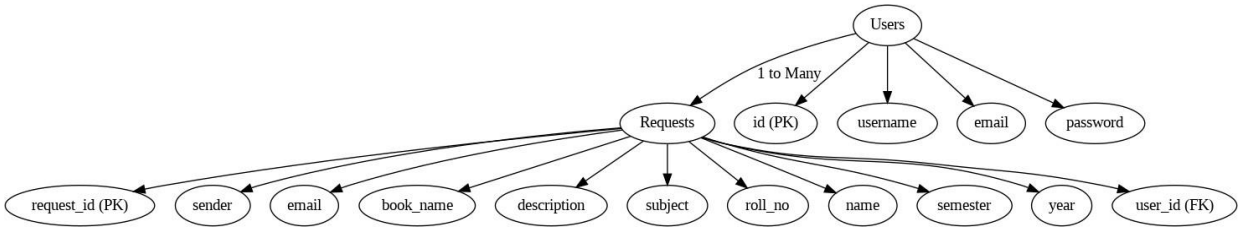
Scenario 3: Personalized User Experience and Recommendations

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**: 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

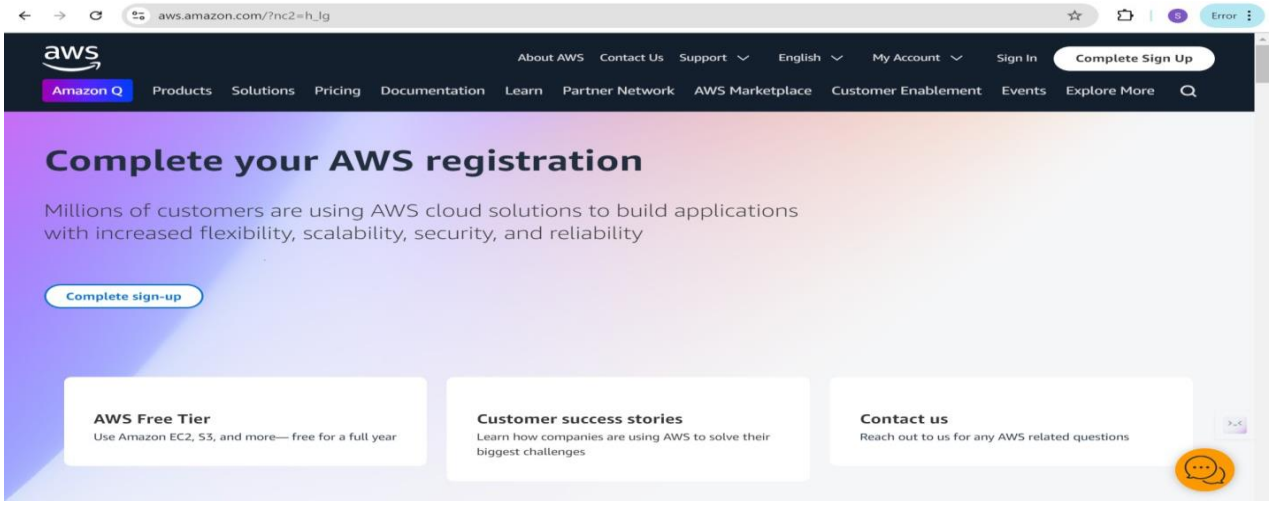
Activity 7.2: Run the Flask App

8. Testing and Deployment

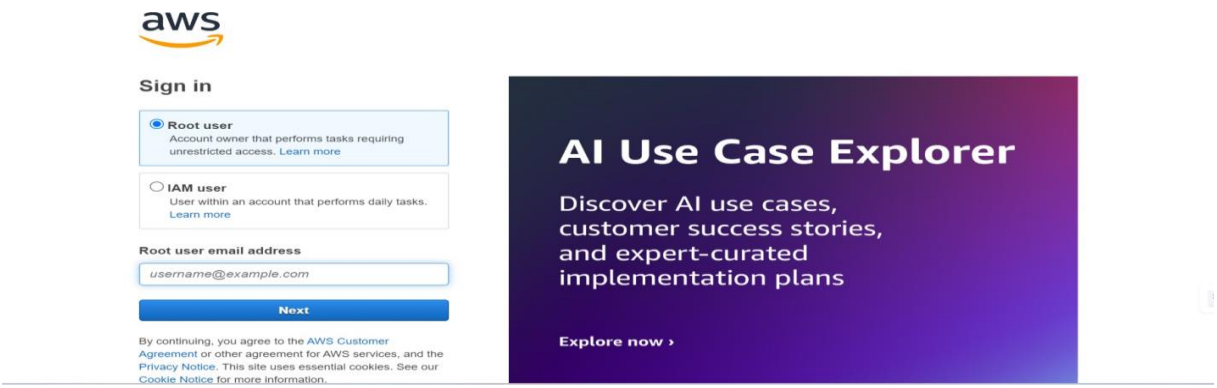
Activity 8.1: Conduct functional testing to verify user registration, login, book requests, and notifications.

Milestone 1: AWS Account Setup and Login

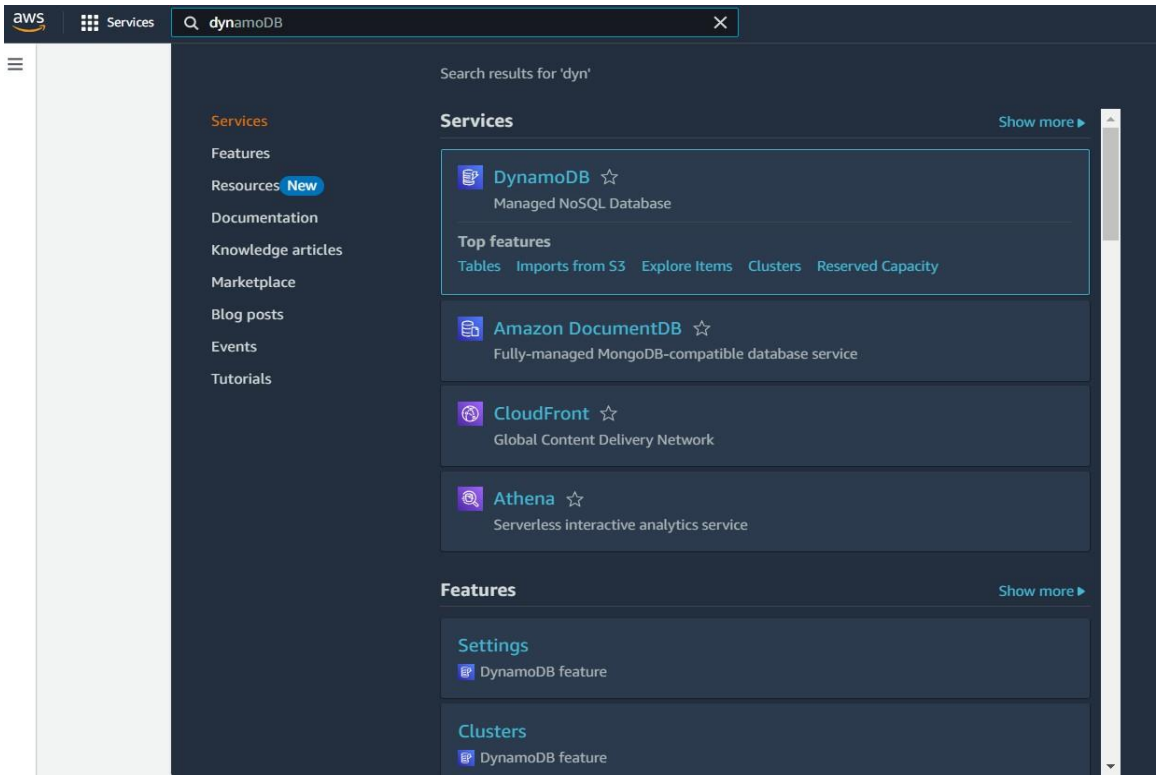
- **Activity 1.1: Set up an AWS account if not already done.**
  - Sign up for an AWS account and configure billing settings.

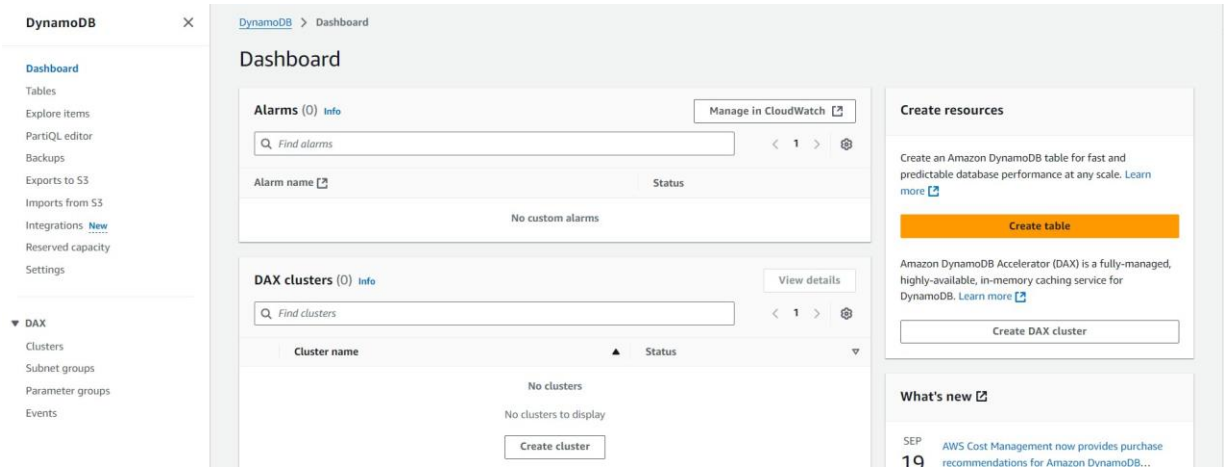


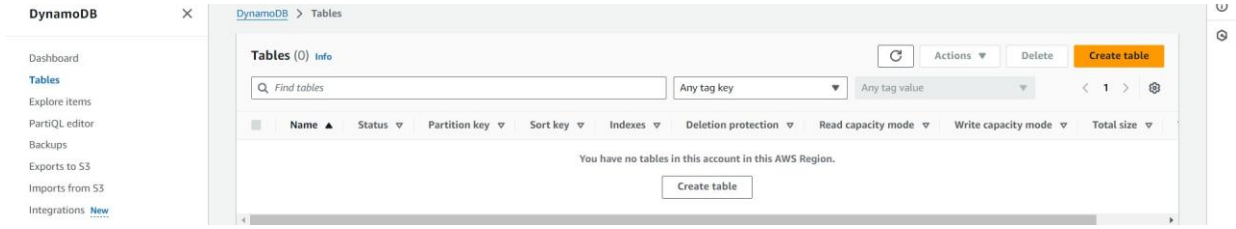
- **Activity 1.2: Log in to the AWS Management Console**
  - After setting up your account, log in to the [AWS Management Console].



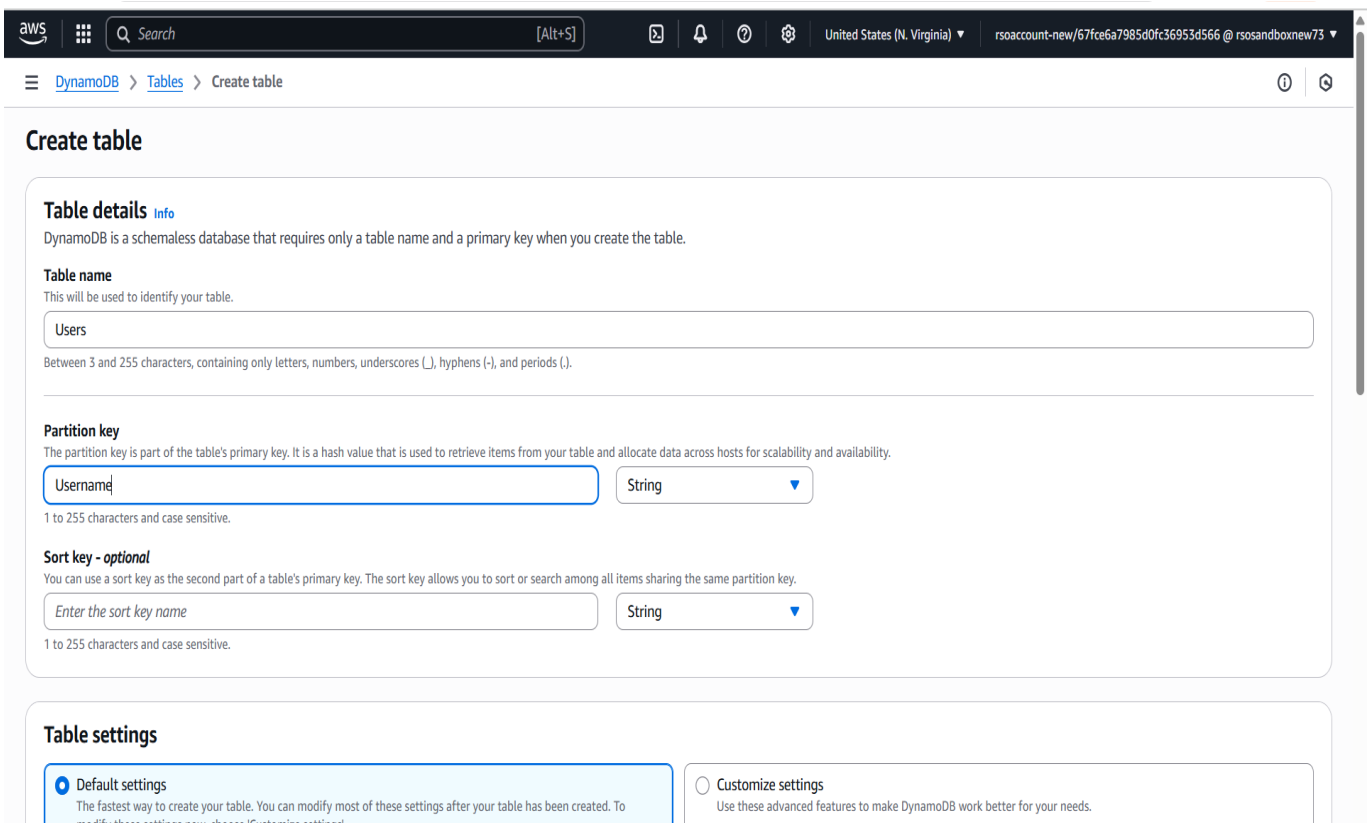
Milestone 2: DynamoDB Database Creation and Setup

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



Default table settings

These are the default settings for your new table. You can change some of these settings after creating the table.

Setting	Value	Editable after creation
Table class	DynamoDB Standard	Yes
Capacity mode	On-demand	Yes
Maximum read capacity units	-	Yes
Maximum write capacity units	-	Yes
Local secondary indexes	-	No
Global secondary indexes	-	Yes
Encryption key management	AWS owned key	Yes
Deletion protection	Off	Yes
Resource-based policy	Not active	Yes

Tags

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

No tags are associated with the resource.

Add new tag

You can add 50 more tags.

This table will be created with auto scaling deactivated. You do not have permissions to turn on auto scaling.

Cancel | Create table

DynamoDB > Tables

DynamoDB

- Dashboard
- Tables
- Explore items
- PartiQL editor
- Backups
- Exports to S3
- Imports from S3
- Integrations New
- Reserved capacity
- Settings

The Users table was created successfully.

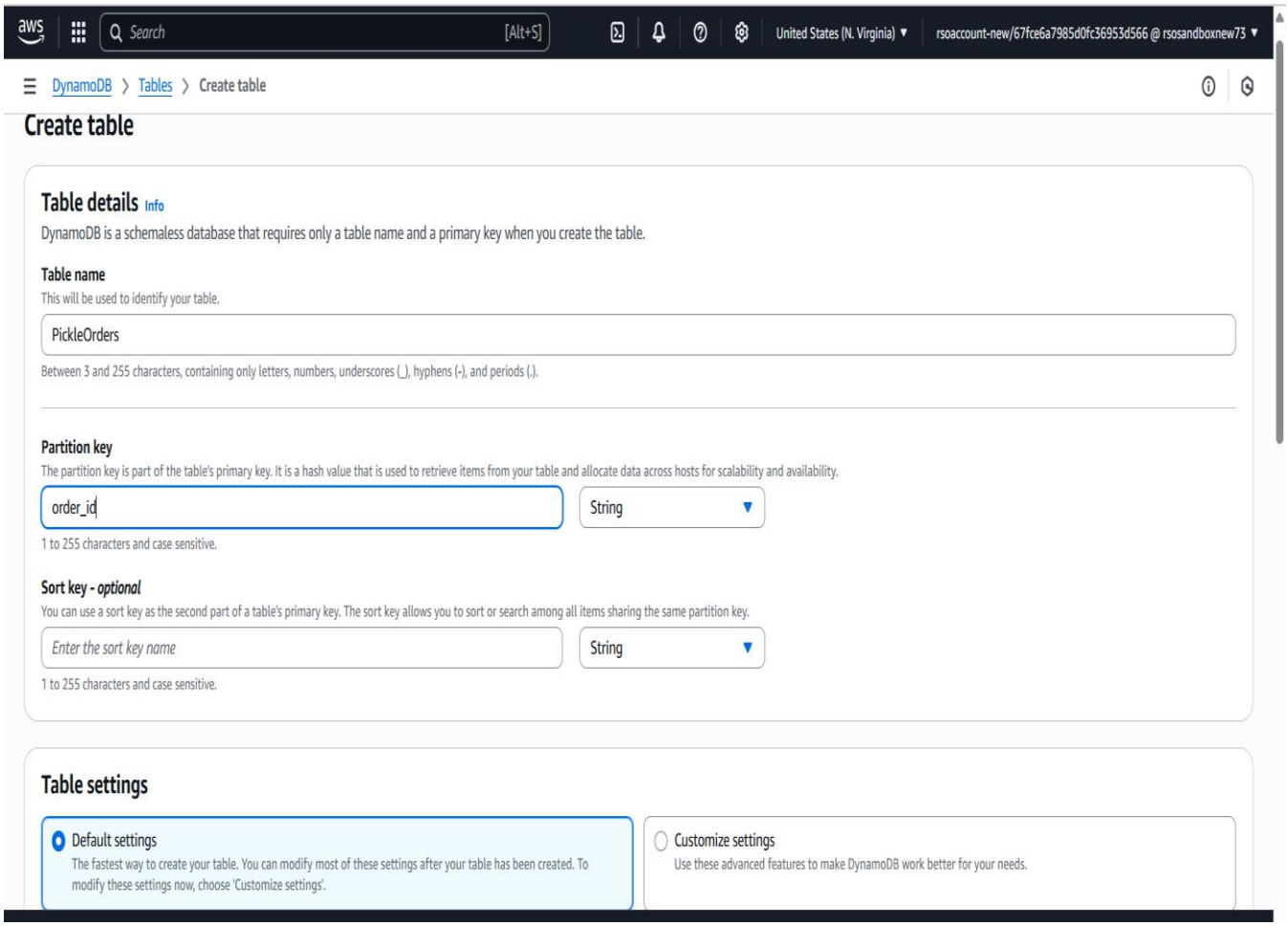
Tables (1) Info

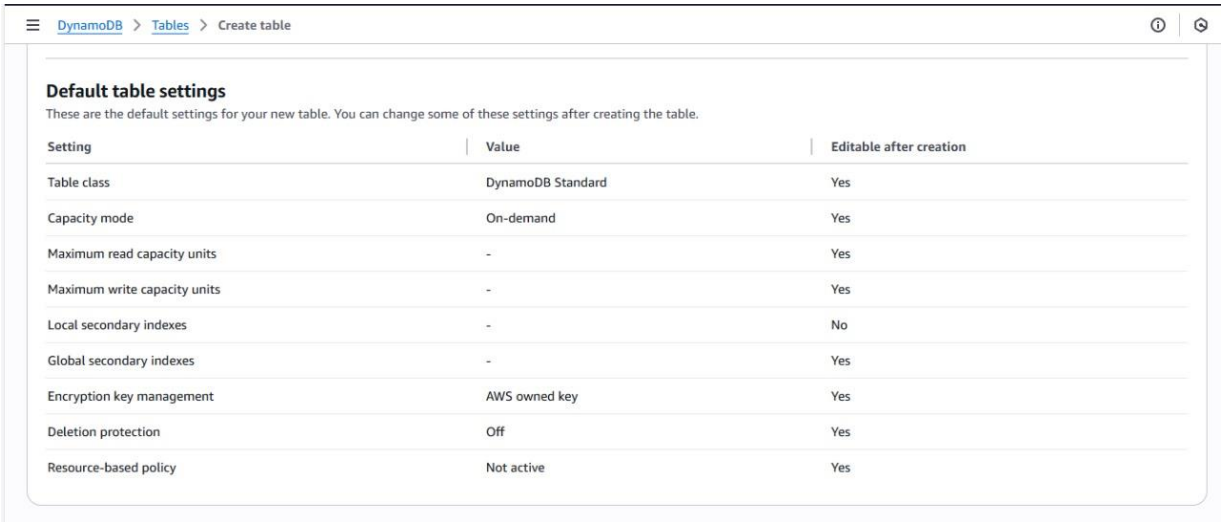
Actions | Delete | Create table

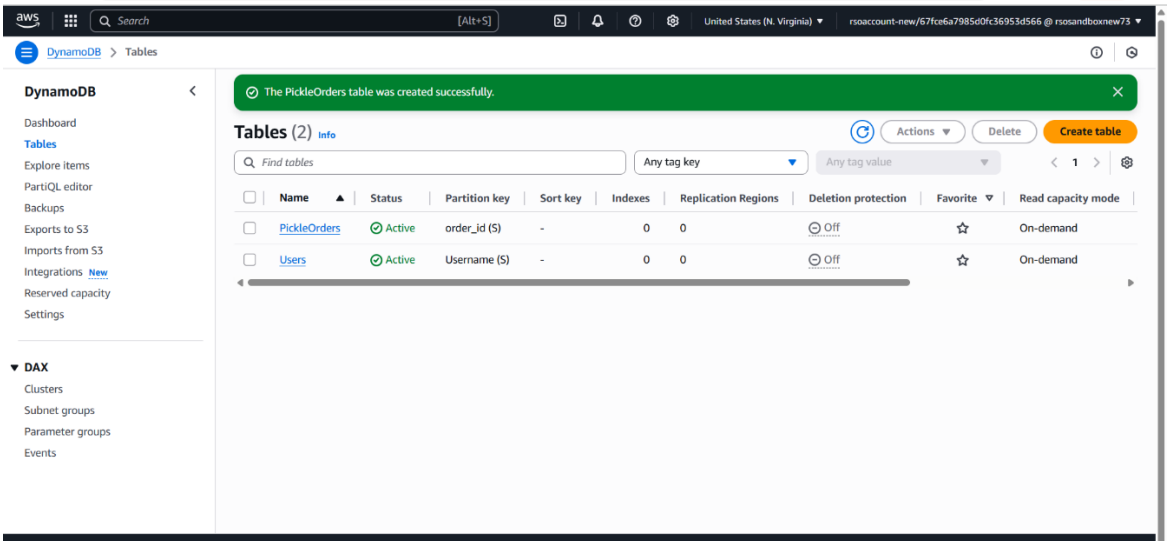
Find tables | Any tag key | Any tag value | 1

Name	Status	Partition key	Sort key	Indexes	Replication Regions	Deletion protection	Favorite	Read capacity mode
Users	Active	Username (S)	-	0	0	Off		On-demand

- Follow the same steps to create an Orders table with Order_id as the primary key to store Order details.



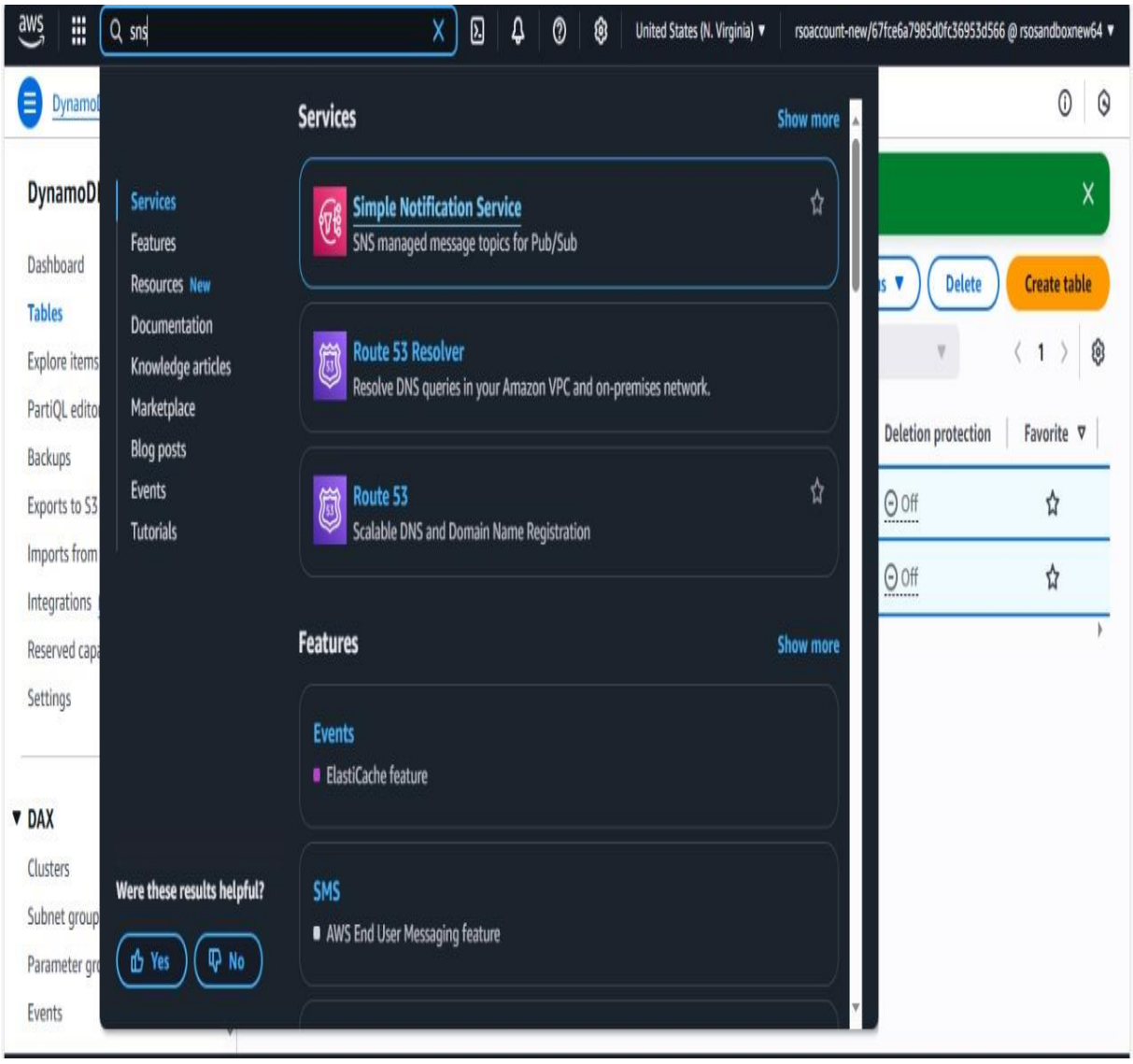


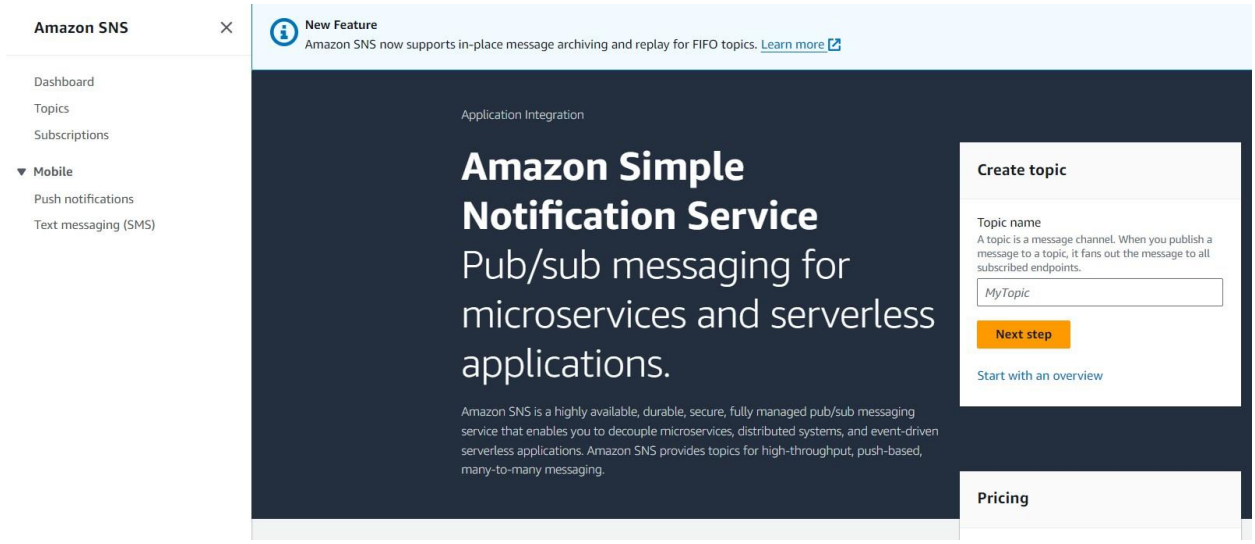


Milestone 3: SNS Notification Setup

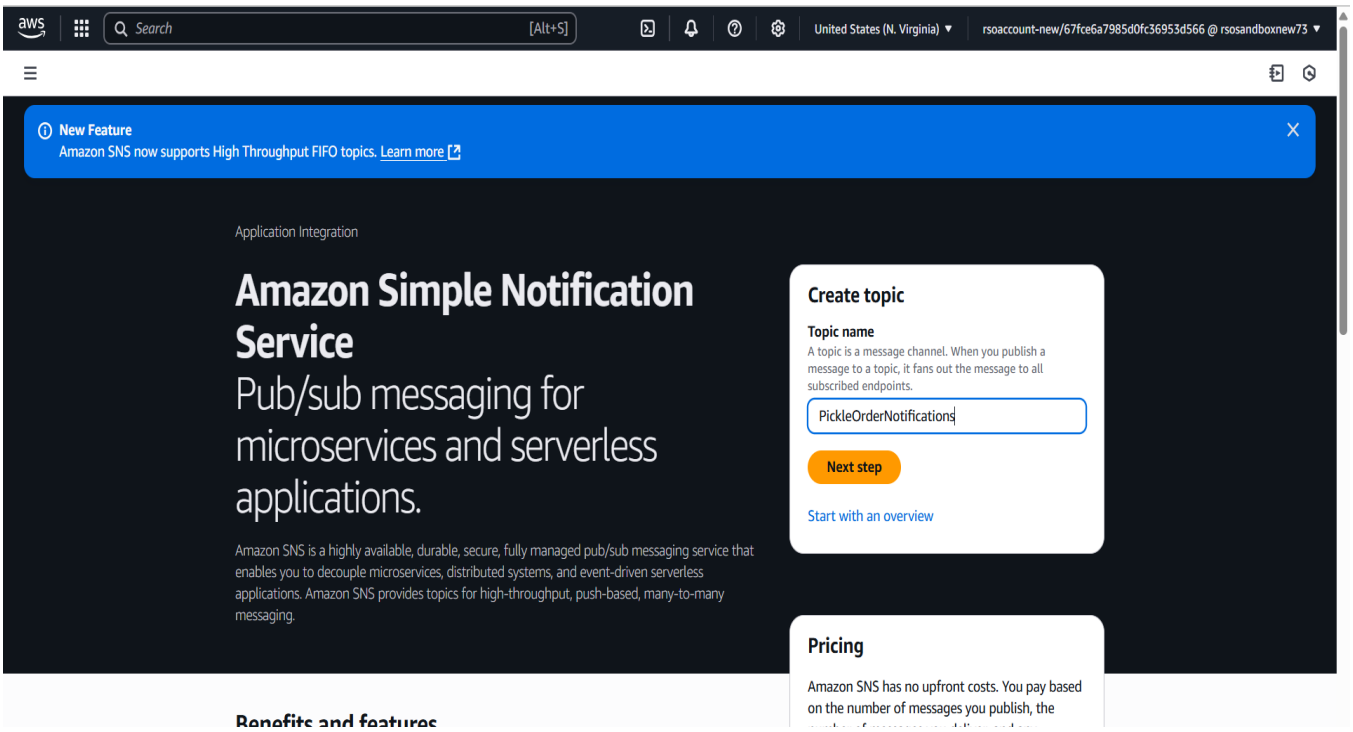
- **Activity 3.1：Create SNS topics for sending email notifications to users and library staff.**

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

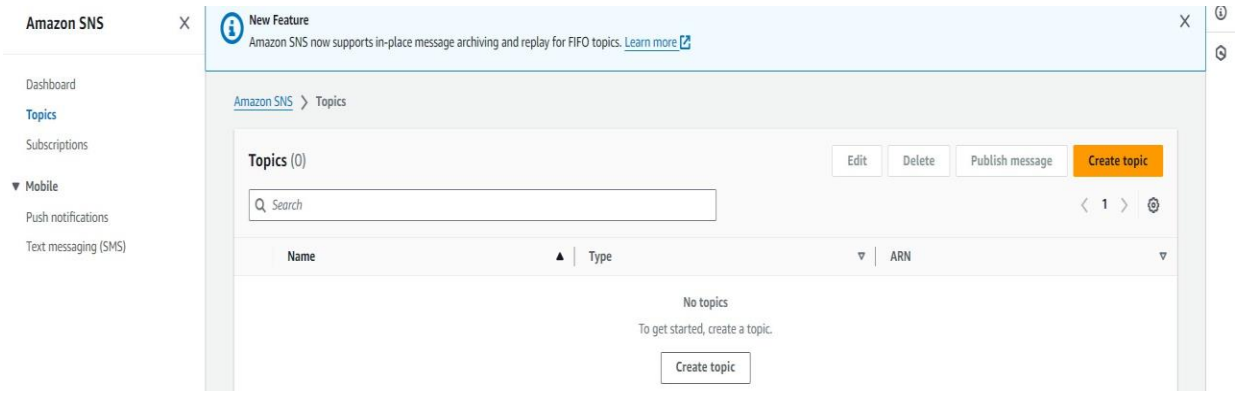




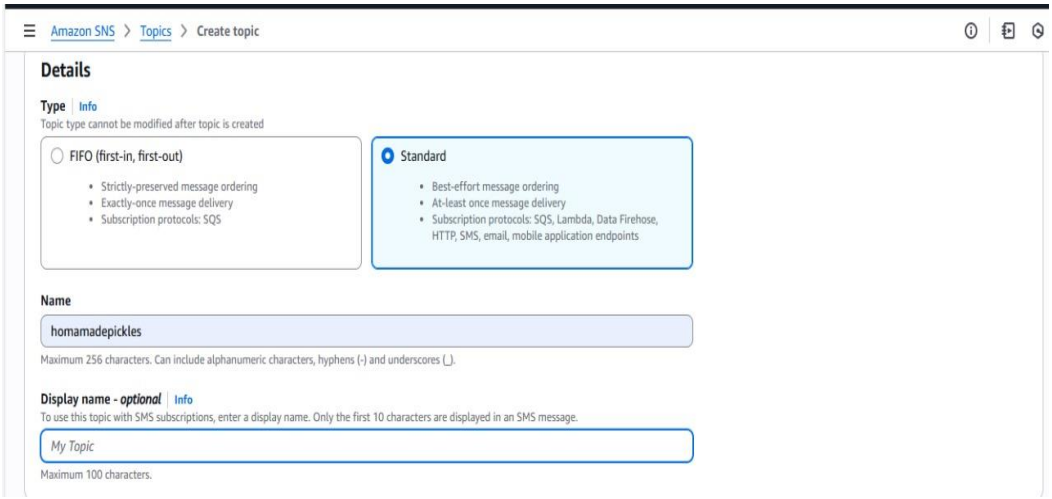
- Enter Topic Name and Click Next Step

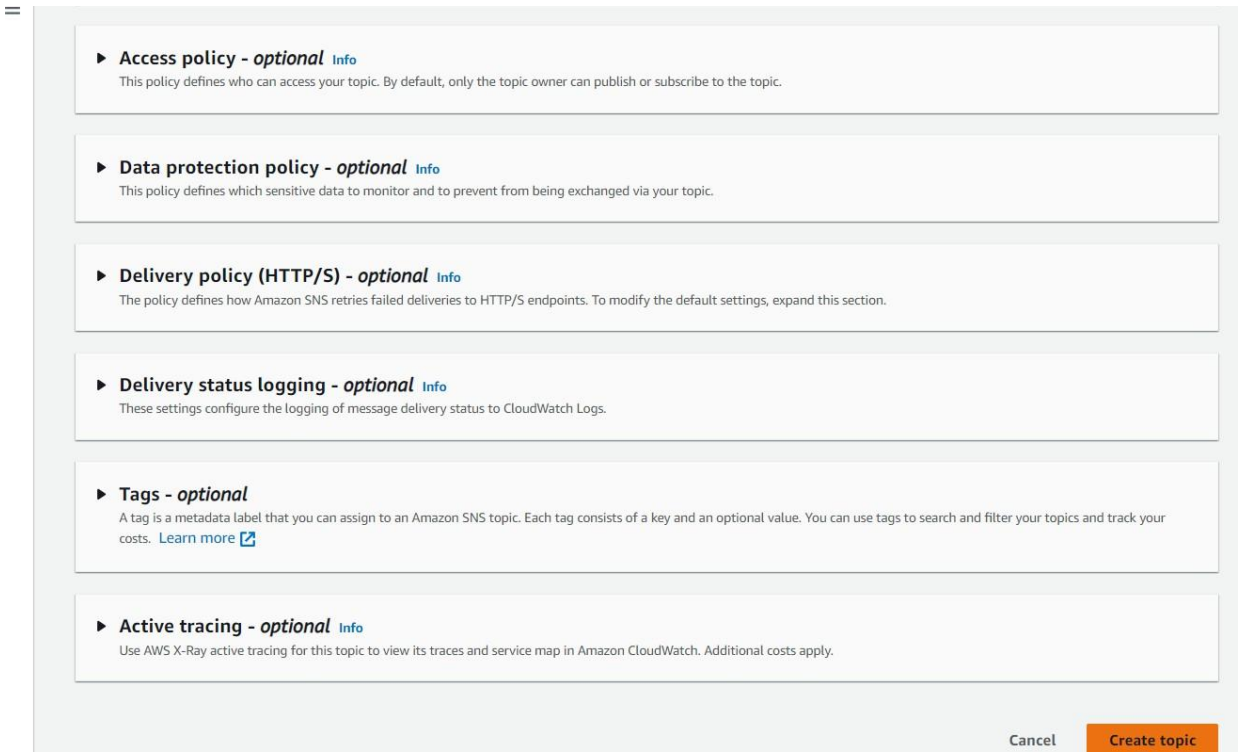


- Click on **Create Topic** and choose a name for the topic.

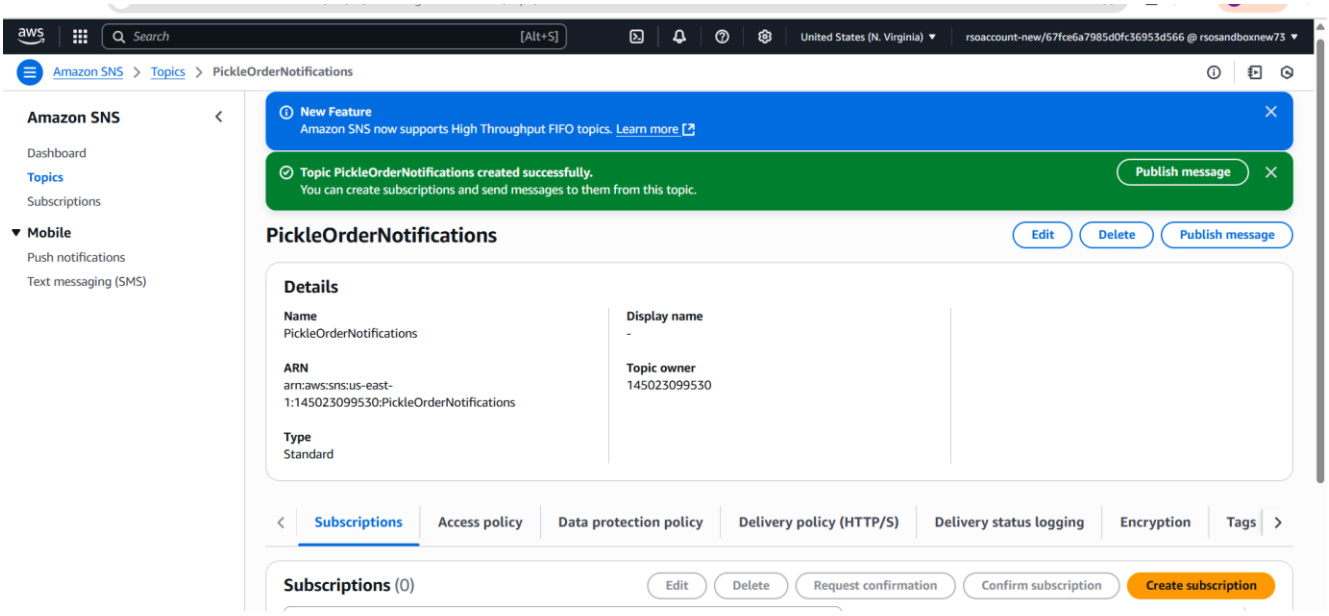


- Choose Standard type for general notification use cases and Click on Create Topic.



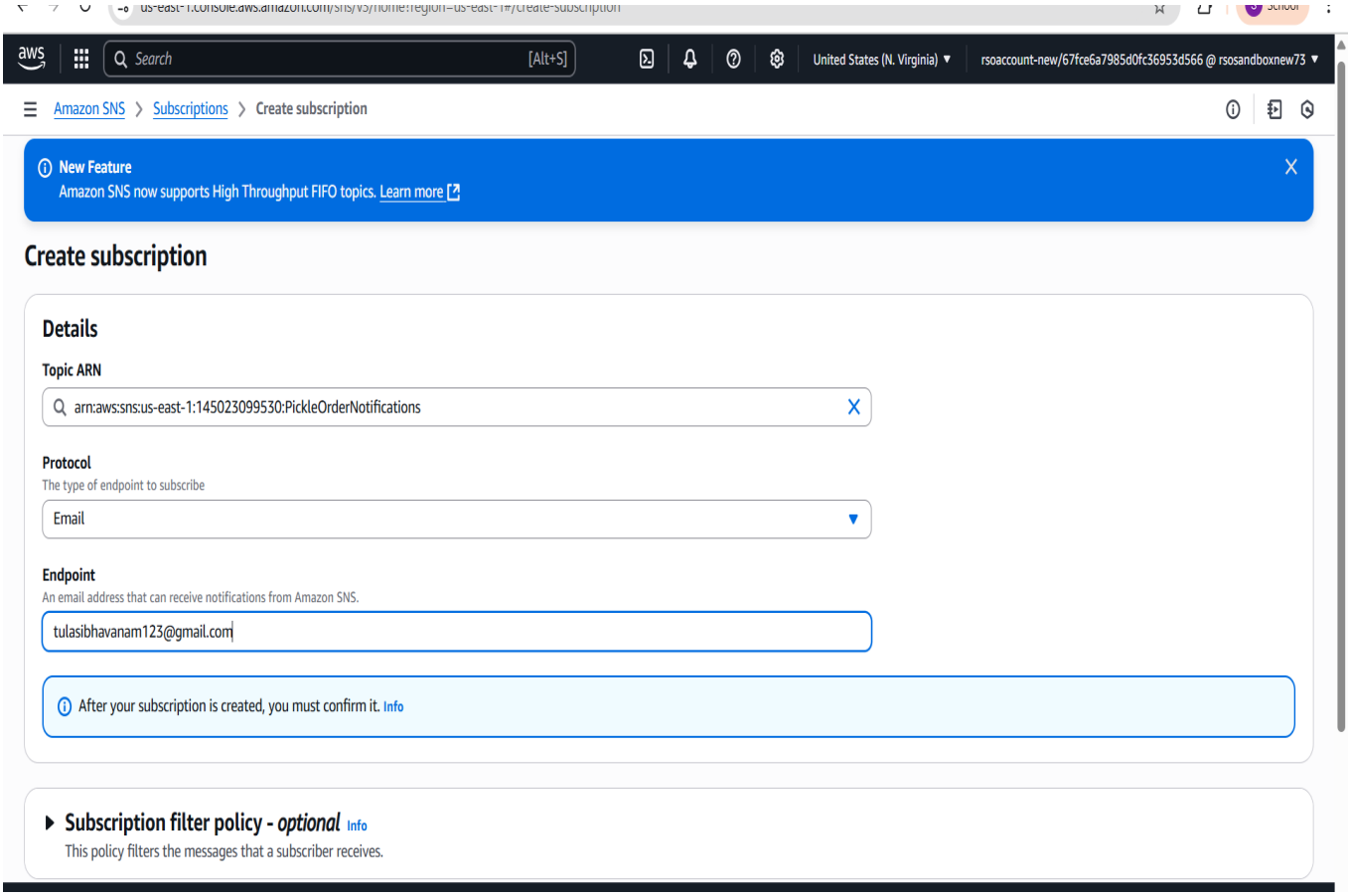


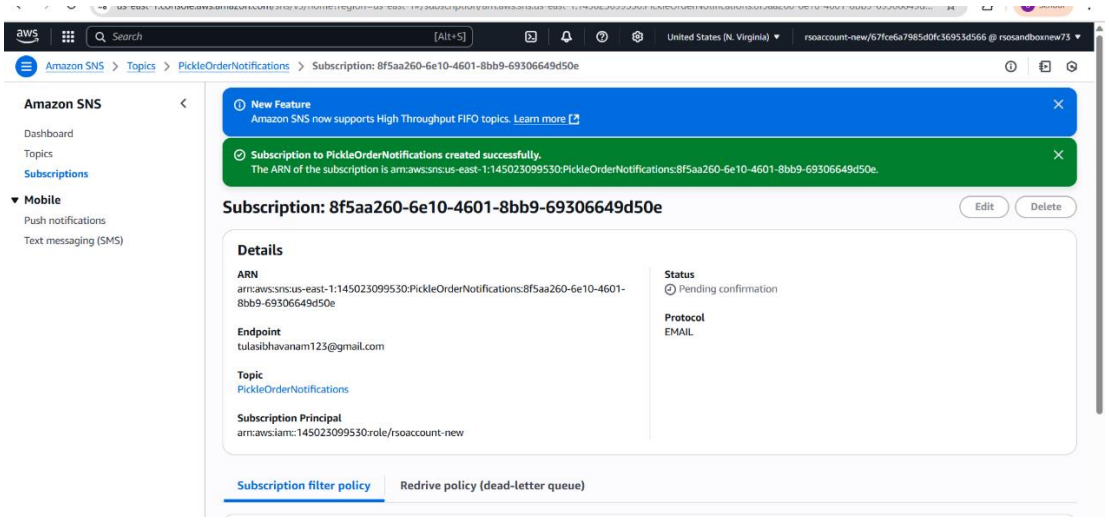
- Configure the SNS topic and note down the **Topic ARN**.



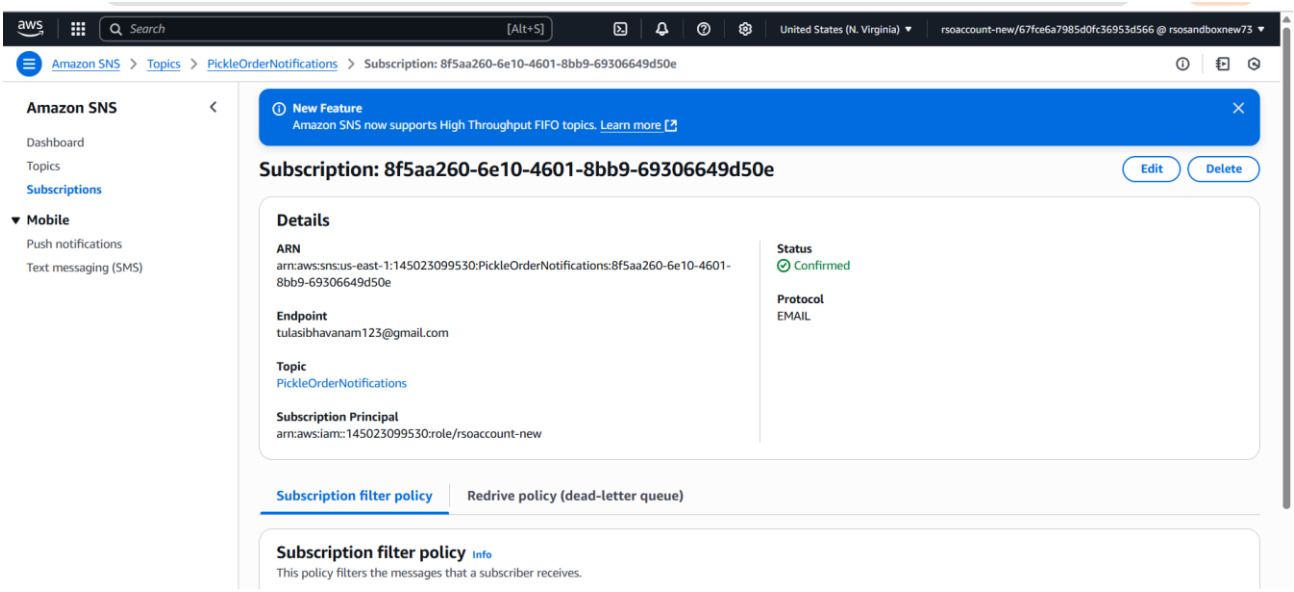
- **Activity 3.2: Subscribe users and staff to relevant SNS topics to receive real-time notifications when a order equest is made.**

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

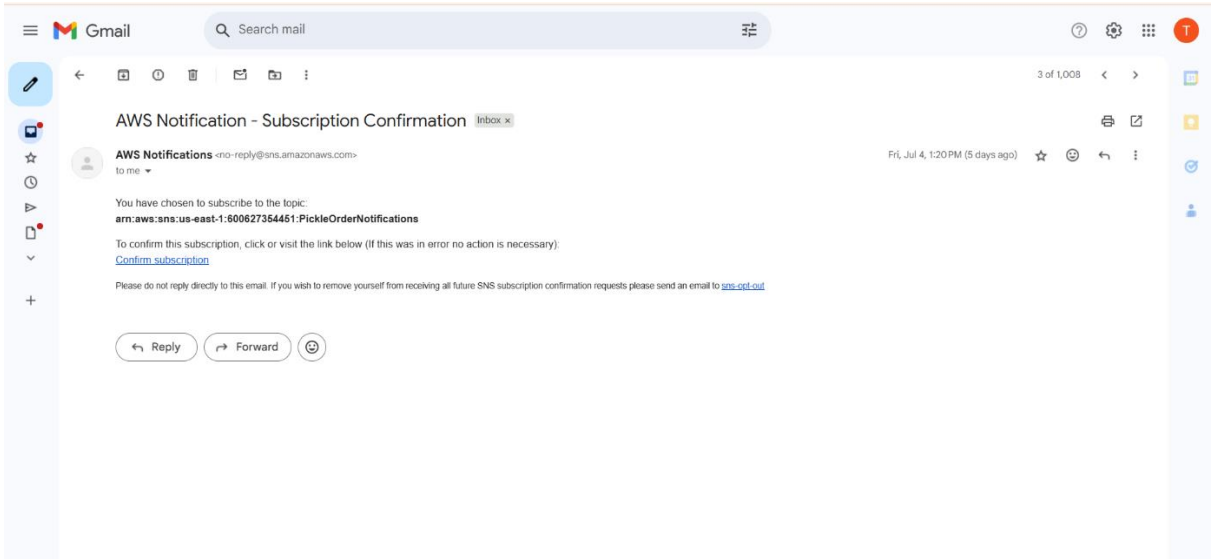


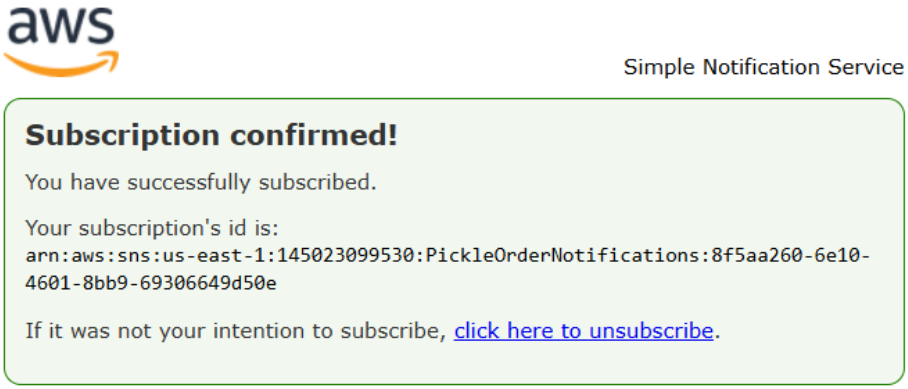


- After subscription request for the mail confirmation



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

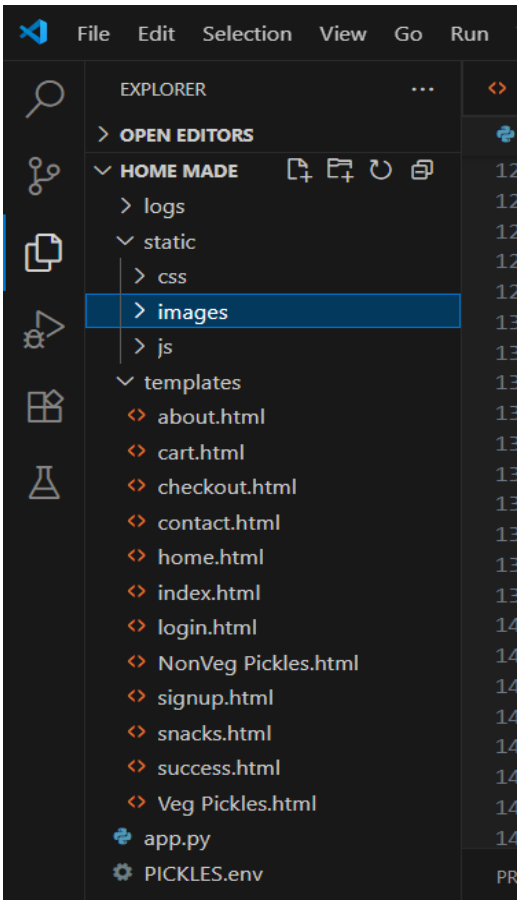




- Successfully done with the SNS mail subscription and setup, now store the ARN link.

Milestone 4:Backend Development and Application Setup

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



Description:

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

Description of the code :

- **Flask App Initialization**

```
from flask import Flask, render_template, request, redirect, url_for
import boto3
from boto3.dynamodb.conditions import Key
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from bcrypt import hashpw, gensalt, checkpw
```

Description: import essential libraries including Flask utilities for routing, Boto3 for DynamoDB operations, SMTP and email modules for sending mails, and Bcrypt for password hashing and verification

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

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

- **Dynamodb Setup:**

```
AWS_REGION = 'us-east-1'
SNS_TOPIC_ARN = 'arn:aws:sns:us-east-1:600627354451:PickleOrderNotifications'

dynamodb = boto3.resource('dynamodb', region_name=AWS_REGION)
orders_table = dynamodb.Table('PickleOrders')
users_table = dynamodb.Table('users')
sns = boto3.client('sns', region_name=AWS_REGION)
```

Description: initialize the DynamoDB resource for the ap-south-1 region and set up access to the Users and Requests tables for storing user details and book requests.

- **SNS Connection**

```
# Email settings
EMAIL_HOST = os.getenv('EMAIL_HOST', 'smtp.gmail.com')
EMAIL_PORT = int(os.getenv('EMAIL_PORT', 587))
EMAIL_USER = os.getenv('EMAIL_USER')
EMAIL_PASSWORD = os.getenv('EMAIL_PASSWORD')

@app.context_processor
def inject_theme():
    return {"color": app.config["THEME_COLOR"], "year": datetime.now().year}

# -------------------- Product Inventory --------------------
products = {
    "mango": {"name": "Mango Pickle", "price": 200, "stock": 10, "image": "mango pickle.webp"},
    "tomato": {"name": "Tomato Pickle", "price": 150, "stock": 7, "image": "Tomato pickle.webp"},
    "lemon": {"name": "Lemon Pickle", "price": 180, "stock": 8, "image": "Lemon pickle.jpg"},
    "chicken": {"name": "Chicken Pickle", "price": 250, "stock": 9, "image": "chicken pickle.webp"},
    "fish": {"name": "Fish Pickle", "price": 250, "stock": 6, "image": "Fish pickle.webp"},
    "mutton": {"name": "Mutton Pickle", "price": 300, "stock": 7, "image": "Mutton pickle.webp"},
    "banana_chips": {"name": "Banana Chips", "price": 100, "stock": 8, "image": "Banana Chips.jpg"},
    "ama_papad": {"name": "Ama Papad", "price": 80, "stock": 8, "image": "Aam papad.jpg"},
    "chekka_pakodi": {"name": "Chekka Pakodi", "price": 110, "stock": 5, "image": "Chekka Pakodi.jpg"}
}

def get_products(prefix=None):
    if not prefix:
        return products
    return {k: v for k, v in products.items() if k.startswith(prefix)}
```

Description: Configure **SNS** to send notifications when a book request is submitted. Paste your stored ARN link in the sns_topic_arn space, along with the region_name where the SNS topic is created. Also, specify the chosen email service in SMTP_SERVER (e.g., Gmail, Yahoo, etc.) and enter the subscribed email in the SENDER_EMAIL section. Create an ‘App password’ for the email ID and store it in the SENDER_PASSWORD section.

- **Routes for Web Pages**

.

- **Register Route:**

```
@app.route("/signup.html", methods=["GET", "POST"])
def signup():
    if request.method == 'POST':
        name = request.form.get('name')
        email = request.form.get('email')
        password = request.form.get('password')
        confirm = request.form.get('confirm')

        if password != confirm:
            return render_template('signup.html', error="Passwords do not match!")

        # Save user to dictionary (in real app, use database)
        users[email] = {
            'name': name,
            'password': password  # For security, use hashing (next step)
        }

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

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

Description: define /register route to validate registration form fields, hash the user password using Bcrypt, store the new user in DynamoDB with a login count, and send an SNS notification on successful registration

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

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

        user = users.get(email)
        if user and user['password'] == password:
            session['user'] = email
            return redirect(url_for('home'))
        else:
            return render_template('login.html', error="Invalid email or password")

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

Description: define /login route to validate user credentials against DynamoDB, check the password using Bcrypt, update the login count on successful authentication, and redirect users to the home page

- **Home routes:**

```python
# ------------------- Routes -------------------

@app.route('/')
def home():
    user = session.get('user')  # ✅ FIXED: get user safely
    if user:
        return render_template('home.html', user=user)
    return redirect(url_for('login'))
@app.route('/index.html')
def index():
    return render_template("index.html")
```

Description:

define the home route / to automatically redirect users to the register page when they access the base URL.

.

- **Request Routes:**

```
@app.route('/checkout.html', methods=['GET', 'POST'])
def checkout():
    if 'user' not in session:
        flash('Please login to checkout', 'warning')
        return redirect(url_for('login'))

    if request.method == 'POST':
        name = request.form.get('name')
        email = request.form.get('email')
        phone = request.form.get('phone')
        address = request.form.get('address')
        notes = request.form.get('notes')
        payment = request.form.get('payment')

        if not name or not email or not phone or not address or not payment:
            flash('All fields including payment method are required!', 'danger')
            return redirect(url_for('checkout'))

        order_id = str(uuid.uuid4())[:8].upper()
        session['cart'] = []

        flash(f'Order #{order_id} placed successfully!', 'success')
        return redirect(url_for('success'))

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

Description: define /request-form route to capture book request details from users, store the request in DynamoDB, send a thank-you email to the user, notify the admin, and confirm submission with a success message.

LogoutRoute:

```
@app.route('/logout')
def logout():
    session.pop('user', None)
    return redirect(url_for('login'))
```

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

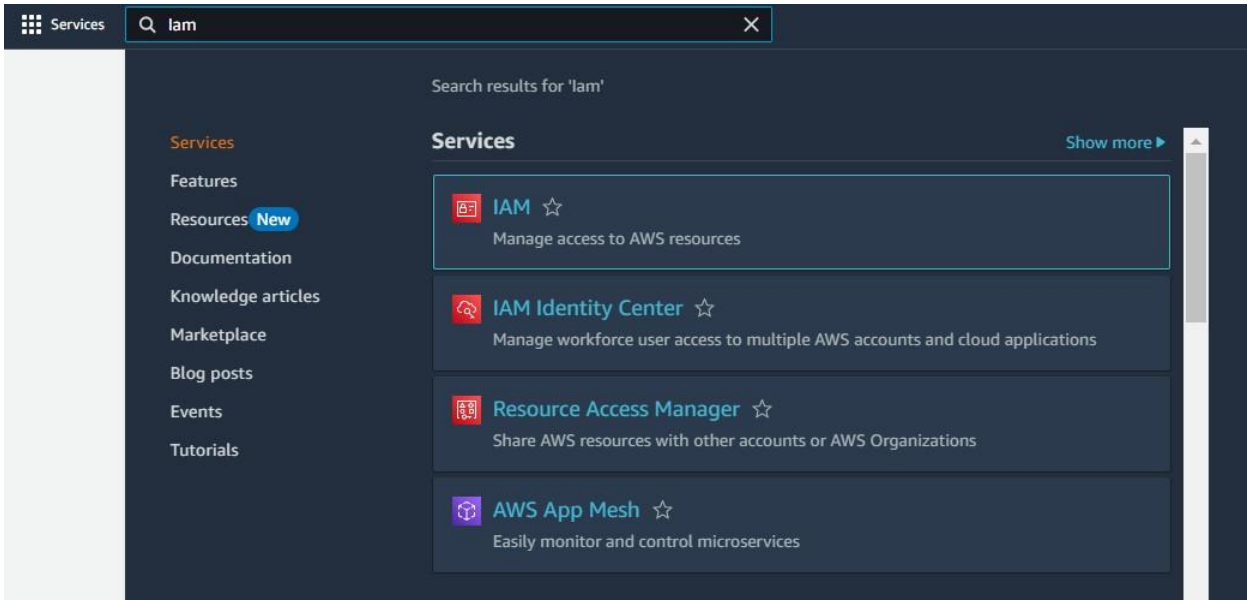
Deployment Code:

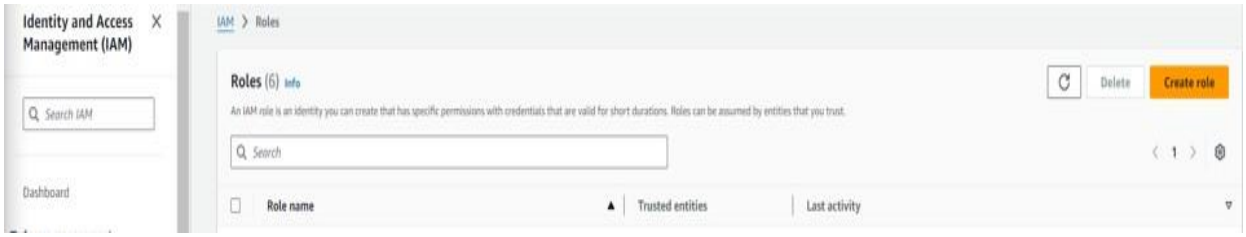
```
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000, debug=True)
```

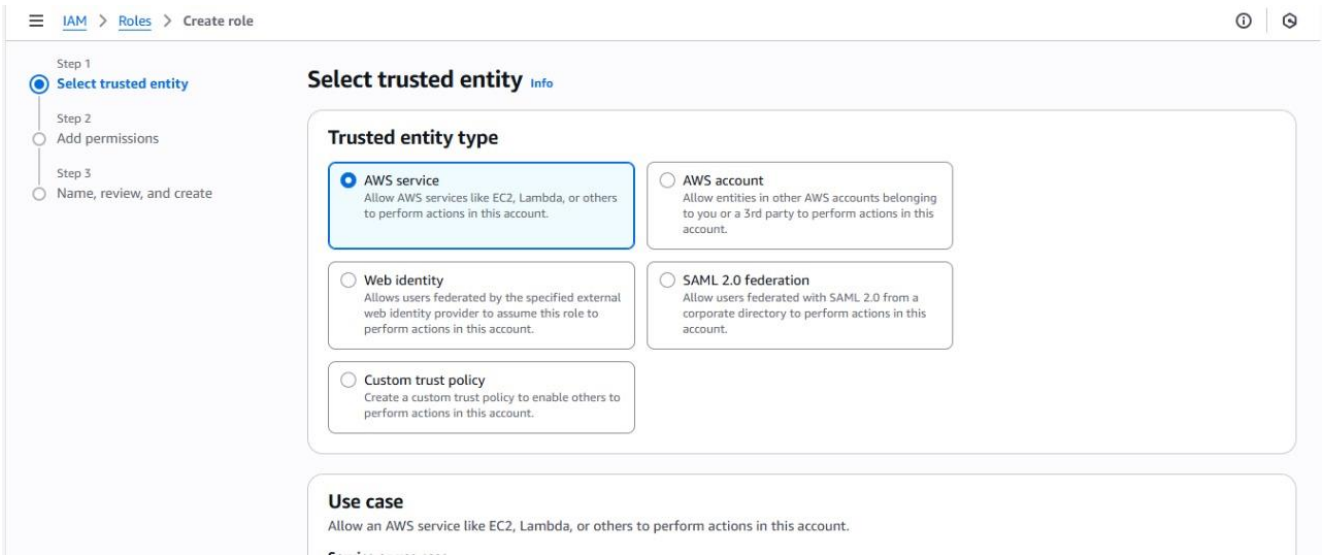
Description: start the Flask server to listen on all network interfaces (0.0.0.0) at port 80 with debug mode enabled for development and testing.

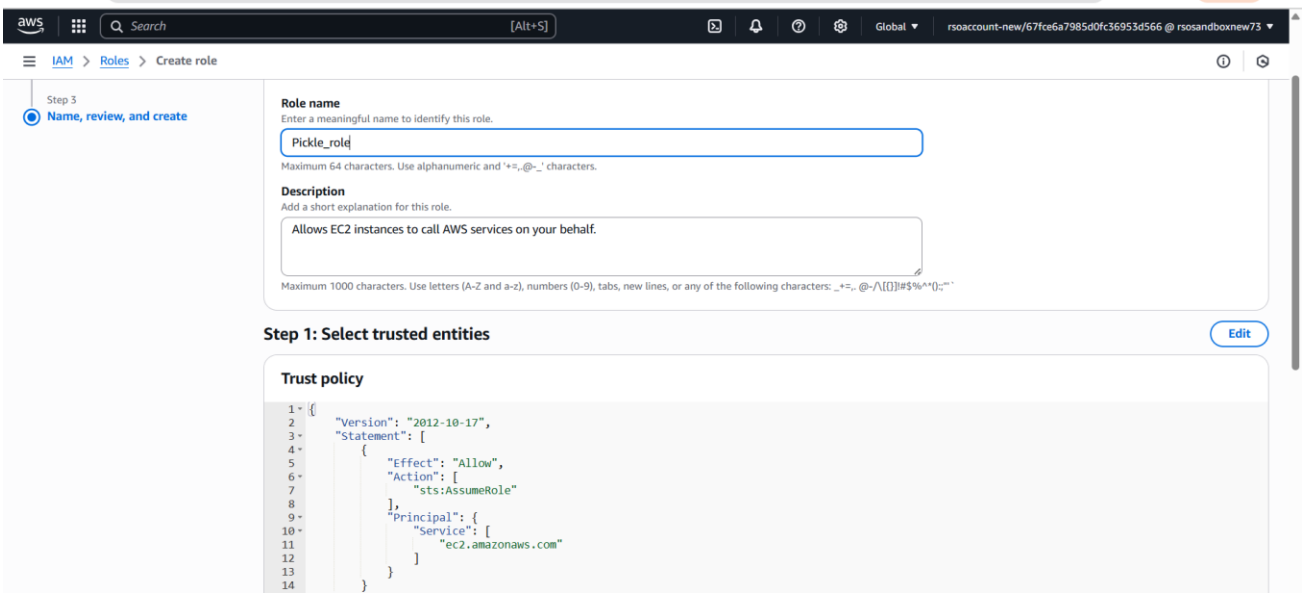
Milestone 5: IAM Role Setup

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





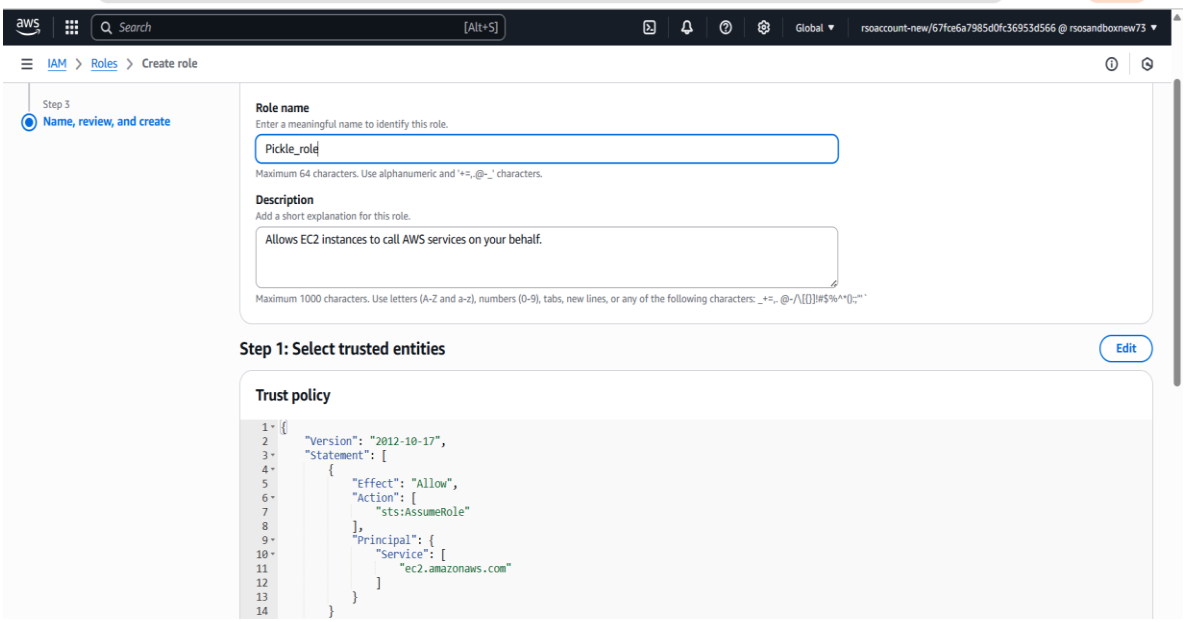


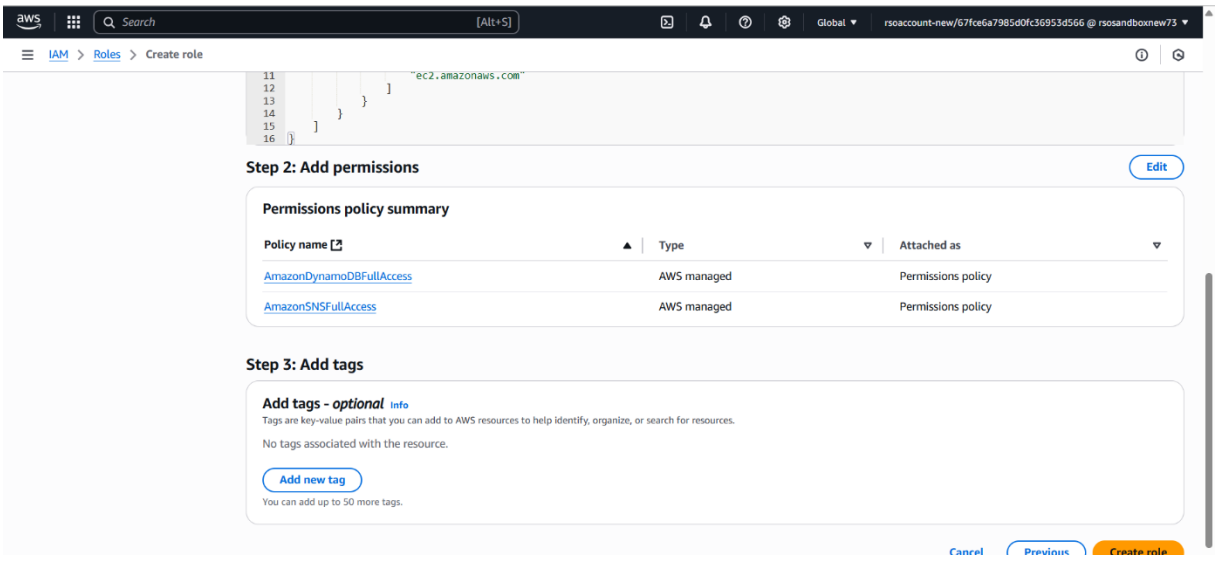


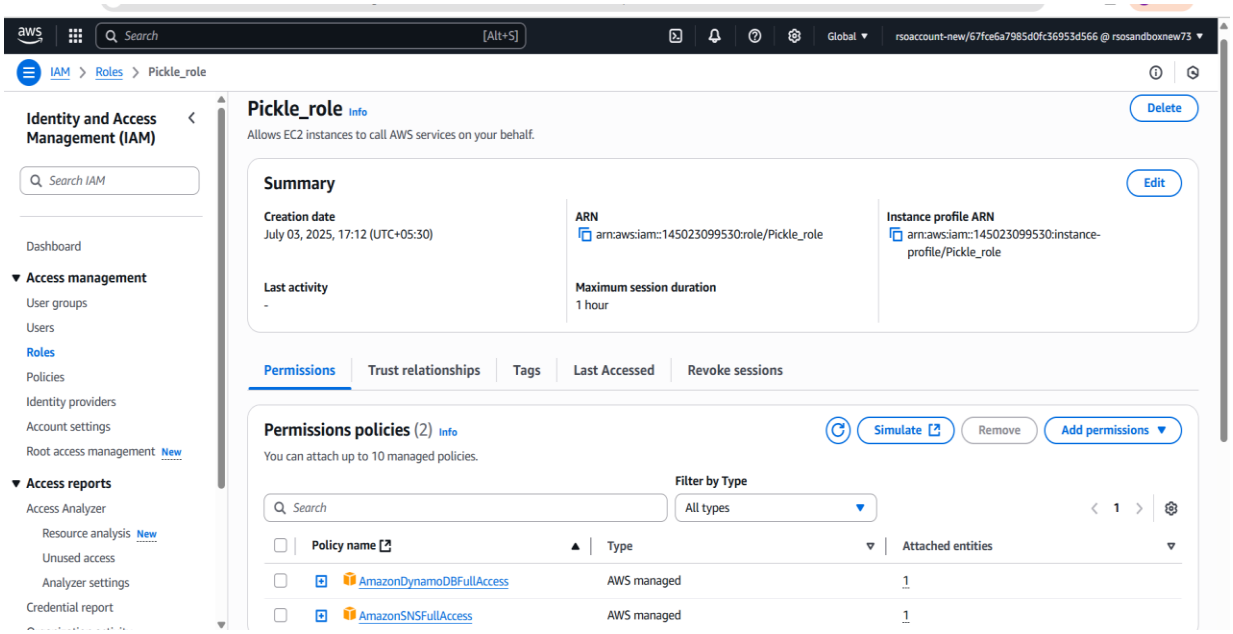
- **Activity 5.2: Attach Policies.**

  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.

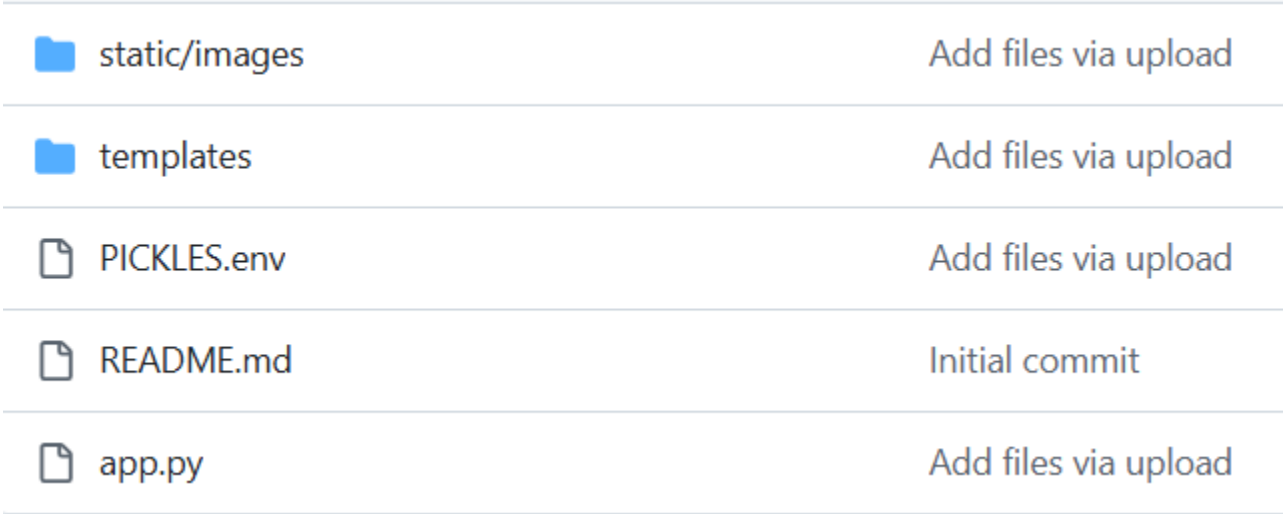

Search [Alt+S]
Global
rsoaccount-new/67fce6a7985d0fc36953d566 @ rsosandboxnew73
IAM > Roles > Create role
Step 3
Name, review, and create
Role name
Enter a meaningful name to identify this role.
Pickle_role
Maximum 64 characters. Use alphanumeric and '+=,.@-_' characters.
Description
Add a short explanation for this role.
Allows EC2 instances to call AWS services on your behalf.
Maximum 1000 characters. Use letters (A-Z and a-z), numbers (0-9), tabs, new lines, or any of the following characters: _+=,. @-/\[{}]!#$%^*();"'`
Step 1: Select trusted entities
Edit
Trust policy
{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Action": [
"sts:AssumeRole"
],
"Principal": {
"Service": [
"ec2.amazonaws.com"
]
}
}

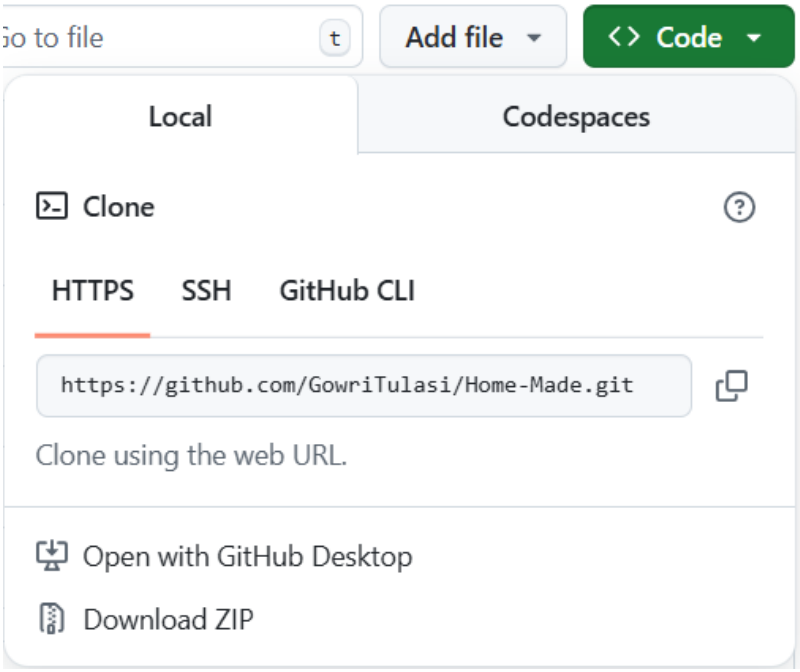

Search [Alt+S]
Global
rsoaccount-new/67fce6a7985d0fc36953d566 @ rsosandboxnew73
IAM > Roles > Create role
"ec2.amazonaws.com"
]
}
}
]
}
Step 2: Add permissions
Edit
Permissions policy summary
Policy name
Type
Attached as
AmazonDynamoDBFullAccess
AWS managed
Permissions policy
AmazonSNSFullAccess
AWS managed
Permissions policy
Step 3: Add tags
Add tags - optional Info
Tags are key-value pairs that you can add to AWS resources to help identify, organize, or search for resources.
No tags associated with the resource.
Add new tag
You can add up to 50 more tags.
Cancel
Previous
Create role



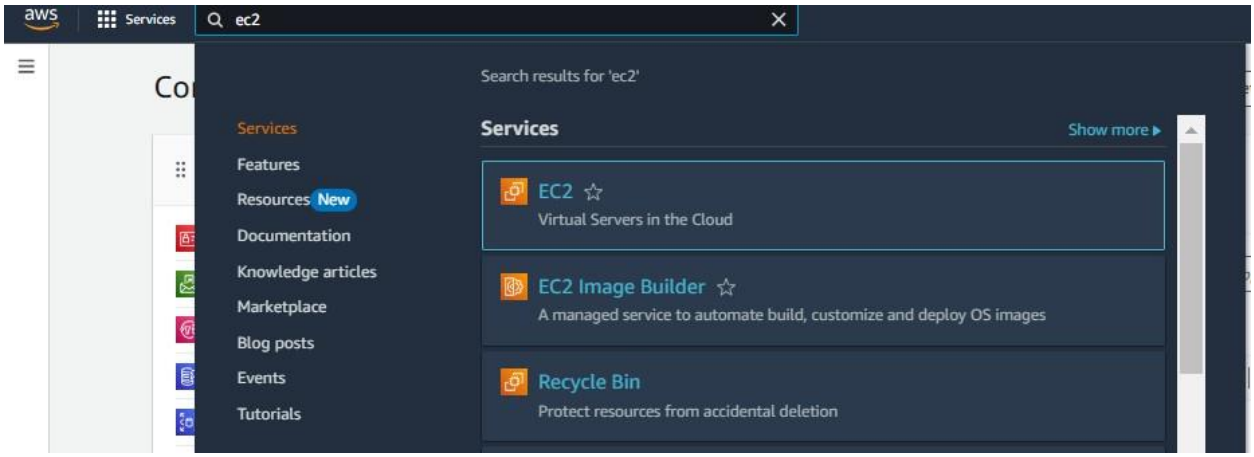
Milestone 6: EC2 Instance Setup

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

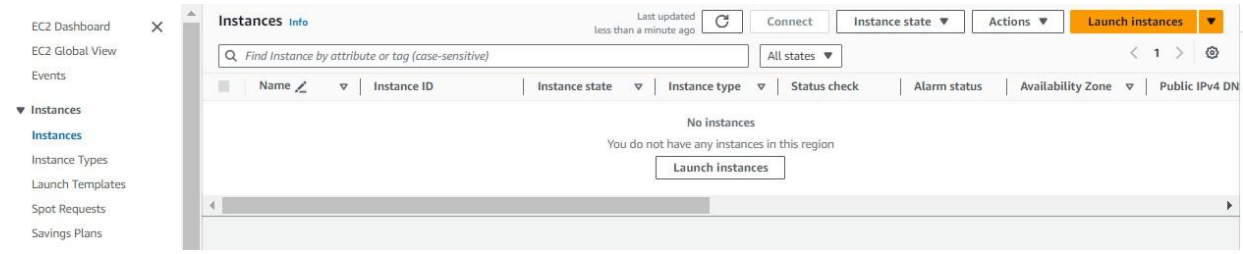


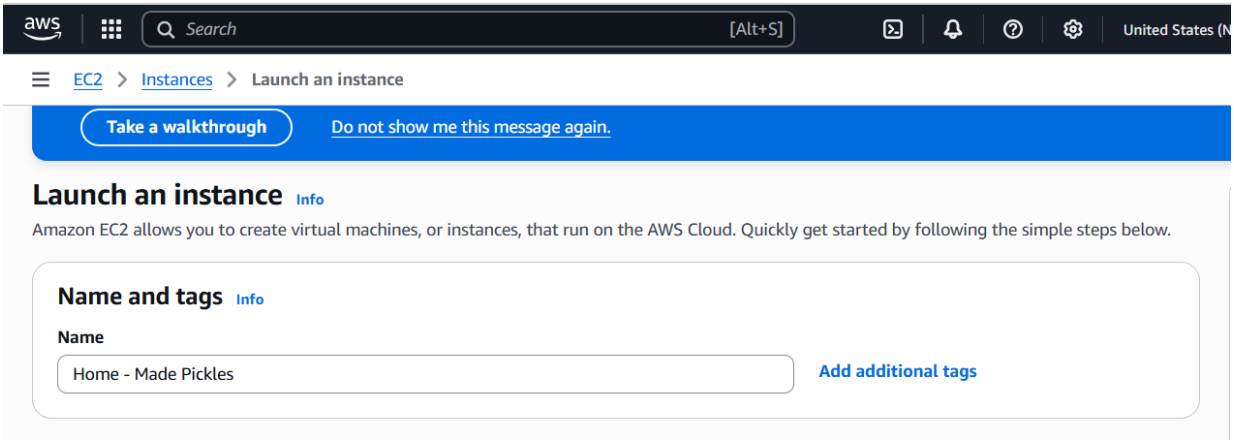


- **Activity 6.1: Launch an EC2 instance to host the Flask application.**
    - **Launch EC2 Instance**
        - In the AWS Console, navigate to EC2 and launch a new instance.

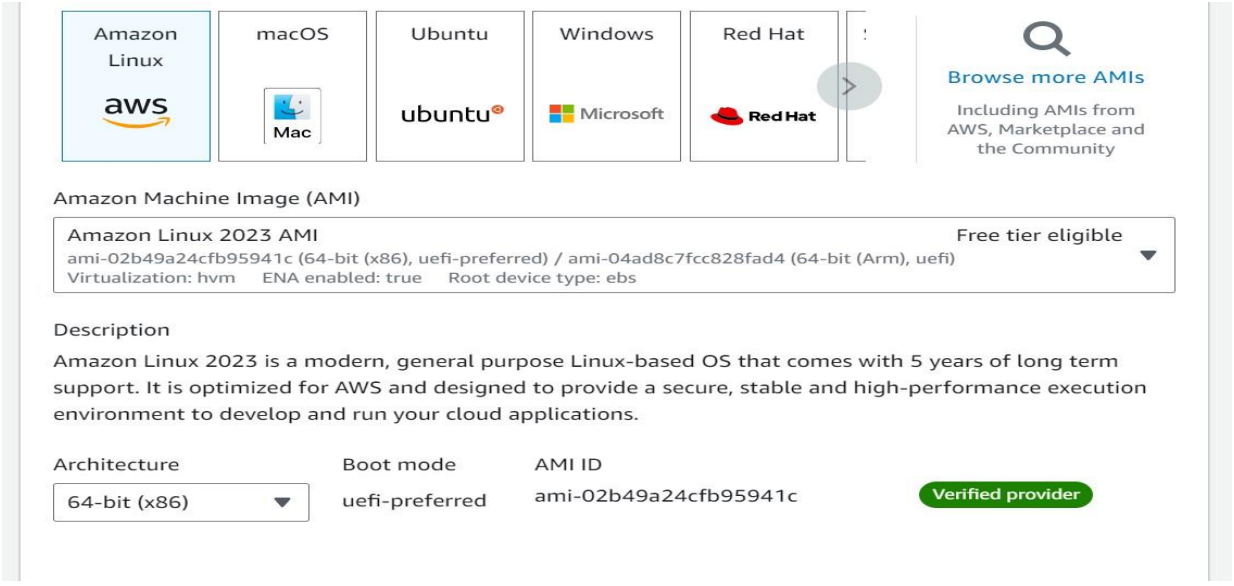


- Click on Launch instance to launch EC2 instance

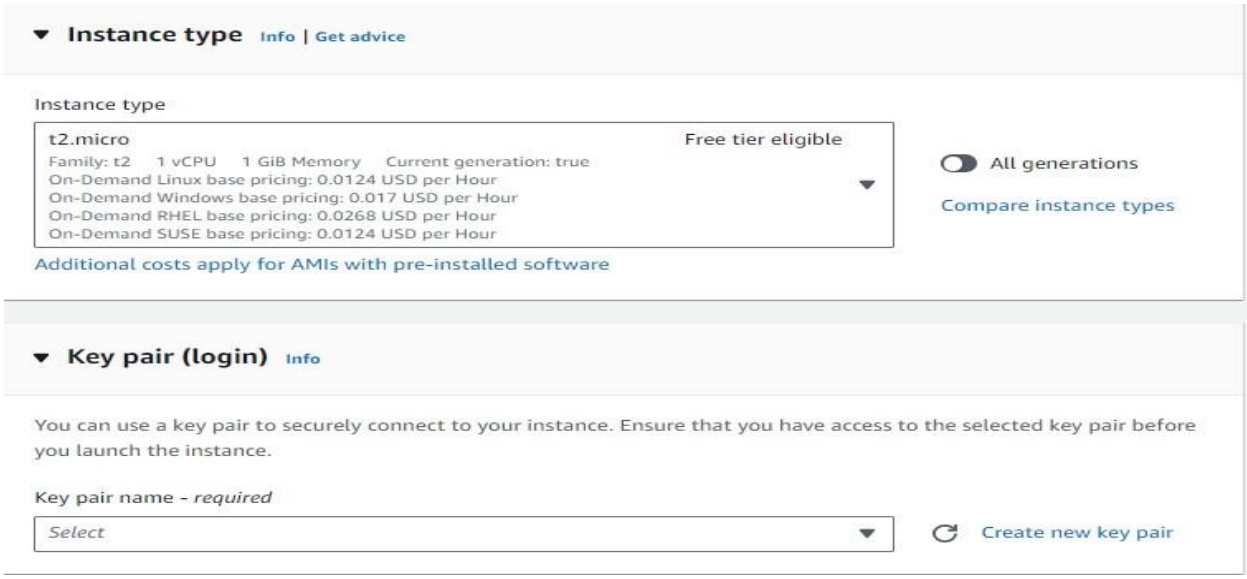


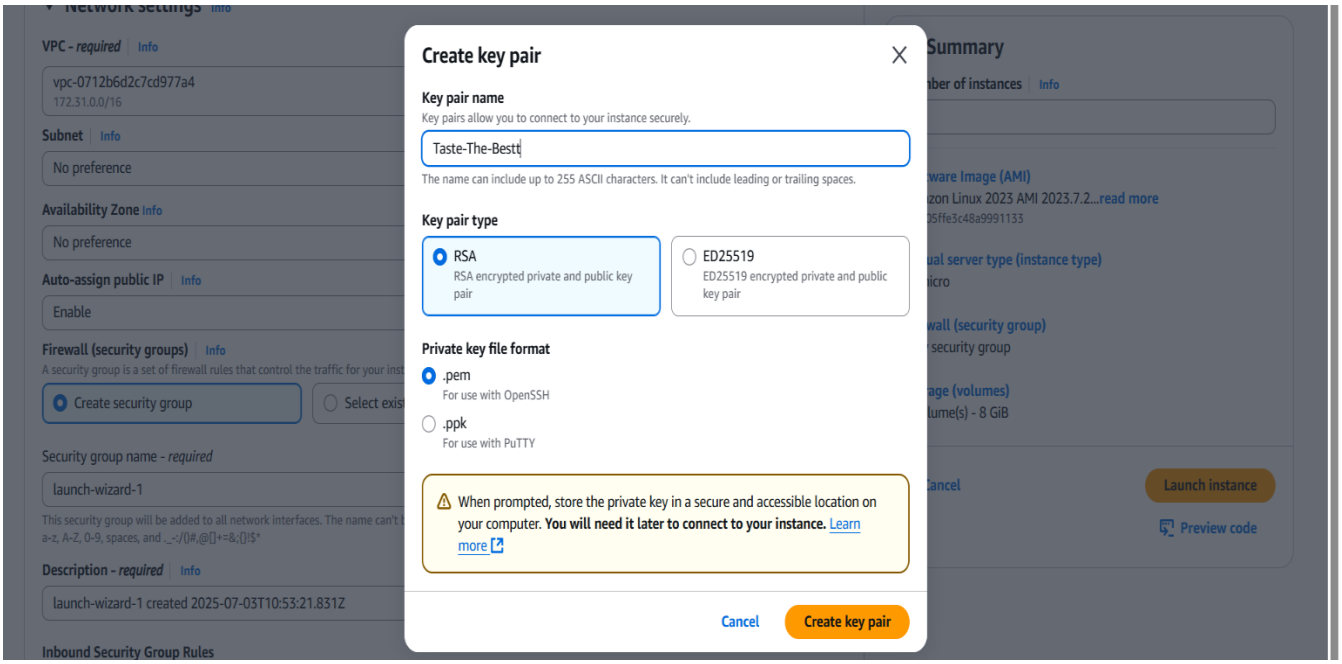


- Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).

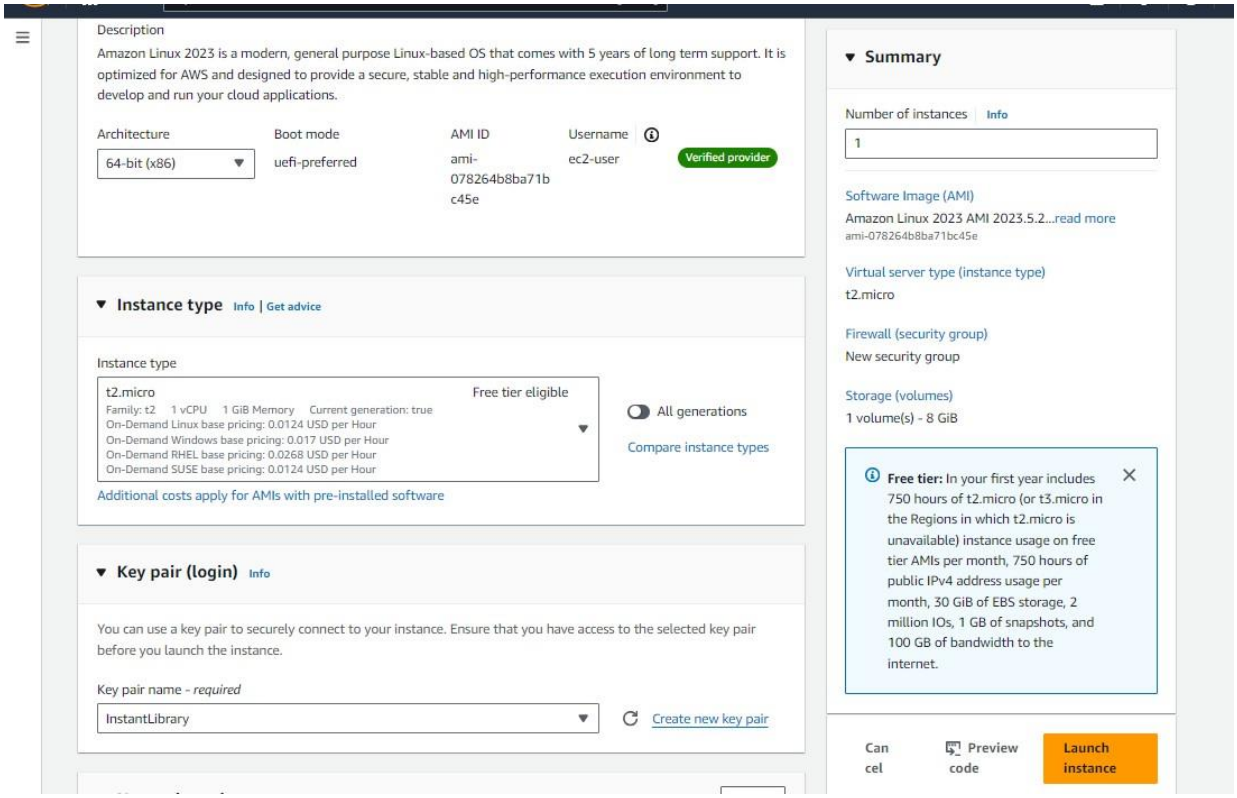


- Create and download the key pair for Server access.

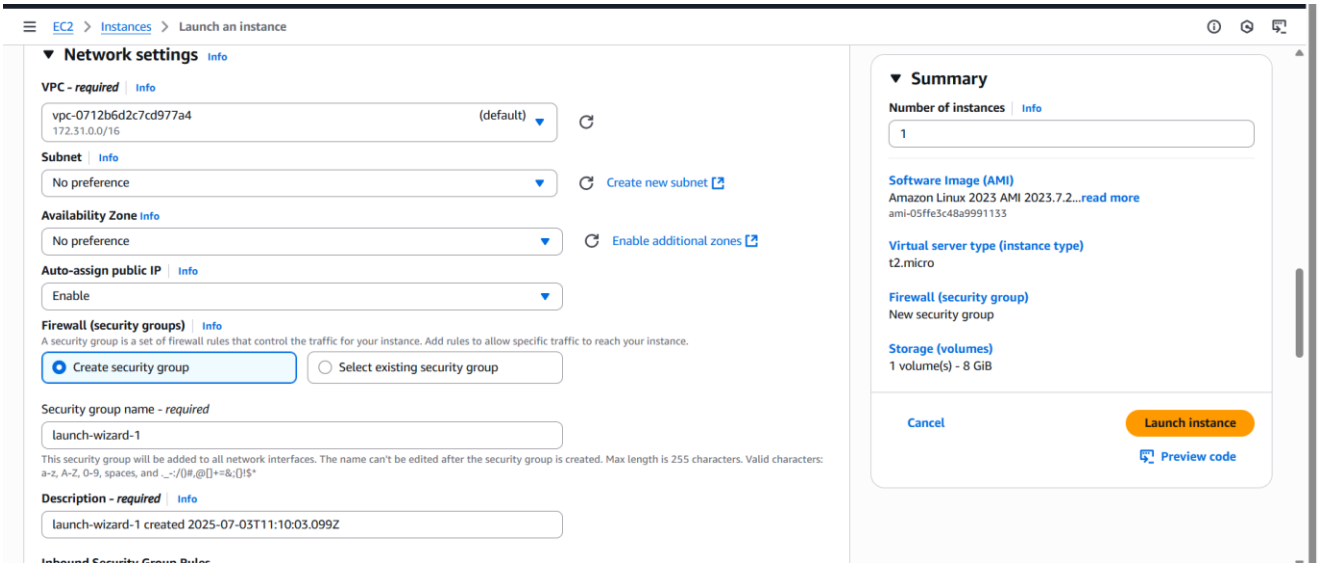


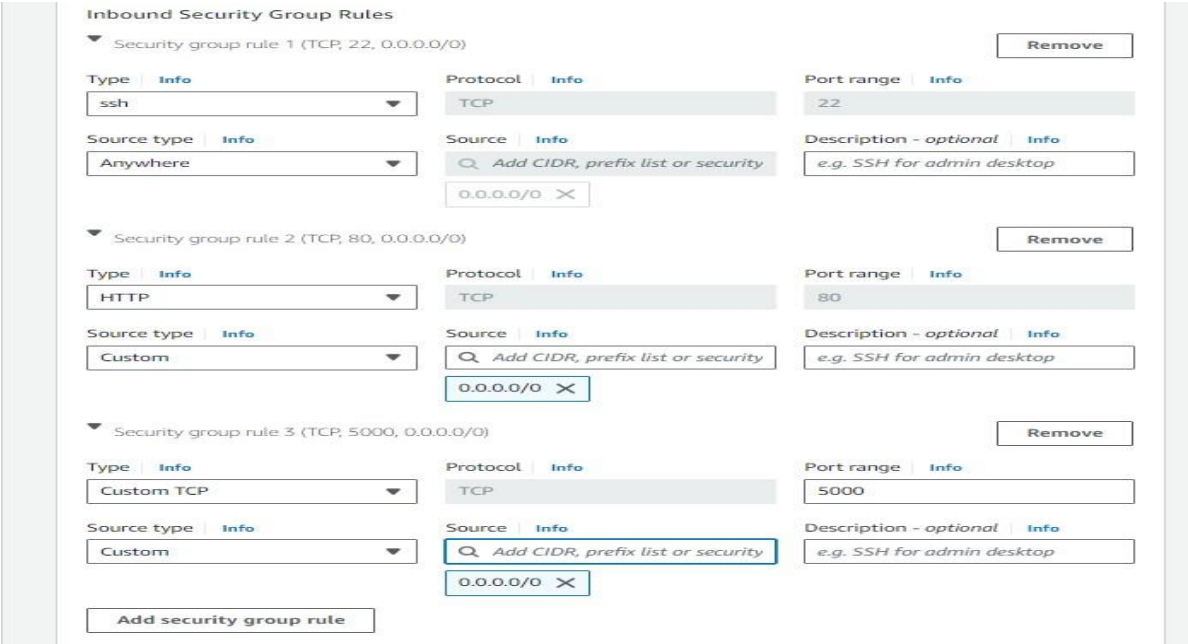


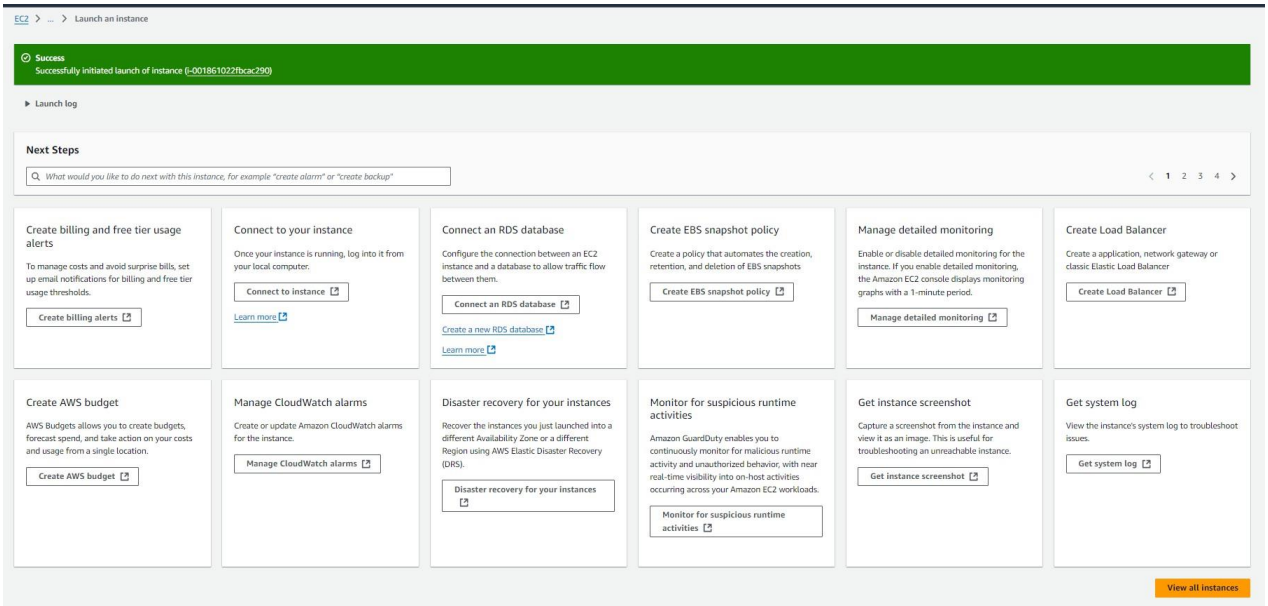
InstantLibrary.pem



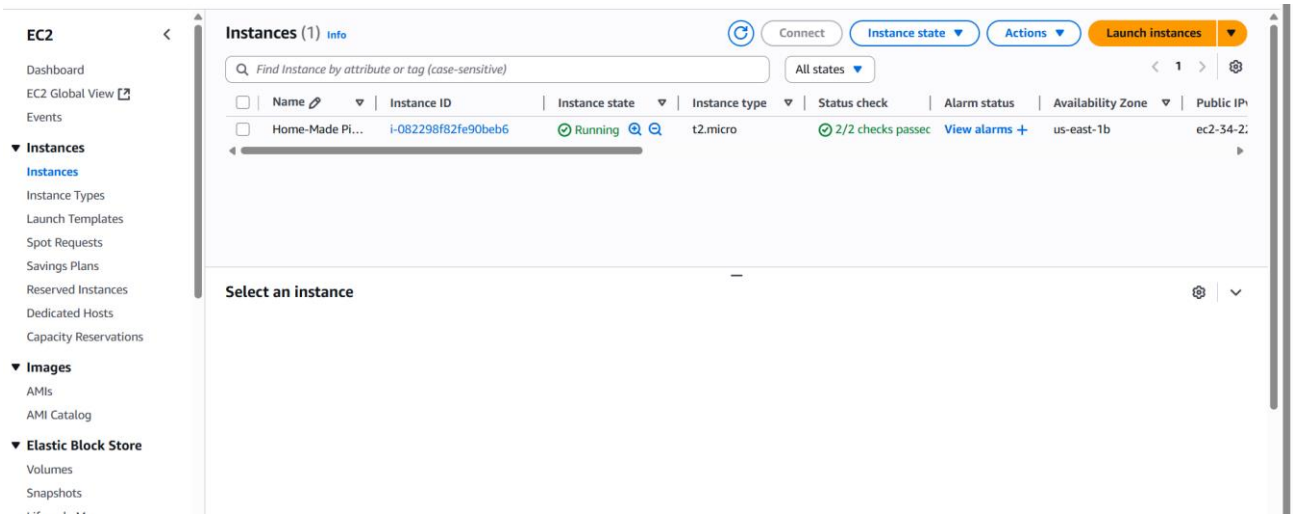
- **Activity 6.2 : Configure security groups for HTTP, and SSH access.**

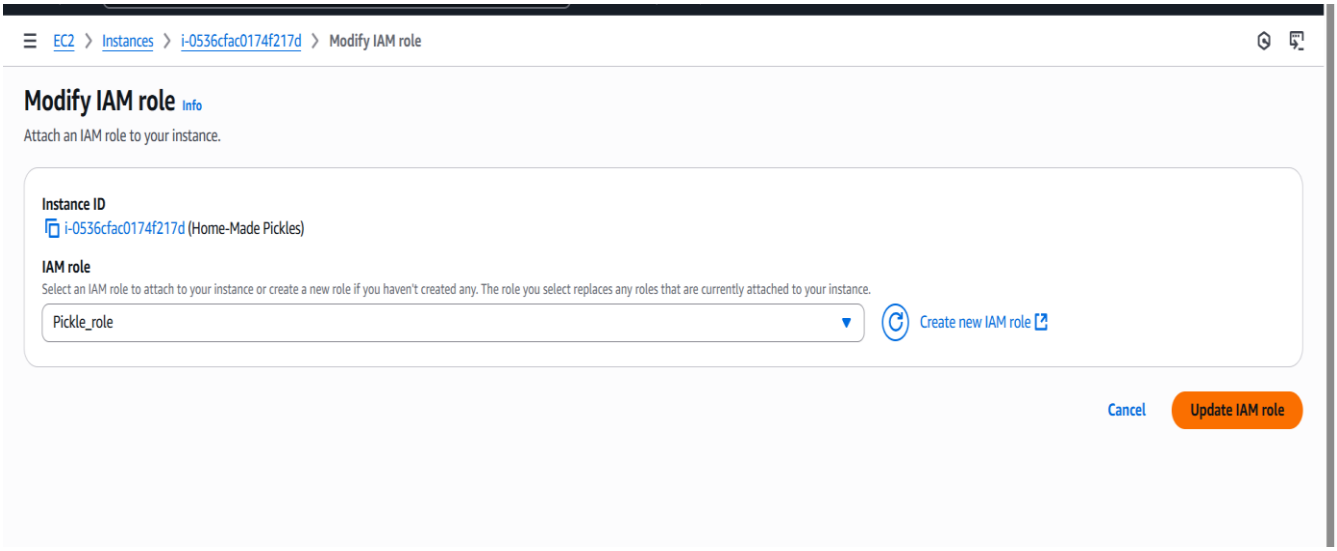


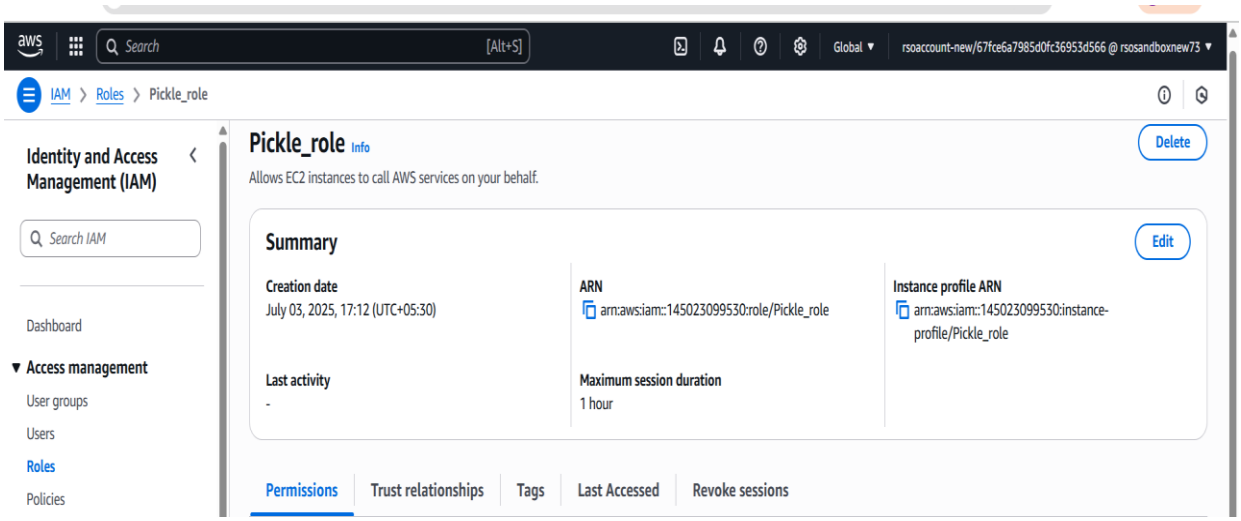




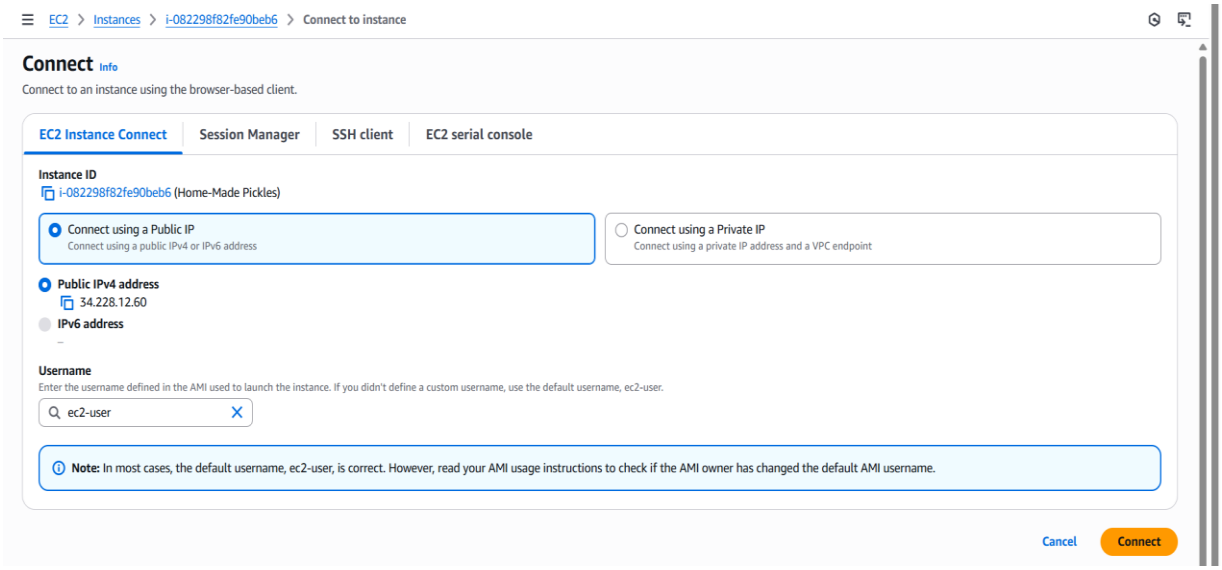
- To connect to EC2 using **EC2 Instance Connect**, start by ensuring that an **IAM role** is attached to your EC2 instance. You can do this by selecting your instance, clicking on **Actions**, then navigating to **Security** and selecting **Modify IAM Role** to attach the appropriate role. After the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



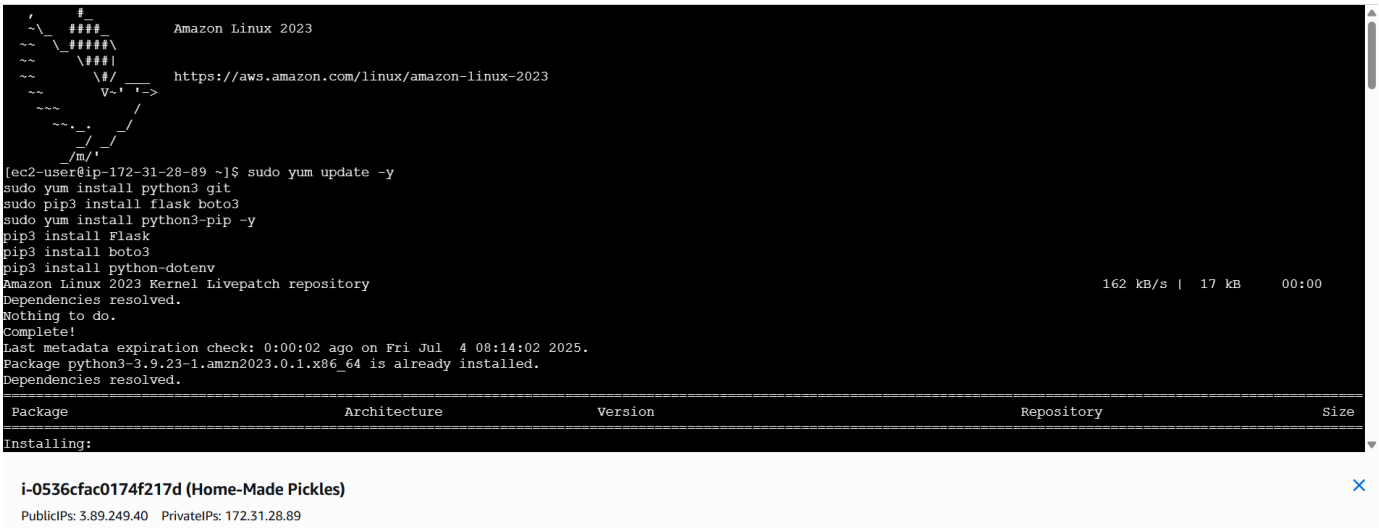




- Now connect the EC2 with the files



- Enter all the commands and Run



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/GowriTulasi/Home-Made.git

This will download your project to the EC2 instance.

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

```
cd Home-Made
```

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

Run the Flask Application

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

Verify the Flask app is running:

http://your-ec2-public-ip

- Run the Flask app on the EC2 instance

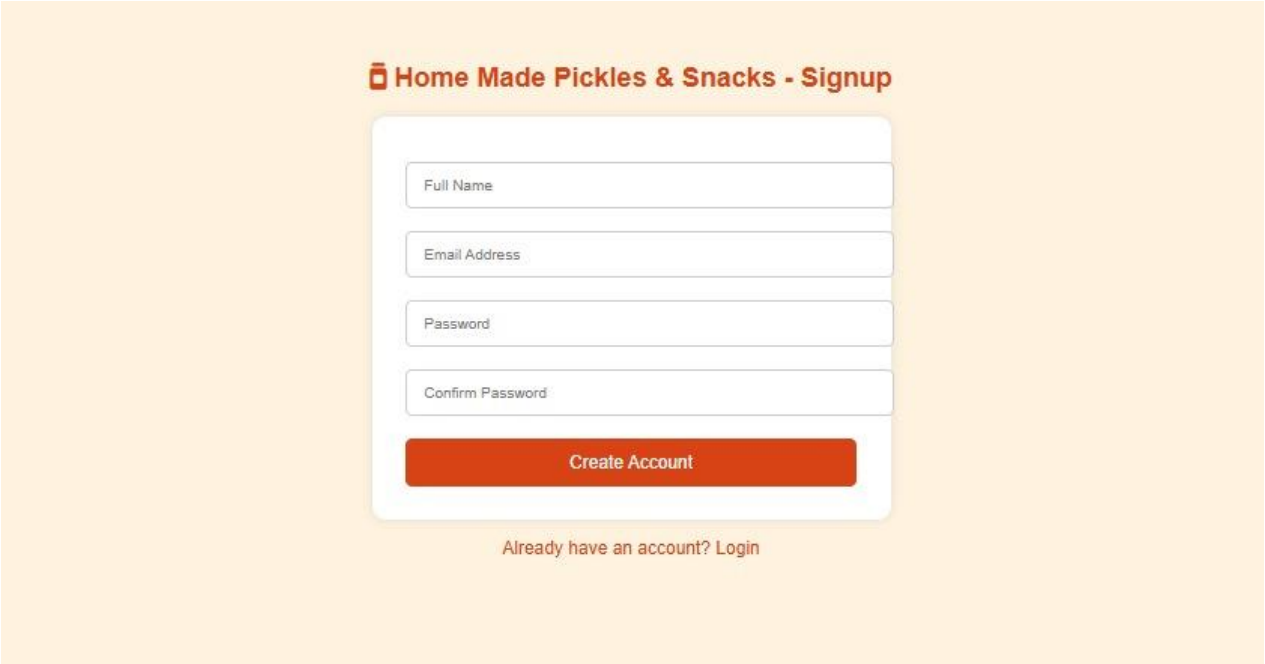
Access the website through:

PUBLIC IP :https://3.89.249.40:5000

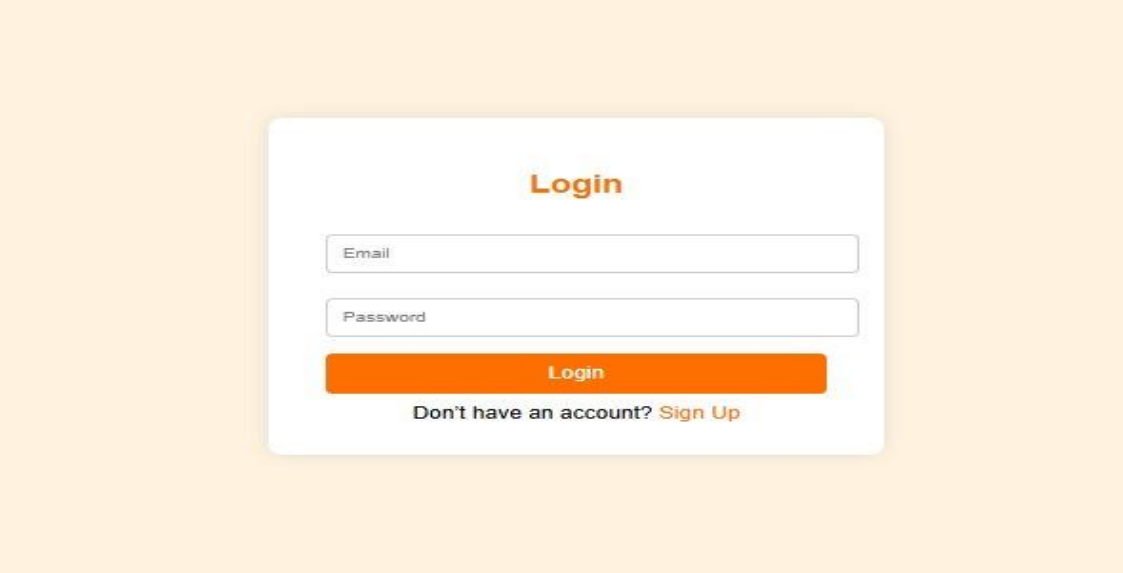
Milestone 8: Testing and Deployment

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

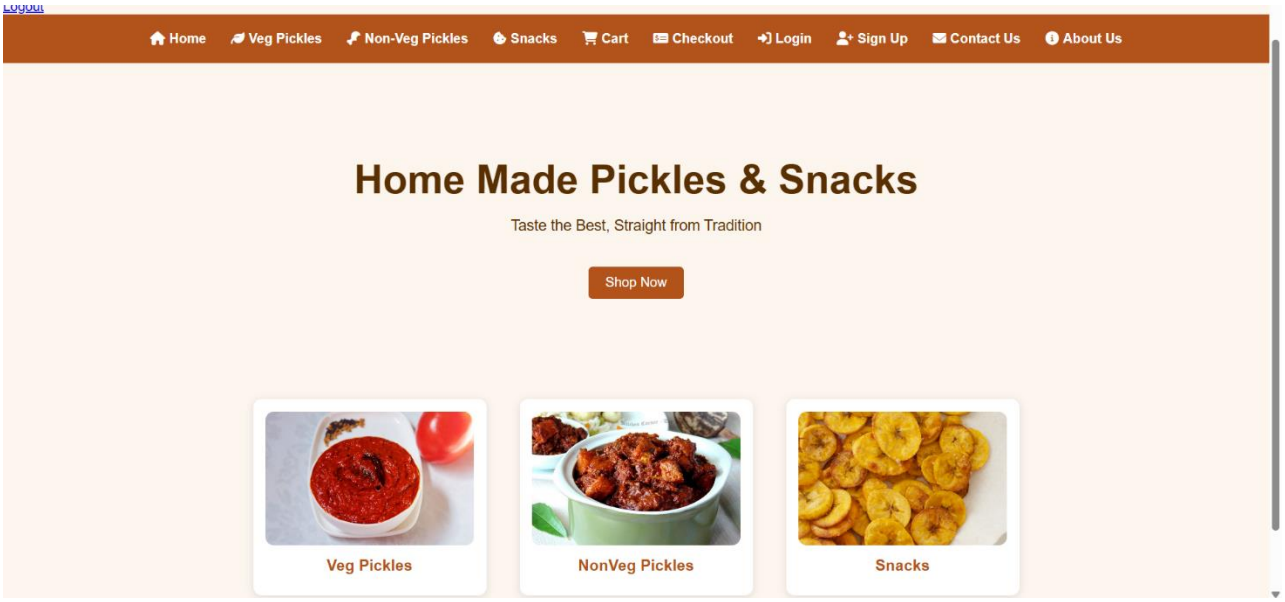
Sign Up page



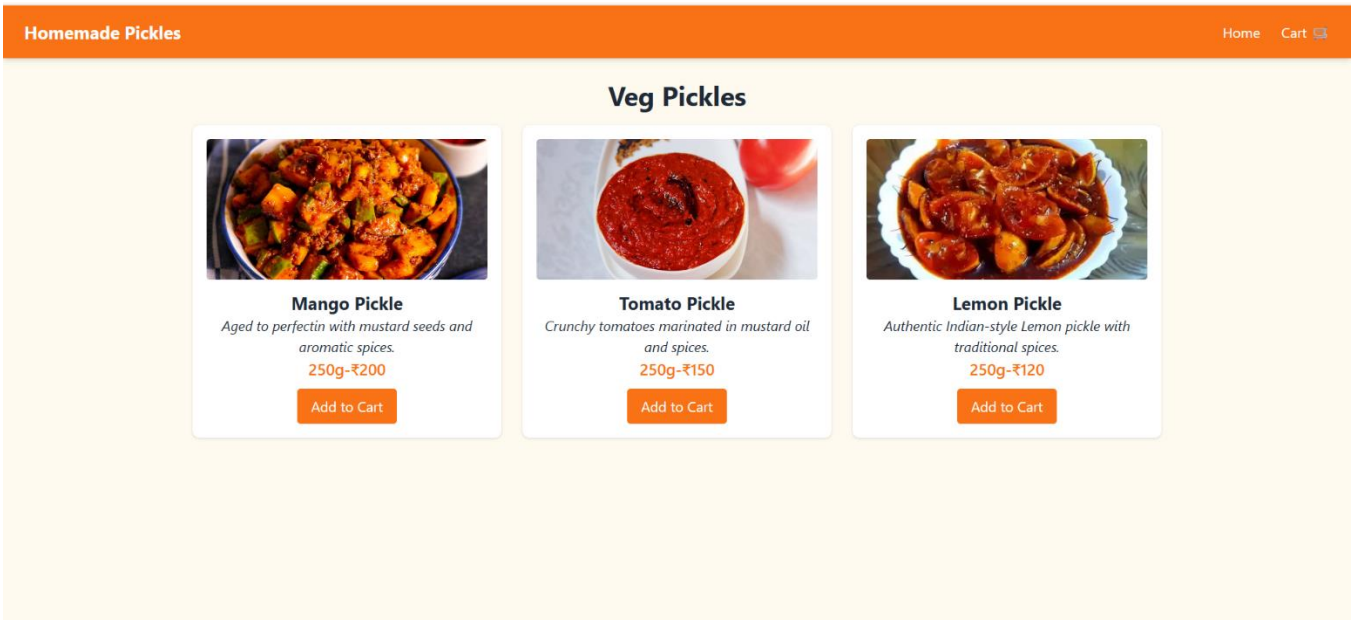
Login Page:



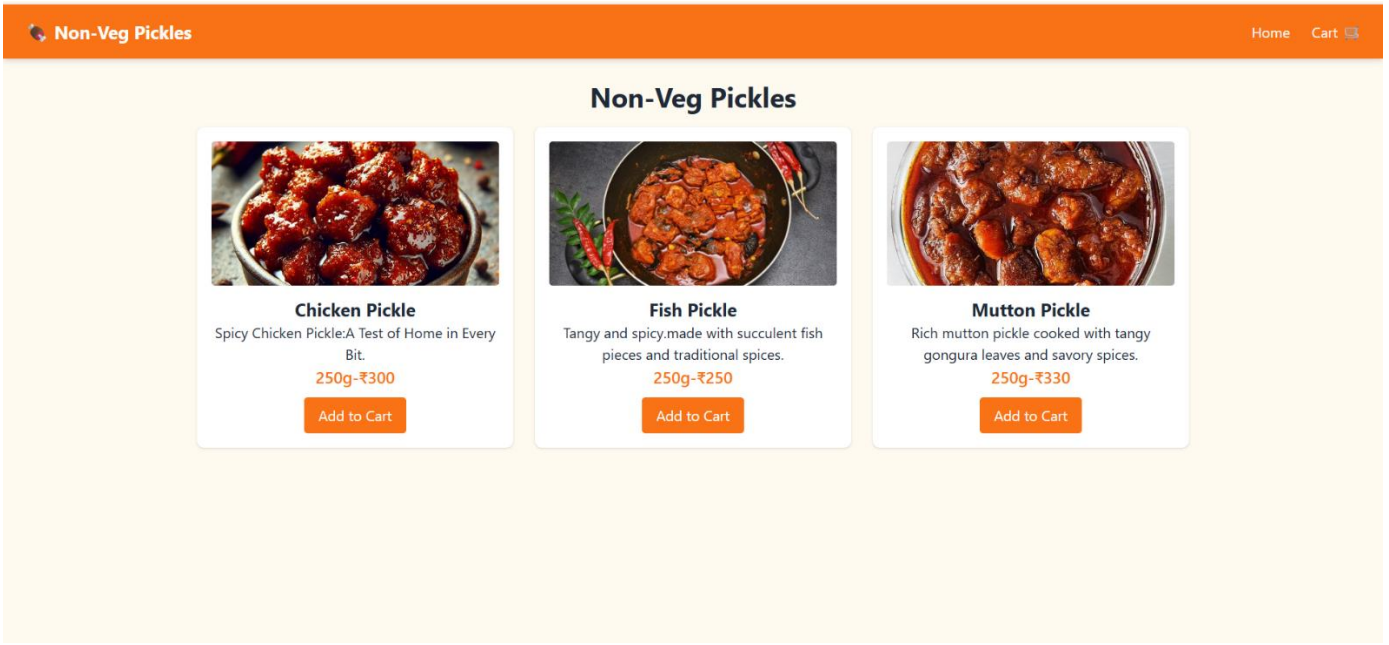
Home Page



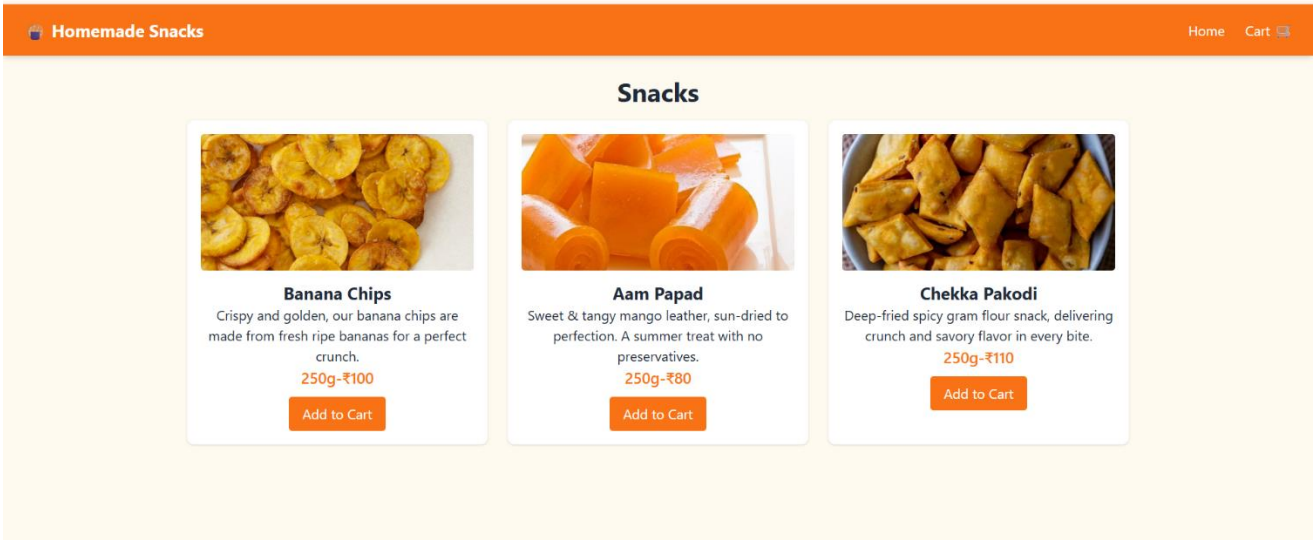
Veg Pickles



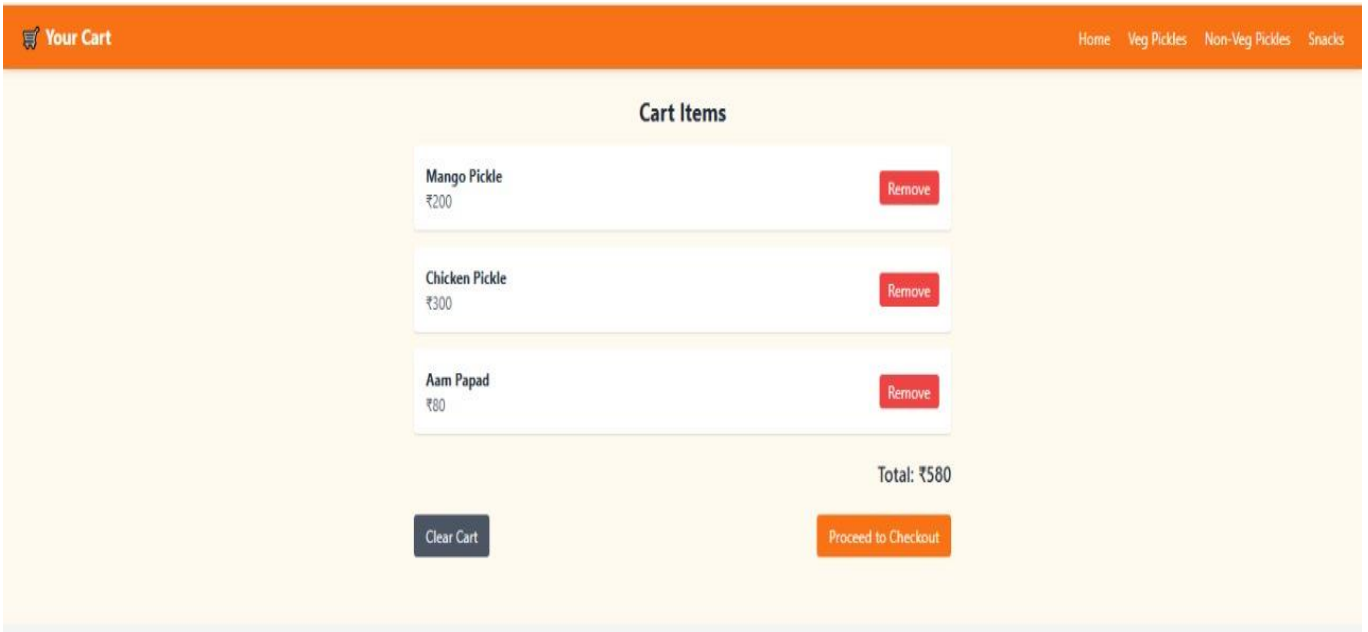
Non Veg Pickles



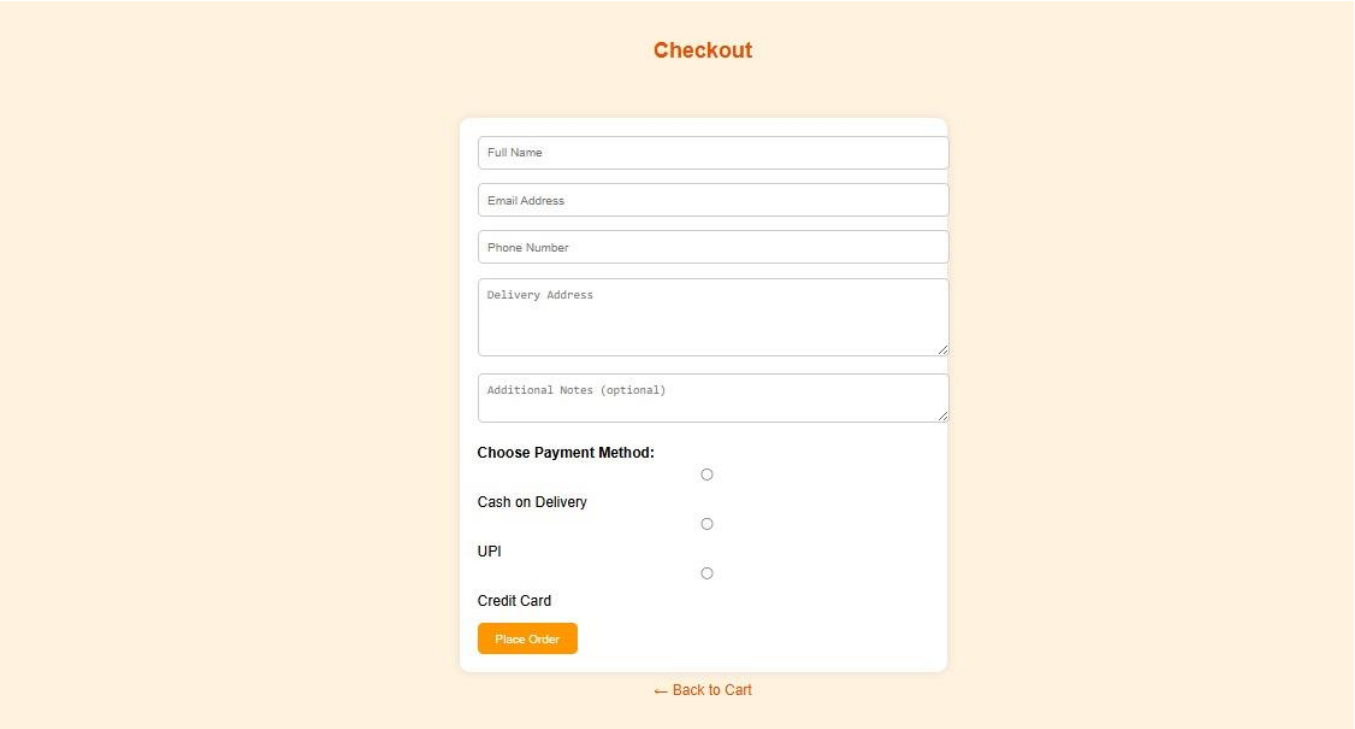
Snacks Page



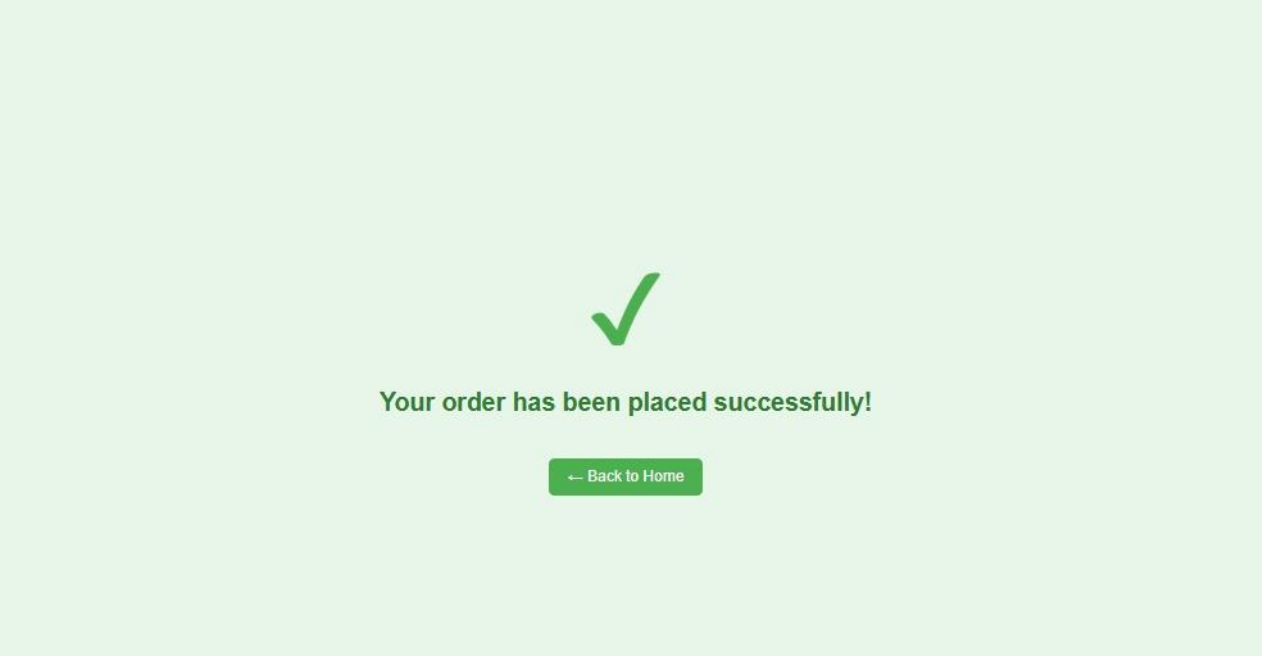
Cart Page



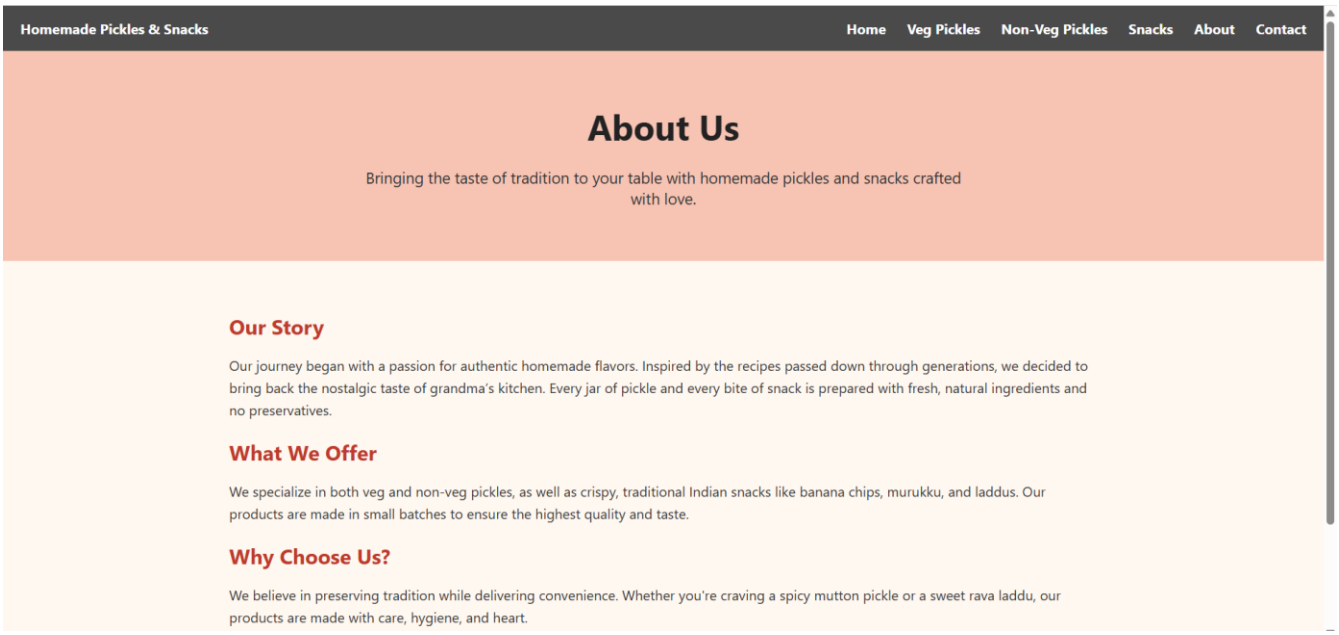
Check out Page



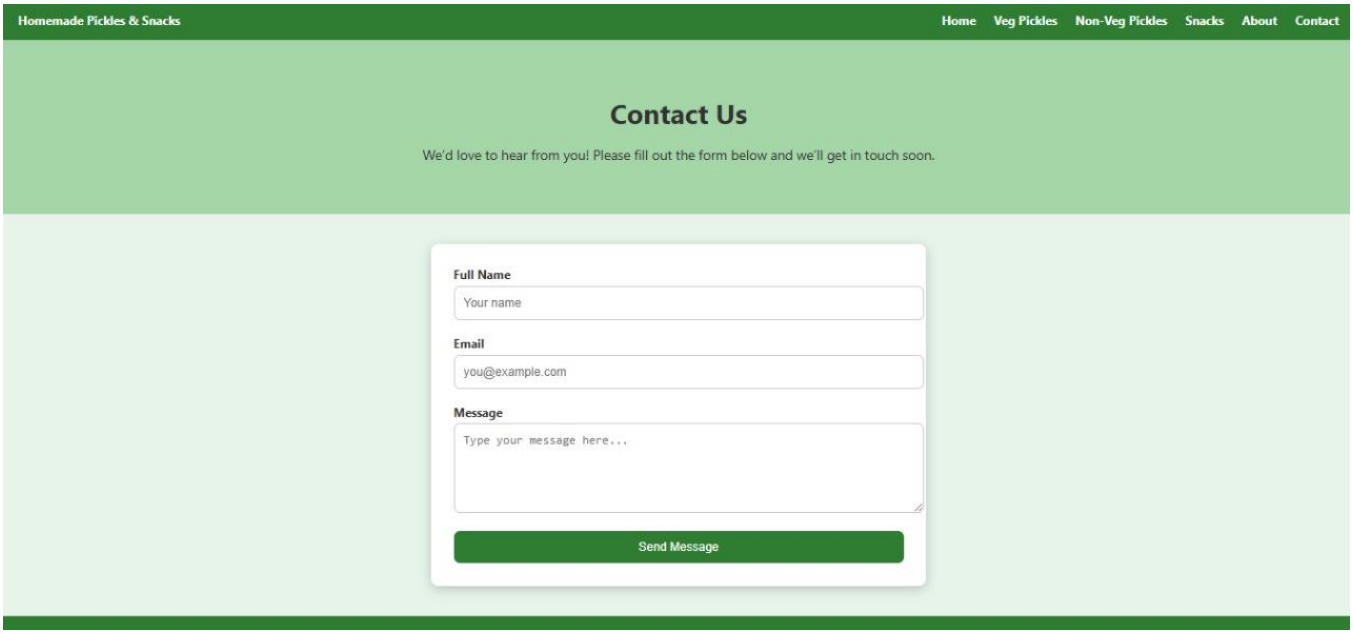
Success page



About us page



Contact Us



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.