Be Practical Tech Solutions
Top Software Training Institute in Bangalore

BE PRACTICAL TECH SOLUTION

BENGALURU

Project Report

This is to state that **Mr. Madhu C M** has successfully completed the **AWS Cloud Project** titled **"Small Book Store"**, The project involved the practical implementation of various AWS services, demonstrating a sound understanding of cloud architecture, deployment strategies, and best practices in cloud-based application development."

Mr. Hemanth
Project Guide

Be Practical Tech Solution
Software Training Institute in Bangalore

INDEX

CHAPTER 01: INTRODUCTION

Introduction:

In today's digital era, online shopping has become a vital part of modern commerce, with users demanding seamless, fast, and secure access to products and services. A full-stack **Book Store website** addresses this need by offering an efficient, user-friendly platform where customers can browse, search, and purchase books online. This system not only simplifies the book purchasing process but also enhances customer experience with personalized features and secure payment integration.

The Book Store application is built using modern web technologies, with a **React.js frontend** and a **Flask backend**, and is fully deployed on **Amazon Web Services (AWS)** to ensure scalability, reliability, and high availability. Utilizing AWS services such as **EC2, S3, RDS, Application Load Balancer, and VPC**, the deployment ensures a production-grade environment capable of handling real-world traffic.

This introduction explores the key components of the Book Store system, the importance of cloud deployment, and how leveraging AWS enables a secure, scalable, and efficient platform for both customers and administrators. From dynamic book listings and user authentication to cart management and database storage, this system demonstrates a complete end-to-end solution for managing an online bookstore.

1.1 Objectives:

The objectives of the Book Store website project include:

- **Efficient Book Management:** Allow admins to add, update, or remove book listings including title, author, price, genre, and description.
- **User Registration & Authentication:** Enable secure sign-up and login features with user role management.
- **Seamless User Experience:** Provide a clean and responsive frontend built using React.js for easy navigation, search, and checkout.
- **Secure Backend Operations:** Handle server-side logic using Flask, including database queries, session handling, and API routing.
- **Cloud Deployment:** Utilize AWS infrastructure to host the frontend, backend, and database with services like EC2, S3, RDS, and Load Balancers.
- **High Availability and Scalability:** Ensure the application remains responsive and resilient under varying loads through AWS Auto Scaling and Load Balancing

1.2 Purpose, Scope, and Applicability

1.2.1 Purpose:

The purpose of this project is to design, develop, and deploy a robust full-stack Book Store web application that simplifies the book purchasing process for users while providing administrators with the tools needed to manage the platform effectively. The deployment on AWS ensures a scalable and reliable production environment, reducing the complexity of infrastructure management

1.2.2 Scope:

The scope of this Book Store system includes:

- **Frontend:** Developed in React.js for an interactive and responsive UI.
- **Backend:** Developed in Python Flask, responsible for business logic and database interaction.
- **Database:** Managed using Amazon RDS (POSTGRESQL), storing user data, book details,
- **Deployment:** Hosted on AWS using EC2 for compute, RDS for database, and Load Balancer for traffic distribution.
- **Security and Networking:** Implemented via AWS IAM, VPC, Security Groups, and HTTPS for secure communication.

The system supports both customer and admin roles with dedicated functionalities.

1.2.3 Applicability:

This Book Store system is applicable to:

- **Small and Medium Businesses** seeking to launch a digital storefront for book sales.
- **Educational Institutions** offering books and materials online.
- **Startup Developers and Students** wanting to learn full-stack development with real cloud deployment.
- **E-Commerce Enthusiasts** experimenting with scalable cloud solutions for online selling.

The modular design and cloud-native deployment make it easily adaptable for other product categories as well.

1.3 Achievements:

The major achievements of this project include:

- **Successful full-stack development** of a Book Store with integrated frontend, backend, and database systems.
- **Deployment on AWS** using services like EC2, RDS, and Load Balancer, ensuring high availability and performance.
- **Implementation of secure user authentication**
- **Use of version control (Git/GitHub)** for code management and collaboration.
- **Real-time scalability and fault tolerance** through AWS cloud architecture.

These achievements highlight not just the technical aspects of full-stack web development but also cloud deployment best practices.

CHAPTER 02: REQUIRMENTS AND ANALYSIS

2.1 Problem Definition:

The problem definition for the Simple Bookstore Application focuses on addressing the challenges of managing a scalable and reliable online bookstore. Traditional monolithic or local deployment models lack flexibility, scalability, and high availability, which limits the user experience and system performance under load. This project aims to build a cloud-based bookstore system to improve application scalability, ensure high availability, enable secure data handling, and enhance system observability using AWS services. By leveraging AWS components such as EC2, VPC, RDS (PostgreSQL), ALB, and monitoring services, the system ensures seamless performance, cost optimization, and resilience. Additionally, it provides a practical understanding of cloud architecture and deployment processes.

2.2Requirement Specification:

This System Requirement Specification (SRS) provides a detailed overview of the Simple Bookstore system, focusing on its functions, behavior, and architecture. The application is designed for users to browse, search, and purchase books online through a clean and responsive interface

The system is developed using React for the frontend and Flask for the backend, with a PostgreSQL database hosted on AWS RDS. The deployment utilizes key AWS services to simulate a production-grade cloud infrastructure. The goal is not only to deliver a functional bookstore application but also to enhance understanding of **cloud computing** concepts, infrastructure design, and deployment practices using AWS.

Key Functionalities:

- Book listing and search
- User authentication and login
- Book purchase or order management
- Secure and efficient data storage in PostgreSQL
- Cloud-based deployment with high availability and monitoring

2.3 System Requirements:

Hardware and Software Requirement :

Hardware Requirement :

2.3.1 PC :- Laptop or Desktop

2.3.2 Ram :- 8GB(min)

2.3.3 Storage :- 256GB(min)

2.3.4 Processor :- i3/i5/ryzen3/ryzen5, etc..

2.3.5 Operating System :- Windows 10 or 11

Software Requirement :

Frontend:

2.3.6 React JS
2.3.7 HTML
2.3.8 CSS
2.3.9 JavaScript

Backend:

2.3.10 PYTHON (flask Framework)

DATABASE:

2.3.11 PostgreSQL (Hosted on AWS RDS)

PLATFORM and CLOUD SERVICES:

2.3.12 **Amazon EC2:** For hosting backend Flask server and possibly frontend

2.3.13 **Amazon RDS (PostgreSQL):** For persistent and scalable database storage

2.3.14 **Amazon VPC:** For secure and isolated networking of AWS resources

2.3.15 **Application Load Balancer (ALB):** For distributing traffic across EC2 instances

2.3.16 **CloudWatch:** For monitoring logs and system performance

2.3.17 **IAM:** For secure access management

2.4 Cloud Service i Used:

1. Amazon EC2 (Elastic Compute Cloud):

Amazon EC2 provides resizable virtual servers (instances) in the cloud to host applications. In this project, EC2 is used to deploy the Flask backend and optionally the React frontend. It allows you to choose the instance type, configure security groups, and manage the application server environment. EC2 provides full control over the operating system and installed software. It supports auto scaling and load balancing for high availability. The flexibility and control offered by EC2 make it ideal for deploying and managing custom applications. Instances can be monitored and managed through the AWS Console or CLI.

2. Amazon VPC (Virtual Private Cloud):

Amazon VPC allows you to create an isolated network environment in the cloud. In this project, a custom VPC is created to securely host EC2 instances and the RDS database. Subnets, route tables, internet gateways, and NAT gateways are configured to manage traffic and provide access control. VPC ensures that your bookstore application components are securely segmented and protected from public exposure. Security groups and Network ACLs within the VPC offer fine-grained access control. Using a VPC mimics traditional on-premises networking while offering cloud scalability. It plays a key role in ensuring network-level security and isolation.

3. Amazon RDS (Relational Database Service) – PostgreSQL:

Amazon RDS simplifies the setup, operation, and scaling of relational databases in the cloud. For this project, PostgreSQL is used as the backend database to store book records, user information, and order data. RDS automates database administration tasks such as backups, patching, and monitoring. It provides high availability using Multi-AZ deployments and automatic failover. Performance monitoring and scaling are easy through the AWS console. By offloading database management, RDS allows the developer to focus on application logic. It also supports secure access via VPC and IAM roles.

4. Application Load Balancer (ALB):

The Application Load Balancer is used to distribute incoming traffic across multiple EC2 instances. It ensures that the bookstore application remains available and responsive under varying loads. ALB operates at the application layer (HTTP/HTTPS) and supports advanced routing features, such as path-based and host-based routing. It helps achieve fault tolerance by rerouting traffic if one instance fails. SSL termination at ALB enhances application security. Integration with EC2 Auto Scaling allows seamless scaling based on demand. ALB is crucial for maintaining high availability and user experience.

6. IAM (Identity and Access Management):

AWS IAM allows secure control of access to AWS services and resources. For this project, IAM is used to define roles and policies that grant least-privilege access to EC2, RDS, and other services. IAM ensures that only authorized users or applications can interact with critical infrastructure components. It supports multi-factor authentication (MFA) and granular permissions. IAM roles can be attached to EC2 instances to securely access AWS resources without storing credentials. Managing access through IAM is essential for maintaining security in a cloud environment. It also supports audit and compliance with detailed access logs.

CHAPTER 03: SYSTEM DESIGN

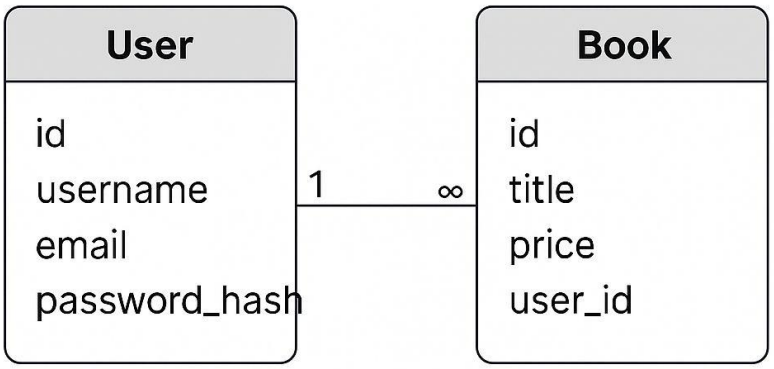
3.1 ER Diagram:

An ER (Entity-Relationship) diagram is a graphical representation used in database design to illustrate the relationships between entities in a system. It helps in organizing and defining the structure of a database.

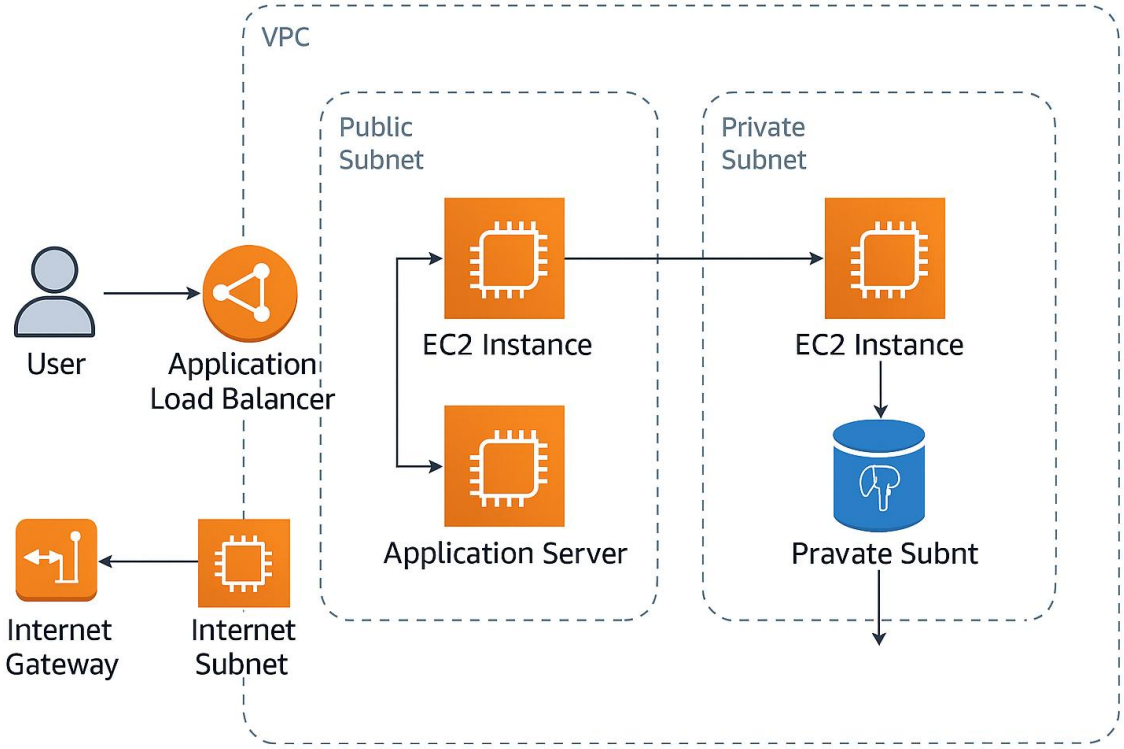
In my current project, a **simple Book Store system**, the ER diagram captures the core entities and their relationships. The main entities include:

- **User**: Represents individuals who can register and log in to the system.
- **Book**: Contains information about available books, including the name and price.

The ER diagram defines how users interact with the system and how books are stored and managed. It ensures that the database is structured efficiently to support user authentication and book-related operations.



3.2 Architecture Flow:



This is a **Three-Tier Architecture** implemented using core AWS services to securely deploy a Book Store web application. It separates concerns into **Presentation**, **Application**, and **Data** tiers for better scalability, isolation, and management.

🔷 **1. Presentation Tier (Public Subnet)**

- **User** accesses the application through a browser.
- Request goes through the **Internet Gateway** into the VPC.
- **Application Load Balancer (ALB)** receives the request and distributes it to EC2 instances in the **public subnet** (serving as the frontend or web server).
- These EC2 instances host the static frontend (HTML, CSS, JavaScript).

🔷 **2. Application Tier (Private Subnet)**

- Frontend EC2 communicates with **Application EC2 instances** located in the **private subnet**.
- This backend is built using a **Flask API** that handles:
  - User login/registration
  - Book addition logic
  - Fetching book data for the homepage
- These EC2 instances are not publicly accessible and only allow internal communication from the frontend layer.

🔷 **3. Data Tier (Private Subnet with RDS)**

- The application layer interacts with **Amazon RDS (MySQL/PostgreSQL)** in a separate **private subnet**.
- RDS securely stores:
  - User login credentials
  - Book details (name, price)
- RDS is **not exposed to the internet**, only accessible from backend EC2 instances within the private

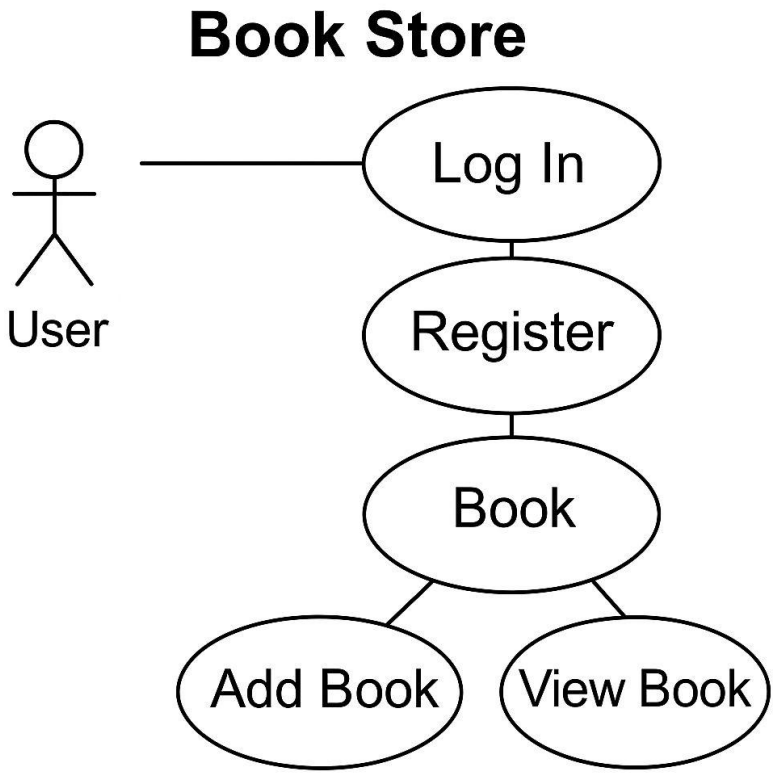
subnet.

Security & Network Controls

- **VPC** provides full network isolation.
- **Security Groups** restrict access:
  - Frontend EC2 only accessible via ALB
  - Backend EC2 only accessible from frontend EC2
  - RDS only accessible from backend EC2
- Optionally, a **NAT Gateway** can be added to allow backend EC2s to access the internet for updates without exposing them publicly.

3.3 Use Case Diagram:

Use case diagram is a visual representation in Unified Modeling Language (UML) used to show the functional requirements of a system. It highlights the interactions between "actors" (which can be users or other systems) and "use cases" (which represent the system's functionalities or services).



CHAPTER 04: IMPLEMENTATION AND TESTING

4.1 Implementation Approaches:

Implementing a BookStore application involves various approaches to address different aspects of the system, such as user interface, database management, and AWS service integration. Here are some common implementation approaches:

1. Centralized System (with AWS)
    - Single Database: All data related to book information, user details, and transactions are stored in a centralized database (e.g., Amazon RDS).
    - Single Server: All system operations are handled by a central server (e.g., an EC2 instance).
        - Pros: Simplified data management, centralized control, easier maintenance.
        - Cons: Limited scalability, potential single point of failure

1. Cloud Based System (AWS):

Cloud Infrastructure: Uses AWS services (e.g., EC2 for hosting, RDS for database management, S3 for storing book images) to scale the application and manage data.

- **Pros**: Scalability, flexibility, reduced hardware costs, and access from anywhere.
- **Cons**: Dependency on internet connectivity, potential concerns about data privacy.
- **Elastic Load Balancing**: Distributes traffic to multiple EC2 instances to ensure high availability and scalability.
- **Auto Scaling**: Adjusts the number of EC2 instances to match traffic demands, optimizing costs and performance.

2. Web-Based System:

Web Application: Provides users with access via a browser, enabling them to log in, add books, and view books on the homepage.

- **Features**: User authentication, book addition (name, price), homepage display of added books.
- **Pros**: Accessible from any device with a browser, easy to update and maintain.
- **Cons**: Requires internet access, may need optimization for mobile devices.

3. Mobile-Based System:

Mobile Compatibility: While the core system is web-based, it can be optimized for mobile access by ensuring the frontend is responsive (using frameworks like Bootstrap or media queries).

- **Pros**: Real-time updates, convenience for mobile users.
- **Cons**: Mobile optimization is required for better user experience on smaller screens.

.

4.2Coding details and code efficiency:

Frontend

a. Login.js

```
import React, { useState, useEffect } from 'react';
import { Link, useNavigate } from 'react-router-dom';
import { useAuth } from '../context/AuthContext';

function Login() {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');
  const [loading, setLoading] = useState(false);

  const { login, isAuthenticated } = useAuth();
  const navigate = useNavigate();

  // Redirect if already logged in
  useEffect(() => {
    if (isAuthenticated) {
      navigate('/');
    }
  }, [isAuthenticated, navigate]);

  const handleSubmit = async (e) => {
    e.preventDefault();
    setError('');
    setLoading(true);

    if (!username || !password) {
      setError('Username and password are required');
      setLoading(false);
```

```
    return;
  }

  try {
    console.log('Attempting login with:', { username });
    const success = await login(username, password);
    if (success) {
      navigate('/');
    } else {
      setError('Invalid username or password');
    }
  } catch (err) {
    console.error('Login error:', err);
    setError('Network error. Please check if the backend server is running.');
  } finally {
    setLoading(false);
  }
};

return (
  <div className="row justify-content-center">
    <div className="col-md-6 col-lg-4">
      <div className="card shadow">
        <div className="card-body p-4">
          <h2 className="text-center mb-4">Login</h2>

          {error && <div className="alert alert-danger">{error}</div>}

          <form onSubmit={handleSubmit}>
            <div className="mb-3">
              <label htmlFor="username" className="form-label">Username</label>
              <input
```

```
    type="text"
    id="username"
    className="form-control"
    value={username}
    onChange={(e) => setUsername(e.target.value)}
    required
   />
  </div>

  <div className="mb-3">
   <label htmlFor="password" className="form-label">Password</label>
   <input
    type="password"
    id="password"
    className="form-control"
    value={password}
    onChange={(e) => setPassword(e.target.value)}
    required
   />
  </div>

  <button
   type="submit"
   className="btn btn-primary w-100 mb-3"
   disabled={loading}
  >
   {loading ? 'Logging in...' : 'Login'}
  </button>

  <div className="text-center">
   <p className="mb-0">
    Don't have an account? <Link to="/register">Register</Link>
```

```
            </p>
           </div>
          </form>
         </div>
        </div>
       </div>
     </div>
  );
}
```

a.

b. Register.js

```
import React, { useState, useEffect } from 'react';
import { Link, useNavigate } from 'react-router-dom';
import { useAuth } from '../context/AuthContext';

function Register() {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [confirmPassword, setConfirmPassword] = useState('');
  const [error, setError] = useState('');
  const [loading, setLoading] = useState(false);

  const { register, isAuthenticated } = useAuth();
  const navigate = useNavigate();

  // Redirect if already logged in
  useEffect(() => {
    if (isAuthenticated) {
      navigate('/');
    }
  }, [isAuthenticated, navigate]);

  const handleSubmit = async (e) => {
    e.preventDefault();
    setError('');

    // Validate passwords match
    if (password !== confirmPassword) {
      setError('Passwords do not match');
      return;
    }

    // Validate password length
    if (password.length < 6) {
      setError('Password must be at least 6 characters');
      return;
```

```
  }

  setLoading(true);

  try {
    const success = await register(username, password);
    if (success) {
      navigate('/login');
    } else {
      setError('Username already exists');
    }
  } catch (err) {
    setError('An error occurred during registration');
  } finally {
    setLoading(false);
  }
};

return (
  <div className="row justify-content-center">
    <div className="col-md-6 col-lg-4">
      <div className="card shadow">
        <div className="card-body p-4">
          <h2 className="text-center mb-4">Register</h2>

          {error && <div className="alert alert-danger">{error}</div>}

          <form onSubmit={handleSubmit}>
            <div className="mb-3">
              <label htmlFor="username" className="form-label">Username</label>
              <input
                type="text"
                id="username"
                className="form-control"
                value={username}
                onChange={(e) => setUsername(e.target.value)}
                required
              />
            </div>

            <div className="mb-3">
              <label htmlFor="password" className="form-label">Password</label>
              <input
                type="password"
                id="password"
                className="form-control"
                value={password}
                onChange={(e) => setPassword(e.target.value)}
                required
              />
            </div>

            <div className="mb-3">
              <label htmlFor="confirmPassword" className="form-label">Confirm Password</label>
              <input
                type="password"
```

```
              id="confirmPassword"
              className="form-control"
              value={confirmPassword}
              onChange={(e) => setConfirmPassword(e.target.value)}
              required
             />
            </div>

            <button
             type="submit"
             className="btn btn-primary w-100 mb-3"
             disabled={loading}
            >
             {loading ? 'Registering...' : 'Register'}
            </button>

            <div className="text-center">
             <p className="mb-0">
              Already have an account? <Link to="/login">Login</Link>
             </p>
            </div>
           </form>
          </div>
         </div>
        </div>
       </div>
  );
}

export default Register;
```

c. App.js

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Navigate } from 'react-router-dom';
import { AuthProvider, useAuth } from './context/AuthContext';

// Components
import Login from './components/Login';
import Register from './components/Register';
import BookList from './components/BookList';
import AddBook from './components/AddBook';
import Navbar from './components/Navbar';

// Protected route wrapper
function RequireAuth({ children }) {
  const { isAuthenticated } = useAuth();
  return isAuthenticated ? children : <Navigate to="/login" />;
}

function App() {
  return (
    <AuthProvider>
     <Router>
      <Navbar />
```

```
      <div className="container py-4">
        <Routes>
          <Route path="/" element={<BookList />} />
          <Route path="/login" element={<Login />} />
          <Route path="/register" element={<Register />} />
          <Route
            path="/add-book"
            element={
              <RequireAuth>
                <AddBook />
              </RequireAuth>
            }
          />
          <Route path="*" element={<Navigate to="/" />} />
        </Routes>
      </div>
    </Router>
  </AuthProvider>
 );
}

export default App;
```

d. App.js

```
<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="utf-8" />

    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />

    <meta name="viewport" content="width=device-width, initial-scale=1" />

    <meta name="theme-color" content="#000000" />

    <meta

      name="description"

      content="BookStore application"

    />

    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />

    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />

    <title>BookStore</title>

  </head>

  <body>

    <noscript>You need to enable JavaScript to run this app.</noscript>
```

```
    <div id="root"></div>
  </body>
</html>
```

Backend

a. App.py

```
from flask import Flask, request, jsonify, send_from_directory
from flask_sqlalchemy import SQLAlchemy
from flask_cors import CORS
from flask_login import LoginManager, UserMixin, login_user, logout_user, login_required, current_user
from werkzeug.security import generate_password_hash, check_password_hash
import os
from datetime import datetime, timedelta
from functools import wraps
import jwt as pyjwt

app = Flask(__name__, static_folder='static')
CORS(app, supports_credentials=True)

# Database configuration
app.config['SQLALCHEMY_DATABASE_URI'] = os.getenv('DATABASE_URL', 'sqlite:///bookstore.db')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['SECRET_KEY'] = os.getenv('SECRET_KEY', 'your-secret-key')
app.config['JWT_SECRET_KEY'] = os.getenv('JWT_SECRET_KEY', 'your-jwt-secret-key')
app.config['UPLOAD_FOLDER'] = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'static/uploads')

# Ensure upload folder exists
os.makedirs(app.config['UPLOAD_FOLDER'], exist_ok=True)

db = SQLAlchemy(app)
```

```
login_manager = LoginManager()
login_manager.init_app(app)

# Serve static files
@app.route('/static/<path:filename>')
def serve_static(filename):
    return send_from_directory(app.static_folder, filename)

# Models
class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    password_hash = db.Column(db.String(120), nullable=False)
    books = db.relationship('Book', backref='owner', lazy=True)

class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    price = db.Column(db.Float, nullable=False)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)

def token_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = None
        auth_header = request.headers.get('Authorization')

        if auth_header:
            if auth_header.startswith('Bearer '):
                token = auth_header.split(' ')[1]
```

```
        if not token:
            return jsonify({'error': 'Token is missing'}), 401

        try:
            data = pyjwt.decode(token, app.config['JWT_SECRET_KEY'], algorithms=["HS256"])
            current_user = User.query.get(data['user_id'])
            if not current_user:
                return jsonify({'error': 'User not found'}), 401
        except Exception as e:
            return jsonify({'error': f'Token is invalid: {str(e)}'}), 401

        return f(current_user, *args, **kwargs)
    return decorated

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

# Routes
@app.route('/register', methods=['POST'])
def register():
    data = request.get_json()

    if not data or not data.get('username') or not data.get('password'):
        return jsonify({'error': 'Username and password are required'}), 400

    if User.query.filter_by(username=data['username']).first():
        return jsonify({'error': 'Username already exists'}), 400

    hashed_password = generate_password_hash(data['password'])

    user = User(
```

```
        username=data['username'],
        password_hash=hashed_password
    )

    db.session.add(user)
    db.session.commit()

    return jsonify({'message': 'User created successfully'}), 201

@app.route('/login', methods=['POST'])
def login():
    data = request.get_json()

    if not data or not data.get('username') or not data.get('password'):
        return jsonify({'error': 'Username and password are required'}), 400

    user = User.query.filter_by(username=data['username']).first()

    if not user or not check_password_hash(user.password_hash, data['password']):
        return jsonify({'error': 'Invalid username or password'}), 401

    # Generate token using PyJWT
    token = pyjwt.encode({
        'user_id': user.id,
        'exp': datetime.utcnow() + timedelta(hours=24)
    }, app.config['JWT_SECRET_KEY'])

    return jsonify({
        'message': 'Login successful',
        'token': token,
        'user_id': user.id,
        'username': user.username
```

```
    }), 200

@app.route('/logout', methods=['POST'])
@token_required
def logout(current_user):
    return jsonify({'message': 'Logged out successfully'}), 200

@app.route('/books', methods=['GET'])
def get_books():
    books = Book.query.all()
    return jsonify([{
        'id': book.id,
        'title': book.title,
        'price': book.price,
        'user_id': book.user_id
    } for book in books]), 200

@app.route('/books/<int:book_id>', methods=['GET'])
def get_book(book_id):
    book = Book.query.get_or_404(book_id)
    return jsonify({
        'id': book.id,
        'title': book.title,
        'price': book.price,
        'user_id': book.user_id
    }), 200

@app.route('/books', methods=['POST'])
@token_required
def add_book(current_user):
    data = request.get_json()
```

```
    if not data or not data.get('title') or not data.get('price'):
        return jsonify({'error': 'Title and price are required'}), 400

    try:
        price = float(data['price'])
    except ValueError:
        return jsonify({'error': 'Price must be a number'}), 400

    book = Book(
        title=data['title'],
        price=price,
        user_id=current_user.id
    )
    db.session.add(book)
    db.session.commit()

    return jsonify({
        'id': book.id,
        'title': book.title,
        'price': book.price
    }), 201

@app.route('/books/<int:book_id>', methods=['PUT'])
@token_required
def update_book(current_user, book_id):
    book = Book.query.get_or_404(book_id)

    # Check if the current user owns the book
    if book.user_id != current_user.id:
        return jsonify({'error': 'You do not have permission to edit this book'}), 403

    data = request.get_json()
```

```
    if data.get('title'):
        book.title = data['title']

    if data.get('price'):
        try:
            book.price = float(data['price'])
        except ValueError:
            return jsonify({'error': 'Price must be a number'}), 400

    db.session.commit()

    return jsonify({
        'id': book.id,
        'title': book.title,
        'price': book.price
    }), 200

@app.route('/books/<int:book_id>', methods=['DELETE'])
@token_required
def delete_book(current_user, book_id):
    book = Book.query.get_or_404(book_id)

    # Check if the current user owns the book
    if book.user_id != current_user.id:
        return jsonify({'error': 'You do not have permission to delete this book'}), 403

    db.session.delete(book)
    db.session.commit()

    return jsonify({'message': 'Book deleted successfully'}), 200
```

```
if __name__ == '__main__':
    with app.app_context():
        db.create_all()

        # Create a test user if none exists
        if not User.query.filter_by(username='test').first():
            test_user = User(
                username='test',
                password_hash=generate_password_hash('password')
            )
            db.session.add(test_user)
            db.session.commit()
            print("Created test user: username='test', password='password'")

    app.run(debug=True)
```

README FILE

a. README.md

```
# BookStore Application

A full-stack bookstore application with React frontend and Flask backend, designed for AWS deployment.

## Features

- User authentication (login/register)
- Add new books with title, price, and image
- View all available books
- AWS RDS for database
- AWS EC2 for hosting
```

- Application Load Balancer for traffic distribution

Prerequisites

- Node.js and npm
- Python 3.8+
- AWS Account with necessary services configured:
  - RDS PostgreSQL instance
  - EC2 instance
  - Application Load Balancer
  - VPC

Backend Setup

1. Create a virtual environment:

```bash
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

2. Install dependencies:

```bash
pip install -r requirements.txt
```

3. Set up environment variables:

```bash
export DATABASE_URL=your_rds_endpoint
export SECRET_KEY=your_secret_key
export AWS_ACCESS_KEY_ID=your_aws_access_key
export AWS_SECRET_ACCESS_KEY=your_aws_secret_key
```

4. Run the Flask application:

```bash
python backend/app.py
```

Frontend Setup

1. Install dependencies:

```bash
cd frontend
npm install
```

2. Start the development server:

```bash
npm start
```

AWS Deployment

1. RDS Setup:
   - Create a PostgreSQL instance
   - Note the endpoint, username, and password
   - Configure security groups

2. EC2 Setup:
   - Launch an EC2 instance
   - Install required software (Python, Node.js)
   - Configure security groups
   - Set up environment variables

3. Application Load Balancer:
   - Create a new load balancer
   - Configure target groups
   - Set up listeners

4. VPC Configuration:
   - Create a VPC
   - Set up subnets
   - Configure route tables
   - Set up security groups

Environment Variables

Create a `.env` file in the backend directory:

```
DATABASE_URL=your_rds_endpoint
SECRET_KEY=your_secret_key
AWS_ACCESS_KEY_ID=your_aws_access_key
AWS_SECRET_ACCESS_KEY=your_aws_secret_key
```

Security Considerations

- Use HTTPS for all API calls
- Implement proper CORS policies
- Use environment variables for sensitive data
- Regularly rotate AWS credentials
- Implement proper error handling
- Use secure password hashing
- Implement rate limiting
- Regular security audits

Contributing

1. Fork the repository
2. Create your feature branch
3. Commit your changes
4. Push to the branch
5. Create a Pull Request

4.3 Testing Approach

4.3.1 Unit Testing:

Unit testing involves testing individual components of the BookStore web application—such as functions in the backend (Flask APIs), frontend React components, or database queries—in isolation to verify their correctness. This testing ensures that each part of the application, such as login, book listing, cart management, and order processing, performs as expected. Tools like pytest (Python) and Jest (JavaScript) are used to automate these tests. Unit testing helps in identifying bugs early and ensures robustness before integration into the full system.

4.3.2 Integrated Testing:

Integration testing is used to validate the interactions between integrated modules like the React frontend, Flask backend, RDS database, and AWS services. For instance, it tests if user login details submitted from the frontend are correctly processed by the backend and stored or retrieved from RDS. This phase also includes checking interactions with AWS components such as S3 for image storage, API Gateway (if used), or application responses behind a Load Balancer. Integration testing ensures that the system works cohesively across all services and components.

4.4 Modification and improvements:

- **AWS-Based Deployment**: Deployed the application using **Amazon EC2**, **RDS**, **S3**, and **Elastic Load Balancer**, providing scalability, high availability, and cost-effective hosting.

- **CI/CD Integration**: Implemented continuous integration and deployment using **AWS CodePipeline** and **CodeDeploy** for automated updates and version control.

- **Responsive Frontend**: Ensured that the React frontend is mobile-friendly and responsive across devices using modern UI libraries.

- **Secure Authentication**: Implemented secure user login with hashed passwords and integrated **AWS IAM roles and security groups** for access control and isolation.

- **Database Backup and Monitoring**: Enabled **automated backups for Amazon RDS** and configured **CloudWatch** for monitoring system logs and application performance.

- **Auto Scaling**: Set up **Auto Scaling Groups** to dynamically manage the number of EC2 instances based on traffic load.

- **CloudFront for Caching**: Integrated **Amazon CloudFront** to cache and deliver static content globally, improving performance for users worldwide.

- **Disaster Recovery**: Established **disaster recovery strategies** including multi-AZ RDS deployment and periodic EC2 snapshot backups.

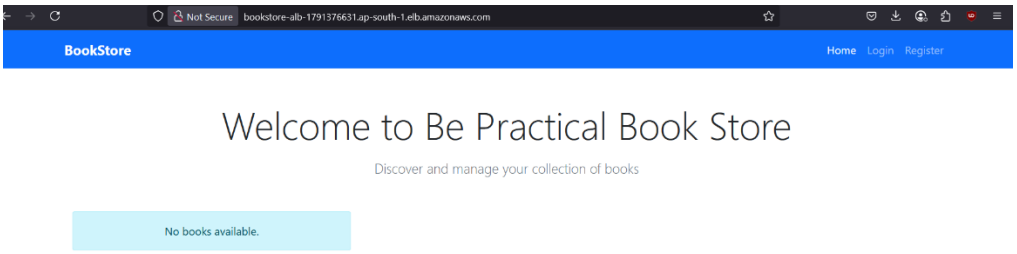
- **Future Enhancements**:

  - Add **multi-language support** for a wider audience.
  - Integrate **online payment gateways** (e.g., Stripe) for purchasing books.
  - Develop **analytics dashboards** using **Amazon QuickSight** to track user behavior and sales performance.
  - Enable **user reviews and feedback** functionality for books.
  - Build a **mobile version** or PWA (Progressive Web App) for enhanced accessibility.

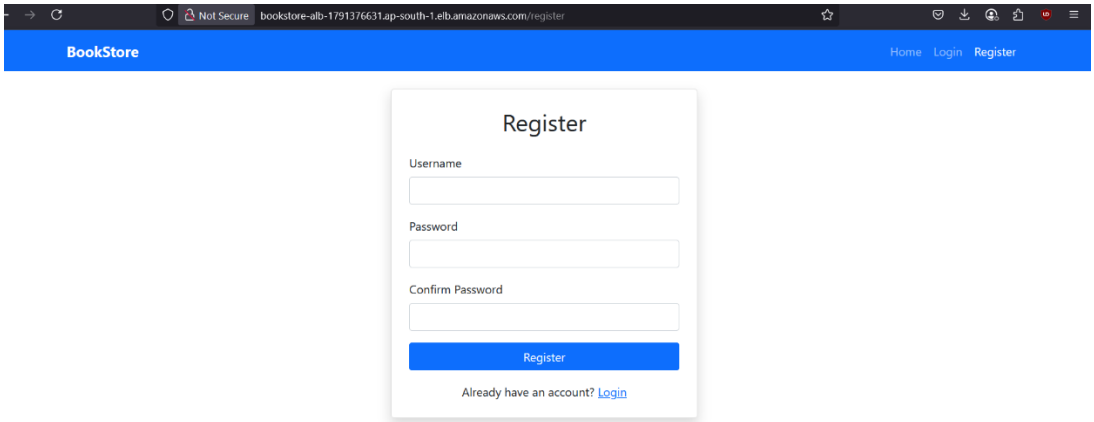
Chapter 5: RESULTS AND DISCUSSIONS

USER VIEW

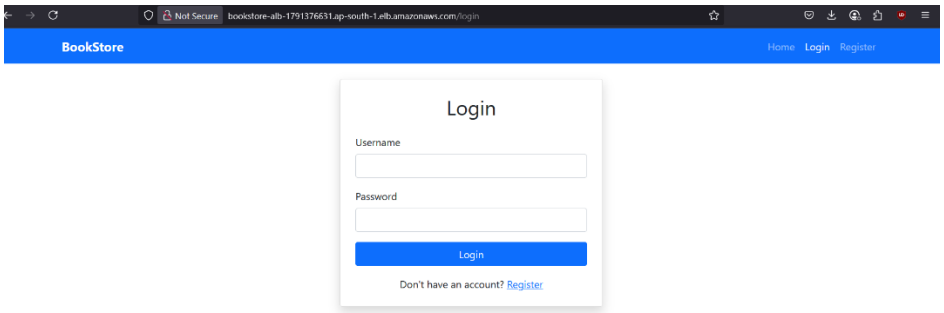
Home Page



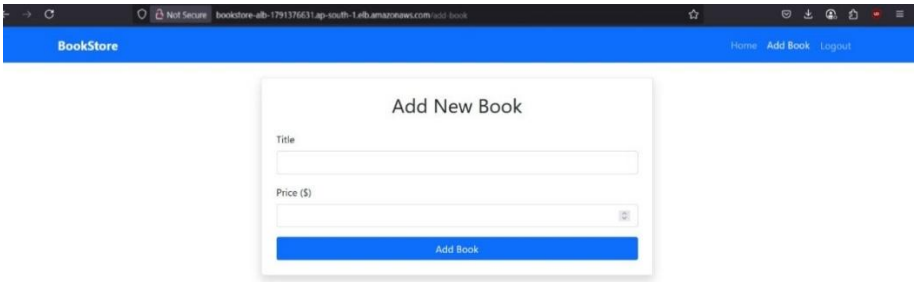
Register Page

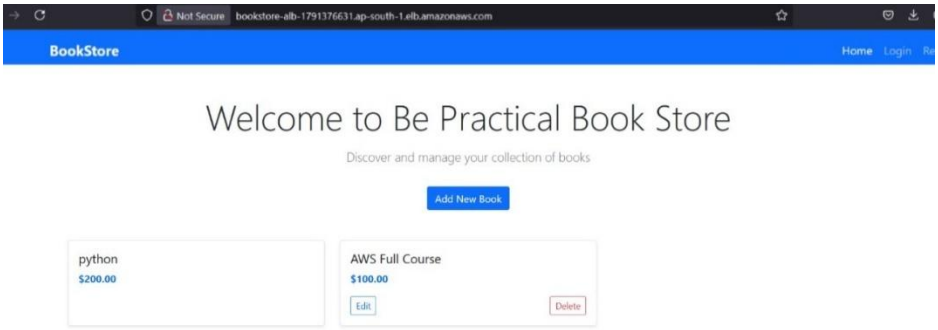


Login Page



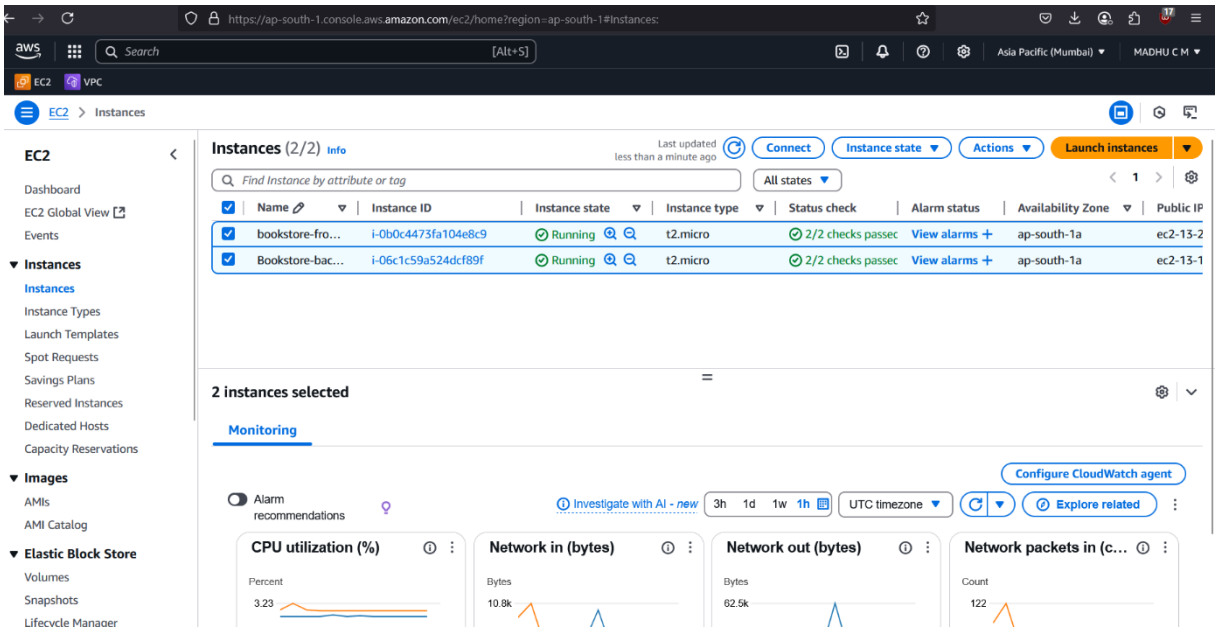
Book Adding Page



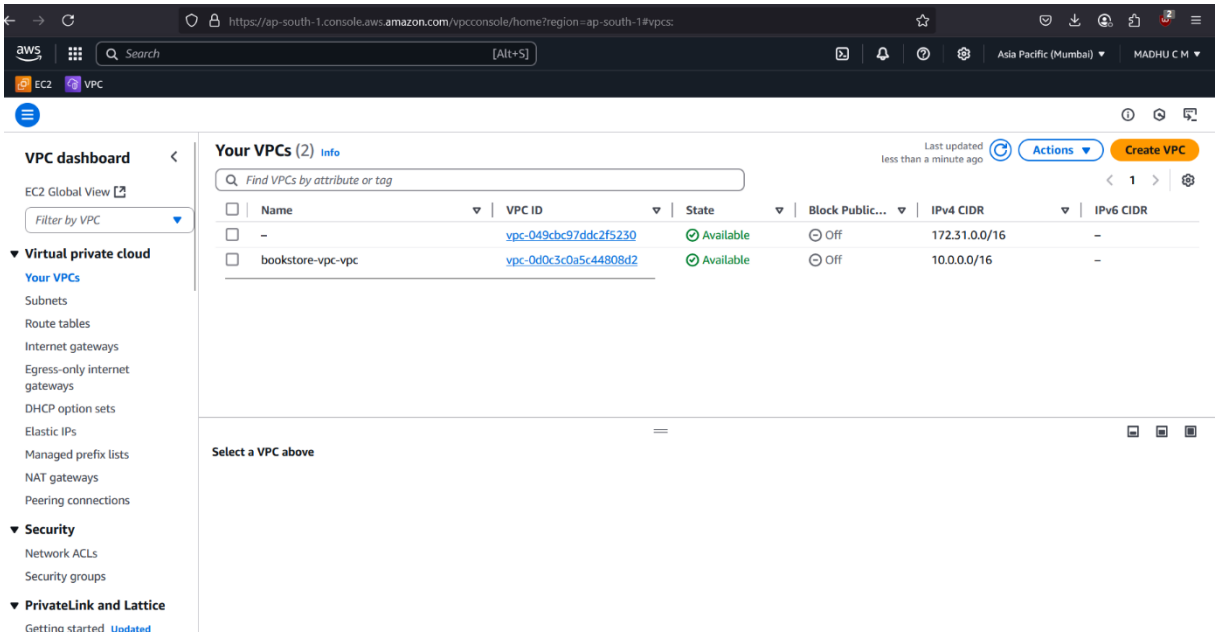


Services I Used

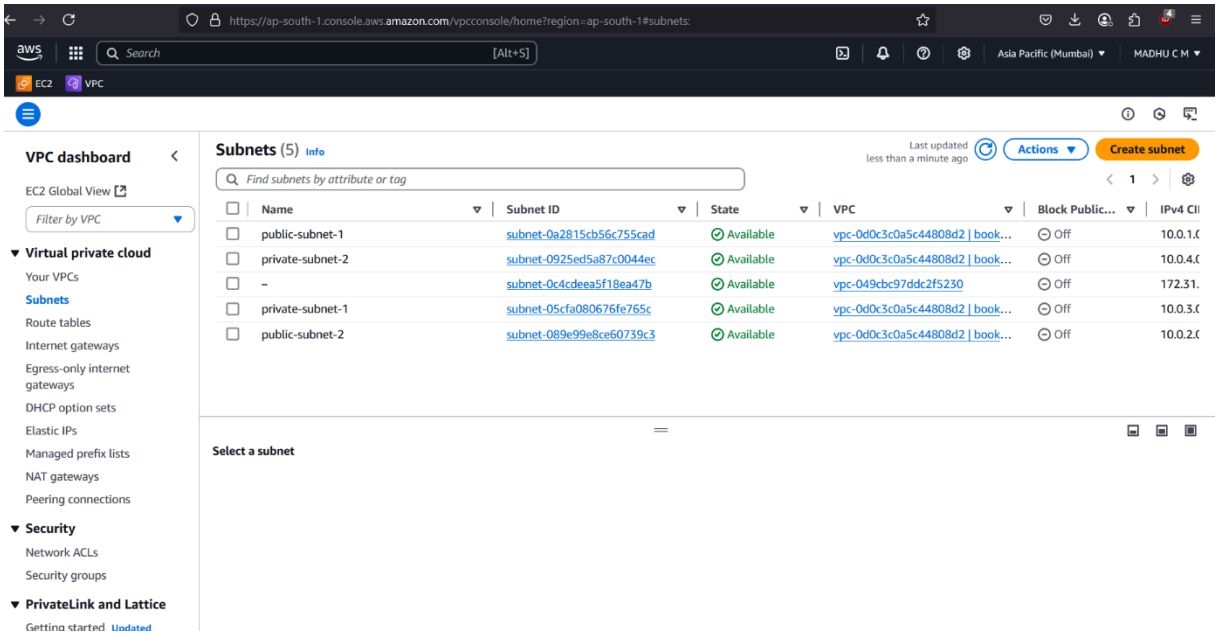
EC2: two Instances



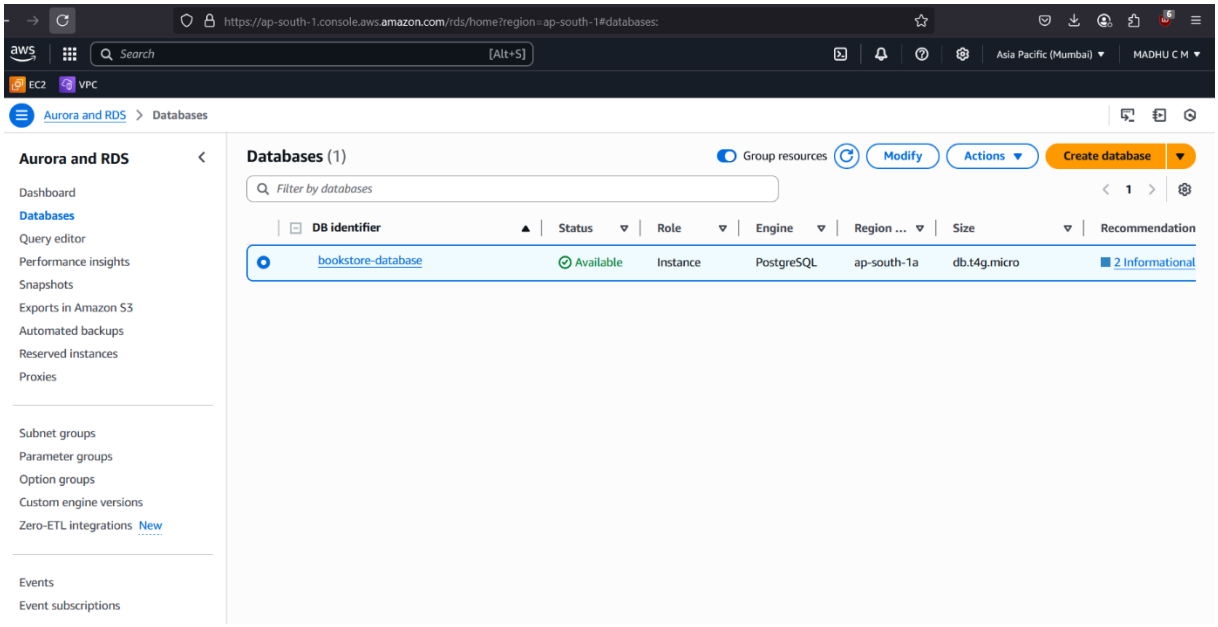
VPC:



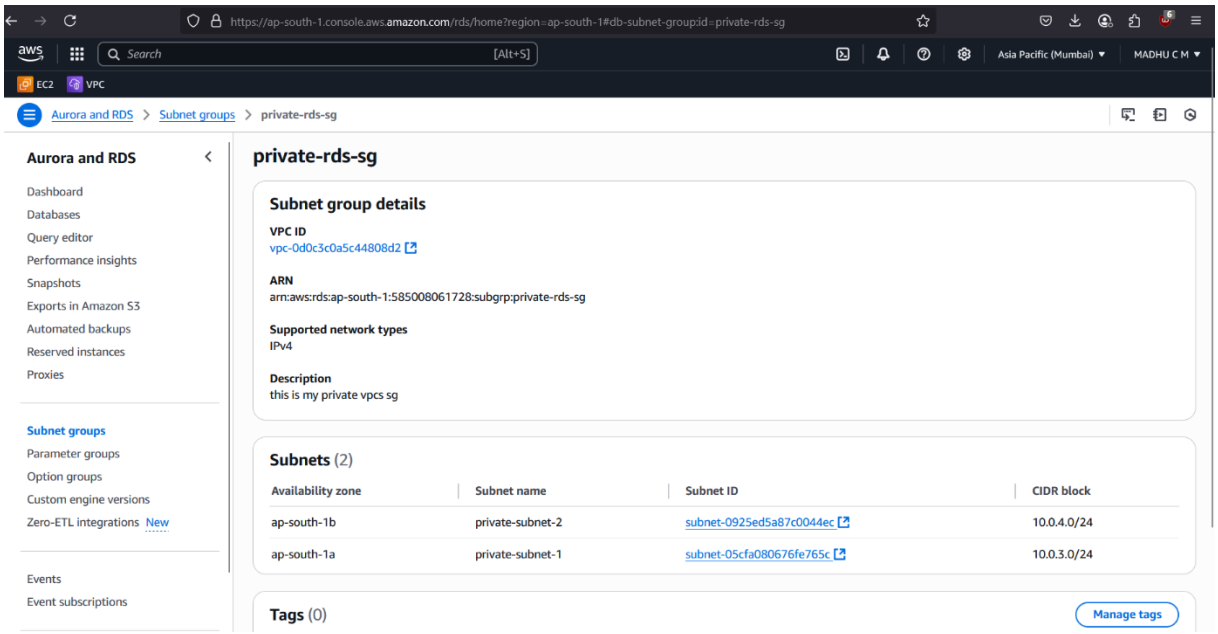
4 Subnets : Two Private & two Public



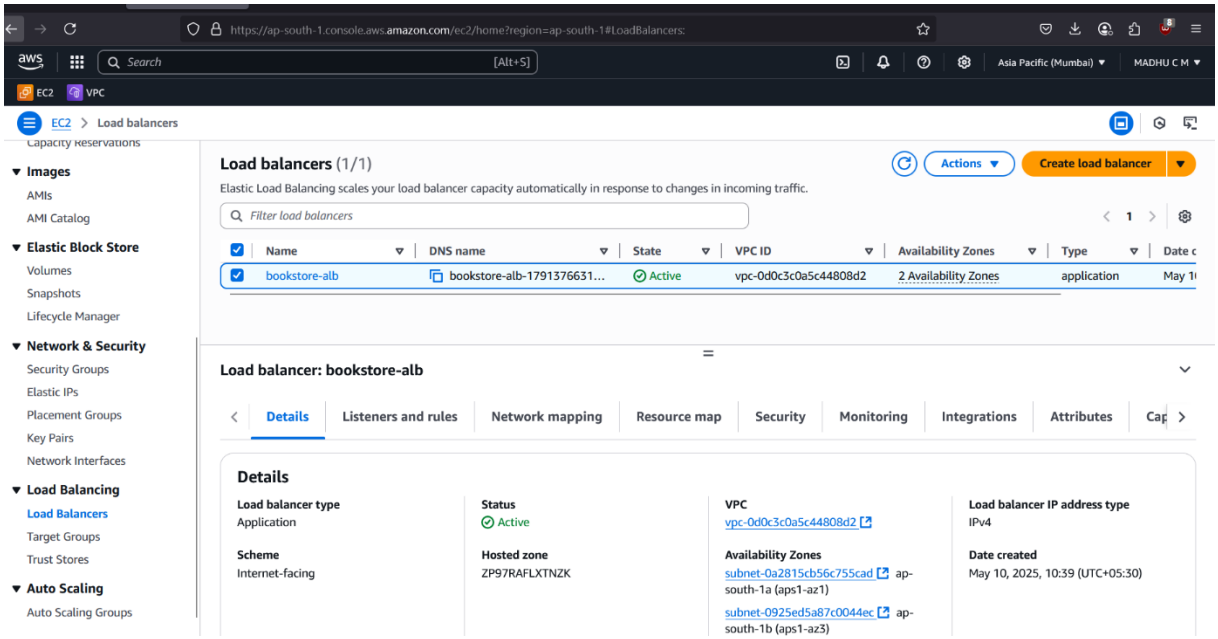
RDS: PostgreSql



Adding the two private subnets



Load Balancer: Application Load Balancer



Chapter 06 : CONCLUSION

7.1 Conclusion:

The BookStore web application serves as a comprehensive platform for browsing, purchasing, and managing book-related activities in a digital environment. Designed with a full-stack architecture using React for the frontend and Flask for the backend, and deployed on AWS, the application leverages modern cloud services to ensure high availability, scalability, and security.

By integrating core functionalities such as user authentication, book browsing, shopping cart management, order placement, and backend administration, the system provides a smooth and intuitive experience for both users and administrators. Deployment on AWS services like EC2, RDS, S3, and Elastic Load Balancer ensures performance optimization, automated scaling, and robust fault tolerance.

Rigorous unit and integration testing, along with continuous monitoring via AWS CloudWatch, help maintain system reliability and fast issue resolution. Security practices such as IAM-based access control and encrypted connections safeguard user data and application integrity

In summary, this cloud-based BookStore application demonstrates how leveraging AWS services can result in a scalable, reliable, and secure e-commerce platform. With future enhancements like analytics, payment integration, and mobile access, the system is well-positioned to meet evolving user expectations and support a growing user base in a dynamic digital marketplace.

7.3 Future Scope:

The future scope of the BookStore web application is extensive, supported by continuous advancements in cloud computing, user experience design, and emerging technologies. Integration with **AI and machine learning** can enable personalized book recommendations, dynamic pricing models, and intelligent search capabilities. Expanding to a **mobile app** or **progressive web app (PWA)** will enhance accessibility and user engagement across platforms.

Features such as **voice-assisted navigation**, **chatbots for customer support**, and **predictive analytics** for inventory and sales trends can further enrich the user experience. **Blockchain technology** may be integrated to ensure secure digital transactions, transparent reviews, and anti-piracy measures for digital content. Leveraging **serverless architectures** (like AWS Lambda) and **containerization** can improve scalability and deployment efficiency.

Overall, the application can evolve into a smart, efficient, and user-centric platform by continuously adopting cutting-edge technologies and aligning with changing consumer behavior in the digital commerce space.