Cloud-hosted Banking Data Analytics and Reporting System on AWS

"CloudBank Analytics" is a comprehensive cloud-based banking data analytics and reporting system designed to streamline financial data processing, analysis, and visualization. The project leverages Amazon Web Services (AWS) for robust and scalable infrastructure, utilizing Amazon RDS for secure and efficient data storage, AWS Elastic Beanstalk for reliable web application hosting, and AWS Lambda for automated data processing and report generation.

The user-friendly web interface, built with Flask, allows bank administrators to log in to their personal accounts, creating a seamless experience for data visualization and report management. Once logged in, users are presented with a dashboard that serves as a central hub for all banking analytics activities. The dashboard prominently features real-time financial data, key performance indicators (KPIs), and options for generating custom reports.

By combining modern cloud technology with an intuitive user interface, "CloudBank Analytics" aims to provide financial institutions with powerful tools for monitoring customer behavior, managing risk, and ensuring regulatory compliance, ultimately improving decision-making and operational efficiency.

Scenarios

Scenario 1: Real-time Transaction Monitoring

Sarah, a bank's fraud detection analyst, logs into CloudBank Analytics during her morning routine. The dashboard immediately alerts her to unusual transaction patterns detected overnight. Using the real-time analytics feature, Sarah quickly investigates the flagged transactions, confirms a potential fraud attempt, and takes immediate action to protect the affected accounts.

Scenario 2: Custom Report Generation

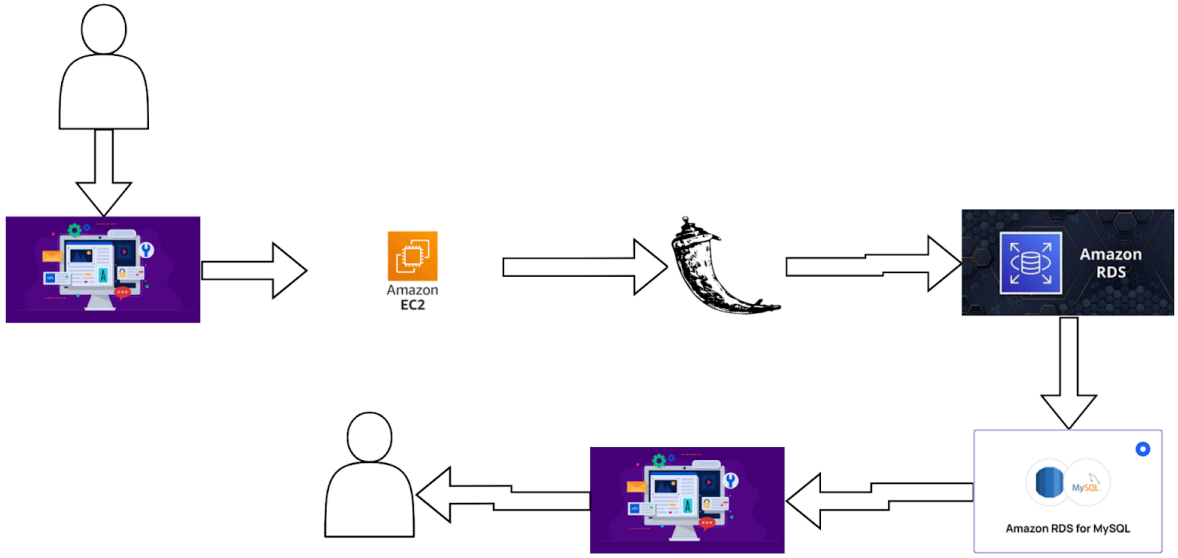
John, a financial manager, needs to prepare a comprehensive quarterly report for the board meeting. He logs into CloudBank Analytics and uses the custom report generation feature. John selects various metrics such as loan performance, deposit growth, and customer acquisition rates. The system, leveraging AWS Lambda, quickly

processes the vast amount of data stored in Amazon RDS and generates a detailed report, which is then stored in Amazon S3 for easy access and sharing.

Scenario 3: Regulatory Compliance Monitoring

Lisa, a compliance officer, uses CloudBank Analytics to ensure the bank meets all regulatory requirements. She accesses a specialized dashboard that tracks key compliance metrics in real-time. When she notices that a particular metric is approaching a regulatory threshold, she uses the system to drill down into the underlying data, identify the root cause, and initiate corrective actions before any compliance issues arise.

Architecture:



Prior Knowledge:

1. AWS Account Setup: https://youtu.be/CjKhQoYeR4Q?si=ui8Bvk_M4FfVM-Dh

2.Web Application Stack : FLask || MySQL Connector using flask || HTML/JS/CSS

3. AWS EC2 Instance: https://www.youtube.com/results?search_query=aws+ec2+oneshot

4. RDS Database: https://www.youtube.com/results?search_query=rds+oneshot

5. MySQL: https://www.youtube.com/results?search_query=mysql+tutorial

6. RDS to MySQL:https://www.youtube.com/results?search_query=mysql+connector+for+rds

7. Clone Git repo: https://www.youtube.com/results?search_query=clone+github+repository

8. AWS Cost Management: https://youtu.be/OKYJCHHSWb4?si=aY3DQJ1v26CfZxXA

9. AWS IAM: https://www.youtube.com/results?search_query=aws+iam

10. AWS CloudWatch:https://www.youtube.com/results?search_query=aws+cloudwatch

Project Flow:

Project Initialization:

- Define objectives, scope, and KPIs; set up the AWS environment.

EC2 Instance Setup:

- Launch and configure an EC2 instance to host the web application.

RDS Database Setup:

- Create and configure an RDS instance with MySQL engine.

Web Application Development:

- Develop the web application with registration, login, and dashboard features.

Database Integration:

- Connect the web application to the RDS database using appropriate drivers.

User Interface Implementation:

- Create user-friendly interfaces for registration, login, and blood request management.

Testing and Optimization:

- Conduct thorough testing of all features and optimize for performance.

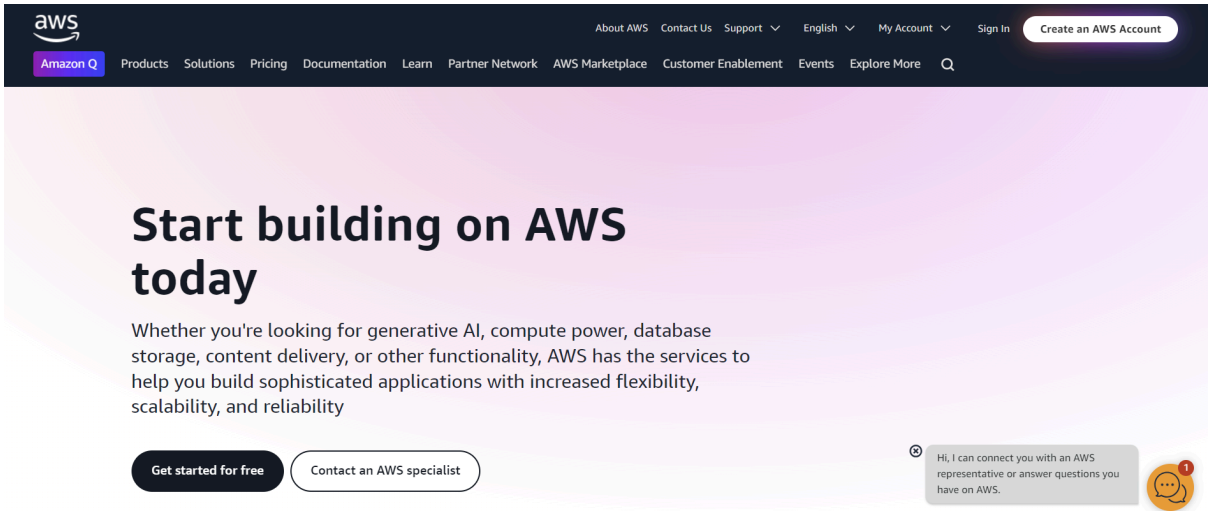
Milestone 1: AWS Account Creation

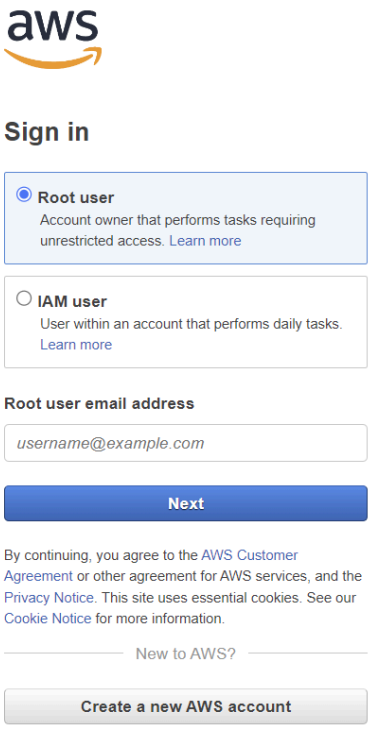
In this milestone, we will set up an AWS account to access the necessary services for the BloodBridge project.

Activity 1: Create AWS Account

1. Go to the AWS website.
2. Click on the "Create an AWS Account" button.
3. Follow the prompts to enter your email address and choose a password.

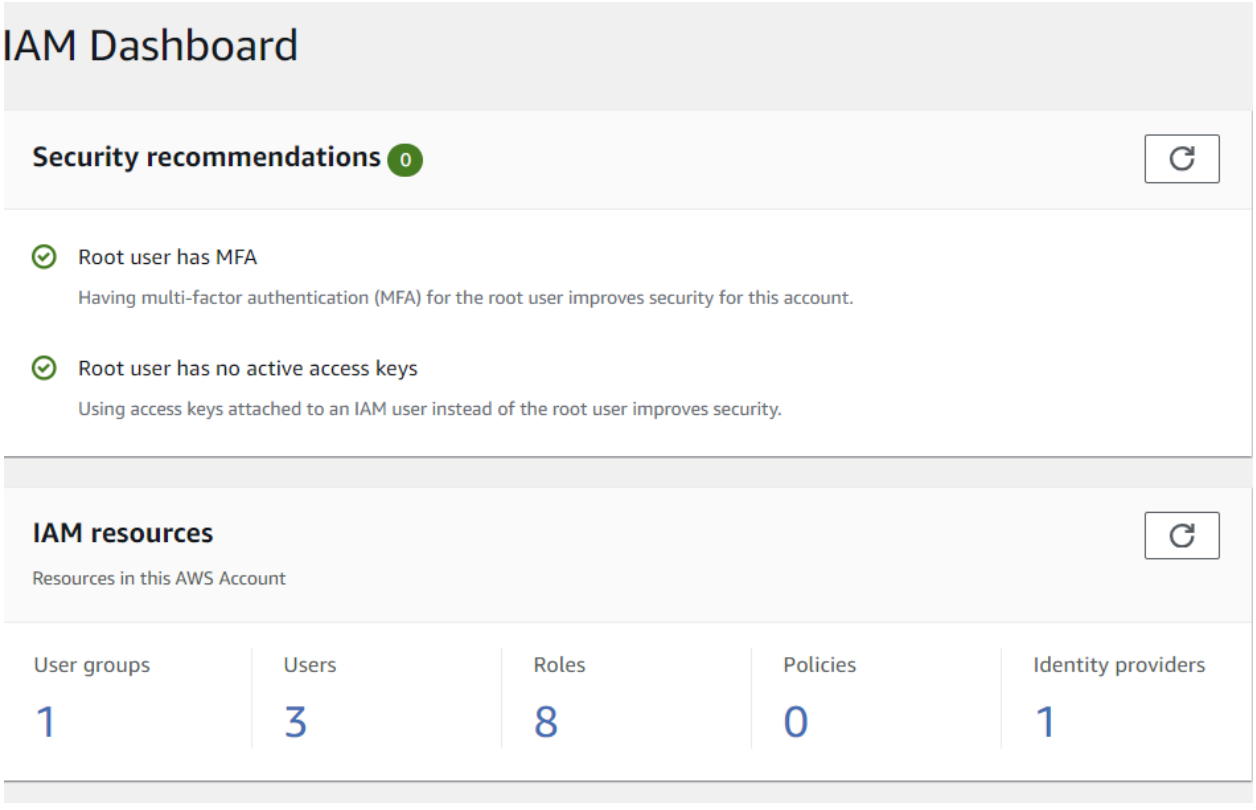
4. Provide the required account information, including your name, address, and phone number.
5. Enter your payment information. (Note: While AWS offers a free tier, a credit card is required for verification.)
6. Complete the identity verification process.
7. Choose a support plan (the basic plan is free and sufficient for starting).
8. Sign in to your new AWS account.

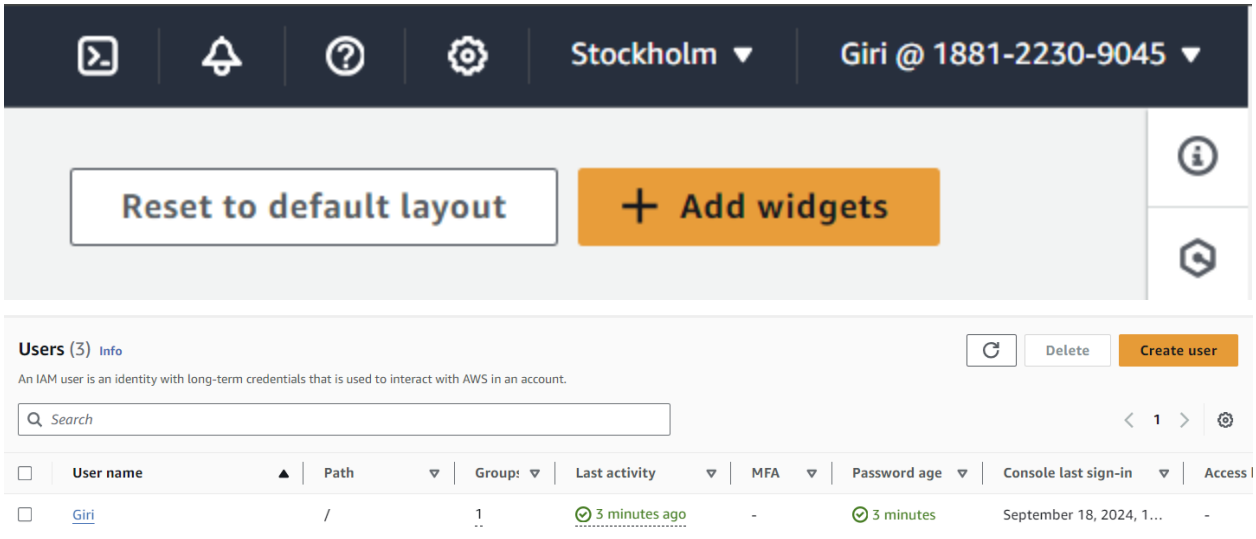




Activity 2: Set Up IAM Users and Permissions

1. Access the IAM console from the AWS Management Console.
2. Create a new IAM user for yourself with administrative access.
3. Set up multi-factor authentication (MFA) for added security.
4. Create a group for developers and assign necessary permissions.
5. Generate access keys for programmatic access if needed.

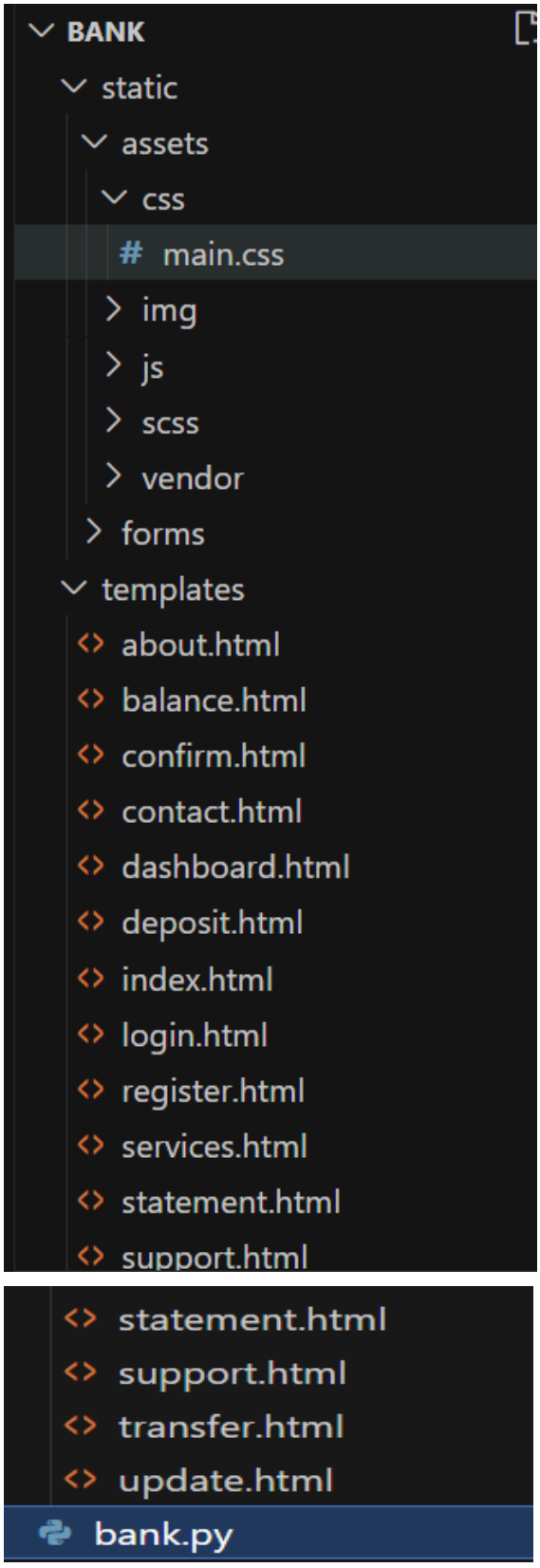




Milestone 2: Local Development Environment Setup

Activity 1: Set Up Local Development Environment
1. Install Python and pip on your local machine.
2. Install Flask and other necessary Python packages (e.g., `flask-sqlalchemy`, `mysql-connector-python`).
3. Set up a virtual environment for your project.

Activity 2: Create Flask Application Structure
1. Create a new directory for your Easybank project.
2. Set up a basic Flask application structure:


BANK
static
assets
css
main.css
img
js
scss
vendor
forms
templates
about.html
balance.html
confirm.html
contact.html
dashboard.html
deposit.html
index.html
login.html
register.html
services.html
statement.html
support.html
statement.html
support.html
transfer.html
update.html
bank.py

Milestone 3: Develop Easybank Web Application

Activity 1: Create Web Pages

1. Design and create HTML templates for your banking application, including:
   - Home page
   - User registration page
   - Login page
   - Account dashboard
   - Transaction page
   - Account creation page
   - Check Balance page
   - Deposit page
   - Services page
   - Contact Page
   - confirm page
   -Account Login page
   - Statement page
2. Place these templates in the `templates/` directory.
3. Create CSS files in the `static/css/` directory for styling.
4. Add any necessary JavaScript files in the `static/js/` directory.

Activity 2: Implement Flask Routes and Views

1. In `app.py`, create Flask routes for each of your web pages.
2. Implement view functions to render the appropriate templates.
3. Add form handling for user input (registration, login, transactions, etc.).

bank.py > ...

```python
from flask import Flask, render_template, request, redirect, url_for
import mysql.connector
from flask import flash, session
from datetime import datetime

app = Flask(__name__)
app.secret_key = "your_secret_key"  # Needed for flash msgs

# Database config
db_config = {
    'host': 'bank.crqmssgockvo.ap-south-1.rds.amazonaws.com',
    'user': 'admin',
    'password': 'Surya123456',
    'database': 'bank'
}

cnxpool = mysql.connector.pooling.MySQLConnectionPool(pool_name="mypool",
                                                      pool_size=5,
                                                      **db_config)

# Function to establish a database connection
def get_db_connection():
    try:
        return cnxpool.get_connection()
    except mysql.connector.Error as err:
        print(f"Error: {err}")
        return None

@app.route("/test-db-connection")
def test_db_connection():
    try:
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT DATABASE();")  # Test query to check connection
        db_name = cursor.fetchone()
        cursor.close()
        conn.close()
        return f"Connected to the database: {db_name[0]}"
    except mysql.connector.Error as err:
        return f"Error: {err}"
```

```python
# Define your routes here
@app.route("/")
def index():
    try:
        conn = get_db_connection()
        cursor = conn.cursor()
    except mysql.connector.Error as err:
        print(f"Error: {err}")
    finally:
        if conn:
            cursor.close()
            conn.close()
    return render_template("index.html")


@app.route("/register", methods=['post', 'get'])
def register():
    if request.method == 'POST':
        # session.init_app(app)  # Initialize the session
        full_name = request.form['full_name']
        email = request.form['email']
        password = request.form['password']
        phone = request.form['phone']
        address = request.form['address']
        aadhar_number = request.form['aadhar_number']
        pan_card = request.form['pan_card']

        conn = get_db_connection()
        cursor = conn.cursor()

        # Check if the user already exists
        cursor.execute("SELECT * FROM users WHERE email = %s", (email,))
        user = cursor.fetchone()

        if user:
            flash("email already exists! Please log in")
            return redirect(url_for('login', email= email)) # redirect to login page

        # Validate phone number and Aadhar number
        if len(phone) != 10:
            flash("Phone number must be 10 digits")
            return render_template("register.html")
        if len(aadhar number) != 12:
```

bank.py > ...

```python
def register():
            return render_template("register.html")
        if len(aadhar_number) != 12:
            flash("Aadhar number must be 12 digits")
            return render_template("register.html")

        # Insert the new user into the database
        cursor.execute("INSERT INTO users (full_name, email, password, phone, address, aadhar_number, pan_card) VALUES (%s,%s, %s, %s, %s, %s, %s, %s)",
                       (full_name, email, password, phone, address, aadhar_number, pan_card))
        conn.commit()
        cursor.close()
        conn.close()

        user_data = {
            'full_name': full_name,
            'email': email,
        }
        session['user'] = user_data
        flash("Registration successful! Please log in.")
        return redirect(url_for('confirm',user=user_data))

    return render_template("register.html")

@app.route("/confirm")
def confirm():
    user = session.get('user')
    if user:
        return render_template("confirm.html", user=user)
    else:
        return redirect(url_for('login'))

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


        conn = get_db_connection()
        cursor = conn.cursor()

        # Verify login credentials
```

bank.py > ...

```python
def login():

        # Verify login credentials
        cursor.execute("SELECT * FROM users WHERE email = %s AND password = %s",
                       (email, password))
        user = cursor.fetchone()

        cursor.close()
        conn.close()

        if user:
            user_data = {
                'fullname' : user[4],
                'email': user[1],
                'user_id':user[0]
            }
            session['user'] = user_data
            print('this is email', email)
            return redirect(url_for('dashboard'))
        else:
            flash("Invalid login credentials!")
            return redirect(url_for('login'))


    return render_template("login.html")


@app.route("/dashboard")
def dashboard():
    user_data = session.get('user')
    if user_data:
        email = user_data['email']
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM users WHERE email = %s", (email,))
        user = cursor.fetchone()
        cursor.close()
        conn.close()
        return render_template("dashboard.html", user=user)
    else:
        return redirect(url_for('login'))

@app.route("/deposit", methods=['post', 'get'])
```

bank.py > ...

```python
@app.route("/deposit", methods=['post', 'get'])
def deposit():
    user_data = session.get('user')
    if user_data:
        if request.method == 'POST':
            amount = float(request.form['deposit_amount'])
            account_type = request.form['account_type']
            conn = get_db_connection()
            cursor = conn.cursor()
            cursor.execute("SELECT 1 FROM accounts WHERE user_id = %s", (user_data['user_id'],))
            if not cursor.fetchone():
                # Insert a new row
                cursor.execute("INSERT INTO accounts (user_id, balance, account_type) VALUES (%s, %s, %s)", (user_data['user_id'], amount, account_type))
                cursor.execute("INSERT INTO account_statements (user_id, transaction_type,transaction_amount, transaction_date) VALUES (%s, 'Credit', %s, %s)", (user_data['user_id'], amount, datetime.r
            else:
                # Update the existing row
                cursor.execute("UPDATE accounts SET balance = balance + %s WHERE user_id = %s", (amount, user_data['user_id']))
                cursor.execute("INSERT INTO account_statements (user_id, transaction_type,transaction_amount,  transaction_date) VALUES (%s, 'Credit', %s, %s)", (user_data['user_id'], amount, datetime.
            conn.commit()
            cursor.close()
            conn.close()
            flash("Funds deposited successfully!")
            return redirect(url_for('dashboard'))
        return render_template("deposit.html")
    else:
        return redirect(url_for('login'))

@app.route("/balance", methods=['get'])
def check_balance():
    user_data = session.get('user')
    if user_data:
        email = user_data['email']
        conn = cnxpool.get_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT balance FROM accounts WHERE user_id = %s", (user_data['user_id'],))
        balance = cursor.fetchone()[0]
        cursor.close()
        conn.close()
        return render_template("balance.html", balance=balance)
    else:
        return redirect(url_for('login'))

```

bank.py > ...

```python
@app.route("/account-statement", methods=['get'])
def account_statement():
    user_data = session.get('user')
    if user_data:
        email = user_data['email']
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM transactions WHERE email = %s", (email,))
        transactions = cursor.fetchall()
        cursor.close()
        conn.close()
        return render_template("account_statement.html", transactions=transactions)
    else:
        return redirect(url_for('login'))

@app.route("/transfer", methods=['post', 'get'])
def transfer():
    user_data = session.get('user')
    if user_data:
        if request.method == 'POST':
            user_id = request.form.get('user_id')
            amount = request.form.get('amount')

            if user_id and amount:
                try:
                    amount = float(amount)

                    # Check if the user_id exists
                    conn = get_db_connection()
                    cursor = conn.cursor()
                    cursor.execute("SELECT user_id FROM users WHERE user_id = %s", (user_id,))
                    recipient_user_id = cursor.fetchone()

                    if recipient_user_id is None:
                        flash("Recipient user not found!")
                        return redirect(url_for('transfer'))

                    # Check if the sender has sufficient balance
                    cursor.execute("SELECT balance FROM accounts WHERE user_id = %s", (user_data['user_id'],))
                    sender_balance = cursor.fetchone()

                    if sender_balance is None:
                        flash("Sender account not found!")
```

bank.py > ...

```python
def transfer():
                    if sender_balance is None:
                        flash("Sender account not found!")
                        return redirect(url_for('transfer'))

                    sender_balance = sender_balance[0]

                    if sender_balance >= amount:
                        # Update sender's balance
                        cursor.execute("UPDATE accounts SET balance = balance - %s WHERE user_id = %s", (amount, user_data['user_id']))

                        # Update recipient's balance
                        cursor.execute("UPDATE accounts SET balance = balance + %s WHERE user_id = %s", (amount, user_id))

                        # Insert a new transaction into the account_statements table
                        cursor.execute("INSERT INTO account_statements (user_id, transaction_type, transaction_amount, transaction_date) VALUES (%s, 'Debit', %s, %s)", (user_data['user_id'], amount, d
                        cursor.execute("INSERT INTO account_statements (user_id, transaction_type, transaction_amount, transaction_date) VALUES (%s, 'Credit', %s, %s)", (user_id, amount, datetime.now(

                        conn.commit()
                        cursor.close()
                        conn.close()
                        flash("Funds transferred successfully!")
                        return redirect(url_for('dashboard'))
                    else:
                        flash("Insufficient balance!")
                        return redirect(url_for('transfer'))
                except ValueError:
                    flash("Invalid transfer amount!")
                    return redirect(url_for('transfer'))
            else:
                flash("Please fill in all fields!")
                return redirect(url_for('transfer'))
        return render_template("transfer.html")
    else:
        return redirect(url_for('login'))

@app.route("/statement", methods=['GET'])
def statements():
    user_data = session.get('user')
    if user_data:
        user_id = user_data['user_id']
        conn = get_db_connection()
        cursor = conn.cursor()
```

bank.py > ...

```python
def statements():
    user_data = session.get('user')
    if user_data:
        user_id = user_data['user_id']
        conn = get_db_connection()
        cursor = conn.cursor()

        # Fetch the account statements for the logged-in user
        cursor.execute("""
            SELECT transaction_type, transaction_amount, transaction_date, description
            FROM account_statements
            WHERE user_id = %s
            ORDER BY transaction_date DESC
        """, (user_id,))

        transactions = cursor.fetchall()

        cursor.close()
        conn.close()

        return render_template("statement.html", transactions=transactions)
    else:
        return redirect(url_for('login'))


@app.route("/customer-support")
def customer_support():
    return render_template("customer_support.html")


@app.route("/services")
def services():
    return render_template("services.html")

@app.route("/contact")
def contact():
    return render_template("contact.html")


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

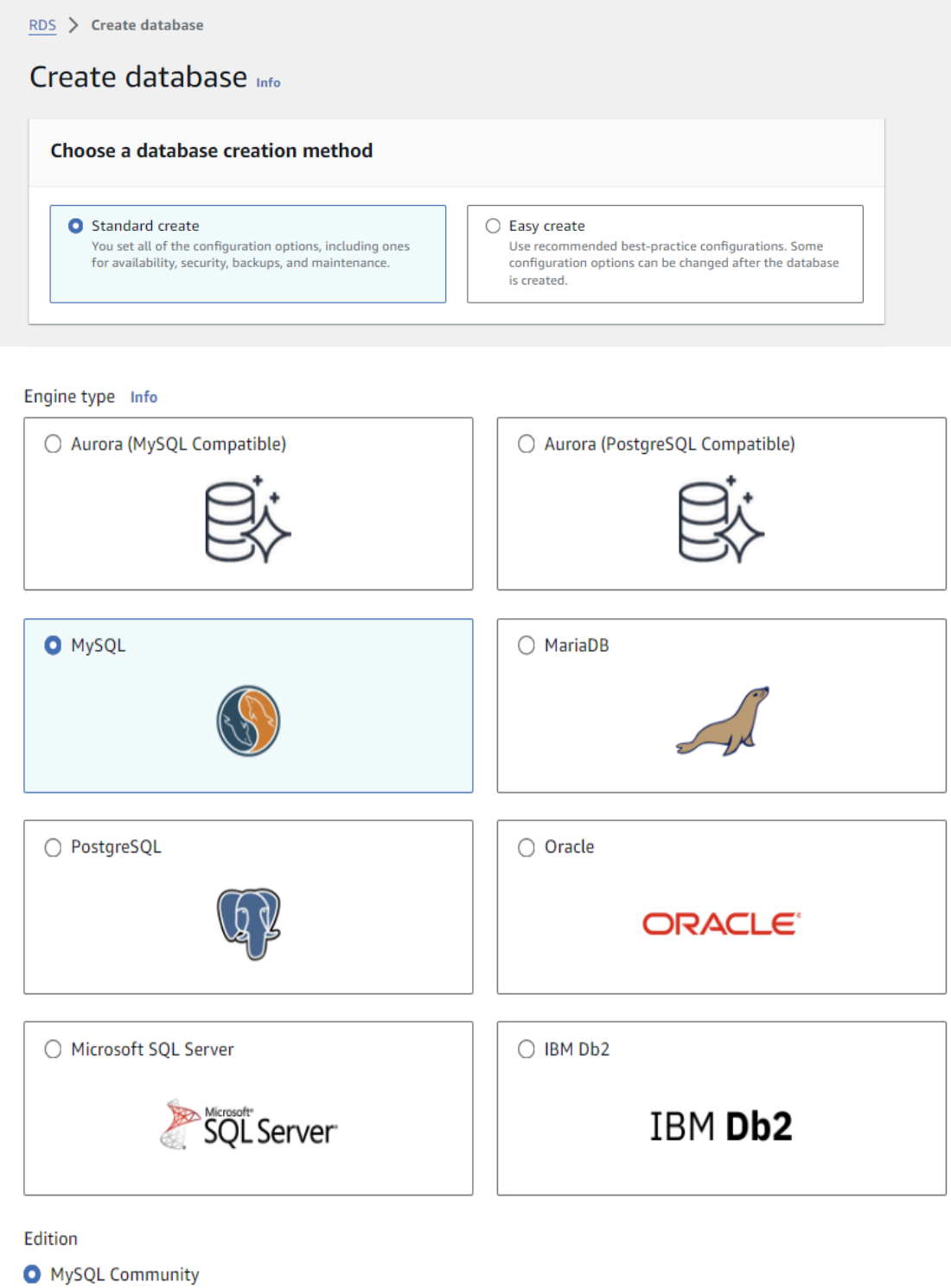
Milestone 3: AWS RDS Setup and MySQL Integration

Activity 1: Create Amazon RDS Instance

1. Access RDS Console from the AWS Management Console.
2. Create a new RDS instance:
    - Choose MySQL as the engine type.
    - Select an appropriate instance size (e.g., db.t3.micro for testing).

- Configure storage, network settings, and security groups.
- Set up the master username and password.
- Make sure to allow connections from your local IP for development and testing.

3. Note down the endpoint, port, database name, username, and password.



▼ **Credentials Settings**

Master username Info

Type a login ID for the master user of your DB instance.

admin

1 to 16 alphanumeric characters. The first character must be a letter.

Credentials management

You can use AWS Secrets Manager or manage your master user credentials.

- ○ Managed in AWS Secrets Manager - *most secure*
  RDS generates a password for you and manages it throughout its lifecycle using AWS Secrets Manager.
- ◉ Self managed
  Create your own password or have RDS create a password that you manage.

☐ Auto generate password

Amazon RDS can generate a password for you, or you can specify your own password.

Master password Info

••••••••

Minimum constraints: At least 8 printable ASCII characters. Can't contain any of the following symbols: / ' " @

Confirm master password Info

••••••••

Templates

Choose a sample template to meet your use case.

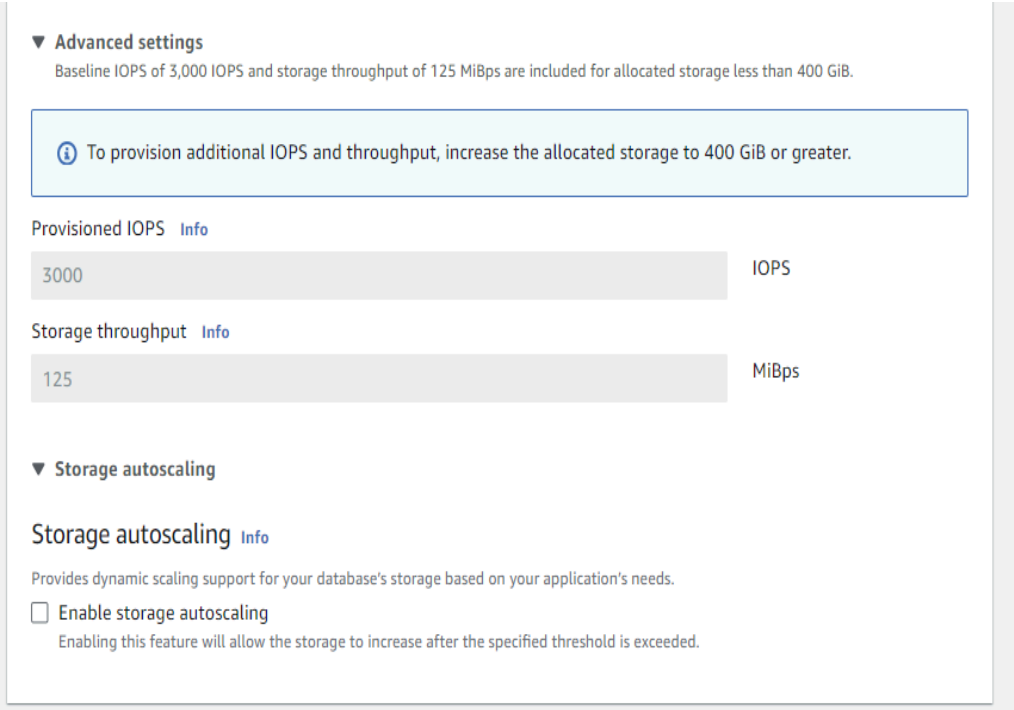
- ○ Production
  Use defaults for high availability and fast, consistent performance.
- ○ Dev/Test
  This instance is intended for development use outside of a production environment.
- ◉ Free tier
  Use RDS Free Tier to develop new applications, test existing applications, or gain hands-on experience with Amazon RDS. Info

Availability and durability

Deployment options Info

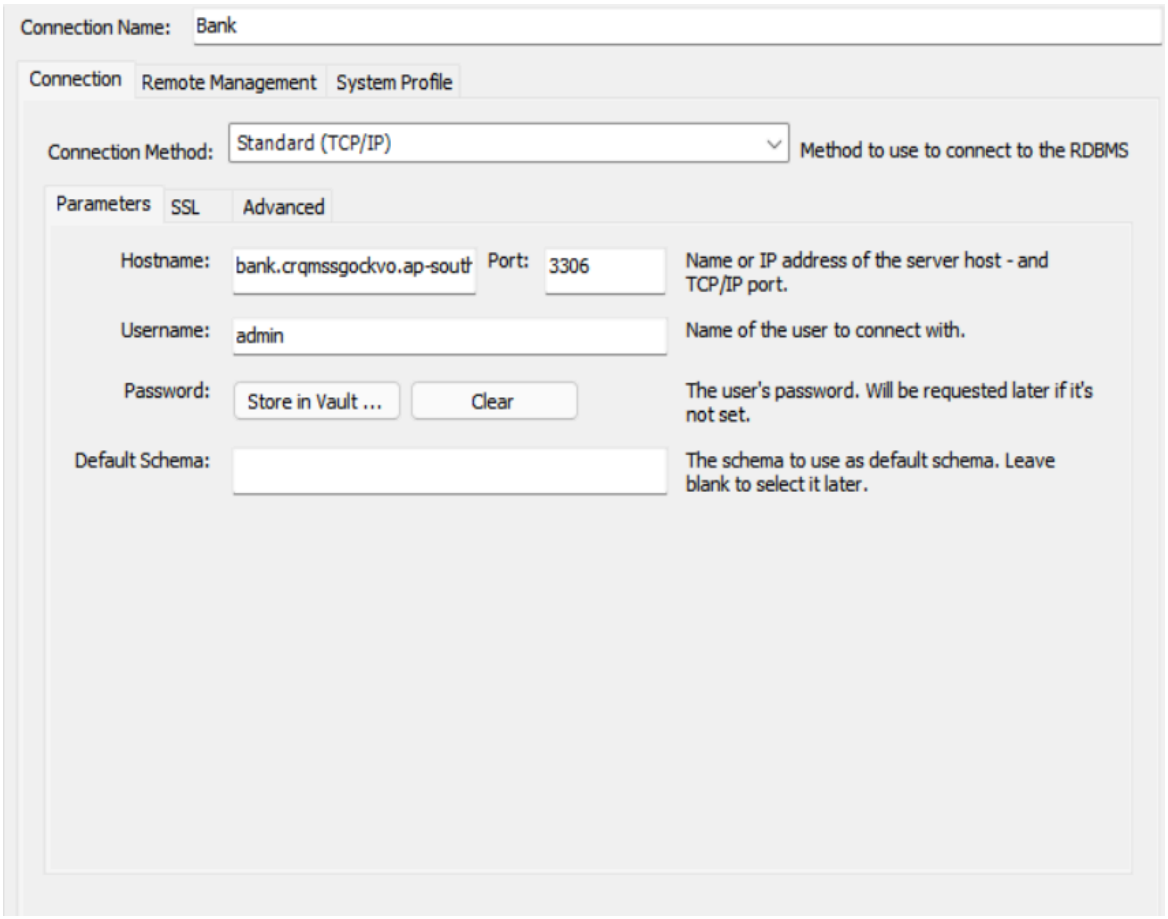
The deployment options below are limited to those supported by the engine you selected above.

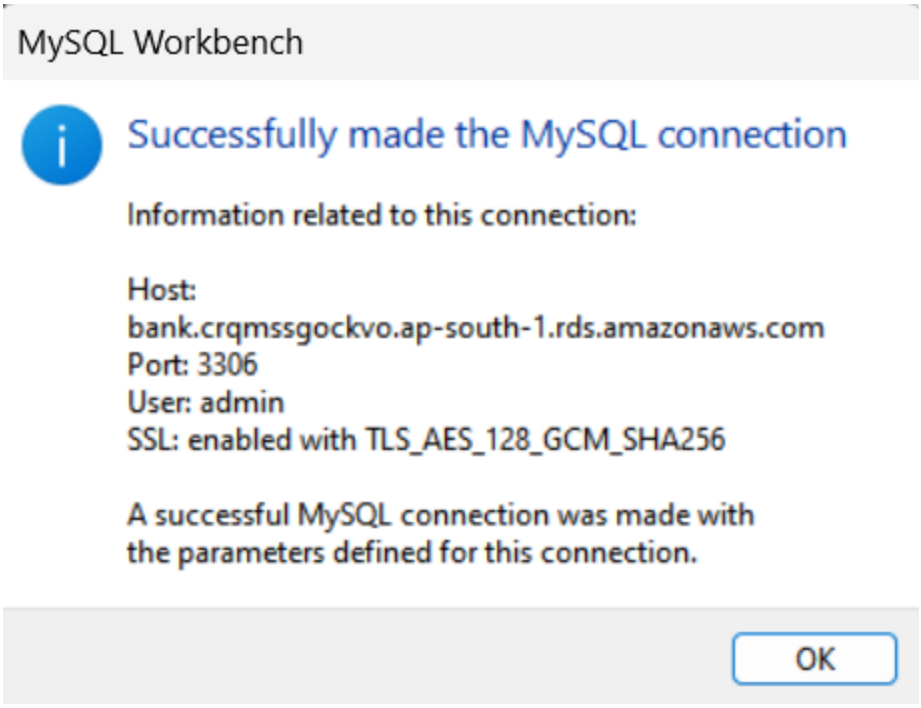
- ○ Multi-AZ DB Cluster
  Creates a DB cluster with a primary DB instance and two readable standby DB instances, with each DB instance in a different Availability Zone (AZ). Provides high availability, data redundancy and increases capacity to serve read workloads.
- ○ Multi-AZ DB instance (not supported for Multi-AZ DB cluster snapshot)
  Creates a primary DB instance and a standby DB instance in a different AZ. Provides high availability and data redundancy, but the standby DB instance doesn't support connections for read workloads.
- ◉ Single DB instance (not supported for Multi-AZ DB cluster snapshot)
  Creates a single DB instance with no standby DB instances.



Activity 2: Connect to RDS using MySQL Client

1. Install MySQL client on your local machine if not already done.
2. Use the following command to connect to your RDS instance:
   Copy
   ```
   mysql -h <your-rds-endpoint> -P 3306 -u <your-username> -p
   ```
3. Enter your RDS master password when prompted.


Connection Name:
Bank
Connection
Remote Management
System Profile
Connection Method:
Standard (TCP/IP)
Method to use to connect to the RDBMS
Parameters
SSL
Advanced
Hostname:
bank.crqmssgockvo.ap-south
Port:
3306
Name or IP address of the server host - and TCP/IP port.
Username:
admin
Name of the user to connect with.
Password:
Store in Vault ...
Clear
The user's password. Will be requested later if it's not set.
Default Schema:
The schema to use as default schema. Leave blank to select it later.



Milestone 5: EC2 Deployment

Activity 1: Launch EC2 Instance

1. Access EC2 Console from the AWS Management Console.
2. Launch a new EC2 instance:
   - Choose an Amazon Linux 2 AMI.
   - Select an appropriate instance type (e.g., t2.micro for testing).
   - Configure instance details, including VPC and subnet.
   - Add storage as needed.
   - Configure security group to allow SSH (port 22) and HTTP/HTTPS (ports 80/443).
3. Create or select an existing key pair for SSH access.

EC2 > Instances > Launch an instance

Launch an instance Info

Amazon EC2 allows you to create virtual machines, or instances, that run on the AWS Cloud. Quickly get started by following the simple steps below.

Name and tags Info

Name

flask application

Add additional tags

Application and OS Images (Amazon Machine Image) Info

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below

Search our full catalog including 1000s of application and OS images

Recents | Quick Start

Amazon Linux | macOS | Ubuntu | Windows | Red Hat | SU

Browse more AMIs

Including AMIs from AWS, Marketplace and the Community

Amazon Machine Image (AMI)

Amazon Linux 2023 AMI — Free tier eligible

ami-02b49a24cfb95941c (64-bit (x86), uefi-preferred) / ami-04ad8c7fcc828fad4 (64-bit (Arm), uefi)

Virtualization: hvm ENA enabled: true Root device type: ebs

Description

Amazon Linux 2023 is a modern, general purpose Linux-based OS that comes with 5 years of long term support. It is optimized for AWS and designed to provide a secure, stable and high-performance execution environment to develop and run your cloud applications.

Architecture	Boot mode	AMI ID	
64-bit (x86)	uefi-preferred	ami-02b49a24cfb95941c	Verified provider

Instance type Info | Get advice

Instance type

t2.micro — Free tier eligible

Family: t2 1 vCPU 1 GiB Memory Current generation: true

On-Demand Linux base pricing: 0.0124 USD per Hour

On-Demand Windows base pricing: 0.017 USD per Hour

On-Demand RHEL base pricing: 0.0268 USD per Hour

All generations

Compare instance types

Key pair (login) Info

You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.

Key pair name - *required*

Select

Create new key pair

Network settings Info

VPC - *required* Info

vpc-00053d268636862c3 (default)
172.31.0.0/16

Subnet Info

No preference

Create new subnet

Auto-assign public IP Info

Enable

Additional charges apply when outside of free tier allowance

Firewall (security groups) Info

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

Create security group

Select existing security group

Security group name - *required*

launch-wizard-2

This security group will be added to all network interfaces. The name can't be edited after the security group is created. Max length is 255 characters. Valid characters: a-z, A-Z, 0-9, spaces, and ._-:/()#,@[]+=&;{}!$*

Description - *required* Info

launch-wizard-2 created 2024-08-29T19:00:44.460Z

Inbound Security Group Rules

Security group rule 1 (TCP, 22, 0.0.0.0/0)

Remove

Type Info

ssh

Protocol Info

TCP

Port range Info

22

Source type Info

Anywhere

Source Info

Add CIDR, prefix list or securi

0.0.0.0/0

Description - *optional* Info

e.g. SSH for admin desktop

Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

Add security group rule

Activity 2: Configure EC2 Instance

1. Connect to your EC2 instance using SSH.
2. Update the system and install necessary software:
   sudo yum update -y
   sudo yum install python3 python3-pip mysql -y
3. Install required Python packages:`

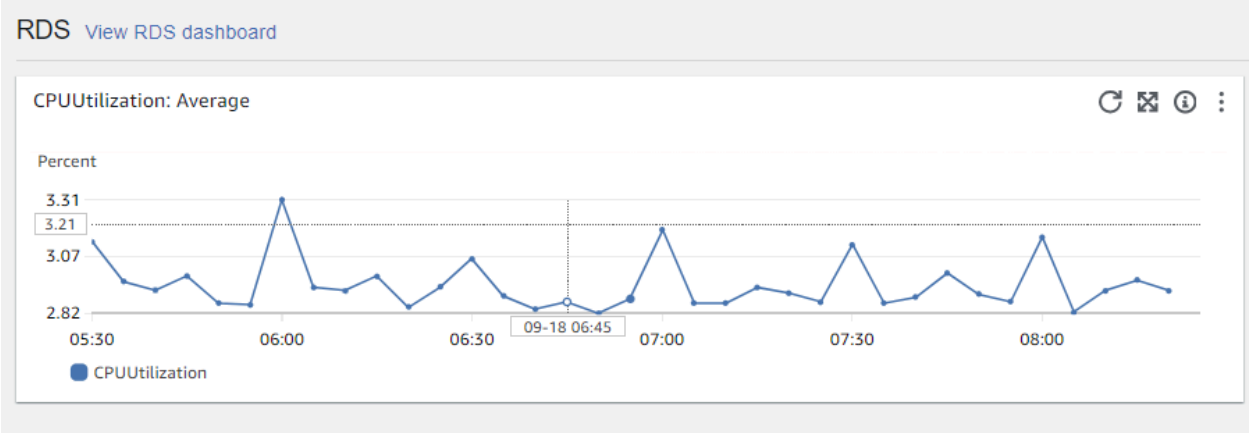
Activity 3: Deploy Flask Application to EC2

1. Transfer your application code to the EC2 instance.
2. Set up any necessary environment variables, including database connection strings.
3. Configure the web server to serve your application.
4. Start your application and ensure it's accessible via the EC2 instance's public IP or domain.
5. Run the below commands on ec2 terminal
6. sudo yum update -y
7. sudo yum install python3 -y
8. sudo pip3 install virtualenv
9. python3 -m venv venv
10. source venv/bin/activate
11. pip install flask
12. git clone https://github.com/your-repo/your-flask-app.git
13. cd your-flask-app
14. python3 app.py

Milestone 6: Monitoring and Management

Activity 1: Set Up CloudWatch Monitoring

1. Access the CloudWatch console from the AWS Management Console.
2. Create a new dashboard for your Easybank application.
3. Add widgets to monitor key metrics:
   - EC2 instance CPU utilization, network traffic, and status checks
   - RDS instance CPU utilization, free storage space, and database connections
   - Application-specific metrics (e.g., number of transactions, active users)
4. Set up CloudWatch Alarms for critical thresholds (e.g., high CPU usage, low free storage).



Activity 2: Configure CloudWatch Logs
1. Install and configure the CloudWatch Logs agent on your EC2 instance.
2. Set up log groups for your application logs, EC2 system logs, and Nginx logs (if applicable).
3. Create log filters to extract and analyze important log events.

Milestone 7: Testing, Optimization, and Maintenance

Activity 1: Conduct Thorough Testing
1. Perform functionality testing of all banking features.
2. Conduct security testing, including penetration testing if possible.
3. Perform load testing to ensure the application can handle expected traffic.

Activity 2: Optimize Performance
1. Analyze and optimize database queries.
2. Implement caching mechanisms where appropriate (e.g., Flask-Caching).
3. Optimize front-end assets (minify CSS/JS, optimize images).

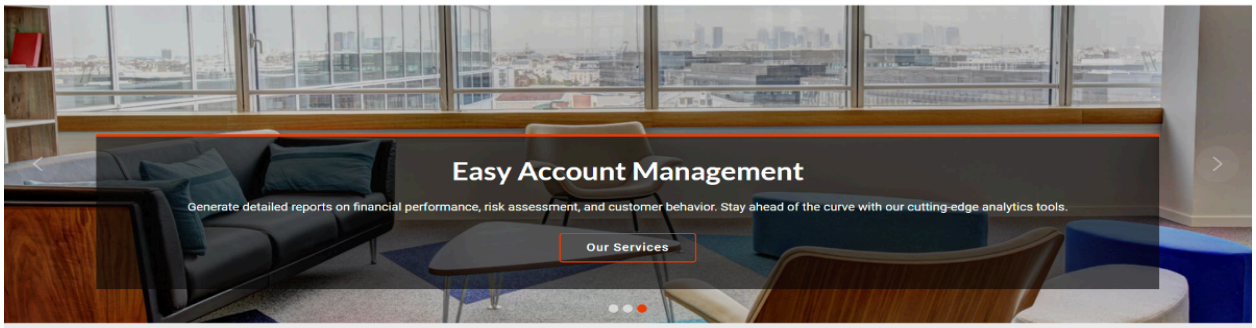
Activity 3: Implement Backup and Disaster Recovery
1. Configure automated RDS snapshots for database backups.
2. Set up a disaster recovery plan, including steps for failover to a different region if necessary.

Activity 4: Continuous Improvement
1. Regularly update EC2 instances and RDS with the latest security patches.
2. Monitor CloudWatch metrics and logs to identify areas for improvement.
3. Gather user feedback and iterate on the application features and user experience.
4. Continuously optimize AWS resource usage for cost-effectiveness.

INDEX.HTML:

EASYBANK

HOME SERVICES REGISTER LOGIN CONTACT



Financial Performance Analytics

Gain deep insights into your institution's financial health with comprehensive performance metrics and trend analysis.

Call To Action

Online Account Opening

Open a new bank account from the comfort of your home. It's quick, easy, and secure.

24/7 Account Access

Log into your account anytime, anywhere to manage your finances

View statements

Access your account statements online. View, download, or print them as needed.

Analyse your spends

Analyse your spends and give you insights on your monthly spend data.

Easy Fund Transfer

Transfer to anyone who has a valid account number within our bank with no extra fee

Easy Support

Get Easy customer support anytime and from anywhere and get your issues resolved within no time.

Partnered With Us

Flattern

MyBank

Your trusted partner for online banking services.

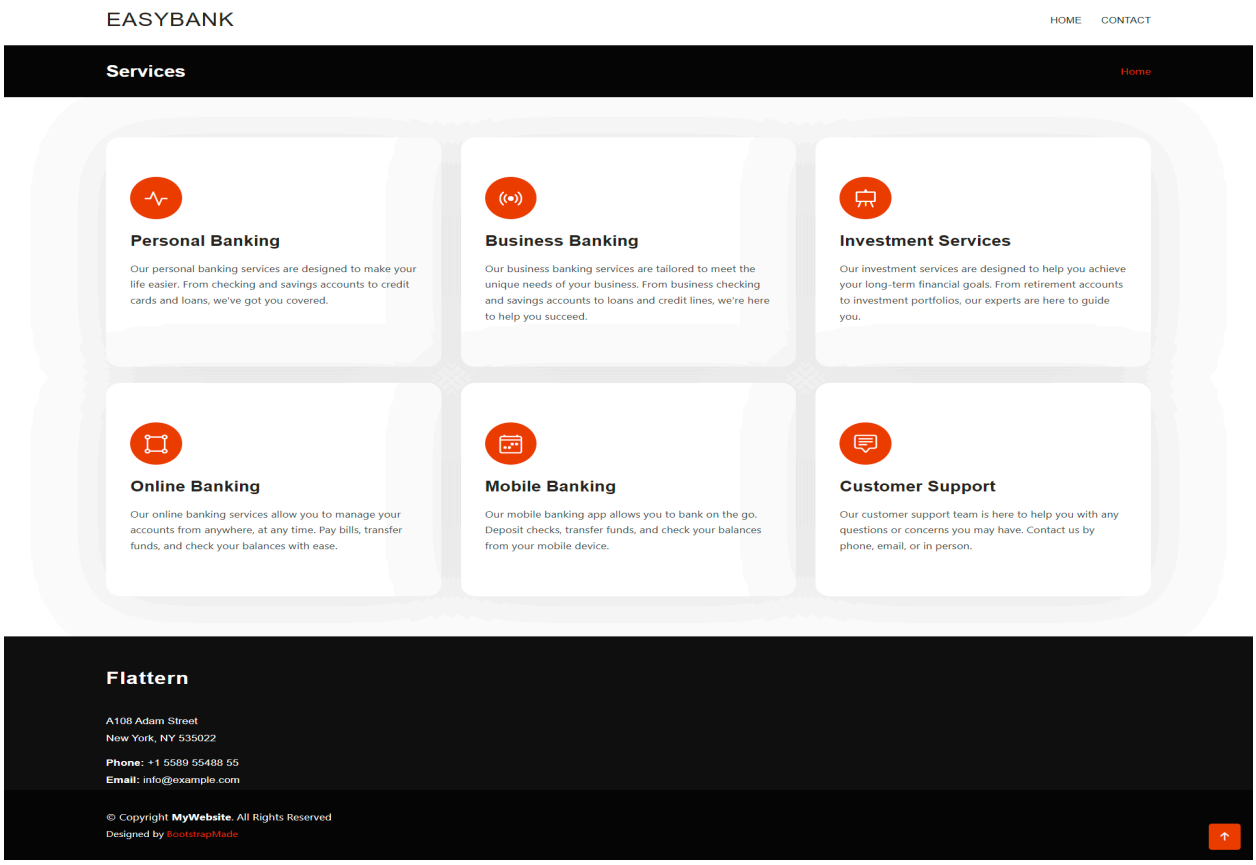
A108 Adam Street
New York, NY 535022

Phone: +91 554466 554466
Email: info@EasyBank.com

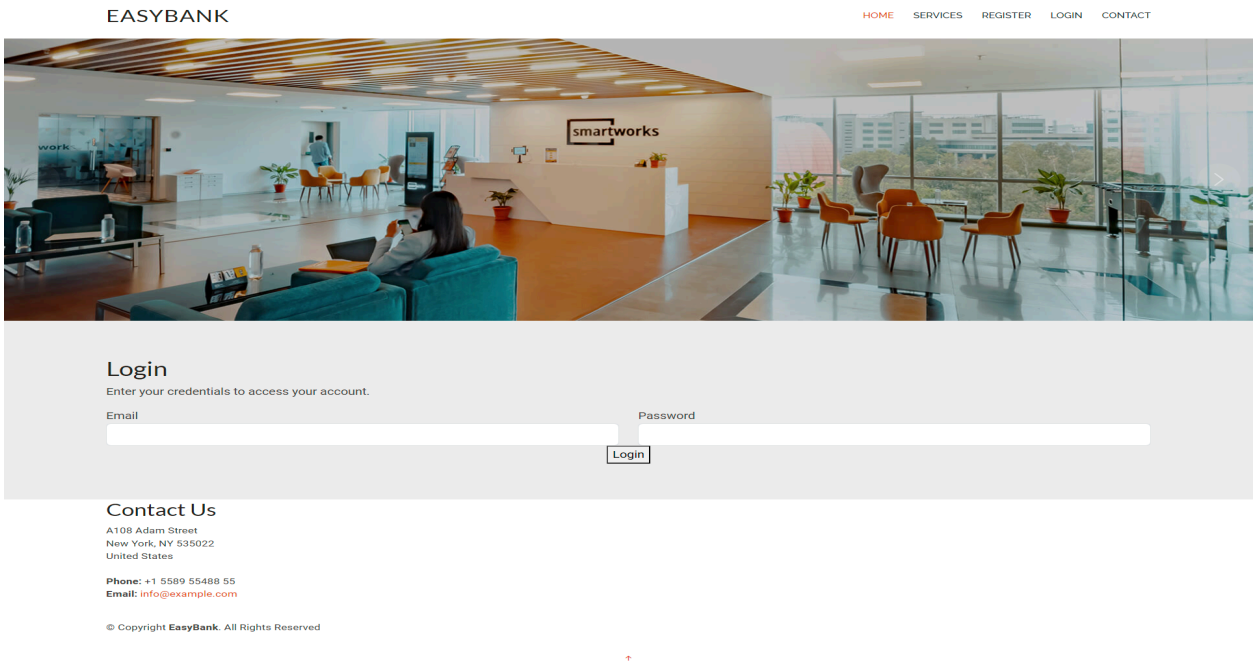
Useful Links

- Home
- About us
- Services
- Terms of service
- Privacy policy

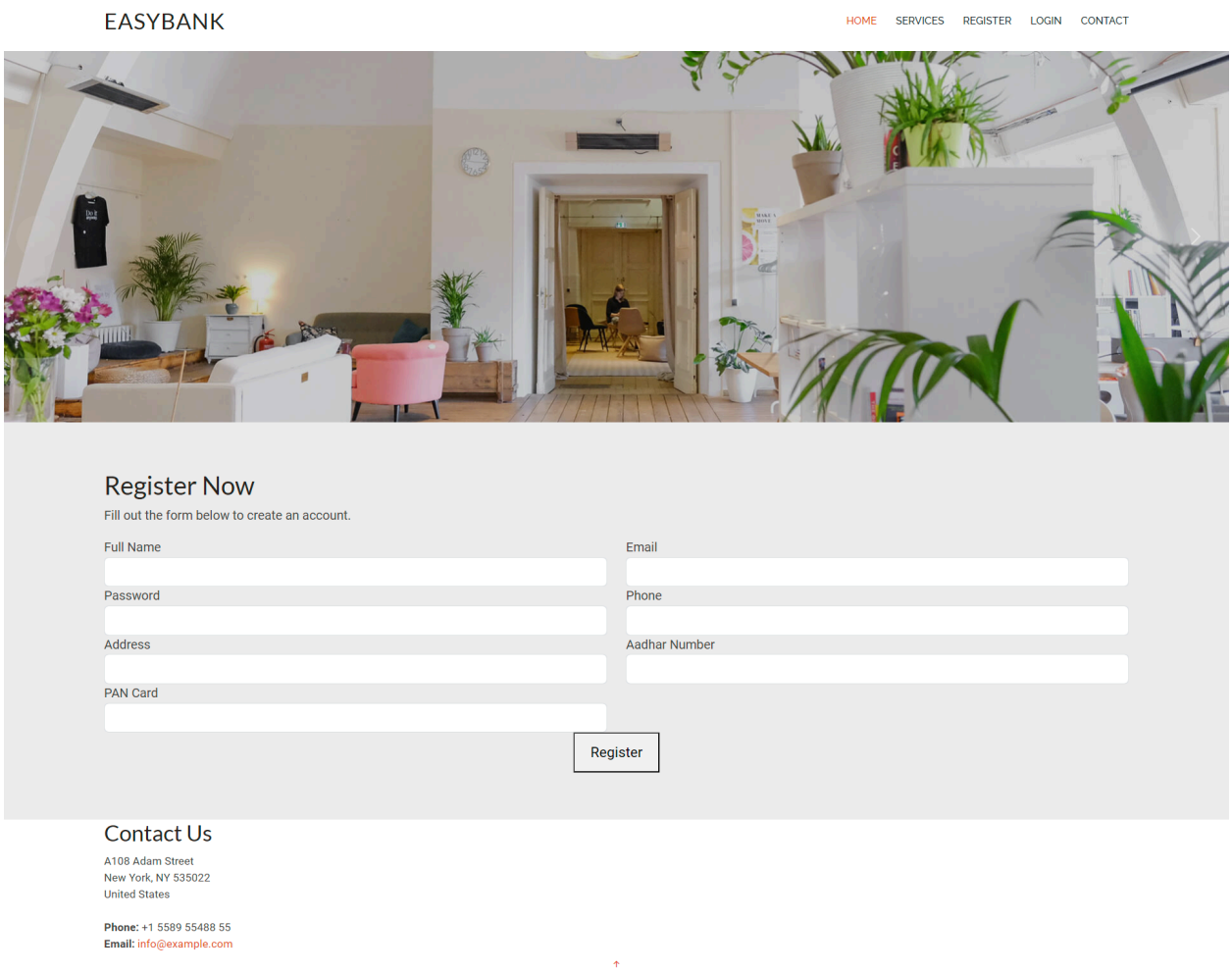
SERVICE.HTML:



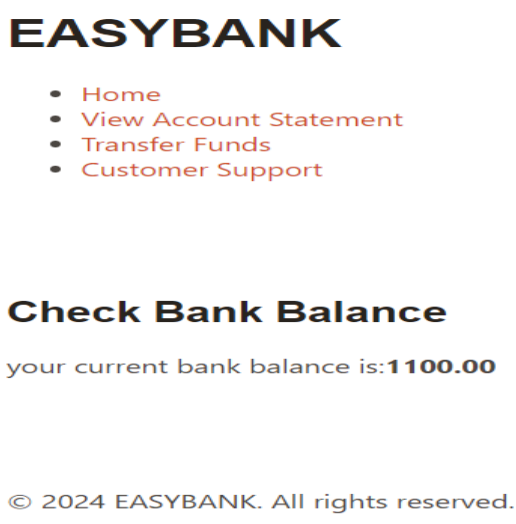
LOGIN.HTML:



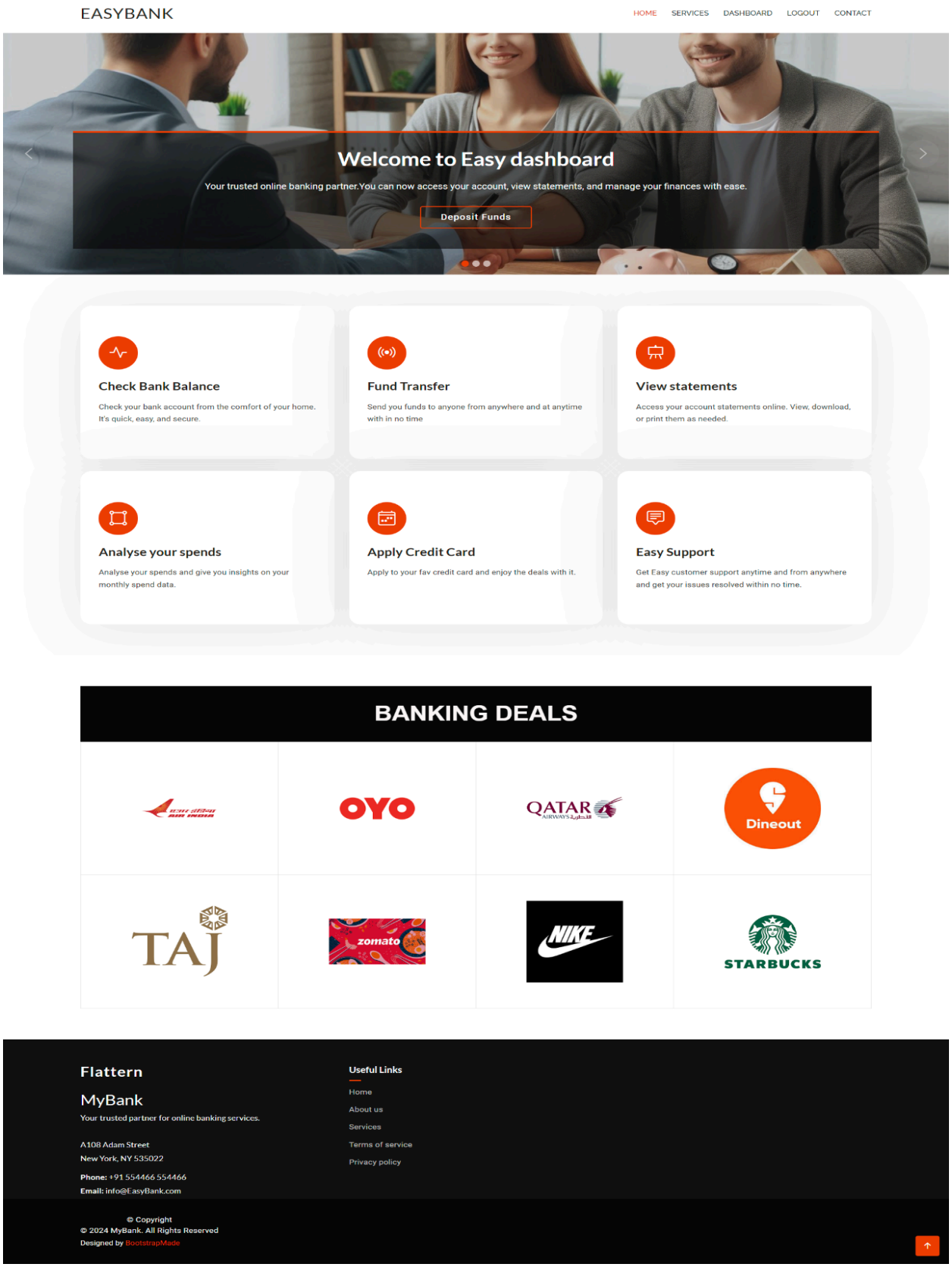
REGISTER.HTML:

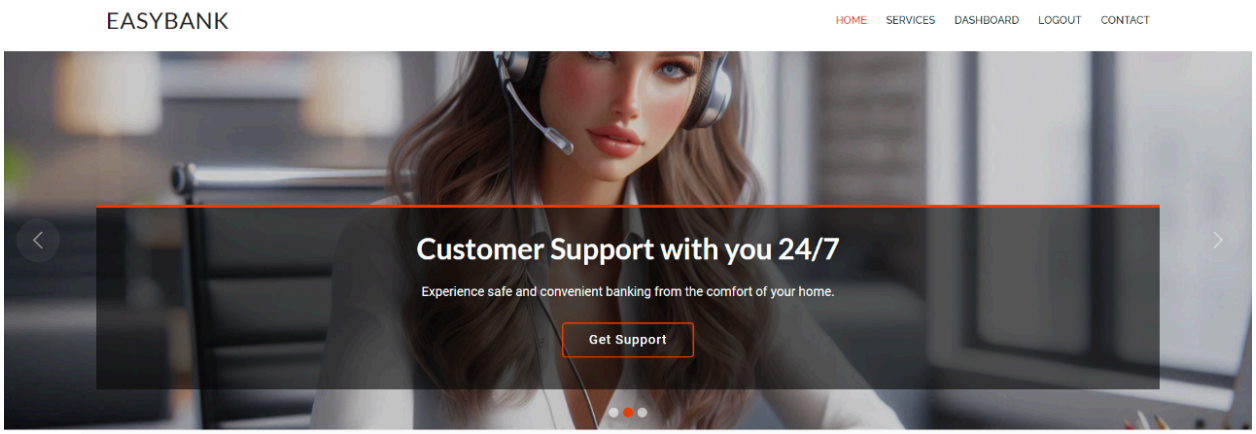


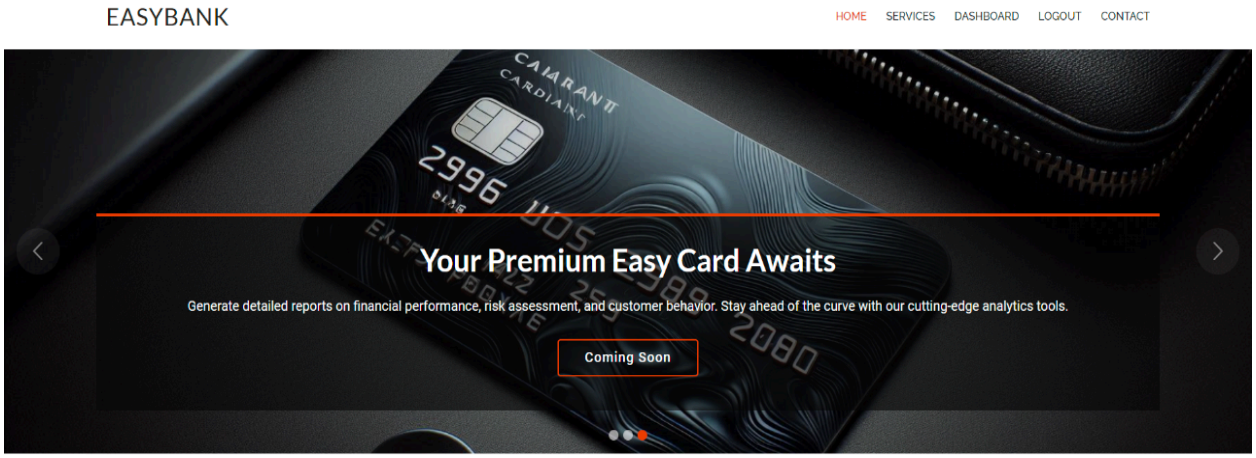
BALANCE.HTML:



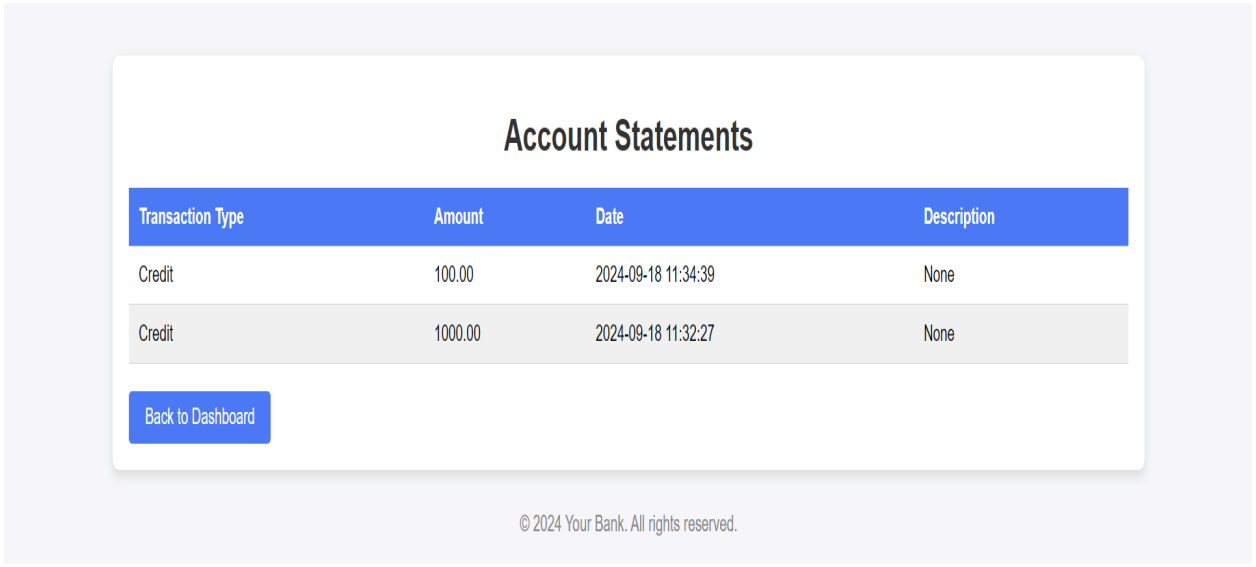
DASHBOARD.HTML:



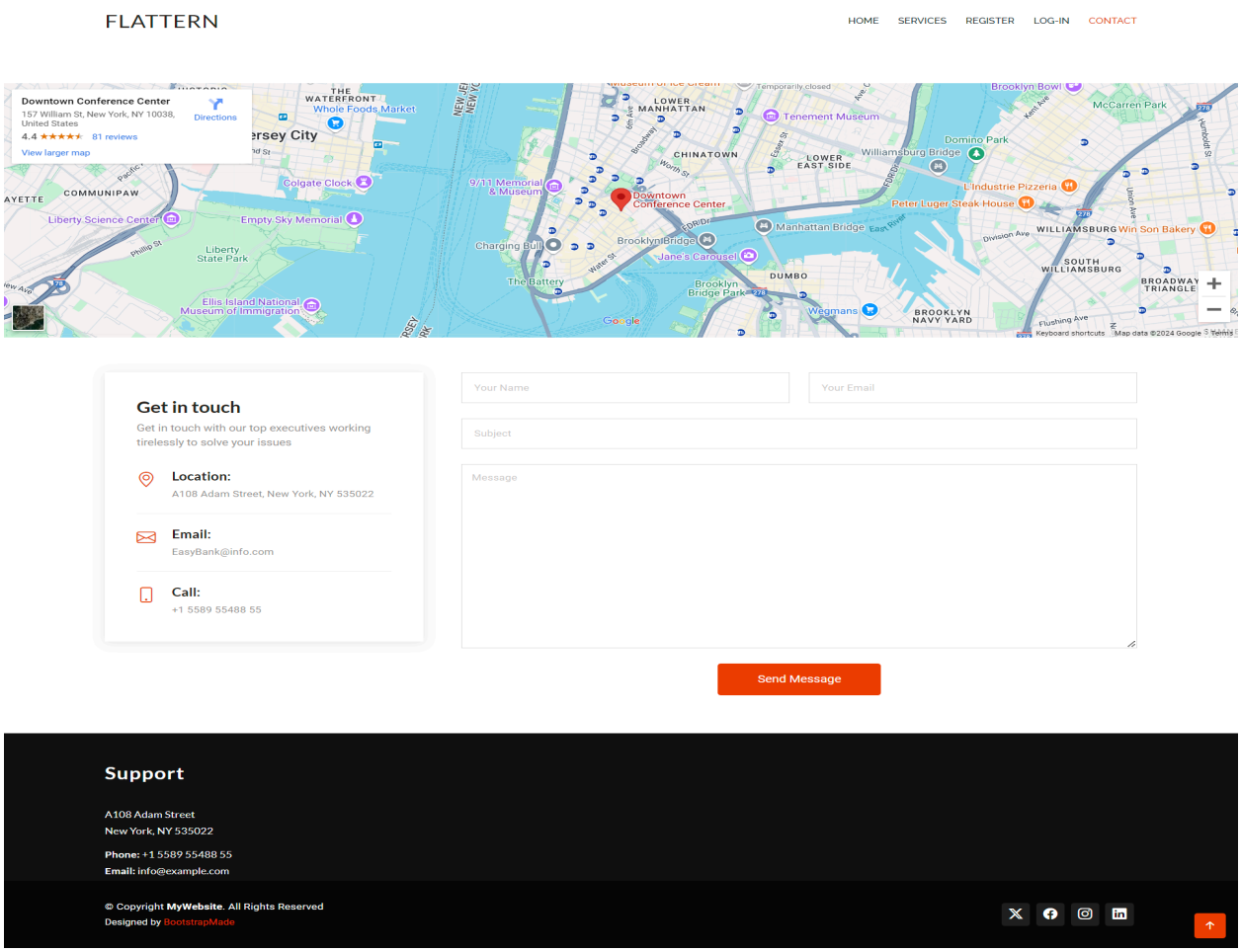




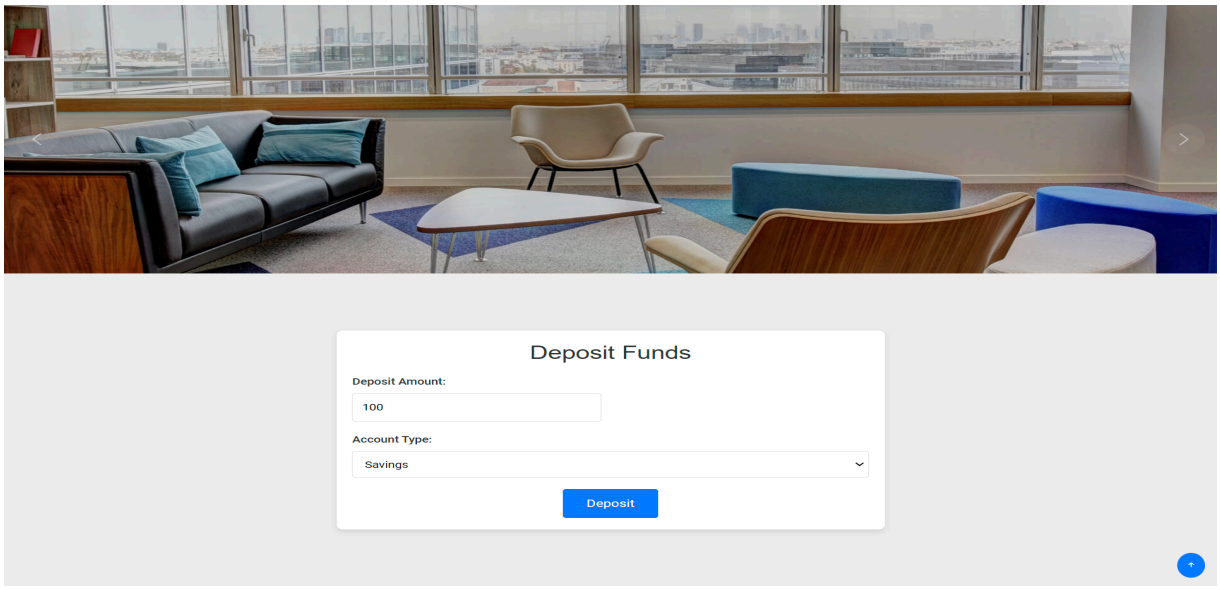
STATEMENTS.HTML:



CONTACT.HTML:



DEPOSIT.HTML:



CONFIRM.HTML:

EASYBANK

HOME SERVICES REGISTER LOGIN CONTACT

Account Created Successfully!

Welcome, akhil!

Your account has been created successfully. You can now log in to access our services.

Log in

Contact Us

A108 Adam Street
New York, NY 535022
United States

Phone: +1 5589 55488 55
Email: info@example.com

Useful Links

- Home
- About us
- Services
- Terms of service
- Privacy policy

TRANSFER.HTML:

Transfer Funds

Securely transfer funds between accounts with ease.

Funds transferred successfully!

Registration successful! Please log in.

Invalid login credentials!

Recipient's User ID:

Transfer Amount:

Transfer

CONCLUSION:

The Easybank project serves as a robust foundation for a cloud-based banking system. By leveraging AWS services, we've created a scalable, secure, and efficient platform that can adapt to the evolving needs of modern banking. This project not only demonstrates technical proficiency in cloud technologies but also highlights the complexities and considerations involved in developing financial applications in the cloud.

As the financial sector continues to embrace digital transformation, projects like Easybank pave the way for more innovative, secure, and user-friendly banking solutions. The lessons learned and the architecture developed here can serve as a valuable reference for future fintech projects and cloud-based applications in sensitive domains.