Case Study Solution: Consumer Services for Gas Utilities

Prepared by: Kushal Zinzuvadia

Executive Summary

This case study addresses the problem faced by a gas utility company experiencing a high volume of customer service requests. The proposed solution is a Django application designed to streamline customer service operations by enabling customers to submit service requests online, track their status, and access account information. Additionally, the application provides tools for customer support representatives to manage and resolve requests efficiently. This solution ensures improved customer satisfaction through reduced wait times and enhanced service quality.

Introduction

Background: A gas utility company is struggling to manage a large volume of customer service requests, leading to long wait times and dissatisfaction among customers.

Problem Statement: The current system cannot handle the volume of requests, causing delays and poor service delivery. A robust digital solution is required to streamline operations and improve customer satisfaction.

Scope and Goals: Develop a Django application to provide consumer services, enabling customers to submit service requests, track their status, and view account information, while equipping support representatives with tools to manage and resolve requests.

Problem Analysis

- **Key Issues Identified**:
    - Inability of the current system to handle high request volumes.
    - Lack of a centralized platform for customers to submit and track service requests.
    - Inefficiency in request management by customer support representatives.
- **Impact Assessment**:
    - Increased customer dissatisfaction.

- Delays in service resolution.
- Operational inefficiencies for customer support staff.

Proposed Solution

Solution Overview: A Django-based web application that addresses the identified issues through its robust architecture and features.

Technical Details:

1. **Architecture**:
    - A modular Django application with a well-organized codebase structured as follows:
        - consumer-services-portal
            - manage.py: Entry point for Django commands.
            - db.sqlite3: SQLite database for local development.
            - customer_service/: Main application folder containing customer-related functionalities:
                - models/: Contains customer.py and service_request.py for database models.
                - views/: Contains customer_views.py and track_request.py for handling customer actions.
                - forms/: Includes customer_edit_form.py and request_form.py for form handling.
                - templates/customer_service/: HTML templates for the frontend (e.g., customer_account.html, submit_request.html, track_request.html).
                - migrations/: Database migrations.
            - gas_utility_service/: Project-level folder with settings, URL routing, and WSGI/ASGI configuration.
2. **Database Design**:
    - Tables include Customer (user data) and ServiceRequest (service request details).
    - Relationships:

        - Customer (One-to-Many) → ServiceRequest.

3. **API Endpoints**:
    - POST /api/service-request/: Submit a new service request.
    - GET /api/service-request/<id>/: Retrieve details of a specific service request.
    - GET /api/customer/<id>/requests/: List all service requests for a customer.
    - PATCH /api/service-request/<id>/: Update the status of a service request.
4. **Frameworks and Tools**:
    - Backend: Django with modular design.
    - Frontend: Django templates for server-side rendering.
    - Database: SQLite (development) with PostgreSQL for production.
    - Deployment: Gunicorn, Nginx, and Docker for containerized deployment.

Implementation Plan:

- **Step 1**: Clone the repository from GitHub.
- **Step 2**: Set up a virtual environment and install dependencies using pip install -r requirements.txt.
- **Step 3**: Configure database settings for production in gas_utility_service/settings.py.
- **Step 4**: Run migrations using python manage.py migrate.
- **Step 5**: Test the application using python manage.py test.
- **Step 6**: Deploy the application using Docker and a CI/CD pipeline.

Alternative Solutions

1. **Custom-built application without Django**:
    - **Pros**: Tailored solution with no reliance on external frameworks.
    - **Cons**: Increased development time and cost.
2. **Using a third-party customer service platform**:
    - **Pros**: Quick deployment with pre-built features.

- **Cons**: Limited customization and potentially higher long-term costs.

Expected Outcomes

- **Performance Improvements**:
  - Reduction in customer wait times.
  - Streamlined request management for support representatives.
- **Metrics for Success**:
  - Average response and resolution times.
  - Number of successful service request submissions.
  - Customer satisfaction ratings.

Challenges and Mitigations

- **Challenge 1**: Ensuring the application handles high traffic during peak times.
  - **Mitigation**: Optimize database queries and implement caching.
- **Challenge 2**: Ensuring secure handling of user data and attachments.
  - **Mitigation**: Use Django's built-in security features and HTTPS.

Conclusion

The proposed Django application, as implemented in the GitHub repository, addresses the challenges faced by the gas utility company by providing a robust platform for customers and support representatives. With a well-structured codebase and scalable architecture, the solution ensures enhanced customer satisfaction and operational efficiency.