Project Documentation

For

Smart SWM System

Date: 01/04/2023

Contents

1. Document Management

1.1 Contributors

Sl No	Name
1	KARTHIK K PAI
2	ARYANJEET SINGH
3	CSV DEVISH

2 OVERVIEW

2.1 Problem Statement

Bangalore, also known as Bengaluru, is a rapidly growing city in India that has been facing various issues related to waste management and collection. The city has a limited number of waste collection trucks and staff, which makes it difficult to collect and dispose of waste effectively. In some areas, waste is collected irregularly or not at all, leading to piling up of garbage on the streets.Due to the lack of proper waste management facilities and services, many people resort to dumping their waste in unauthorised areas, such as vacant lots or open spaces. This leads to health hazards, environmental pollution, and the spread of diseases. There is a lack of monitoring and accountability, which leads to the continuation of improper waste disposal practices. Lack of credibility and monitoring systems in the conductor system is also highly contributing.

2.2 Solution

Presenting a centralised solution that aims to solve the issue of contractor credibility and holds the contractor liable for different issues faced. The SWM Management tool is a good fusion of hardware and software as well as a front end app that improves citizen usability. A centralised dashboard ensures efficient monitoring so that all issues can be solved.

2.3 Functional Requirements

1. New route addition: Creation of new routes on dashboard in order to facilitate waste collection

2. Waste collection centre addition: Addition of specialised waste collection centres for categories like Ewaste for drop offs.

3. Real time tracking of collection vehicles: Use of embedded systems in collection vehicles to track location and compute delay.

4. Wrong trash dump tracking: Weight sensor embedded on vehicles to ensure waste is not dumped anywhere else other than the stipulated location.

5. Citizen friendly app: App from which citizens can track municipality vehicles to drop trash off from the comfort of their houses.

2.4 Non Functional Requirements

Performance: The waste management application must perform efficiently and respond quickly to user requests and agent requests . It should be able to handle a large volume of data and transactions without slowing down.

Security: The application must ensure the security and privacy of user data, as it may contain sensitive information. It should have robust security measures, such as encryption, access control, and user authentication. Sensors systems while transmitting data must take security measures to avoid man in the middle.

Reliability: The application must be reliable and available at all times, as any downtime or system failure can disrupt waste management operations. It should have backup and recovery mechanisms to ensure that data is not lost in case of a system failure.

Usability: The application must be user-friendly and easy to use, with a simple and intuitive interface. It should provide clear instructions and feedback to users and have features like search, filtering, and sorting to make it easy to find and manage waste-related data.

Scalability: The application should be scalable and able to handle an increasing number of users, waste management sites, and data volumes. It should be designed to accommodate future growth and changes in waste management policies and practices.

2.5 Code , Description and Component details

```python
from django.shortcuts import render
from django.shortcuts import render
import matplotlib.pyplot as plt

from io import StringIO
import folium
import datetime
from .models import static_route, collection_location, realtime_location

def maps(request):

    points = static_route.objects.all()
    f = folium.Figure(width = 900, height = 800)

    m = folium.Map(location= [12.972442, 77.580643], tiles="openstreetmap",
    zoom_start=-12, min_zoom = 11).add_to(f)

    all_points = []

    for pin_points in points:
        coordinates = (pin_points.Latitude, pin_points.Longitude)
        all_points.append(coordinates)
        folium.Marker(coordinates).add_to(m)

    folium.PolyLine(all_points, color='red', weight=2, opacity=0.8).add_to(m)


    if request.method == "POST":
        lat = float(request.POST.get("lat"))
        lon = float(request.POST.get("lon"))
        time = request.POST.get("time")
        stud = static_route.objects.all()

        static_route.objects.create(Time = time, Latitude = lat, Longitude = lon)

    return render(request, "maps.html", {'maps' : m._repr_html_()})

```

```python
"""SegFault URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/4.1/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path
from SWM import views
from django.conf import settings
from django.conf.urls.static import static
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name = "index"),
    path('maps/', views.maps, name = "maps"),
    path('collection/', views.collection, name = "collection"),
    path('display/', views.display, name = "display"),
    path('live_dashboard/', views.live_dashboard, name = "live_dashboard"),
]
```

```python
from django.db import models


class static_route(models.Model):
    Time = models.TimeField()
    Latitude = models.FloatField()
    Longitude = models.FloatField()

    def __str__(self):
        return '%s %s %s' % (self.Time, self.Latitude, self.Longitude)


class realtime_location(models.Model):
    Date = models.DateField(auto_now_add=True)
    Time = models.TimeField()
    Latitude = models.FloatField()
    Longitude = models.FloatField()
    Weight = models.FloatField()

    def __str__(self):
        return '%s %s %s %s %s' % (self.Date, self.Time, self.Latitude, self.Longitude, self.Weight)

class collection_location(models.Model):

    waste=(
        ("CW", "Cloth Waste"),
        ("GW", "Glass Waste"),
        ("EW", "Electronic Waste")
    )

    Garbage_type = models.CharField(choices=waste,max_length = 10)

    Name = models.CharField(max_length=20)
    Phone = models.IntegerField()
    Start_date = models.DateField()
    End_date = models.DateField()
    Start_time = models.TimeField()
    End_time = models.TimeField()
    Latitude = models.FloatField()
    Longitude = models.FloatField()

    def __str__(self):
        return '%s %s %s %s %s %s %s %s %s %s' % (self.Name, self.Phone, self.Start_date, self.End_date, self.Start_time, self.End_time, self.Latitude, self.Longit
```

2.6 Limitations

- Accuracy of GPS
- How are interfaces setup (on demand, push, pull, nightly)?
- Are there any web services or APIs the application provides?

2.7 Scope of work

- Development of user interface
- Feedback collection
- Crowd sourced information
- Efficient routing

2.8 Financial information

- Cost of hardware -1100/-
- Costing cost for scaling

2.9 End User Details

- agents
  manages the system
- citizens
  accesses the benefits of the system

2.10 Prototype to Market

- use of industrial sensors
- improvement of user interface
- multi platform scalability
- adherence to sensor data transmission protocol