Table of content

1 Setting up the project files

* 1. structure logic
  2. file structure
  3. initiating docker files

1. Setting up backend
   1. creating project
   2. installing dependencies
   3. testing backend

**Building a Charity Management System: Flutter, Django REST Framework, and Docker**

**Project Overview**

This project aims to develop a comprehensive charity management system utilizing Flutter for the frontend, Django REST Framework for the backend, and Docker for containerization. The system will enable charities to efficiently manage their activities, including donor management, volunteer management, project management, financial tracking, and reporting.

**Backend (Django REST Framework)**

**Model Structure**

**Core Models:**

* **CharityProfile:** Contains details about the charity (name, mission, vision, contact information).
* **User:** Represents users (donors, volunteers, staff) with roles and permissions.
* **Project:** Defines charity projects with goals, budget, and status.
* **Donor:** Stores donor information, donation history, and preferences.
* **Volunteer:** Manages volunteer details, skills, availability, and assignments.
* **Expense:** Tracks financial expenditures for projects.
* **Income:** Records income sources (donations, grants).

**Additional Models:**

* **News/Announcement:** For sharing updates with stakeholders.
* **Event:** Manages charity events and registrations.
* **Report:** Generates various reports (financial, donor, volunteer, project).

**API Endpoints**

* User authentication and authorization
* Charity profile management
* Project CRUD operations
* Donor management (registration, donation history, preferences)
* Volunteer management (registration, availability, assignments)
* Expense and income tracking
* Report generation
* News and announcements
* Event management

**Key Features**

* Robust user authentication and authorization with role-based access control.
* Efficient data management for donors, volunteers, and projects.
* Comprehensive financial tracking with expense and income categorization.
* Customizable reporting capabilities for various stakeholders.
* Integration with payment gateways for secure donations.
* Notification system for donors, volunteers, and staff.

**Frontend (Flutter)**

**User Interface**

* **Dashboard:** Provides an overview of key metrics (donations, volunteers, projects).
* **Donor Management:** Allows managing donor information, tracking donations, and sending thank-you messages.
* **Volunteer Management:** Handles volunteer recruitment, scheduling, and task assignments.
* **Project Management:** Enables creating, updating, and tracking project progress.
* **Financial Management:** Facilitates expense tracking, income recording, and generating financial reports.
* **Reporting:** Offers various report templates for analysis and decision-making.
* **News and Announcements:** Displays updates and notifications.
* **Event Management:** Creates and manages charity events.

**Key Features**

* Intuitive and user-friendly interface.
* Responsive design for different screen sizes.
* Offline capabilities for data synchronization.
* Integration with backend API for data retrieval and updates.
* Secure data storage and transmission.

**Dockerization**

* Create a Dockerfile for the backend and frontend separately.
* Define dependencies, build environment, and entry points.
* Create a Docker Compose file to manage both containers.
* Configure network settings and environment variables.

**Deployment**

* Choose a suitable cloud platform (AWS, GCP, Azure).
* Deploy the Dockerized application using container orchestration tools (Kubernetes, Docker Swarm).
* Implement continuous integration and continuous deployment (CI/CD) pipelines.
* Monitor application performance and resource utilization.

**Additional Considerations**

* **Data Security:** Implement robust security measures to protect sensitive data (encryption, access controls, regular audits).
* **Scalability:** Design the system to handle increasing user loads and data volumes.
* **Accessibility:** Ensure the application is accessible to users with disabilities.
* **Testing:** Conduct thorough testing (unit, integration, system, user acceptance) to identify and fix issues.
* **Maintenance:** Establish regular maintenance and update schedules.

**Project Roadmap**

1. **Backend development:** Define models, create API endpoints, implement authentication, and core functionalities.
2. **Frontend development:** Design UI, integrate with backend API, develop core features.
3. **Dockerization:** Create Dockerfiles and Compose file, test containerization.
4. **Testing:** Conduct thorough testing of both frontend and backend.
5. **Deployment:** Deploy the application to a cloud platform.
6. **Iterative improvement:** Gather user feedback, implement enhancements based on requirements.

By following this structured approach, you can develop a robust and scalable charity management system that effectively supports the organization's mission.

**Would you like to delve deeper into any specific aspect of this project?**

**1.3 File Structure and Dockerfiles for Charity Management System**

**Project Structure**

Let's break down the project into two main directories: backend and frontend.

**Backend (Django REST Framework)**

backend/

├── charity\_management/

│ ├── \_\_init\_\_.py

│ ├── settings.py

│ ├── urls.py

│ ├── asgi.py

│ ├──wsgi.py

│ ├── db.sqlite3

│ ├── manage.py

│ ├── charity\_app/

│ │ ├── \_\_init\_\_.py

│ │ ├── models.py

│ │ ├── serializers.py

│ │ ├── views.py

│ │ ├── migrations/

│ │ └── tests/

│ ├── static/

│ └── templates/

└── Dockerfile

* **charity\_management:** The main Django project directory.
* **charity\_app:** The Django app containing models, serializers, views, and other app-specific logic.
* **static:** For static files (CSS, JavaScript, images).
* **templates:** For Django templates.
* **Dockerfile:** Defines the Docker image for the backend.

**Frontend (Flutter)**

frontend/

├── lib/

│ ├── main.dart

│ ├── screens/

│ │ ├── home\_screen.dart

│ │ ├── donor\_screen.dart

│ │ ├── ...

│ ├── components/

│ │ ├── custom\_button.dart

│ │ ├── ...

│ ├── models/

│ │ ├── user.dart

│ │ ├── donation.dart

│ │ ├── ...

│ ├── services/

│ │ ├── api\_service.dart

│ │ ├── ...

│ ├── utils/

│ │ ├── constants.dart

│ │ ├── helpers.dart

│ │ ├── ...

└── Dockerfile

* **lib:** Contains the Flutter application code.
* **Dockerfile:** Defines the Docker image for the frontend.

**Dockerfiles**

**Backend Dockerfile**

Dockerfile

# Use a lightweight Python base image

FROM python:3.9-slim-buster

# Set the working directory

WORKDIR /app

# Copy the requirements file

COPY requirements.txt   requirements.txt

# Install   dependencies

RUN pip install -r requirements.txt

# Copy the rest of the code

COPY . .

# Set environment variables (optional)

ENV DJANGO\_SETTINGS\_MODULE=charity\_management.settings

# Command to run the Django application

CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]

**Frontend Dockerfile**

Dockerfile

# Use a Flutter base image

FROM dart:latest

# Set the working directory

WORKDIR /app

# Copy the Flutter project

COPY . .

# Install dependencies

RUN flutter pub get

# Command to run the Flutter application

CMD ["flutter", "run"]

**Docker Compose**

To run both containers together, use a docker-compose.yml file:

YAML

version: '3.7'

services:

backend:

build: ./backend

ports:

- "8000:8000"

volumes:

- ./backend:/app

frontend:

build: ./frontend

ports:

- "3000:3000"

depends\_on:

- backend

**Additional Considerations**

* **Environment Variables:** Use environment variables to store sensitive information like database credentials.
* **Data Persistence:** Consider using volumes to persist data (e.g., database files).
* **Image Optimization:** Minimize image size by using multi-stage builds and excluding unnecessary files.
* **Security:** Follow best practices for Docker image security (e.g., use non-root user, keep images up-to-date).

This structure provides a solid foundation for your charity management system. You can adapt it based on your specific project requirements and preferences.

**Would you like to dive deeper into any specific part of the structure or Dockerfiles?**

## Backend Setup: Django REST Framework

### Prerequisites

* Python and pip installed
* Basic understanding of Django and RESTful APIs

### Project Initialization

1. **Create a new Django project:**

Bash

django-admin startproject charity\_management

1. **Create a Django app:**

Bash

cd charity\_management

python manage.py startapp charity\_app

### Install Required Packages

Bash

pip install djangorestframework

### Configure Django Settings

1. **Add 'charity\_app' to INSTALLED\_APPS in charity\_management/settings.py:**

Python

INSTALLED\_APPS = [

# ...

'charity\_app',

'rest\_framework',

]

1. **Set up REST framework:**

Python

REST\_FRAMEWORK = {

# Configure default permissions and authentication here

}

### Create Models

In charity\_app/models.py, define your models:

Python

from django.db import models

class CharityProfile(models.Model):

name = models.CharField(max\_length=255)

mission = models.TextField()

vision = models.TextField()

# ... other fields

class Donor(models.Model):

name = models.CharField(max\_length=255)

email = models.EmailField()

# ... other fields

# ... other models

### Create Serializers

In charity\_app/serializers.py, define serializers to convert models to JSON:

Python

from rest\_framework import serializers

from .models import CharityProfile, Donor

class CharityProfileSerializer(serializers.ModelSerializer):

class Meta:

model = CharityProfile

fields = '\_\_all\_\_'

class DonorSerializer(serializers.ModelSerializer):

class Meta:

model = Donor

fields = '\_\_all\_\_'

### Create Views

In charity\_app/views.py, create viewsets to handle API requests:

Python

from rest\_framework import viewsets

from .serializers import CharityProfileSerializer, DonorSerializer

from .models import CharityProfile, Donor

class CharityProfileViewSet(viewsets.ModelViewSet):

queryset = CharityProfile.objects.all()

serializer\_class = CharityProfileSerializer

class DonorViewSet(viewsets.ModelViewSet):

queryset = Donor.objects.all()

serializer\_class = DonorSerializer

### Register API Endpoints

In charity\_management/urls.py, include the API URLs:

Python

from django.contrib import admin

from django.urls import path, include

from rest\_framework import routers

from charity\_app import views

router = routers.DefaultRouter()

router.register(r'charity\_profiles', views.CharityProfileViewSet)

router.register(r'donors', views.DonorViewSet)

urlpatterns = [

path('admin/', admin.site.urls),

path('', include(router.urls)),

]

### Run the Development Server

Bash

python manage.py runserver

This is a basic setup. You can now access your API endpoints at http://127.0.0.1:8000/charity\_profiles/ and http://127.0.0.1:8000/donors/.

**Next steps:**

* Implement authentication and authorization.
* Create more complex models and serializers.
* Optimize API performance.
* Add error handling and validation.

**Would you like to proceed with implementing authentication and authorization?**