Django Project Setup Guide

Create a Django project called Growbal having a typical authentication, user management and routing systems with two models (User management info (type of user -i.e. Investors, Entrepreneurs-, location, session status, tier, privileges and permissions, etc.), and service data containing web scrapped data with fields like Company Name, Logo, Country, Website, Social media links (LinkedIn, Facebook, Instagram), Services offered, Contact numbers, Emails, Office locations, Ratings, Key Individuals, Pricing, and Service Descriptions). The Django project should connect to a PosgreSQL database.

I want ready to use code, files, and commands for setting up the above project.

To generate a fully working Django project tailored to your needs, could you please clarify a few things:

1. Should user registration and login include email verification and password reset flows?
2. Do you need Django admin customization for managing users and service data?
3. For the web-scraped service data, do you want a script to simulate scraping from a particular source (e.g., Crunchbase, LinkedIn)?
4. What operating system and Python version are you using for deployment?
5. Do you want this project to use Django REST Framework for APIs or is it primarily for server-rendered HTML views?

Once I have these details, I'll generate everything you need to get started.

↪ For the web-

Should user registration and login include email verification and password reset flows? No, keep it simple for now

Do you need Django admin customization for managing users and service data? Yes I want to use Django's admin panel, etc.

For the web-scraped service data, do you want a script to simulate scraping from a particular source (e.g.,

Crunchbase, LinkedIn)? No, assume that you have already built a web scraping logic, he web scraping task could be handled by that web scraping module. But make sure to reserve files and code blocks for that built logic, and consider incorporating it into the project.

What operating system and Python version are you using for deployment? Python 3.11.11 on Ubuntu

Do you want this project to use Django REST Framework for APIs or is it primarily for server-rendered HTML views? Use Django REST Framework for building API endpoints

Great. I'll create a Django project called Growbal with PostgreSQL, Django REST Framework, and admin support. It will include:

- A user management system with user types (Investor, Entrepreneur), location, session status, tier, privileges, and permissions
- A model for web-scraped service data (with all specified fields)
- Django admin setup for both models
- API endpoints using Django REST Framework
- Reserved structure and files for a future scraping module
- Compatible setup for Python 3.11.11 on Ubuntu

I'll generate the full directory structure, settings, models, serializers, URLs, and necessary commands for setting up the project.

Growbal Django Project with User Management and Service Data Models

Growbal Django Project: Setup and Configuration

In this guide, we create a **Django** project called **"Growbal"** using Python 3.11.11 on Ubuntu. We will configure it to use a **PostgreSQL** database. The project includes two main apps: one for **user management** (with authentication and profiles) and one for **service data** (information acquired via web scraping). We also set up an admin interface and expose **REST API** endpoints with token-based authentication. All necessary configuration, code, and commands are provided for a ready-to-use project.

Project Setup and Dependencies

To begin, make sure you have **Python 3.11** and **PostgreSQL** installed on Ubuntu. We'll use **Django 4.x** (latest at time of writing) and the Django REST Framework. Install the required packages using pip:

- **Django** (web framework) – `pip install Django`
- **psycopg2-binary** (PostgreSQL adapter) – `pip install psycopg2-binary` stackoverflow.com
- **djangorestframework** (DRF for APIs) – `pip install djangorestframework` pypi.org
- **djangorestframework-simplejwt** (optional, for JWT auth if needed) – *not used in this project since we use DRF's built-in tokens*
- **Pillow** (image processing, required for ImageField) – `pip install Pillow` (Django's `ImageField` requires Pillow geeksforgeeks.org)

With dependencies installed, create the Django project and two apps:

```bash
django-admin startproject growbal .
cd growbal
python manage.py startapp accounts      # app for user profiles
python manage.py startapp services     # app for scraped service data
```

This will create a project directory `growbal/` and two app directories `accounts/` and `services/`. We will also prepare a placeholder `scraper/` folder for future scraping scripts.

Next, open `growbal/settings.py` to configure the project:

- **Installed Apps**: Add `'accounts'`, `'services'`, `'rest_framework'`, and `'rest_framework.authtoken'` to `INSTALLED_APPS`. (The `authtoken` app is needed for token authentication django-rest-framework.org .) Django's default auth apps (like `'django.contrib.auth'` and `'django.contrib.admin'`) should already be present.
- **Database**: Configure the default database to use PostgreSQL. For example:

  ```python
  DATABASES = {
      'default': {
          'ENGINE': 'django.db.backends.postgresql',
          'NAME': 'growbal_db',
          'USER': 'postgres',
          'PASSWORD': 'your_db_password',
          'HOST': 'localhost',
          'PORT': '5432',
      }
  }
  ```

  This format is based on Django's standard PostgreSQL settings w3schools.com . Make sure to replace the database name, user, and password with your own. Ensure that a PostgreSQL database with the given name exists and the user has access (you can create a DB and user in PostgreSQL before proceeding).
- **Media Files**: Since we will upload company logos, add media settings in `settings.py`:

  ```python
  MEDIA_URL = '/media/'
  MEDIA_ROOT = BASE_DIR / 'media'
  ```

  This will store uploaded images in a `media/` directory. (During development, you can serve these files by adding `urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)` when `DEBUG=True`.)
- **Django REST Framework**: Configure DRF to use token authentication globally:

  ```python
  REST_FRAMEWORK = {
      'DEFAULT_AUTHENTICATION_CLASSES': [
          'rest_framework.authentication.TokenAuthentication',
  ```

```
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ]
}
```

This ensures all API requests require a valid token by default geeksforgeeks.org . We included `rest_framework.authtoken` in INSTALLED_APPS and will run migrations to create the token model django-rest-framework.org .

With these settings in place, run the initial migrations and create a superuser:

```bash
python manage.py makemigrations   # generate migrations for our apps
python manage.py migrate          # apply migrations (creates tables including auth tables)
python manage.py createsuperuser  # create an admin user for login
```

Now let's implement each part of the project in detail.

1. Authentication and User Management

User model and profile: We will use Django's built-in **User** model for authentication (username, password, email, etc.) and extend it with a **profile model** to store extra fields. Django's documentation recommends using a one-to-one linked model (profile) to add non-auth fields to the user docs.djangoproject.com . Our `accounts/models.py` will have:

```python
# accounts/models.py
from django.db import models
from django.contrib.auth.models import User

class UserProfile(models.Model):
    # Link to the built-in User model
    user = models.OneToOneField(User, on_delete=models.CASCADE, related_name='profile')
    # Additional profile fields
    USER_TYPES = [
```

```
        ('Investor', 'Investor'),
        ('Entrepreneur', 'Entrepreneur'),
    ]
    user_type = models.CharField(max_length=20, choices=USER_TYPES)
    location = models.CharField(max_length=100, blank=True)
    # Session status: e.g., whether the user is active in a session (online/offline)
    STATUS_CHOICES = [('active', 'Active'), ('inactive', 'Inactive')]
    session_status = models.CharField(max_length=8, choices=STATUS_CHOICES,
default='inactive')
    tier = models.CharField(max_length=50, blank=True)  # e.g., membership level
    # (Privileges and permissions will be handled by Django's auth
groups/permissions)

    def __str__(self):
        return f"{self.user.username} Profile"
```

This `UserProfile` is linked one-to-one with `User`. It includes fields for **user_type** (Investor or Entrepreneur), **location**, **session_status** (which we define as Active/Inactive for simplicity), and **tier**. We rely on Django's built-in permission system for privileges (the User model has flags like `is_staff`, `is_superuser` and supports group/permission assignments).

Whenever a new User is created, you can create a corresponding UserProfile. This can be done manually in the admin or automatically using a signal (e.g., Django's post_save signal on User model) – for brevity we won't include the signal code here, but it's a common practice.

Authentication views (login/logout): Django provides out-of-the-box views for logging in and out. We can enable them by including Django's auth URLs in the project `urls.py` . For example:

python

```
# growbal/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')),  # Enable login/logout
views
    path('api/', include('api.urls')),  # (We'll set up API routes in a separate
```

```
urls module)
]
```

Including `django.contrib.auth.urls` will automatically set up routes like `/accounts/login/` and `/accounts/logout/` using Django's default views and templates for authentication. This covers basic login/logout functionality (no email verification or password reset, which suits our requirements). Users can log in with their username and password (for the superuser or any users you create via admin).

Note: By default, only users with `is_staff=True` can log into the admin site. Our superuser will have that. Regular users (Investors/Entrepreneurs) can use the `/accounts/login/` page if needed for non-admin authentication flows.

2. Service Data Model

Next, we create a model to represent the **service data** gathered from web scraping. This model will store various details about companies and their services. In `services/models.py`:

python

```python
# services/models.py
from django.db import models

class ServiceData(models.Model):
    company_name = models.CharField(max_length=255)
    logo = models.ImageField(upload_to='logos/', blank=True, null=True)  # Company
logo image
    country = models.CharField(max_length=100, blank=True)
    website = models.URLField(blank=True, null=True)
    linkedin = models.URLField(blank=True, null=True)
    facebook = models.URLField(blank=True, null=True)
    instagram = models.URLField(blank=True, null=True)
    services_offered = models.TextField(blank=True)      # description or list of
services
    contact_numbers = models.TextField(blank=True)       # phone numbers (possibly
multiple)
    emails = models.TextField(blank=True)                # contact emails (possibly
multiple)
    office_locations = models.TextField(blank=True)      # addresses (possibly
multiple)
```

```
    ratings = models.CharField(max_length=50, blank=True)  # e.g. "4.5/5" or other
rating info
    key_individuals = models.TextField(blank=True)       # names of key people
    pricing = models.TextField(blank=True)               # pricing information
    service_description = models.TextField(blank=True)   # descriptions of services
or company

    def __str__(self):
        return self.company_name
```

This **ServiceData** model has fields for all the required information:

- **Company Name** (string)
- **Logo** (image upload; uses an ImageField so we can store an image file for the logo)
- **Country** (string)
- **Website** (URL)
- **Social media links** for LinkedIn, Facebook, Instagram (URLs)
- **Services offered** (text field, to list or describe services)
- **Contact numbers** (text, can store multiple numbers separated by commas or newlines)
- **Emails** (text, for one or more email addresses)
- **Office locations** (text, for one or more addresses)
- **Ratings** (stored as text or a small numeric string – e.g. "4.5/5")
- **Key individuals** (text, list of people)
- **Pricing** (text, pricing details)
- **Service description** (text, additional description of the service or company)

All these fields are optional (blank allowed) because scraped data might not have every field for every entry. We used appropriate field types (e.g., URLField for links, ImageField for logos, TextField for potentially long text).

ImageField considerations: Using an ImageField requires installing Pillow, as noted earlier . We set `upload_to='logos/'` so that uploaded logos are saved under a `media/logos/` directory (within MEDIA_ROOT). Ensure `MEDIA_ROOT` and `MEDIA_URL` are set in settings (which we did) and that Pillow is installed.

After defining this model, run `python manage.py makemigrations && python manage.py migrate` to create the corresponding database table (since we already did migrate globally, make sure migrations for the `services` app are applied).

3. Admin Panel Configuration

Django's admin site will be used to manage both the **UserProfile** and **ServiceData** models. We need to **register these models** in their respective app's **admin.py** and customize the display for better usability.

In `accounts/admin.py`:

```python
from django.contrib import admin
from .models import UserProfile

@admin.register(UserProfile)
class UserProfileAdmin(admin.ModelAdmin):
    list_display = ('user', 'user_type', 'tier', 'location', 'session_status')
    list_filter = ('user_type', 'tier')
```

In `services/admin.py`:

```python
from django.contrib import admin
from .models import ServiceData

@admin.register(ServiceData)
class ServiceDataAdmin(admin.ModelAdmin):
    list_display = ('company_name', 'country', 'website', 'ratings')
    search_fields = ('company_name', 'country')
```

Here we use the `@admin.register` decorator to register the models with the admin site, and define a `ModelAdmin` for each to customize how they appear. We configured:

- **list_display**: This tuple specifies which fields to show in the list view of the admin. For example, for UserProfile we show the user (username), type, tier, location, and session status. For ServiceData we show company name, country, website, and ratings. Django allows us to easily control which fields are displayed in admin lists w3schools.com .
- **list_filter**: Adds filters in the sidebar for certain fields (e.g., user_type, tier) to quickly filter the list.
- **search_fields** (for ServiceData): Enables a search box in the admin list to search by company name or country.

By registering these, an admin user can log into the Django admin (`/admin/`) and add or edit service entries or user profiles. The **UserProfile** will appear inline with the user if we configured it that way; however, another approach (not shown here) is to use an inline admin to edit profile info on the same page as the User model docs.djangoproject.com . Our approach is simpler: we manage profiles as a separate model in admin.

Now you should be able to log in as the superuser at `/admin/` and see **User Profiles** and **Service Data** sections.

4. Django REST Framework API (CRUD with Token Authentication)

We will expose a RESTful API for both the **UserProfile** and **ServiceData** models. This will allow Create, Read, Update, Delete operations via HTTP (CRUD). We use **Django REST Framework (DRF)** to build these endpoints.

First, ensure **djangorestframework** and **rest_framework.authtoken** are in `INSTALLED_APPS` (as done in settings). DRF provides a powerful toolkit for serializing models and handling requests.

Serializers: Create serializers for our models in a new file `accounts/serializers.py` and `services/serializers.py`:

python

```python
# accounts/serializers.py
from rest_framework import serializers
from .models import UserProfile
```

```python
class UserProfileSerializer(serializers.ModelSerializer):
    class Meta:
        model = UserProfile
        fields = '__all__'  # serialize all fields of the profile
```

python

```python
# services/serializers.py
from rest_framework import serializers
from .models import ServiceData

class ServiceDataSerializer(serializers.ModelSerializer):
    class Meta:
        model = ServiceData
        fields = '__all__'  # serialize all fields of the service data
```

Using `ModelSerializer` automatically creates fields corresponding to the model. We use `'__all__'` to include all model fields. (In a real project, you might limit fields or make nested serializers for the related User, but we'll keep it simple.)

Views: We can use DRF **ViewSets** to handle CRUD for each model. Let's create `accounts/views.py` and `services/views.py` with viewsets:

python

```python
# accounts/views.py
from rest_framework import viewsets, permissions
from .models import UserProfile
from .serializers import UserProfileSerializer

class UserProfileViewSet(viewsets.ModelViewSet):
    queryset = UserProfile.objects.all()
    serializer_class = UserProfileSerializer
    permission_classes = [permissions.IsAuthenticated]  # require auth for all
actions
```

python

```python
# services/views.py
from rest_framework import viewsets, permissions
from .models import ServiceData
```

```
from .serializers import ServiceDataSerializer

class ServiceDataViewSet(viewsets.ModelViewSet):
    queryset = ServiceData.objects.all()
    serializer_class = ServiceDataSerializer
    permission_classes = [permissions.IsAuthenticated]  # require token auth
```

These viewsets provide standard actions (`list`, `retrieve`, `create`, `update`, `destroy`). We restrict access so only authenticated requests (with a valid token) can use them. By default, our DRF settings also enforce `IsAuthenticated` globally , so this is just an extra safeguard or could even be omitted due to the global setting.

URLs: We will route these viewsets using a router. It's convenient to create a separate `api/urls.py` (included in the main urls as shown earlier):

python

```
# api/urls.py
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from accounts.views import UserProfileViewSet
from services.views import ServiceDataViewSet
from rest_framework.authtoken import views as authtoken_views

router = DefaultRouter()
router.register(r'profiles', UserProfileViewSet, basename='profiles')
router.register(r'services', ServiceDataViewSet, basename='services')

urlpatterns = [
    path('', include(router.urls)),  # includes /profiles/ and /services/ endpoints
    path('api-token-auth/', authtoken_views.obtain_auth_token, name='api-token-
auth'),
]
```

We use `DefaultRouter` to automatically generate URL routes for the two viewsets. This will create endpoints such as:

- `/api/profiles/` for listing and creating user profiles, and `/api/profiles/<id>/` for retrieve, update, delete.
- `/api/services/` for listing/creating service entries, and `/api/services/<id>/` for detail operations.

We also include a URL for obtaining auth tokens: `api-token-auth/` is wired to DRF's built-in view `obtain_auth_token`, which accepts a POST with username and password and returns a token if credentials are valid geeksforgeeks.org .

At this point, run `python manage.py makemigrations && python manage.py migrate` again to ensure the `authtoken` app creates its Token model/table. You should also generate tokens for existing users. One easy way is to use the provided endpoint: for example, using curl or HTTPie:

```bash
# Obtain token for a user (e.g. the superuser created earlier)
curl -X POST -d "username=<your_username>&password=<your_password>"
http://127.0.0.1:8000/api/api-token-auth/
```

This will return a JSON response with a token, e.g. `{"token": "abcdef123456..."}`. Copy this token. Now you can authenticate any API requests by including an **Authorization** header with the token, as follows:

```makefile
Authorization: Token <your_token_here>
```

For example, to get the list of services via API, you could do:

```bash
curl -H "Authorization: Token <your_token>" http://127.0.0.1:8000/api/services/
```

The DRF browsable API (if you open these endpoints in a browser) will also allow you to login (using the token or session if logged in via admin) and test the API. The token-based auth we set up is a simple mechanism where each user has one static token string (stored in the database) django-rest-framework.org . (For a more complex setup, one could use JWT with `djangorestframework-simplejwt`, but that's beyond our scope.)

5. Project Structure and Scraper Module

Your project directory structure should look like this (simplified):

graphql

```
growbal/                # Django project root
├── manage.py
├── growbal/            # Project settings package
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   ├── asgi.py
│   └── wsgi.py
├── accounts/           # App for user profiles
│   ├── models.py
│   ├── admin.py
│   ├── views.py
│   ├── serializers.py
│   └── ... (apps.py, etc.)
├── services/           # App for scraped service data
│   ├── models.py
│   ├── admin.py
│   ├── views.py
│   ├── serializers.py
│   └── ...
├── api/
│   ├── urls.py         # API URL routes (viewsets and token auth)
│   └── __init__.py
├── scraper/            # Placeholder for future web scraping module
│   ├── __init__.py
│   └── scraper_demo.py (e.g., an empty module or example scraping script)
└── media/              # Media directory for uploaded files (logo images)
    └── logos/
```

We included a `scraper/` folder (not a Django app, just a Python package) to hold future web scraping code. For now it can contain placeholder files or a basic script. For example, `scraper/scraper_demo.py` could be a starting point for scraping routines (this is outside the scope of Django's MVC, but you might later turn it into a custom command or integrate with the ServiceData model).

6. Running the Project and Usage

Finally, here are the steps and commands to get the project up and running:

1. **Install dependencies** (Django, DRF, Postgres adapter, Pillow):

```bash
pip install Django psycopg2-binary djangorestframework Pillow
```

(Make sure PostgreSQL is installed and running on your system.)

2. **Start a Django project and apps** (if you haven't already done so):

```bash
django-admin startproject growbal .
python manage.py startapp accounts
python manage.py startapp services
```

(The structure and code provided above should then be placed into the respective files.)

3. **Configure the database**: In `growbal/settings.py`, set up the `DATABASES` section to use PostgreSQL (as shown earlier) with your database name, user, password, host, and port. Ensure the database and user exist in PostgreSQL (you can create them using psql or PGAdmin). For example, your settings might include:

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'growbal_db',
        'USER': 'postgres',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Also, add `'rest_framework'` and `'rest_framework.authtoken'` to **INSTALLED_APPS** django-rest-framework.org, and include `'accounts'` and `'services'` apps.

4. **Apply migrations** to create tables in the PostgreSQL database:

```bash
```

```
python manage.py makemigrations
python manage.py migrate
```

This will set up Django's auth tables, our UserProfile and ServiceData tables, and the token auth table.

5. **Create a superuser** to access the admin:

```bash
python manage.py createsuperuser
```

Follow the prompts to set username and password. This account will be able to log into `/admin/` and also can be used to obtain an API token.

6. **Run the development server**:

```bash
python manage.py runserver
```

The site will be accessible at **http://127.0.0.1:8000/**. You can log into the admin interface at **http://127.0.0.1:8000/admin/** using the superuser credentials.

7. **Using the API**: Obtain an auth token for a user by POSTing to `/api/api-token-auth/` with username & password (as JSON or form data). You will get a token string in response . Then include this token in the `Authorization` header for subsequent API calls. For example, to list all services via API, GET `/api/services/` with the header `Authorization: Token <your_token>`.

8. **Future web scraping integration**: You can write scripts in the `scraper/` module to fetch data from external sources and then create `ServiceData` records (via the Django ORM or Django management commands). The project is structured to accommodate this – for instance, you might create a Django management command that calls your scraping code and populates the database.

By following these steps, you now have a Django project "**Growbal**" with a PostgreSQL database backend. It supports user authentication (with profile data for Investors and Entrepreneurs), an admin panel to manage users and scraped service data, and a token-protected REST API for both models. You can extend this foundation by implementing the actual web scraping logic in the `scraper` module and hooking it into the Django app (e.g., schedule it to run periodically or trigger via an admin action).

With the provided setup, everything is in place to start building out the additional features as needed. Enjoy your new Growbal project!

Sources:

- Django documentation on extending the User model with a one-to-one profile docs.djangoproject.com
- Django documentation on built-in auth views for login/logout consideratecode.com
- Django ImageField requires Pillow for image uploads geeksforgeeks.org
- W3Schools example of customizing admin list display fields w3schools.com
- Django REST Framework installation and setup pypi.org django-rest-framework.org
- DRF TokenAuthentication usage and endpoint setup geeksforgeeks.org geeksforgeeks.org
- Installing psycopg2 for PostgreSQL (using binary wheel) stackoverflow.com
- Example Django settings for PostgreSQL connection w3schools.com

Sources