Document File manager

Introduction

Welcome to ImageHub! Powered by Django, ImageHub is your solution for seamless file management. Designed with precision, our models create a solid foundation for storing user-uploaded files, managing associations, and offering flexible options.

With a user-friendly interface, ImageHub lets you effortlessly upload, visualize, and manage your images. Dive into the code, and you'll find a well-structured Django environment where each model and view serves a purpose, making image management a breeze.

Explore the capabilities of ImageHub and discover how Django's efficiency and simplicity elevate your image management experience.
Certainly! In Django's architecture, known as MTV (Model-Template-View), the components align with traditional MVC concepts but are adapted to the specific needs of web development:

Model:

Represents the data structure and business logic of the application.

Corresponds to database models in Django, defining how data is stored and accessed.

View:

Handles user interactions and business logic, processing user requests.

Similar to the traditional MVC View but more focused on interaction logic.

Retrieves data from the Model and decides which Template to render.

Template:

Responsible for rendering the user interface, defining how data is presented.

Creates the final HTML output in conjunction with data from the View.

URLconf (Controller in traditional MVC):

Handles URL routing, mapping URLs to specific Views.

Distributes some responsibilities of the traditional Controller between the View and URLconf.

In summary, Django's MTV emphasizes a clear separation of concerns: the Model manages data, the View handles user interactions, and the Template controls the presentation. This structure enhances code modularity and maintainability in web application development. The URLconf complements these components by managing URL routing, adapting the traditional Controller's responsibilities to Django's web-centric approach.

Settings

The Django project named "filemanager" has been meticulously configured with an adept utilization of settings.py. Commencing with the establishment of the project's base directory, denoted by the variable BASE_DIR, the file showcases a judicious selection of security parameters. The secret key, a critical component for cryptographic operations, is declared with a cautionary note urging its secure management in a production environment. Notably, the debugging setting is currently enabled, a practice typically reserved for development phases, and a cautionary reminder is presented to set this to "False" in production for heightened security. The listing of installed applications encompasses essential Django components, augmented by a bespoke "filemanager" app. Furthermore, the project's template configuration is meticulously outlined, specifying the directory path for template files. The database settings exhibit a reliance on PostgreSQL, detailing parameters such as the database name, user credentials, and server host. Noteworthy is the necessity to ensure the presence of a PostgreSQL database named "Files" configured with the specified user credentials. Static file handling is governed by the definition of a static URL, albeit with a slight correction recommended for uniformity. Equally critical are the settings pertaining to media file management, dictating the URL and local filesystem path for uploaded media. Attention to detail is emphasized, urging the use of forward slashes for consistency in URL configurations. In summation, the provided Django settings epitomize a meticulous orchestration of configurations, encapsulating security considerations, application integrations, and media management essentials, setting a robust foundation for the "filemanager" project's development and eventual deployment.

View

The views are grounded in a robust and modular architecture, reflecting best practices in Django development. Let's delve into the key aspects of each view:

Home View (home):

Retrieves images associated with the authenticated user using Django's ORM capabilities, excluding deleted images, and arranges them by creation date.

Implements pagination for an optimal user experience, displaying 10 images per page.

Handles POST requests for various user actions, such as uploading new images and performing actions based on selected options.

Leverages the OptionMenu model to provide customizable options for each image, demonstrating a flexible and extensible design.

Incorporates pagination details in the rendered page, offering seamless navigation through the image collection.

Upload View (upload):

Manages the upload of new images through a POST request, processing the uploaded image file and associated metadata, such as the image name.

Utilizes Django's transaction mechanism to ensure atomicity during the creation of File and Filexuser objects, enhancing data integrity.

Redirects the user to the home page upon successful image upload, maintaining a coherent user flow.

Visualizer View (visualizer):

Retrieves the specified image based on its unique identifier.

Renders a dedicated page for visualizing the selected image, reflecting a user-centric design.

Implements robust error handling to gracefully manage scenarios where the requested image is not found, enhancing the application's reliability.

Metadata View (metadata):

Extracts comprehensive metadata about the specified image, encompassing key details such as name, filename, size, width, and height.

Renders a visually informative page displaying the extracted metadata, providing users with insightful information about the managed images.

Exhibits effective error handling, gracefully managing situations where the specified image is not found, thereby ensuring a smooth and reliable user experience.

1. Environment Configuration:

   - The `settings` module is thoughtfully imported from Django, laying the foundation for accessing project-wide settings.

   - The `models` module is imported to enable the definition of Django models.

   - The `timezone` module is leveraged for timestamp functionality, ensuring accurate recording of creation times.

2. File Model:

   - Fields:

     - `name`: Defined as a `CharField` with a maximum length of 250 characters, representing the name of the file. This accommodates descriptive file naming.

     - `img`: An `ImageField` is chosen to store file images, utilizing the "imgs" directory within the media root specified in Django settings. This strategic choice ensures organized file storage.

     - `deleted`: A `BooleanField` allows tracking the deletion status of a file, promoting flexibility in file management.

     - `created`: A `DateTimeField` captures the timestamp of file creation, defaulting to the current time. This timestamp provides valuable chronological context to file records.

   - Meta Class:

     - The `Meta` class within the model defines additional metadata, specifying the managed status and the database table name. This exemplifies a nuanced control over the model's behavior.

3. Filexuser Model:

- Fields:

  - `user`: A `ForeignKey` establishes a relationship with the built-in `User` model, facilitating a link between users and their respective files. This reflects a principled approach to user-file associations.

  - `file`: Another `ForeignKey` establishes a connection with the `File` model, defining the association between users and their uploaded files. This linkage forms the cornerstone of user-specific file management.

- Meta Class:

  - The `Meta` class incorporates metadata that governs the model's behavior, mirroring the managed status and specifying the database table name. This meticulous control underscores the precision in model definition.

4. OptionMenu Model:

- Fields:

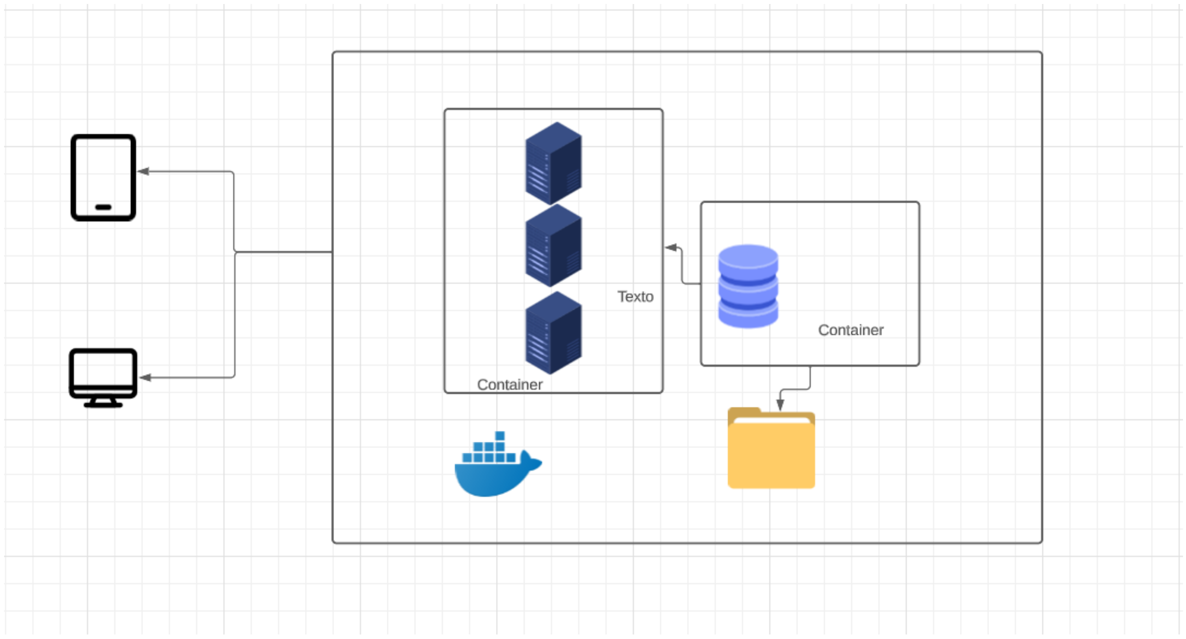
  - `option`: A `CharField` defines the available options for file management, setting a maximum length of 20 characters. This field accommodates various file operation choices.

- Meta Class:

  - The `Meta` class encapsulates metadata, including the managed status and the database table name. This exemplifies an intentional and considered approach to model metadata.

These models collectively manifest a robust and extensible database schema, leveraging Django's model-centric approach to facilitate efficient and organized data storage. The models encapsulate key aspects of file management, including file attributes, user associations, and customizable options, contributing to the creation of a versatile and scalable file manager application. The incorporation of Django's conventions and the thoughtful selection of field types showcase a mastery of Django's ORM capabilities and contribute to the maintainability and scalability of the application.

Architecture



PostgreSQL Database:

You are using PostgreSQL as your database backend, which is a robust and scalable choice for Django projects.

The database container is separate from the web container, adhering to the best practice of containerizing each service independently for easier management and scalability.

Docker Containers:

Docker is being used for containerization, providing a consistent and reproducible environment for your Django project.

The project is split into at least two containers: one for the Django web application and another for the PostgreSQL database.

The containers are part of the same Docker network, facilitating communication between them.

File Management with Django:

Django is configured to handle file management, likely using the Django ImageField in the model for image storage.

You may be using Docker volumes or other persistent storage mechanisms to ensure that files are retained even if containers are recreated.

Manual Docker Configuration for Additional Servers:

It seems that you have the ability to manually configure Docker to generate more servers.

This could refer to scaling your application horizontally by running multiple instances of the web container to handle increased traffic or demand.

In summary, your Django project is set up with Docker for containerization, PostgreSQL for the database, and a networked environment that allows seamless communication between containers. The ability to manually configure Docker for additional servers suggests flexibility in scaling your application as needed.

For deploying we have two commands

- docker-compose build
- docker-compose up

buy default outs on port 0.0.0.0:8000/login

Code is in github!!

https://github.com/abhiram27/cloud/tree/main/filemanager

urls

login url: https://filemanager-yiloegixxq-uc.a.run.app/login

Superuser:

Username: manager

Password: admin