Django Training - 8. Permissions and Django Filters

Materials

1. django-filter
    - Integration with DRF
2. Pagination

Description

Let's assume you're working with another developer who is handling the frontend side of the music platform app, this developer is now working on a screen that will allow users to explore all the albums stored in our database. For a friendly user experience, the developer decided to provide the user with a set of filters on that screen, for example, the user can choose to see only the albums that cost less than 1000 price units.

Let's also assume that our users can be an artist or not, so the artist model we have so far is an optional extension of the user model.

Task

1. Add a relationship field to the `Artist` model that maps an artist to a user instance.
    - Hint: Think carefully about the relationship type you'll use
    - This field should support the following statement: **All artists are users but not all users are artists**
    - When running your migrations, django will complain that you already have artists stored in your database and it doesn't know what to do with them since they aren't mapped to a user model which goes against the relationship you just created assuming this relationship is not nullable. Feel free to reset your database and delete the migrations *or* make this field nullable
2. Create an endpoint `/albums/`
    - `GET` should return a list of **approved** albums
        - Each album should be a JSON object that looks like this:

```
{
    "id": ...,
    "artist": {
        "id": ...,
        "stage_name": ...,
        ...
    },
    "name": ...,
    "release_datetime": ...,
    "cost": ...,

}
```

        - Permit any type of request whether it's authenticated or not.
        - It doesn't make sense to return all albums that we have to the frontend at once, if we have hundreds of thousands of albums, the user's screen will not be able to render that much data, instead we should support pagination.
            - Support `LimitOffsetPagination` for this view.
        - Bonus: Can you create and use custom queryset manager that only returns approved albums?
    - `POST` should accept a JSON body, create an album, and raise proper validation errors for all fields
        - The request body should look like: `{ "name": ..., "release_datetime": ..., "cost": ..., }`
        - Permit only authenticated requests
        - The request must be authenticated by a user who is also an artist
        - The created album will be mapped to the artist who made the request
        - `403 Forbidden` error should be raised if a `POST` request is not authenticated or if it's authenticated by a

user who isn't an artist

3. Using `django-filter`, support the following optional filters for `GET` requests:
    - Cost greater than or equal
    - Cost less than or equal
    - Case-insensitive containment
        - If I have 2 albums named `OK` and `ok`, both should be returned in the response if I filter by `name=ok`
    - Example usage:
        - `/albums/?cost__gte=10&cost__lte=50&name="Album"` This should return all albums within the cost range of 10 and 50 price units inclusive and whose names contain the word `Album` with case insensitivity.
        - `/albums/?cost__lte=100` This should return all albums with a cost less than 100 price units.
    - *Why are IDs passed as url kwargs `/albums/<pk>` but filters are passed as query params `/albums?name="Album"`? What stops us from doing something like `/albums?id=5`* **The convention in Django is to pass unique identifiers as url kwargs and non unique identifiers as query params.**
4. Create another endpoint that supports that same exact features as the `/albums/` endpoint when it receives a `GET` request but implement the filters manually.
    - Raise a validation error if I don't respect the filter's data type, for example if I pass a string to the cost filter.
5. Under `albums/tests` create `test_serializers.py` to test the serializer class (don't make requests to any endpoints)
    - Test that your album serializer serializes the album instance properly and returns the expected data
    - Test that your album serializer deserializes (creates) an album instance properly from given data dictionary and http request
6. Under `albums/tests` create `test_endpoints.py`
    - Test that the view(s) use(s) the expected serializer(s).
    - Test proper response status codes for permitted and unpermitted requests

Guidelines

1. Use this gitignore template
2. List your environment variables (if any) in a `.env.example` file
3. Don't forget to run `manage.py makemigrations` and `python manage.py migrate`
4. You are allowed to refer to any articles, documentation source, questions on StackOverFlow
5. You are not allowed to consult other people through any medium of communication
6. Your grade will be affected if your code isn't clean, you should maintain code cleanliness as much as you can