Advanced Web Development Midterm

How to run the application?

Python version 3 is required to run this project. I tested it using Python version 3.9.6 so just in case of any issue, please ensure the Python version is up to date. I also used Python virtual environments to isolate the packages that the application relied on.

In order to run the application, please perform the following actions and commands in your OS terminal in the given order:

1. Highly recommended but not required: Create a Python virtual environment by running command `python -m venv <my_env>` and then activate the virtual environment by running `my_env/Scripts/activate`.
2. Install the required packages by running `pip install -r requirements.txt`
3. Navigate to directory `/midterm` and then run `python manage.py runserver`

The app should now be running at the default host of `localhost` with IP address of `127.0.0.1` and Port `8000`. To app may now be accessed using a browser or a REST client at the following address: http://127.0.0.1:8000.

In order to run the app at a different address and port, please run the following command:

```
python manage.py runserver localhost:8080
```

Or only change the port:

```
python manage.py runserver 8080
```

The links in this guide assume port `8000` at `localhost` has been used.

Admin

To login to Django admin, please go to http://127.0.0.1:8000/admin and then log in using username: `admin` and password: `admin`.

To create a new Django admin user, please run the following command:

```
python manage.py createsuperuser
```

In `/midterm/proteins/admin.py`, I have registered the Model classes with the Django Admin interface so that basic editing of the entities is possible in the Admin UI. I have also inlined protein domains with Protein.

How to import data?

All the migrations have been run and seed data preloaded in the included `db.sqlite3` file. However, if anything goes wrong, the data may be reimported by taking the following steps:

1. Delete the file `db.sqlite3`.
2. Run `python manage.py makemigrations proteins` (only needed after a change in models)

3. Run `python manage.py migrate`
4. Run `cd scripts`
5. Run `python populate-proteins-db.py`

Models and Serializers

I imported the provided data into the following models/tables:

1. Organism
2. Pfam
3. Domain
4. Protein
5. ProteinDomainMapping (exists solely to map domains to proteins)

In order to ensure the data is normalized and minimal redundant data is present in the tables, I separated out the organisms' data into their own model: `Organism`. Since each `Protein` belongs to an organism, it uses a foreign key `taxonomy` to reference `Organism` model. I noted that `length` property of `Protein` is computable doesn't need to be saved in the database table.

I also separated out the `Domain` and `ProteinDomainMapping` tables because this way, the mapping table didn't have extra columns which would be tricky to include with the relations. Keeping the mapping table separate from the Domain data made the process really simple with Django rest framework doing the include resolution for us.

I preferred to explicitly declare the primary keys for most of my tables because the IDs in Protein, Organism and Pfam table are already unique.

I felt that domain `description` should belong to `Pfam` model and not `Domain` because it changes with Pfam ID and doesn't change with each Domain instance. However, I kept it with `Domain` just to conform with the coursework requirements. The data in `Domain` table is not therefore normalized.

The serializers code is really simple because they are based on `ModelSerializer` from Django Rest Framework, which is not a bad thing because they serve the purpose and there is no need to manually write our serializer methods if the generic ones meet our needs, which they do really well. I only needed to add a few simple custom fields to my serializers e.g. the `length` property and domain coverage property for protein.

REST endpoints

I implemented the following REST endpoints:

```
POST http://127.0.0.1:8000/api/protein/
GET  http://127.0.0.1:8000/api/protein/{protein_id}
GET  http://127.0.0.1:8000/api/pfam/{pfam_id}
GET  http://127.0.0.1:8000/api/proteins/{taxa_id}
GET  http://127.0.0.1:8000/api/pfams/{taxa_id}
GET  http://127.0.0.1:8000/api/coverage/{protein_id}
```

These are listed in the API documentation done using Swagger, however I didn't specify the API in detail because this wasn't part of the coursework requirements. The API specification may be accessed at http://127.0.0.1:8000/

I used Django Rest Framework to create the above endpoints. Since the endpoints above are simple I didn't want to reinvent the wheel and I was able to achieve the required functionality using the generic API views that Django Rest Framework provides. In the following lines I briefly describe any notable features of my endpoints implementations:

In `POST /api/protein/`, I create new proteins and accept basic properties such as `protein_id`, `sequence` and `taxonomy`. I don't create nested objects because I wanted to keep the API RESTful.

In `GET /api/protien/{protein_id}`, I use generic `RetrieveAPIView` and in `ProteinRetreiveSerializer`, I compute the protein length using `len()` which I have overridden in `Protein` model and it simply calculates the number of characters in the protein `sequence` property. Also I used `DomainSerializer` to inline domains into protein and also use `depth = 1` in `Meta` which inlines `taxonomy`.

Again, as I stated above, `protein.domains[].description` field is not normalized but only because I wanted to follow the coursework requirements. Ideally, this description will be better placed in `Pfam` table, even though we will end up with two description fields in `Pfam` table.

In `GET /api/proteins/{taxa_id}`, even though the coursework example shows Django `id` in addition to `protein_id`, my endpoint only lists objects with protein ids because the table has protein ids as primary keys and doesn't use Django sequential primary key. Also, this list would have been better done if the response was an array of protein ids instead of array of objects having protein ids, because the only property the objects have is `protein_id`.

Similarly in endpoint `GET /api/pfams/{taxa_id}`, I didn't use Django sequential ids as primary keys but instead used Pfam id, i.e. `domain_id` as the primary key because it is unique. Therefore, the response from this endpoint differs a bit from the coursework example because it doesn't have `id` field.

To implement the endpoint `GET /api/pfams/{taxa_id}`, I overrode the `get_queryset()` method and I filter the Pfams using Django field lookups functionality with double underscores. This enabled me to get the Pfams that are related to proteins belonging to the given Organism identified by the `texa_id` (I have used the keyword `taxonomy` in my code instead of `texa_id`).

In the endpoint `GET /api/coverage/{protein_id}`, the domain coverage of the protein is calculated in `ProteinDomainCoverageSerializer` and I do this using `serializers.SerializerMethodField()` which is custom field named `coverage` and is computed in `get_coverage()`. In this method I filter the domains by the given protein and then I use `len()` method which I have overridden in `Domain` model and `Protein` model.

Tests

Tests are located in `/midterm/proteins/model_factories.py` and `/midterm/proteins/tests.py`.

To run these tests, please navigate to `/midterm` in the OS terminal, and then run the following command:

```
python manage.py test
```

In the tests classes, I have overridden `setUp()` and `tearDown()` methods from parent classes which ensures that the database is cleaned up before a different class of tests are run.