Instructions

Run the project

Below are the instructions to run the project and to access the API endpoints.

1. Extract the zip archive to a location of your choice.
2. There will be two folders and a file: `script`, `bioweb`, and `requirements.txt`.
3. Open a command prompt or terminal in the directory of the extracted files.
4. Create a Python virtual environment using:

   ```
   python -m virtualenv env
   ```

5. Activate the virtual environment if it is not yet activated using:

   ```
   .\env\Scripts\activate (Windows)
   source env/bin/activate (Mac/Linux)
   ```

6. Install required packages using:

   ```
   pip install -r requirements.txt
   ```

7. Change directory to the `bioweb` folder.
8. Run the project using:

   ```
   python manage.py runserver
   ```

9. The API should now be running and accessible at http://127.0.0.1:8000
10. If you require access to the Django Admin site, a superuser account is available with the following credentials:
    Username: `admin`
    Password: `password123`

Run tests

These instructions assume that the python virtual environment has been activated, with all required packages already installed.

1. Open a command prompt or terminal in the project's root directory.
2. Run the command:

   ```
   python manage.py test
   ```

3. All test cases should be run, and the results will be displayed.

Code Organisation

Introduction

The following sections will describe the general approach, code organisation and logic of the Django project.

In general, the code organisation follows Django's best practices and code is segregated into their respective files such as `models.py, urls.py, api.py, and tests.py`.

The Django project is named `` `bioweb` `` and an application called `` `api` `` was created to contain the codes (models, URLs, views, etc) for the API endpoints.

Models

Related requirements: *R1a - correct use of models and migrations, R2: Implements an appropriate data model for the data*

After inspecting the datasets as well as the REST specification file provided, four models and their relationships were identified. The four models are: `Organism`, `Protein`, `Domain` and `Pfam`.

The sqlite3 database is then produced by first generating the migrations file using the `` `makemigrations` `` command, followed by the `` `migrate` `` command.

Organism

Each row in this model represents an organism. This model is also referenced as a foreign key in the `Protein` model.

Fields: `taxa_id, clade, genus, species`

Protein

Each row in this model represents a protein. This model is also referenced as a foreign key in the `Domain` model.

Fields: `protein_id, sequence, taxonomy (FK), length`

The `taxonomy` field is a foreign key that references the `Organism` model.

In addition, the `Protein` model also has a property called `` `coverage` ``. This property is used by the `` `api/coverage/<protein ID>` `` endpoint.

```
@property
def coverage(self):
    domains = self.domain_set
    return domains.annotate(coverage=(F('stop')-
F('start'))).aggregate(total=Sum('coverage'))['total']/self.length
```

The `` `coverage` `` property is computed by first retrieving a queryset of all domains that belongs to the given protein, then aggregating the sum of the coverage of each domain.

Domain

Each row in this model represents a domain.

Fields: `pfam_id (FK), description, start, stop, protein_id (FK)`

The `pfam_id` field is a foreign key that references the `Pfam` model, while the `protein_id` field is a foreign key that references the `Protein` model.

Pfam

Each row in this model contains the `domain_id` and `domain_description` of a domain. This model is referenced as a foreign key in the `Domain` model.

Fields: `domain_id, domain_description`

URL Routing

Related requirement: *R1d – correct use of URL routing*

To reduce potential clutter in the future, URLs belonging to the API endpoint are contained in its own `urls.py` file in the `api` application (i.e., `bioweb/api/urls.py`).

This file then needs to be included in the main `urls.py` file (i.e., `bioweb/bioweb/urls.py`). This can be done using the `include()` function like so:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('api.urls')),
]
```

This tells Django to look for mappings in the `bioweb/api/urls.py` file if the request path begins with `api/`.

Finally, the actual mappings to the various API endpoints are contained in the `bioweb/api/urls.py` like so:

```
urlpatterns = [
    path('protein/', api.protein_add, name='protein_add_api'),
    path('protein/<str:protein_id>', api.protein_detail, name='protein_api'),
    path('pfam/<str:domain_id>', api.pfam_detail, name='pfam_api'),
    path('proteins/<int:taxonomy_id>', api.proteins_list, name='proteins_api'),
    path('pfams/<int:taxonomy_id>', api.domains_list, name='pfams_api'),
    path('coverage/<str:protein_id>', api.domain_coverage, name='coverage_api'),
]
```

For example, the request `GET http://127.0.0.1/api/protein/ABC12345` will be routed to the `protein_detail()` function in the `api.py` file.

Views and Serialization

Related requirements: *R1b, correct use of form, validators and serialisation, R1c – correct use of django-rest-framework, R3 – Implementation of appropriate code for the required REST endpoints*

The views that serve the various API endpoints are located in a separate file `api.py`. Each endpoint is handled by its respective function, the mapping can be seen in the above section.

Various classes and components from the `django-rest-framework` library were used in the API views and for the serialization of database objects.

The @api_view decorator

The `@api_view` decorator is used in each of the view functions. They are used to dictate which HTTP methods a view should respond to. If a HTTP method is not defined in the decorator, the server will generate and send a `HTTP 405 Method Not Allowed` response back to the requestor.

The benefit of this is twofold, first it clearly informs the requestor that the request method was not allowed for the endpoint, and secondly it also reduces the attack surface of the application as only pre-defined HTTP methods will be processed by the view.

The Response Class

The `Response` class from `django-rest-framework` was also used to return a response to requests. One benefit of using this class is to allow users to view a nice and clean user-interface when accessing the API endpoints through a web browser.

Serializers

Various serializers were used to translate between the database objects and its representation in the json format. For the GET endpoints, the `ModelSerializer` class was used for convenience. The serializers do not simply serialize all fields, but instead only serialize and return specific fields that are required as seen in the REST specifications provided.

For endpoints that require the foreign key fields to be serialized as well, nested serializers were used to achieve this purpose.

An example of this can be seen in the two of the serializers used for the `GET /api/protein/[PROTEIN ID]` endpoint:

```
class PfamSerializer(serializers.ModelSerializer):
    class Meta:
        model = Pfam
        fields = ['domain_id', 'domain_description']


class DomainSerializer(serializers.ModelSerializer):
    pfam_id = PfamSerializer()

    class Meta:
        model = Domain
        fields = ['pfam_id', 'description', 'start', 'stop']
```

From the above, we can see that the `PfamSerializer` is nested within the `DomainSerializer`. This allows us to serialize the `pfam_id` foreign key as well.

The output of the `DomainSerializer` then has this structure:

```
{
    "pfam_id": {
        "domain_id": "PF01650",
        "domain_description": "PeptidaseC13family"
    },
    "description": "Peptidase C13 legumain",
    "start": 40,
    "stop": 94
}
```

Testing

***Related requirement**: R1e – appropriate use of unit testing*

Fixtures

The `factory_boy` library was to provide the fixtures used for testing. These can be found in the `` `bioweb/api/model_factories.py` `` file.

A `DjangoModelFactory` was created for each model in the project. Thus, a total of four factories were created: `PfamFactory, OrganismFactory, ProteinFactory, DomainFactory`.

Test Cases

The test cases are located in the `` `bioweb/api/tests.py` `` file, which is Django's default location to store test codes in Django projects.

The test cases cover two areas of the application: the response of the API endpoints, and the output of the various serializers.

API Endpoints test

These test cases test the endpoint with valid and invalid requests. Requests are valid when they request for an existing object in the database, otherwise they are not valid.

When a valid request is received, we assert that the response status code is 200 and the data returned is of the correct object that we requested. On the other hand, when an invalid request is received, we assert that the response status code is 404.

Serializers test

These tests ensures that each serializer returns exactly the information that it is supposed to return. This is done by first generating an object with `factory_boy`, then passing this object through the serializer and examining its output.

Data loading script

***Related requirement**: R4 – Implementation of appropriate method of bulk loading data*

The data loading script is located in the `script` directory of the zip archive. The script reads all three input files `pfam_descriptions.csv, data_sequences.csv` and `data_set.csv`, stores and manipulate the data, before finally populating them into the database.

Logic flow

Before any file is read, the existing database is cleared using the `x.objects.all().delete()` function. This will allow us to start off with a clean database.

First, the `pfam_descriptions.csv` file is read, and its information stored into a two-dimensional list. Next, the `data_sequences.csv` file is read, and its information is stored into a dictionary, where the key is the protein ID, and its value is the protein sequence.

Finally, the `data_set.csv` file is read, and its information is then split into three different data structures. A dictionary `organisms` is used to store the organisms, with the taxonomy ID as the key and its other attributes as its value. Another dictionary `proteins` is used to store proteins, with the protein ID as the key, and its other attributes as its value. A two-dimensional list `domains` is also used to store the domains.

After all the information from the dataset files have been read and stored, they need to be inserted into the database. For insertion, the `bulk_create()` function was used as this is a Django best practice, especially for large number of rows. Using the bulk functions reduces the number of SQL queries, thus making our operation faster and more efficient.

The order of insertion is important as well, as data needs to be inserted into models that does not have any foreign key first, so that they can subsequently be referenced by other models. The order of insertion is then as follows: `Pfam, Organism, Protein, Domain`.