CM3025 ADVANCED WEB DEVELOPMENT

Student Name: Lawrence Ho Sheng Jin

Date Submitted: 10th Jan 2022

Degree Title: Computer Science

Local Institution: Singapore Institute of Management

Student ID: 10205927

Table of Contents

1.1 Starting the Project

The project context given requires a creation of a simple web API based on the given dataset of organisms. This report will documents the creation of the project and the process of developing the web application. The very first steps include initializing the project in the Django environment through a series of commands that automatically creates the set of packages and files required to construct the app:

```
django-admin startproject bioweb
cd bioweb
python manage.py startapp proteins
```

The bioweb folder contains the .csv files that will populate the database model later on.

```
^C(advanced_web_dev) (django)  lawrenceho~/Documents/CM3035 Adv Web Development/coursework/bioweb[ (ノ°□°) ノ︵ ┻━┻ ls
assignment_data_sequences.csv   bioweb                    instructions.txt              pfam_descriptions.csv
assignment_data_set.csv         db.sqlite3                manage.py                     proteins
```

****Additionally, there are instructions contained in *instructions.txt* on the following details::**
To run tests:

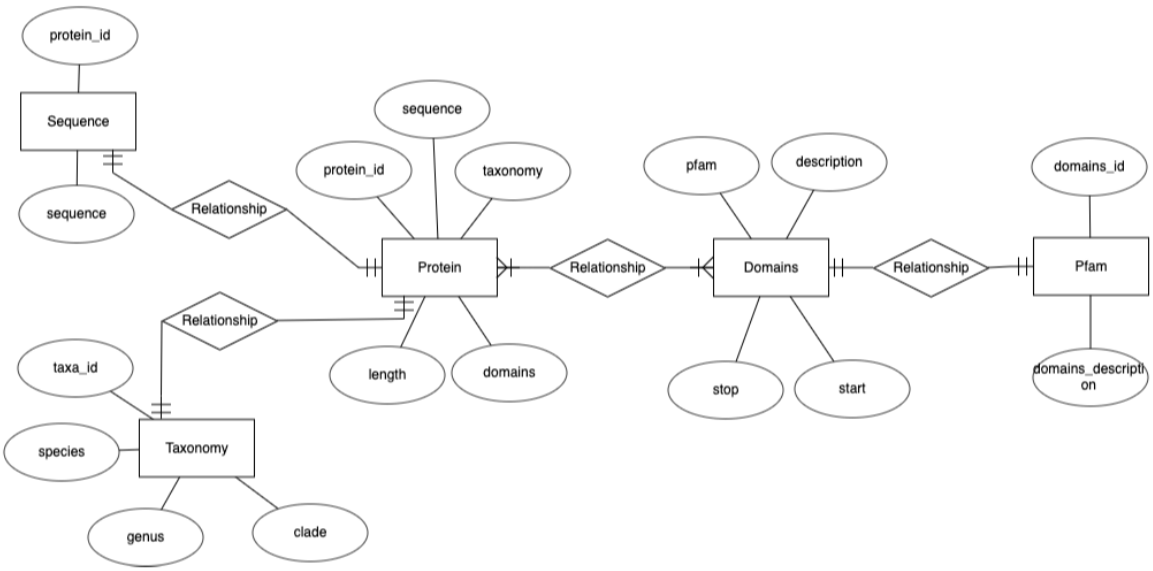
- cd to bioweb directory
- enter the command: `python manage.py test`

To run the data loading script:

- cd to bioweb/proteins directory
- enter the command: `python scripts/load_data.py`

1.2 Database design

For the dataset given, I have arranged the attributes and fields in a suitable manner to ensure a 3NF state and appropriate relations between them. The following ERD represents the database design that I will be adopting for the project:



ERD diagram for organisms

The scope of the project requires the database to be the default '*db.sqlite3*' instantiated with the Django setup, thus, there will be no changes adjusted to the Django project settings.

```
# Database
```

```
# https://docs.djangoproject.com/en/3.0/ref/settings/#databases
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

bioweb.settings.py

The following code snippet allows the implementation of the database based on the procured ERD.

```
class Sequence(models.Model):
    protein_id=models.CharField(max_length=64, null=False, blank=False)
    sequence=models.CharField(max_length=256, null=False, blank=False)
    def __str__(self):
        return "Sequence | " + self.protein_id + " " + self.sequence

class Taxonomy(models.Model):
    taxa_id=models.IntegerField(null=False, blank=False)
    clade=models.CharField(max_length=1, null=False, blank=False, default='E')
    genus=models.CharField(max_length=128, null=False, blank=False)
    species=models.CharField(max_length=128, null=False, blank=False)
    def __str__(self):
        return str(self.taxa_id) + ", " + self.clade + ", " + self.genus + ", " + self.species

class Pfam(models.Model):
    domain_id=models.CharField(max_length=64, null=False, blank=False)
    domain_description=models.CharField(max_length=256, null=False, blank=False)
    def __str__(self):
        return self.domain_id + ", " + self.domain_description

class Domains(models.Model):
    pfam=models.ForeignKey(Pfam, on_delete=models.CASCADE)
    description=models.CharField(max_length=256, null=False, blank=False)
    start=models.IntegerField(null=False, blank=False)
    stop=models.IntegerField(null=False, blank=False)
    # group values together to ensure unique entry
    class Meta:
        unique_together = ['pfam', 'start', 'stop']
    def __str__(self):
        return self.pfam.domain_id + ", " + self.pfam.domain_description + ", " + self.description + ", " + str(self.start) + ", " +
str(self.stop)

class Protein(models.Model):
    protein_id=models.CharField(max_length=64, null=False, blank=False)
    sequence=models.CharField(max_length=256, null=False, blank=False)
```

```
    taxonomy=models.ForeignKey(Taxonomy, on_delete=models.CASCADE)
    length=models.IntegerField(null=False, blank=False)
    domains=models.ManyToManyField(Domains, through='ProteinDomainLink')
    def __str__(self):
        return "Protein | " + self.protein_id + " " + str(self.taxonomy.taxa_id) + " " + str(self.length) + " " + str(self.domains)

class ProteinDomainLink(models.Model):
    protein=models.ForeignKey(Protein, on_delete=models.CASCADE)
    domains=models.ForeignKey(Domains, on_delete=models.CASCADE)
    # group values together to ensure unique entry
    class Meta:
        unique_together = ['protein', 'domains']
```

proteins.models.py

- For the field variable types and settings, all of them have been ensured to contain a value through (null=False, blank=False) with the appropriate length that corresponds to the values in the dataset.
- For Foreign Keys (FKs), the setting to delete them on CASCADE was to ensure easier testing during the development phase, and assignment of the relations.
- There is a many-to-many relation between *Protein* and *Domains*. These two models are connected using the '*through*' keyword, which creates another model '*ProteinDomainLink*' that takes in both *Protein* and *Domains* and its fields to identify the relation between both entities.

```
class ProteinDomainLink(models.Model):
    protein=models.ForeignKey(Protein, on_delete=models.CASCADE)
    domains=models.ForeignKey(Domains, on_delete=models.CASCADE)
    # group values together to ensure unique entry
    class Meta:
        unique_together = ['protein', 'domains']
```

These two fields are then grouped together to ensure unique entries within this model.

Following the code structure to develop the database, Django requires a set of commands necessary to execute the development of the database in the '*db.sqlite3*' file as follows:

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

1.3 Populating the database

After ensuring that the database creation went smoothly, the next step would be to populate it with the given dataset (.csv files). The csv files are contained in the local project directory for easy access within the project. A python script will be created in the local app directory *proteins* and used to read and insert data from the .csv files given.

Path of csv files: "`../bioweb/<csv file name>`"

```
(django)  lawrenceho~/Documents/CM3035 Adv Web Development/coursework/bioweb[ (ﾉ°□°) ﾉ︵ ┻━┻ ls
assignment_data_sequences.csv   bioweb                    manage.py                    proteins
assignment_data_set.csv         db.sqlite3                pfam_descriptions.csv
```

Path of script: "../bioweb/proteins/scripts/load_data.py"

```
(django) lawrenceho~/Documents/CM3035 Adv Web Development/coursework/bioweb/proteins[ (´°□°) ´ ls
__init__.py        api.py             migrations         models.py          serializers.py     tests.py           views.py
admin.py           apps.py            model_factories.py scripts            templates          urls.py
```

The 3 .csv files given contain different sets of data. The dataset file name is assigned to the type of model that it will be inserted to for easier reference.

1. *'data_sequences.csv'* corresponds to [Sequence(protein_id, sequence)]
2. *'data_set.csv'* corresponds to [Protein(protein_id, sequence, taxonomy, length, domains)]
3. *'pfam_descriptions.csv'* corresponds to [Pfam(domain_id, domain_description)]

The first part of the script contains settings and declarations to link the script to the Django app and required files along with local variables of type defaultdict(list) that will store the retrieved data based on a key:value pair. This method of retrieving data from the csv file allows for a unique key value to ensure no duplicates, and at the same time easy access to the value stored in the key for locating shared key values with other variables. There are other dictionary local variables used to store created model objects for appending later on in the script:

```
# RELATIVE PATH for app
sys.path.append(os.path.realpath('..'))
# sys.path.append('..')
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'bioweb.settings')
django.setup()

from proteins.models import *

# RELATIVE PATH for csv files
protein_sequence_file = '../assignment_data_sequences.csv'
protein_file = '../assignment_data_set.csv'
pfam_file = '../pfam_descriptions.csv'

# class variables for caching data
# key:value pair for easy access to certain values depending on matching keys
protein_sequence = defaultdict(dict) #key value pair
taxonomy = defaultdict(list)
domains = defaultdict(list)
protein = defaultdict(list)
# dict to contain created model objects for appending
protein_sequence_rows={}
pfam_rows={}
taxonomy_rows={}

# clear database before insertion
Sequence.objects.all().delete()
Taxonomy.objects.all().delete()
Pfam.objects.all().delete()
Domains.objects.all().delete()
Protein.objects.all().delete()
```

```
ProteinDomainLink.objects.all().delete()
```

bioweb/proteins/scripts/load_data.py

In the second part of the script, it goes through a series of iterations to read the csv rows and assign them to the locally created variables, or directly create model objects.

For Sequence and Pfam models, since there is a direct correlation with the data and attributes, I will be creating the field values directly upon reading from the csv file. These rows of values will be kept in a local variable for access after exiting the loop.

```
# protein_sequence | protein_id
with open(protein_sequence_file) as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    for row in csv_reader:
        protein_sequence[row[0]] = row[1]
        # directly adding to Sequence model
        add = Sequence.objects.create(protein_id=row[0], sequence=row[1])
        add.save()
        protein_sequence_rows[row[0]] = add # save for access later

# pfam | pfam_id
with open(pfam_file) as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    for row in csv_reader:
        # directly adding to Pfam model
        add = Pfam.objects.create(domain_id=row[0], domain_description=row[1])
        add.save()
            pfam_rows[row[0]] = add # save for access later
```

bioweb/proteins/scripts/load_data.py

For the rest of the models (Protein, Taxonomy, Domains), there are PKs and FKs conjoining them, and thus require a more sophisticated approach to populating these entities.

Domains contain a FK connection with Pfam, thus the Pfam model has to be populated first before passing on the object into Domains creation. Likewise, Proteins has a many-to-many connection with Domains, and thus, Domains objects have to be created first before passing it into Proteins creation.

The large dataset contains duplicate values in some rows, so to filter those out, by including a check for pre-existing key in the variable, I can prevent any redundant data from populating the database. Similarly, for the many-to-many relation between Domains and Proteins, if there exist a matching key in the local variable, I will append the rest of the necessary data to the appropriate key:value pairs contained in the variable for me to extract later on in the code.

Basically, the following steps are taken to ensure data integrity while reading from the csv files:

1. Assign values to taxonomy (defaultdict) based on taxa_id
2. Assign values to domains (defaultdict) based on protein_id. Append values to the matching protein_id to contain multiple domains under a single protein_id.
3. Assign values to proteins (defaultdict) based on protein_id. Append values to the matching protein_id to contain multiple domain data under a single protein_id, otherwise create a new entry in the variable with the protein_id.

4. Exit csv_reader.
5. Create Taxonomy model objects based on taxonomy.
6. Create Domains model objects based on pfam_rows and domains variables.
7. Create Protein model objects based on all the above mentioned variables.

```
# protein details
with open(protein_file) as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    for row in csv_reader:
        # retrieve genus and species by splitting
        genus_species = row[3].split(' ', 1)
        # if already exist in taxonomy, then will not create a new entry
        if row[1] not in taxonomy:
            taxonomy[row[1]] = {"taxa_id": row[1], "clade": row[2], "genus": genus_species[0], "species": genus_species[1]}

        # there might be multiple domains per protein_id
        # append multiple domains to a single protein_id key (handling many-to-many)
        domains[row[0]].append({"pfam_id": row[5], "description": row[4], "start": row[6], "stop": row[7]})

        # in the dataset csv, there are rows of recurring proteins with different domain data
        # if there is no pre-existing match for the given protein_id, create a new entry.
        # otherwise, append multiple domains details to a single protein_id (handling many-to-many)
        if row[0] not in protein:
            protein[row[0]] = {"pfam_id": [row[5]], "description": row[4], "taxa_id": row[1], "length": row[8], "start": [row[6]], "stop":
[row[7]]}
        else:
            protein[row[0]]["pfam_id"].append(row[5])
            protein[row[0]]["start"].append(row[6])
            protein[row[0]]["stop"].append(row[7])

# create Taxonomy objects
for taxa_id, data in taxonomy.items():
    row = Taxonomy.objects.create(taxa_id=data["taxa_id"], clade=data["clade"], genus=data["genus"], species=data["species"])
    row.save()
    taxonomy_rows[taxa_id] = row # save to class var for use later when creating Proteins objects

# create Domains objects
for protein_id, data in domains.items():
    for values in data:
        try:
            row = Domains.objects.create(pfam=pfam_rows[values["pfam_id"]], description=values["description"],
start=values["start"], stop=values["stop"])
            row.save()
        except:
            DO_NOTHING
```

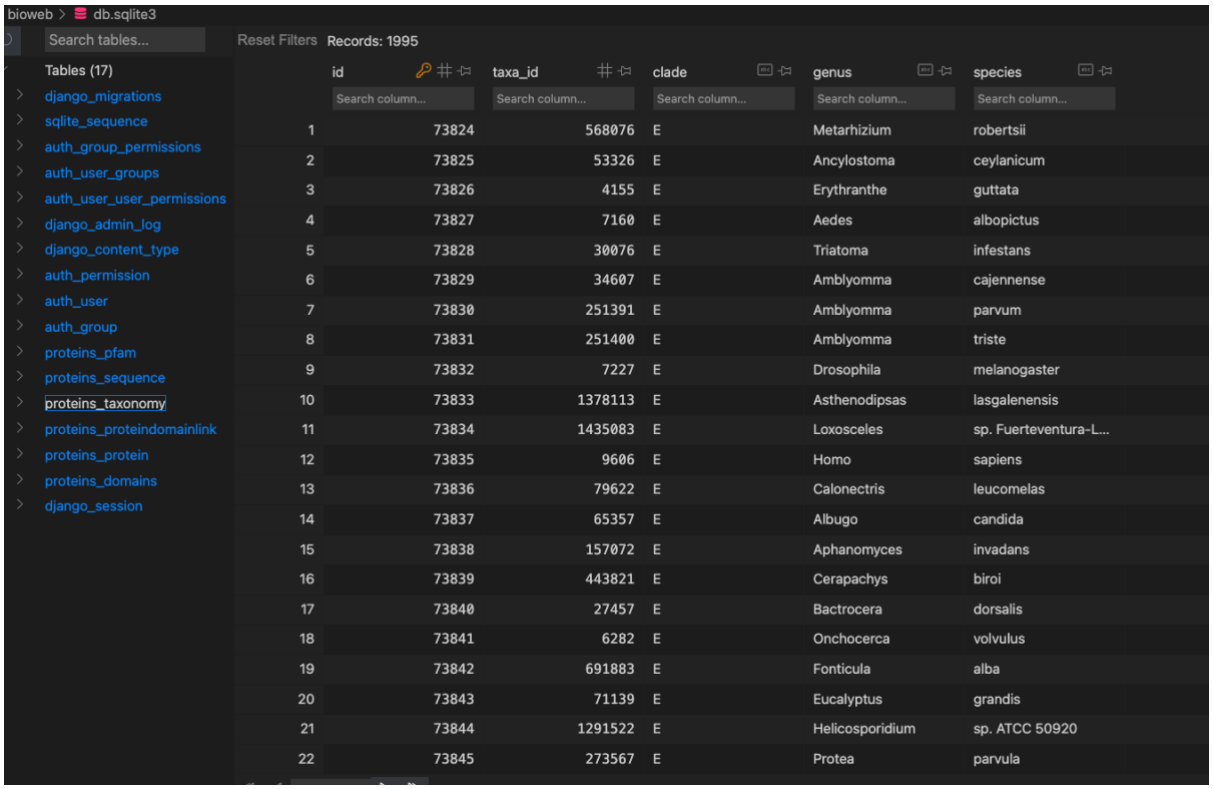
```python
# create Proteins objects
for protein_id, data in protein.items():
    row = Protein.objects.create(
        protein_id=protein_id,
        sequence=protein_sequence[protein_id],
        taxonomy=taxonomy_rows[data["taxa_id"]],
        length=data["length"],
    )
    # unzip appended domain values to a single protein_id
    for pfam_id, start, stop in zip(data['pfam_id'], data['start'], data['stop']):
        # retrieve Domains object that match the above existing conditions
        pfam_object = Pfam.objects.get(domain_id = pfam_id)
        domains_object = Domains.objects.filter(Q(pfam = pfam_object) & Q(start= start) & Q(stop= stop)).get()
        # add the Domains object to the Proteins object
        row.domains.add(*[domains_object])
    row.save()
```

bioweb/proteins/scripts/load_data.py

To run the script, enter the correct directory and call the python command (the following command is made in the proteins directory):

```
python scripts/populate_proteins.py
```

After executing the script, the database should be populated with the correct amount of data. An example screenshot of the populated Taxonomy models looks something like this:



	id	taxa_id	clade	genus	species
1	73824	568076	E	Metarhizium	robertsii
2	73825	53326	E	Ancylostoma	ceylanicum
3	73826	4155	E	Erythranthe	guttata
4	73827	7160	E	Aedes	albopictus
5	73828	30076	E	Triatoma	infestans
6	73829	34607	E	Amblyomma	cajennense
7	73830	251391	E	Amblyomma	parvum
8	73831	251400	E	Amblyomma	triste
9	73832	7227	E	Drosophila	melanogaster
10	73833	1378113	E	Asthenodipsas	lasgalenensis
11	73834	1435083	E	Loxosceles	sp. Fuerteventura-L...
12	73835	9606	E	Homo	sapiens
13	73836	79622	E	Calonectris	leucomelas
14	73837	65357	E	Albugo	candida
15	73838	157072	E	Aphanomyces	invadans
16	73839	443821	E	Cerapachys	biroi
17	73840	27457	E	Bactrocera	dorsalis
18	73841	6282	E	Onchocerca	volvulus
19	73842	691883	E	Fonticula	alba
20	73843	71139	E	Eucalyptus	grandis
21	73844	1291522	E	Helicosporidium	sp. ATCC 50920
22	73845	273567	E	Protea	parvula

sqlviewer showing populated Taxonomy table

1.4 Creating the home view

An home page is created to display and ensure that the models have been populated correctly. This also serves as the default page that will be rendered upon connecting to the localhost link. This is done through a simple index.html page with the path calling the views function that renders the page.

```
path('', views.index, name='index'),
```

proteins/urls.py

```
<html>
    <head></head>
    <body>
        <h1>Law Proteins List</h1>
        <table>
            <tr><th>Protein ID</th></tr>
            <!-- list out all proteins -->
            {% for protein in proteins %}
            <tr><td><a href="/api/protein/{{ protein.protein_id }}">{{ protein }}</a></td></tr>
            {% endfor %}
        </table>
    </body>
</html>
```

proteins/templates/proteins/index.html

```
def index(request):
    proteins = Protein.objects.all()
    return render(request, 'proteins/index.html', {'proteins': proteins})
```

proteins/views.py

1.5 Setting up the API views

There are a total of 5 endpoints to cover for the project's requirements. I will be using the Django Rest Framework (DRF) package to render the API views. In the settings file, I have installed and included the DRF package for importing later on.

```
INSTALLED_APPS = [
    'rest_framework',
    'bootstrap4',
    'proteins.apps.ProteinsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

bioweb/settings.py

Next, creating the python file *'api.py'* that contains the logic to generate these views and importing all the necessary packages required.

```
import json
from django.http import JsonResponse, HttpResponse
from django.utils.translation import get_supported_language_variant #default Response
from django.views.decorators.csrf import csrf_exempt
from django.db.models import Q, query

#rest api packages
from rest_framework.parsers import JSONParser
from rest_framework.decorators import api_view
from rest_framework.response import Response #rest Response, better for development
from rest_framework import status

#packages for easier logic to handle CRUD
from rest_framework import generics
from rest_framework import mixins
# permissions
from rest_framework.permissions import IsAuthenticated

#package for appending querysets
from itertools import chain

from .models import *
from .serializers import *
```

proteins/api.py

1.6 Serializers

Based on the endpoint requirements, various serializer objects will be created that will aid in displaying the required data to the user. These serializer objects corresponds to each model object, and a few more that caters for unique API views.

```
# ============= Serializers that corresponds to Model objects ===================
# ==============================================================================
class TaxonomySerializer(serializers.ModelSerializer):
    class Meta:
        model = Taxonomy
        fields = ['taxa_id', 'clade', 'genus', 'species']

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

```
class DomainsSerializer(serializers.ModelSerializer):
    pfam = PfamSerializer()
    class Meta:
        model = Domains
        fields = ['pfam', 'description', 'start', 'stop']

# ModelSerializer is more dynamic and straightforward
class ProteinSerializer(serializers.ModelSerializer):
    taxonomy = TaxonomySerializer()
    domains = DomainsSerializer(many=True)
    class Meta:
        model = Protein
        fields = ['protein_id', 'sequence', 'taxonomy', 'length', 'domains']
        depth = 1

    # creating domains object and proteins object based on their many-to-many relation
    def create(self, validated_data):
        domains_data = validated_data.pop('domains')
        protein = Protein.objects.create(**validated_data)
        Domains.objects.create(protein=protein, **domains_data)
        return Protein

class ProteinDomainLinkSerializer(serializers.ModelSerializer):
    protein = ProteinSerializer()
    domains = DomainsSerializer(many=True)
    class Meta:
        model = ProteinDomainLink
        fields = ['protein', 'domains']

class ProteinDomainLinkSerializer(serializers.ModelSerializer):
    protein = ProteinSerializer()
    domains = DomainsSerializer(many=True)
    class Meta:
        model = ProteinDomainLink
        fields = ['protein', 'domains']

# ================== Serializers to cater for API Views ========================
# ==============================================================================
# for ENDPOINT 4
class ProteinTaxaIDSerializer(serializers.ModelSerializer):
    class Meta:
        model = Protein
        fields = ['id','protein_id']
# for ENDPOINT 5
class DomainsTaxaIDSerializer(serializers.ModelSerializer):
```

```
    pfam = PfamSerializer()
    class Meta:
        model = Domains
        fields = ['pk','pfam']
```

proteins/serializers.py

1.7 Creating the API views

- **ENDPOINT 1**: POST http://127.0.0.1:8000/api/protein/ - add a new record

Users can take reference from the displayed protein item and enter the details in a json format to add in a new protein record. @api_view will be used to handle the post request, and the ProteinSerializer class will be called to ensure the validity of the request data.

```
# add new protein on 'api/protein/' path
@api_view(['GET','POST'])
def CreateProtein(request):
    # display one protein for reference
    if request.method == 'GET':
        protein = Protein.objects.filter(protein_id="A0A016S8J7")
        serializer = ProteinSerializer(protein, many=True)
        return Response(serializer.data)

    # insert via json data
    if request.method == 'POST':
        # get json data based on POST request
        # assign to ProteinSerializer and render in page if valid
        json_data = json.loads(request.body.decode(encoding='utf-8'))
        protein_serializer = ProteinSerializer(data=json_data)
                # check for valid data
        # if valid, save the data and display it to the user
        if protein_serializer.is_valid():
            protein_serializer.save()
            return Response(protein_serializer.data,status=status.HTTP_201_CREATED)
        return Response(protein_serializer.errors,status=status.HTTP_400_BAD_REQUEST)
```

proteins/api.py

A new url path based on the requirements is created that calls the api_view and renders the web page.

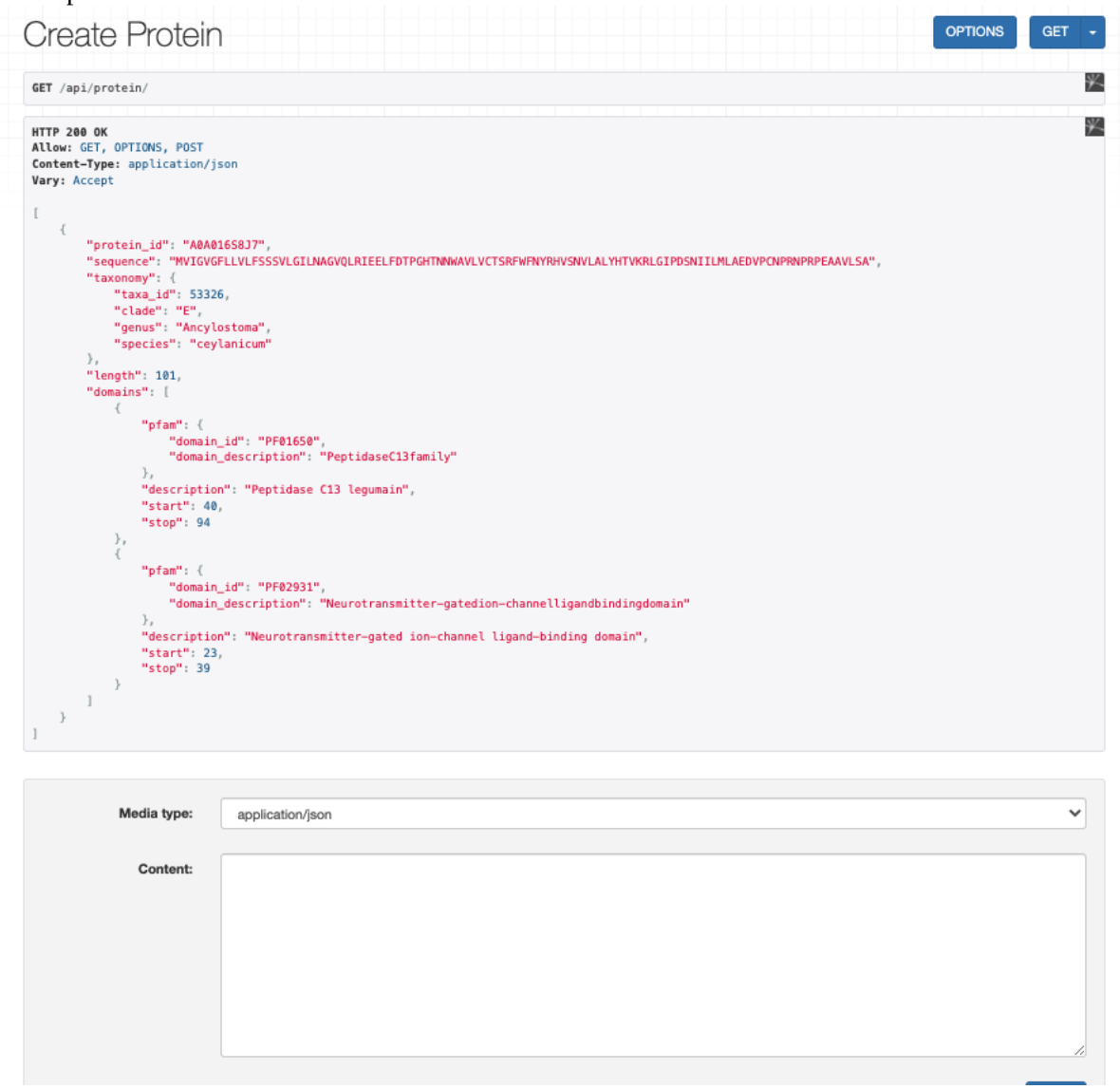
```
path('api/protein/', api.CreateProtein, name="create_protein"),
```

proteins/urls.py

Running the server with the following command in the bioweb directory:

```
python manage.py runserver
```

The browser then renders the page based on the CreateProtein api_view function. It displays a single protein object, and at the bottom of the page, a content box allows users to enter the new protein object details in a json format. Upon successful insertion of the data, the page displays the newly added protein in the same format:



browser view of endpoint-1

- **ENDPOINT 2**: GET http://127.0.0.1:8000/api/protein/[PROTEIN ID] - return the protein sequence and all we know about it

The packages *mixins* and *generics* will be used to render this GET request. The lookup field takes the value appended to the url and uses it as a query parameter along with the *ProteinSerializer* class to get the desired output.

```
# view protein details based on protein_id on 'api/protein/<protein_id>' path
class ProteinDetailsList(mixins.RetrieveModelMixin, generics.GenericAPIView): # generic API view
    # lookup_field based on the input appended to the url
    lookup_field = 'protein_id'
    queryset = Protein.objects.all()
```

```
    serializer_class = ProteinSerializer
    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)
```

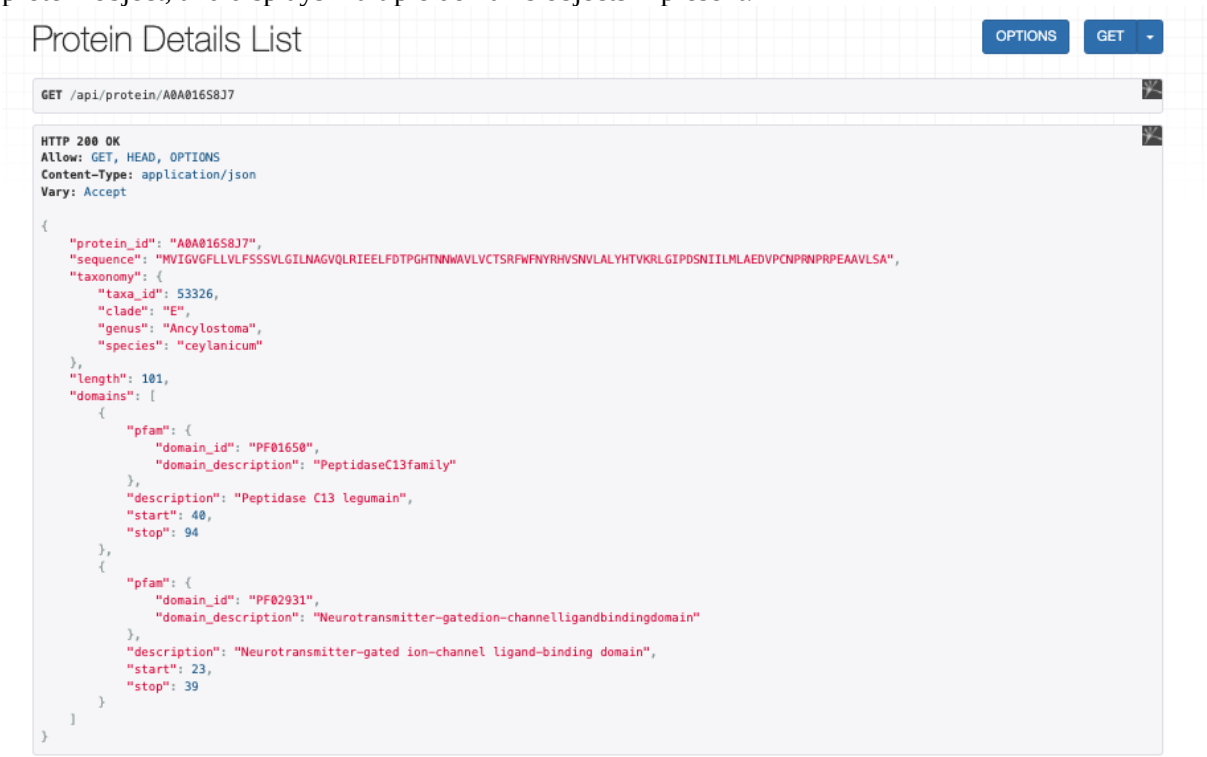
proteins/api.py

A new url path based on the requirements is created that calls the api_view and renders the web page.

```
path('api/protein/<str:protein_id>', api.ProteinDetailsList.as_view(), name="protein_details_list"),
```

proteins/urls.py

The browser renders the page based on the *ProteinDetailsList.as_view()*function. It displays a single protein object, and displays multiple domains objects if present:



browser view of endpoint-2

- **ENDPOINT 3 -** GET http://127.0.0.1:8000/api/pfam/[PFAM ID] - return the domain and it's description

Similar to ENDPOINT 2, the packages *mixins* and *generics* will be used to render this GET request. The lookup field takes the value appended to the url and uses it as a query parameter along with the *PfamSerializer* class to get the desired output.

```
# ENDPOINT 3: get pfam object details based on pfam_id on 'api/pfam/<pfam_id>'
class PfamDetails(mixins.RetrieveModelMixin, generics.GenericAPIView): # generic API view
    lookup_field = 'domain_id' #aka pfam_id
    queryset = Pfam.objects.all()
    serializer_class = PfamSerializer

    def get(self, request, *args, **kwargs):
```

```
    return self.retrieve(request, *args, **kwargs)
```

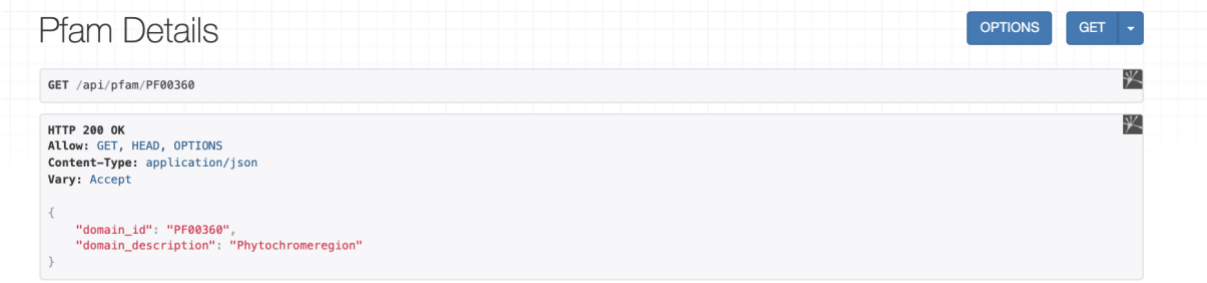
proteins/api.py

A new url path based on the requirements is created that calls the api_view and renders the web page.

```
path('api/pfam/<str:domain_id>', api.PfamDetails.as_view(), name="pfam_details"),
```

proteins/urls.py

The browser renders the page based on the *PfamDetails.as_view()*function. It displays a single pfam object based on the query given by the id:



browser view of endpoint-3

- **ENDPOINT 4 -** GET http://127.0.0.1:8000/api/proteins/[TAXA ID] - return a list of all proteins for a given organism

The query here is a little more sophisticated. There exists a relation between Protein model and Taxonomy models, so through use the *api_view(['GET'])* function, the taxa_id passed together with the request is used to query existing Taxonomy object, followed by all the Protein objects that contain that particular Taxonomy object. The *ProteinTaxaIDSerializer* is then called to display the primary key of the domain data (Proteins' true PK) and the protein_id from the Protein objects.

```
# ENDPOINT 4: get protein_id and id based on taxa_id on 'api/proteins/<taxa_id>'
@api_view(['GET'])
def ProteinTaxaList(request, taxa_id): #takes in taxa_id upon request in url
    if request.method == 'GET':
        # retrieve Taxonomy object, then Protein objects based on taxa_id
        taxa_object = Taxonomy.objects.get(taxa_id = taxa_id)
        protein_id_objects = Protein.objects.filter(taxonomy = taxa_object)
        serializer = ProteinTaxaIDSerializer(protein_id_objects, many=True)
        return Response(serializer.data)
```

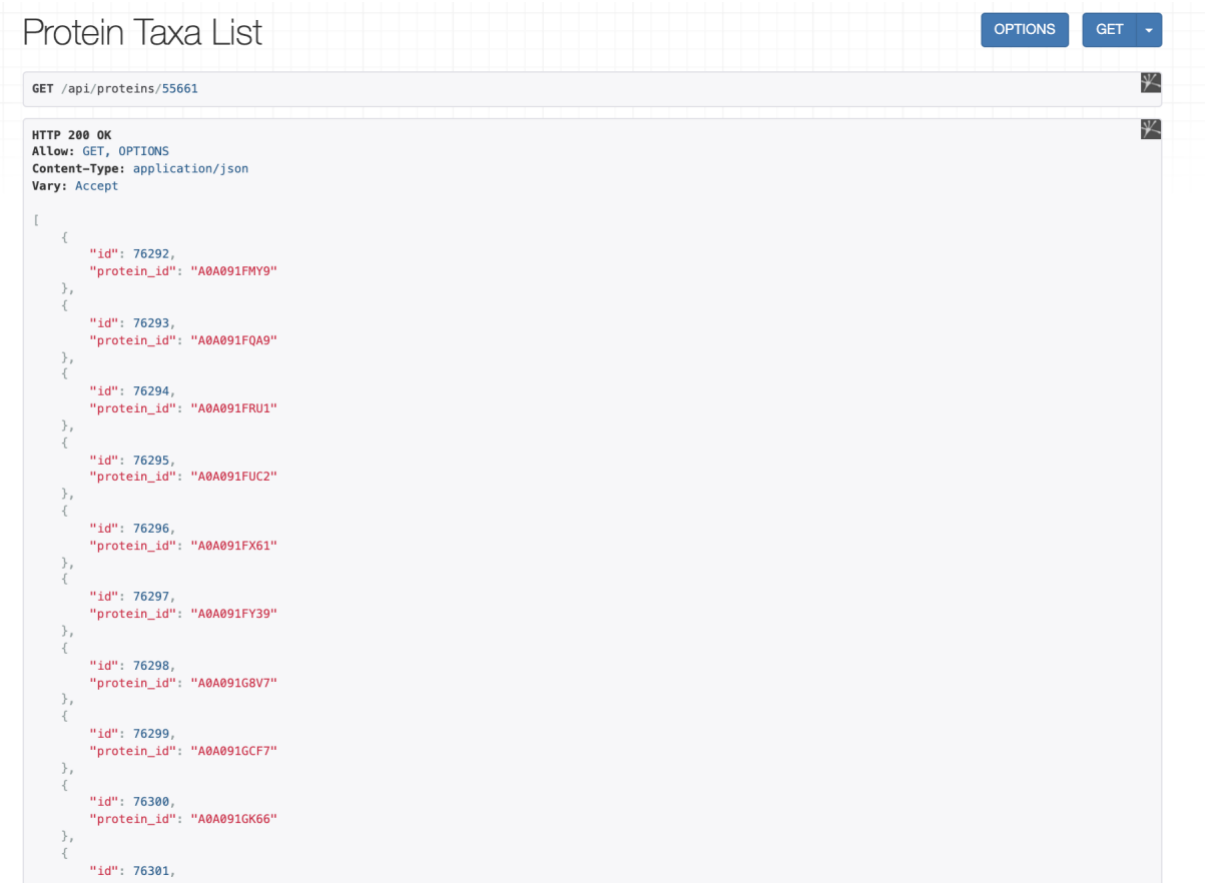
proteins/api.py

A new url path based on the requirements is created that calls the api_view and renders the web page.

```
path('api/proteins/<int:taxa_id>', api.ProteinTaxaList, name="proteins_taxa_list"),
```

proteins/urls.py

The browser renders the page based on the *ProteinTaxaList* function. It displays all protein_pk and protein_id corresponding to the querid Protein objects based on the given taxa_id:



browser view of endpoint-4

- **ENDPOINT 5 -** GET http://127.0.0.1:8000/api/pfams/[TAXA ID] - return a list of all domains in all the proteins for a given organism.

To obtain the list of all domains for a given organism based on the taxa_id requires additional steps to deal with the domains model structure (can have many). Only the Proteins model has a relation with the Domains model, hence, the query process will be from Taxonomy → Proteins → Domains. The following steps are taken to retrieve the domains objects for said organism:

1. Firstly, the Taxonomy objects will be queried based on the taxa_id.
2. Next, the Protein objects will be queried based on the Taxonomy objects retrieved.
3. There can be multiple Protein objects retrieved with the Taxonomy object.
    a. A loop has to be created to go through each individual Protein object.
4. Through the loop, by extracting each individual Protein object, the Domains objects are then retrieved.
    a. Similarly, there can be multiple Domains objects retrieved.
    b. Another loop will be created to look for individual Domains object PK.
    c. This PK values will be appended to a local list variable.
5. The result objects will be queried using the PKs contained in the list variable.
6. Lastly, a customized *DomainsTaxaIDSerializer* serializer object will be used to display the data.

```
# ENDPOINT 5: get domains pfam details based on taxa_id on 'api/pfams/<taxa_id>'
@api_view(['GET'])
def DomainsTaxaList(request, taxa_id): #takes in taxa_id upon request in url
```

```
# local var to store multiple domains id upon query
domains_id_list = []
if request.method == 'GET':
    # retrieve Taxonomy object, then Protein objects based on taxa_id
    taxa_object = Taxonomy.objects.get(taxa_id = taxa_id)
    protein_objects = Protein.objects.filter(taxonomy = taxa_object)
    # loop through respective objects in protein_objects that matched with taxa_id
    for protein in protein_objects:
        # look for all matching Domains objects with protein_objects
        domains_objects = Domains.objects.filter(protein = protein)
        # getting the domain_pk
        for domains in domains_objects:
            domains_id_list.append(domains.pk)


    # retrive domains objects basd on pk
    result = Domains.objects.filter(pk__in = domains_id_list)
    serializer = DomainsTaxaIDSerializer(result, many=True)
    return Response(serializer.data)
```

proteins/api.py

A new url path based on the requirements is created that calls the api_view and renders the web page.

```
path('api/pfams/<int:taxa_id>', api.DomainsTaxaList, name="domains_taxa_list"),
```

proteins/urls.py

The browser renders the page based on the *DomainsTaxaList* function. It displays all domains_pk and pfam objects based on the given taxa_id:

browser view of endpoint-5

- **ENDPOINT 6 -** GET http://127.0.0.1:8000/api/coverage/[PROTEIN ID] - return the domain coverage for a given protein

This is a simple query to look for all the domain coverage values contained within a Protein object.

1. Query Protein objects based on given protein_id
2. Query Domains objects based on retrieved Protein object
3. Loop through all Domains objects retrieved and calculate the total start and stop values.
4. Calculate the coverage based on the values.

```
# ENDPOINT 6: get coverage details
@api_view(['GET'])
def Coverage(request, protein_id): #takes in protein_id upon request in url
    if request.method == 'GET':
        sum_start = 0
        sum_stop = 0
        # retrieve protein object based on protein_id
        protein = Protein.objects.get(protein_id = protein_id)
        # retrive domains objects based on protein object retrieved (might have > 1)
        domains_objects = Domains.objects.filter(protein = protein)
        # loop thru domains_objects to calculate the values
        for domain in domains_objects:
            sum_start += domain.start
            sum_stop += domain.stop
        cal = (sum_start - sum_stop) / protein.length
        return HttpResponse("coverage:    " + str(cal))
```

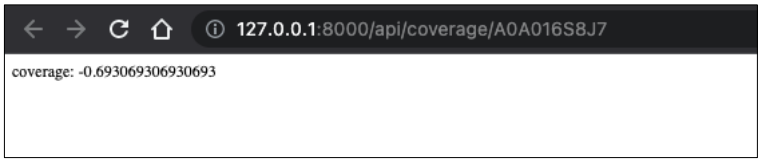
proteins/api.py

A new url path based on the requirements is created that calls the api_view and renders the web page.

```
path('api/coverage/<str:protein_id>', api.Coverage, name="coverage"),
```

proteins/urls.py

The browser renders the page based on the *Coverage* function. It renders a simple message displaying the calculated coverage amount:



1.8 Unit Tests - Serializers

The first set of tests are applied to the serializers, to ensure that the creation of the objects are successful. To aid with the tests, I have imported factory packages in *model_factories.py* of the project. These factory objects will be used to instantiate the test objects.

```
# Factory objects used for testing
class SequenceFactory(factory.django.DjangoModelFactory):
    protein_id = "A123"
    sequence = "SEQUENCE"
```

```
    class Meta:
        model = Sequence

class TaxonomyFactory(factory.django.DjangoModelFactory):
    taxa_id = "123"
    clade = 'E'
    genus = "hocus"
    species = "pocus"
    class Meta:
        model = Taxonomy

class PfamFactory(factory.django.DjangoModelFactory):
    domain_id = "pfam123"
    domain_description = "pfam description"
    class Meta:
        model = Pfam

class DomainsFactory(factory.django.DjangoModelFactory):
    pfam = factory.SubFactory(PfamFactory)
    description = "domains description"
    start = 1
    stop = 99
    class Meta:
        model = Domains

# create many 2 many domains
class ProteinFactory(factory.django.DjangoModelFactory):
    protein_id = factory.Sequence(lambda n: "protein_id%d" % n+str(1)) #protein_id1
    sequence = "2SEQUENCE"
    taxonomy = factory.SubFactory(TaxonomyFactory)
    length = 234

    # many-to-many object creation
    @factory.post_generation
    def domains(self, create, extracted, **kwargs):
        if not create:
            # Simple build, do nothing.
            return
        if extracted:
            for domain in extracted:
                self.domains.add(domain)
    class Meta:
        model = Protein

class ProteinDomainLinkFactory(factory.django.DjangoModelFactory):
```

```
    protein = factory.SubFactory(ProteinFactory)
    domains = factory.SubFactory(DomainsFactory)
    class Meta:
        model = ProteinDomainLink
```

proteins/model_factories.py

Following which, under DRF framework, the APITestCase package will be imported to assist with the numerous test cases to come.

```
# rest api testing packages
from rest_framework.test import APIRequestFactory
from rest_framework.test import APITestCase
```

There are a set of tests for the serializer fields and values contained in them. The logic for serializer testing are as follows:

```
# create protein object based on other sub objects
    def setUp(self):
        # instantiate factory objects
        self.taxonomy1 = TaxonomyFactory.create(taxa_id=1)
        self.pfam1 = PfamFactory.create(pk=1, domain_id="domain1")
        self.domains1 = DomainsFactory.create(pk=1, pfam=self.pfam1)
        self.protein1 = ProteinFactory.create(pk=1, protein_id="protein1", taxonomy=self.taxonomy1)
        self.proteinDomainLink = ProteinDomainLinkFactory.create(protein=self.protein1, domains=self.domains1)
        self.protein1.domains.add(self.domains1)

        # instantiate serializer objects
        self.taxonomySerializer = TaxonomySerializer(instance=self.taxonomy1)
        self.pfamSerializer = PfamSerializer(instance=self.pfam1)
        self.domainsSerializer = DomainsSerializer(instance=self.domains1)
        self.proteinSerializer = ProteinSerializer(instance=self.protein1)
        self.proteinDomainLinkSerializer = ProteinDomainLinkSerializer(instance=self.proteinDomainLink)
        self.proteinTaxaIDSerializer = ProteinTaxaIDSerializer(instance=self.protein1)
        self.domainsTaxaIDSerializer = DomainsTaxaIDSerializer(instance=self.pfam1)

    def tearDown(self):
        # Sequence.objects.all().delete()
        # Taxonomy.objects.all().delete()
        # Pfam.objects.all().delete()
        # Domains.objects.all().delete()
        # Protein.objects.all().delete()
        SequenceFactory.reset_sequence(0)
        TaxonomyFactory.reset_sequence(0)
        PfamFactory.reset_sequence(0)
        DomainsFactory.reset_sequence(0)
        ProteinFactory.reset_sequence(0)
```

```
# ================== SET OF SERIALIZER TESTS ==========================
# ================== comment out when necessary =====================
# ====================================================================
def test_taxonomySerializer(self):
    data = self.taxonomySerializer.data
    # check for correct keys
    self.assertEqual(set(data.keys()), set(['taxa_id', 'clade', 'genus', 'species']))
    # check for correct existing taxa data
    self.assertEqual(data['taxa_id'],1)

def test_pfamSerializer(self):
    data = self.pfamSerializer.data
    # check for correct keys
    self.assertEqual(set(data.keys()), set(['domain_id', 'domain_description']))
    # check for correct existing pfam data
    self.assertEqual(data['domain_id'],'domain1')

def test_domainsSerializer(self):
    data = self.domainsSerializer.data
    # check for correct keys
    self.assertEqual(set(data.keys()), set(['pfam', 'description', 'start', 'stop']))
    # check for correct existing domains data
    self.assertEqual(data['pfam']['domain_id'],self.pfam1.domain_id)

# check for domains existence in protein serializer (many-to-many)
def test_proteinSerializer(self):
    data = self.proteinSerializer.data
    # check for correct keys
    self.assertEqual(set(data.keys()), set(['protein_id', 'sequence', 'taxonomy', 'length', 'domains']))
    # check for existing protein data
    self.assertEqual(data['domains'][0]['pfam']['domain_id'],self.domains1.pfam.domain_id)
```

proteins/tests.py

1.9 Unit Tests - Routes

The following set of test cases caters for the routes and requests handled by the api. There are a set of tests contained in each testing function for each endpoint given.

```
# ================== ROUTE TESTING ====================
class RouteTest(APITestCase):
    taxa1 = None
    pfam1 = None
    domains1 = None
    protein1 = None

    # setup class variables to use for testing
```

```
# override
def setUp(self):
    # instantiate factory objects
    self.taxa1 = TaxonomyFactory.create(pk=1,taxa_id=123)
    self.pfam1 = PfamFactory.create(pk=1, domain_id="domain1")
    self.domains1 = DomainsFactory.create(pk=1, pfam=self.pfam1)
    self.protein1 = ProteinFactory.create(pk=1, protein_id="protein1", taxonomy=self.taxa1)
    self.proteinDomainLink = ProteinDomainLinkFactory.create(protein=self.protein1, domains=self.domains1)
    self.protein1.domains.add(self.domains1)

    # urls (actions) to handle requests and responses for testing
    self.create_url = "/api/protein/"
    self.protein_details_url = reverse('protein_details_list', kwargs={'protein_id': "protein1"})
    self.pfam_details_url = reverse('pfam_details', kwargs={'domain_id': "domain1"})
    self.proteins_taxa_url = reverse('proteins_taxa_list', kwargs={'taxa_id': 123})
    self.domains_taxa_url = reverse('domains_taxa_list', kwargs={'taxa_id': 123})
    self.coverage_url = reverse('coverage', kwargs={'protein_id': "protein1"})
    self.bad_url = "/api/proteins/XXX/"

# override
def tearDown(self):
    # Sequence.objects.all().delete()
    # Taxonomy.objects.all().delete()
    # Pfam.objects.all().delete()
    # Domains.objects.all().delete()
    # Protein.objects.all().delete()
    SequenceFactory.reset_sequence(0)
    TaxonomyFactory.reset_sequence(0)
    PfamFactory.reset_sequence(0)
    DomainsFactory.reset_sequence(0)
    ProteinFactory.reset_sequence(0)

# ======================= SET OF ROUTE TESTS ==========================
# ===================== comment out when necessary =====================
# ======================================================================
def test_CreateProtein(self): # only managed to test for GET
    response = self.client.get(self.create_url, format='json')
    self.assertEqual(response.status_code, 200) # response status PASS
    # FAIL TO TEST FOR POST HERE >>>>
    # response = self.client.post(self.create_url, self.protein1, format='json')
    # data = json.loads(response.content)
    # self.assertEqual(response.status_code, 200) # response status PASS


# test [GET /api/protein/[PROTEIN ID] - return the protein sequence and all we know about it]
```

```
# PASS
def test_ProteinDetailsReturnsSuccess(self):
    response = self.client.get(self.protein_details_url, format='json')
    response.render()
    # check for correct arrived data
    data = json.loads(response.content) # arrived data
    self.assertTrue('sequence' in data) # check for sequence(key) value
    self.assertEqual(data['sequence'], '2SEQUENCE') # check for existing sequence value loaded

# test [GET /api/pfam/[PFAM ID] - return the domain and it's description]
# PASS
def test_PfamDetailsReturnsSuccess(self):
    response = self.client.get(self.pfam_details_url, format='json')
    response.render()
    # check for correct arrived data
    data = json.loads(response.content) # arrived data
    self.assertTrue('domain_description' in data) # check for domain_description(key) value
    self.assertEqual(data['domain_description'], 'pfam description') # check for existing sequence value loaded

# test [GET /api/proteins/[TAXA ID] - return a list of all proteins for a given organism]
# since it is a list, we only want the first item in the list: accessible by index [0]
# PASS
def test_ProteinTaxaDetailsReturnsSuccess(self):
        response = self.client.get(self.proteins_taxa_url, format='json')
        response.render()
        # check for correct arrived data
        data = json.loads(response.content) # arrived data
        self.assertTrue('id' in data[0]) # check for id key in data
        obj = Taxonomy.objects.get(id = data[0]['id']) # retrieve Taxonomy object based on pk
        self.assertEqual(obj.taxa_id, 123) # check for taxa_id match with factory model

# test [GET /api/pfams/[TAXA ID] - return a list of domains pfam details based on taxa_id]
# since it is a list, we only want the first item in the list: accessible by index [0]
# PASS
def test_DomainsTaxaDetailsReturnsSuccess(self):
        response = self.client.get(self.domains_taxa_url, format='json')
        response.render()
        # check for correct arrived data
        data = json.loads(response.content) # arrived data
        self.assertTrue('pfam' in data[0]) # check for taxonomy_id key in data
        taxa_obj = Taxonomy.objects.get(id = data[0]['pk']) # retrieve Taxonomy object based on pk
        protein_obj = Protein.objects.get(taxonomy = taxa_obj) # retrieve Protein object based on taxa_obj
        domain_obj = Domains.objects.get(protein = protein_obj) # retrieve Domain object based on protein_obj
        self.assertEqual(domain_obj.pfam.domain_description, 'pfam description') # check for taxa_id match with factory model
```

```python
# TEST [GET /api/coverage/[PROTEIN ID] - return the domain coverage for a given protein]
# PASS
def test_CoverageReturnsSuccess(self):
    response = self.client.get(self.coverage_url, format='json')
    # check for successful message status
    self.assertEqual(response.status_code, 200)
```

proteins/tests.py

To execute the unit tests, head into the project directory (bioweb) and enter the following command:

```
python manage.py test
```

The logged test cases generally serves its purpose in ensuring the execution of the app is proper. Here is a console log screenshot of the results:

```
^C(advanced_web_dev) (base)  lawrenceho~/Documents/CM3035 Adv Web Development/coursework/bioweb[ (ノ°□°) ノ︵ ┻━┻ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..........
----------------------------------------------------------------------
Ran 10 tests in 0.070s

OK
Destroying test database for alias 'default'...
```