Advanced Web Development Report

R1: The application contains the basic functionality described in class

a) correct use of models and migrations

In the models.py folder, I have included all the tables that I will implement, namely Taxonomy, Pfam, Domain, Protein. I have chose these four models as they are sufficient to implement all the necessary attributes, and they are designed such that it can be called easily to fulfil the REST page requirements.

```python
from django.db import models

# Creation of Taxonomy model
class Taxonomy(models.Model):
    taxa_id = models.CharField(max_length = 30, null = False, blank = False)
    clade = models.CharField(max_length = 1, default ='E')
    genus = models.CharField(max_length = 256, null = False, blank = False)
    species = models.CharField(max_length = 256, null = False, blank = False)
    def __str__(self):
        return str(self.taxa_id)

# Creating the Pfam model
class Pfam(models.Model):
    domain_id = models.CharField(max_length = 30, null = False, blank = False)
    domain_description = models.CharField(max_length = 256, null = False, blank = False)
    def __str__(self):
        return self.domain_id

# Creating the Domain model
class Domain(models.Model):
    # pfam_id is a foreign key
    pfam_id = 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)
```

```python
#Creating the Protein model
class Protein(models.Model):
    protein_id = models.CharField(max_length = 30, null = False, blank = False)
    sequence = models.CharField(max_length = 256, null = False, blank = False)
    # taxonomy is a foeign key
    taxonomy = models.ForeignKey(Taxonomy, on_delete = models.CASCADE)
    length = models.CharField(max_length = 8, null = False, blank = False)
    domains = models.ManyToManyField(Domain, through='Domain_protein_link')
    def __str__(self):
        return str(self.protein_id)

# Creating the linker model to link between domain and protein models
class Domain_protein_link(models.Model):
    #domain and protein are foreign keys
    domain = models.ForeignKey(Domain, on_delete = models.CASCADE)
    protein = models.ForeignKey(Protein, on_delete = models.CASCADE)
    def special_save(self):
        pass
```

For example, in the above Protein table, it contains a link to the table Domain and Taxonomy. When trying to render the first Django webpage and give results shown below, I simply need to call my Protein table in my serializers to initialise protein_id, sequence, Taxonomy table which will contain all the attributes taxa_id, clade, genus, and species, followed by length, and finally the Domains table which is facilitated by the model design, to include pfam_id, which then links to the Pfam table to call domain_id and domain description, followed by start and stop in the Domain table. Hence the model is created after this, such that it follows the sequence and makes it easy to output later.

```
GET  http://127.0.0.1:8000/api/protein/[PROTEIN ID] - return the protein sequence and all we know about it
http://127.0.0.1:8000/api/protein/A0A016S8J7 returns
{
    "protein_id": "A0A016S8J7",
    "sequence": "MVIGVGFLLVLFSSSVLGILNAGVQLRIEELFDTPGHTNNWAVLVCTSRFWFNYRHVSNVLALYHTVKRLGIPDSNIILMLAEDVPCNPRNPRPEAAVLSA",
    "taxonomy": {
        "taxa_id": 53326,
        "clade": "E",
        "genus": "Ancylostoma",
        "species": "ceylanicum"
    },
    "length": 101,
    "domains": [
        {
            "pfam_id": {
                "domain_id": "PF01650",
                "domain_description": "PeptidaseC13family"
            },
            "description": "Peptidase C13 legumain",
            "start": 40,
            "stop": 94
        },
        {
            "pfam_id": {
                "domain_id": "PF02931",
                "domain_description": "Neurotransmitter-gatedion-channelligandbindingdomain"
            },
            "description": "Neurotransmitter-gated ion-channel ligand-binding domain",
            "start": 23,
            "stop": 39
        }
    ]
}
```

```
def __str__(self):
    return str(self.taxa_id)
```

The code above shows how the primary key for a table is initialised.

```
    domain = models.ForeignKey(Domain, on_delete = models.CASCADE)
    protein = models.ForeignKey(Protein, on_delete = models.CASCADE)
```

Foreign keys are initialised by the code shown above.

Subsequently, to make migrations, I used the populate.py file to make migrations and create the script. The CSV files were added by dragging them into a new folder called data, which will be called later.

```
import os
import sys
import django
import csv
from collections import defaultdict

settings_dir = os.path.dirname(__file__)
PROJECT_ROOT = os.path.abspath(os.path.dirname(settings_dir))
sys.path.append(PROJECT_ROOT)
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'organism_data.settings')
django.setup()
from organism_api.models import Domain, Taxonomy, Pfam, Protein, Domain_protein_link

# naming the csv files
data_seq_file = 'data/assignment_data_sequences.csv'
data_set_file = 'data/assignment_data_set.csv'
pfam_data_file = 'data/pfam_descriptions.csv'
```

The code above shows the relevant modules being imported. Each csv file in the data folder is then being set to a variable for example, data_seq_file, to be manipulated later.

```
20  # creating empty sets to put values in later
21  pfam = set()
22  taxonomy = set()
23  domain = set()
24  protein = set()
25  domain_protein_link = set()
26  test = set()
27  testing = set()
28
```

The code above creates empty sets and a temporary storage to hold the necessary data from each relative file, which corresponds to the values needed for each table.

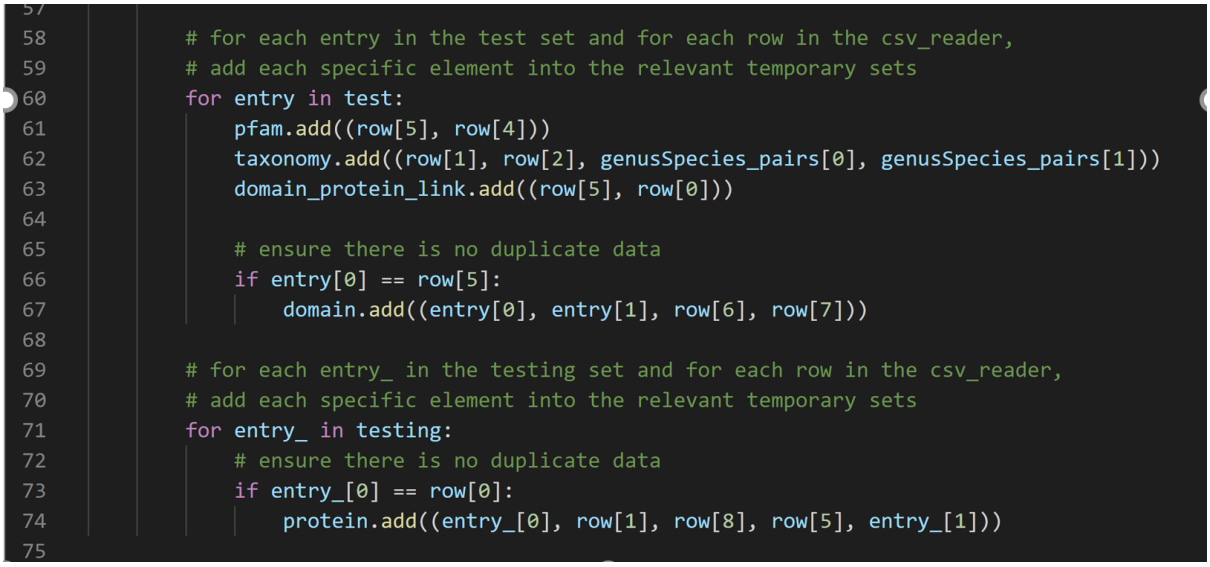
```
28
29  # opening the assignment_data_sequences csv file
30  with open(data_seq_file) as csv_file:
31      csv_reader = csv.reader(csv_file, delimiter=',')
32
33      for row in csv_reader:
34          # saving elements 0 and 1 in csv file into temporary set named testing
35          testing.add((row[0], row[1]))
36
37  # opening the pfam_descriptions csv file
38  with open(pfam_data_file) as csv_file:
39      csv_reader = csv.reader(csv_file, delimiter=',')
40
41      for row in csv_reader:
42          # saving elements 0 and 1 in csv file into temporary set named test
43          test.add((row[0], row[1]))
44
```

The first line of code above opens the csv file and reads it with csv_reader. Each element is separated by the delimiter ‘,’. It then loops each row in the file and adds it to the relevant sets. The code shows row[0] and row[1] from the assignment_data_seq.csv file being put as the first and second item into the empty set testing.

The process is then repeated for the next csv file as shown in line 36 to 43, where row[0] and row[1] from the pfam_data_file.csv is added to the test set.

```
45  # opening the assignment_data_set csv file
46  with open(data_set_file) as csv_file:
47      csv_reader = csv.reader(csv_file, delimiter=',')
48
49      for row in csv_reader:
50          # splitting genus species pairs by spacing
51          genusSpecies_pairs = row[3].split(" ")
52
53          for var in range(len(genusSpecies_pairs)):
54              if (var>1):
55                  # saving element in index 1, 2, 3, and so on into genusSpecies_pairs[1]
56                  genusSpecies_pairs[1] += genusSpecies_pairs[var]
57
```

The screenshot above opens the last csv file title data_set_file. Next, for each row in the file that is separated by the delimited ‘,’, row[3] is split by the spacing and added into a new dictionary named genusSpecies_pairs. Subsequently, for all rows from row[1] onwards, they are combined and added to the genusSpecies[1]. This corresponds to the species element to be stored later. genusSpecies[0] which was extracted previously will be added as the genus element later.



```
58          # for each entry in the test set and for each row in the csv_reader,
59          # add each specific element into the relevant temporary sets
60          for entry in test:
61              pfam.add((row[5], row[4]))
62              taxonomy.add((row[1], row[2], genusSpecies_pairs[0], genusSpecies_pairs[1]))
63              domain_protein_link.add((row[5], row[0]))
64
65              # ensure there is no duplicate data
66              if entry[0] == row[5]:
67                  domain.add((entry[0], entry[1], row[6], row[7]))
68
69          # for each entry_ in the testing set and for each row in the csv_reader,
70          # add each specific element into the relevant temporary sets
71          for entry_ in testing:
72              # ensure there is no duplicate data
73              if entry_[0] == row[0]:
74                  protein.add((entry_[0], row[1], row[8], row[5], entry_[1]))
75
```

The screenshot above shows the same procedure to add all the relevant data into sets related to its table.

In line 60 and 71, for each entry or entry_ in the test and testing sets respectively, they are added into the they respective temporary sets such as the pfam() set, in conjunction with each row in the csv_reader. The nested loops are created to ensure that each set only has the add() function used once.

The elements are added into the respective sets on some conditions to ensure that there is no duplicate data. For instance, line 66 ensures that only matching domain_id from the respective csv files will be added to the domain() set.

This will continue until all the relevant data pertaining to a certain table has been extracted and placed into the set.

```
75
76  # temporary storage for all tables that have fk linked to ur table
77  pfam_rows = {}
78  taxonomy_rows = {}
79  domain_rows = {}
80  protein_rows = {}
81
82  #remove everything from database and repopulate again
83  Pfam.objects.all().delete()
84  Taxonomy.objects.all().delete()
85  Protein.objects.all().delete()
86  Domain_protein_link.objects.all().delete()
87  Domain.objects.all().delete()
88
```

Next, empty arrays are created to store all the tables that have foreign keys linked to the tables. Pfam.objects.all().delete will remove all the data from the database and allow it to repopulate again when it is being run. This is the same for the subsequent lines of code corresponding to the respective tables.

For the code below, data is being added into the table Pfam. For each entry in the temporary set pfam, the data for each table is being stored and matched with the corresponding entry index in the set. This is done and sorted in order with the sequence of variables in each model which were designed earlier in model.py. This continues for all the tables.

```
89
90  # saving the relevant elements into the Pfam table in order
91  for pfam_entry in pfam:
92      row = Pfam.objects.create(domain_id = pfam_entry[0], domain_description = pfam_entry[1])
93      row.save()
94      pfam_rows[pfam_entry[0]] = row
95
96  # saving the relevant elements into the Taxonomy table in order
97  for taxonomy_entry in taxonomy:
98      row = Taxonomy.objects.create(taxa_id = taxonomy_entry[0],
99                                    clade = taxonomy_entry[1],
100                                   genus = taxonomy_entry[2],
101                                   species = taxonomy_entry[3])
102     row.save()
103     # saving data into taxonomy_rows to be called as a foreign key in another table later
104     taxonomy_rows[taxonomy_entry] = row
105
```

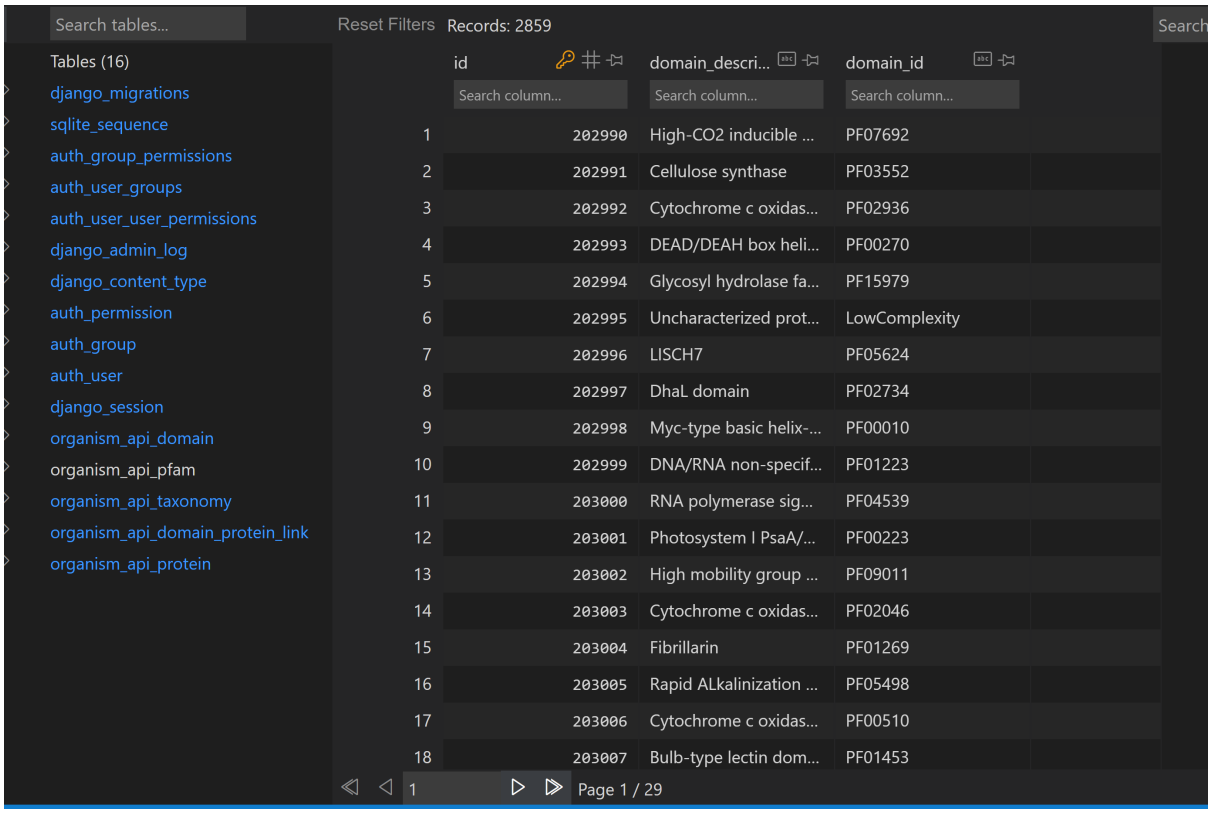
```
105
106 # saving the relevant elements into the Domain table in order
107 for domain_entry in domain:
108     row = Domain.objects.create(
109                                   pfam_id = pfam_rows[domain_entry[0]],
110                                   description = domain_entry[1],
111                                   start = domain_entry[2],
112                                   stop = domain_entry[3])
113     row.save()
114     domain_rows[domain_entry] = row
115
116 # saving the relevant elements into the Protein table in order
117 for protein_entry in protein:
118     row = Protein.objects.create(protein_id = protein_entry[0],
119                                   sequence = protein_entry[4],
120                                   taxonomy = taxonomy_rows[taxonomy_entry],
121                                   length = protein_entry[2])
122     # to populate the linker table
123     row.domains.add(domain_rows[domain_entry])
124     row.save()
125     protein_rows[protein_entry] = row
126
```

The migrations are then set in the command prompt using the following commands:

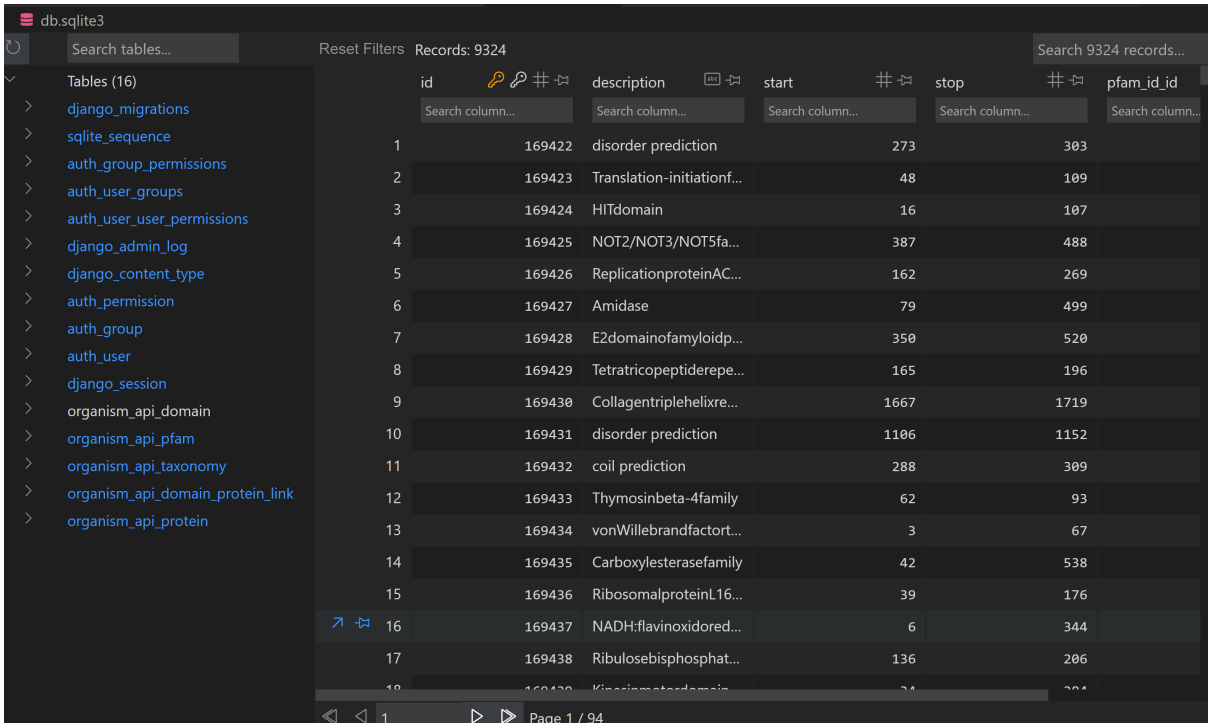
python manage.py makemigrations

python manage.py migrate

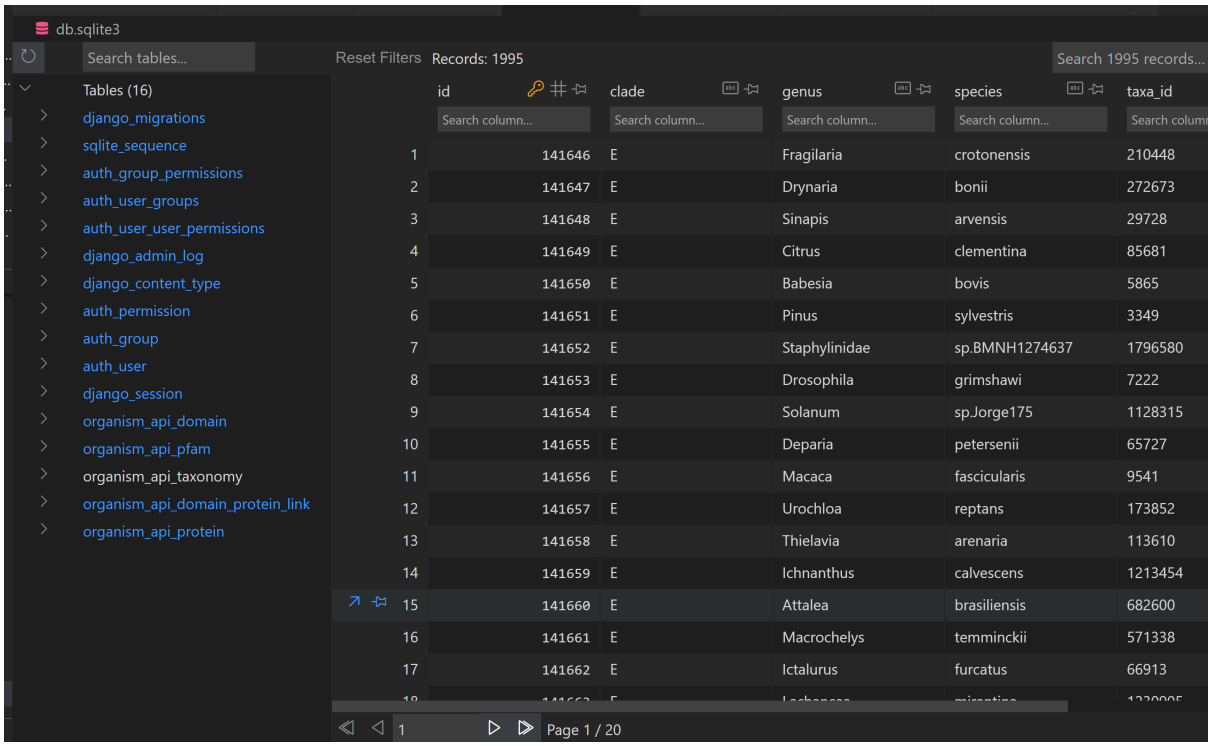
The data can then be displayed using sqlite3. The screenshot below shows how data is successfully stored into the pfam table.



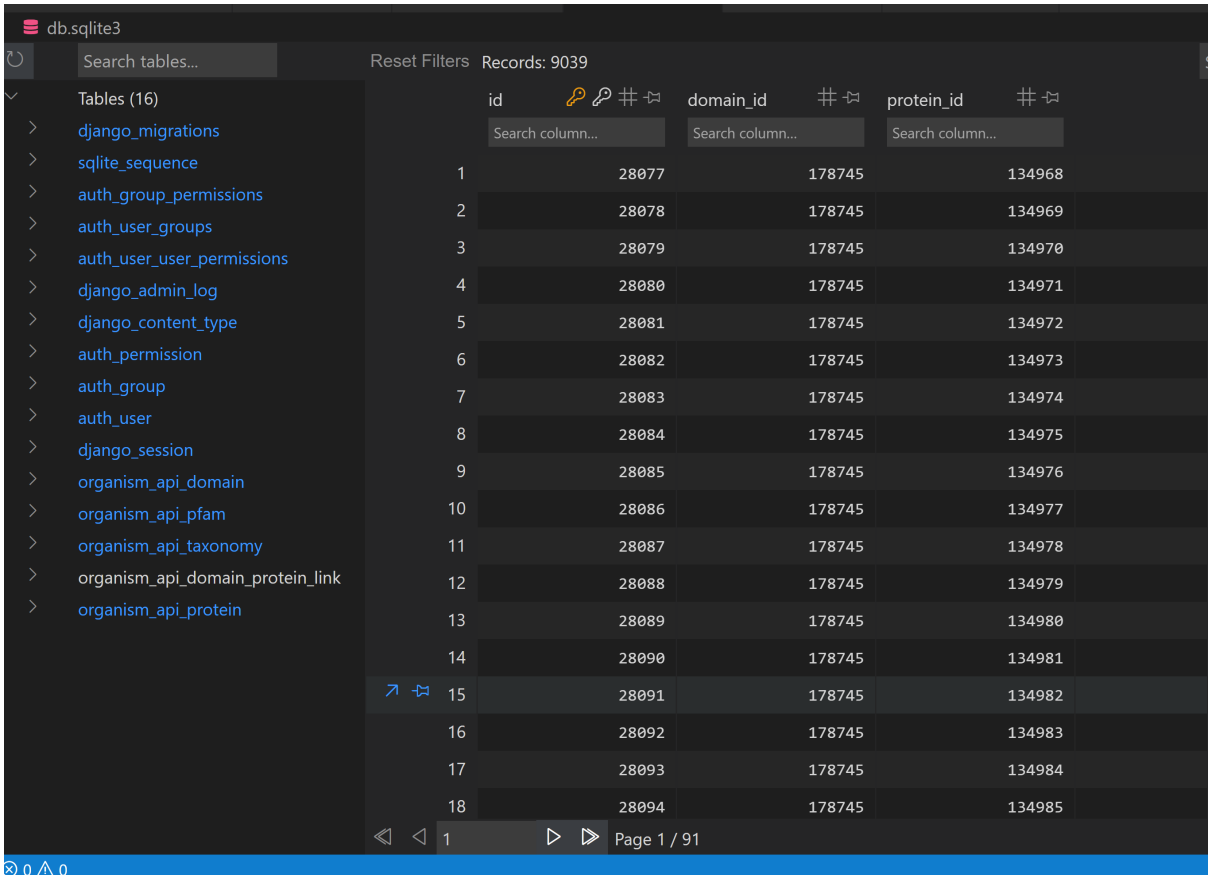
The screenshot below shows how data is successfully stored into the domain table.



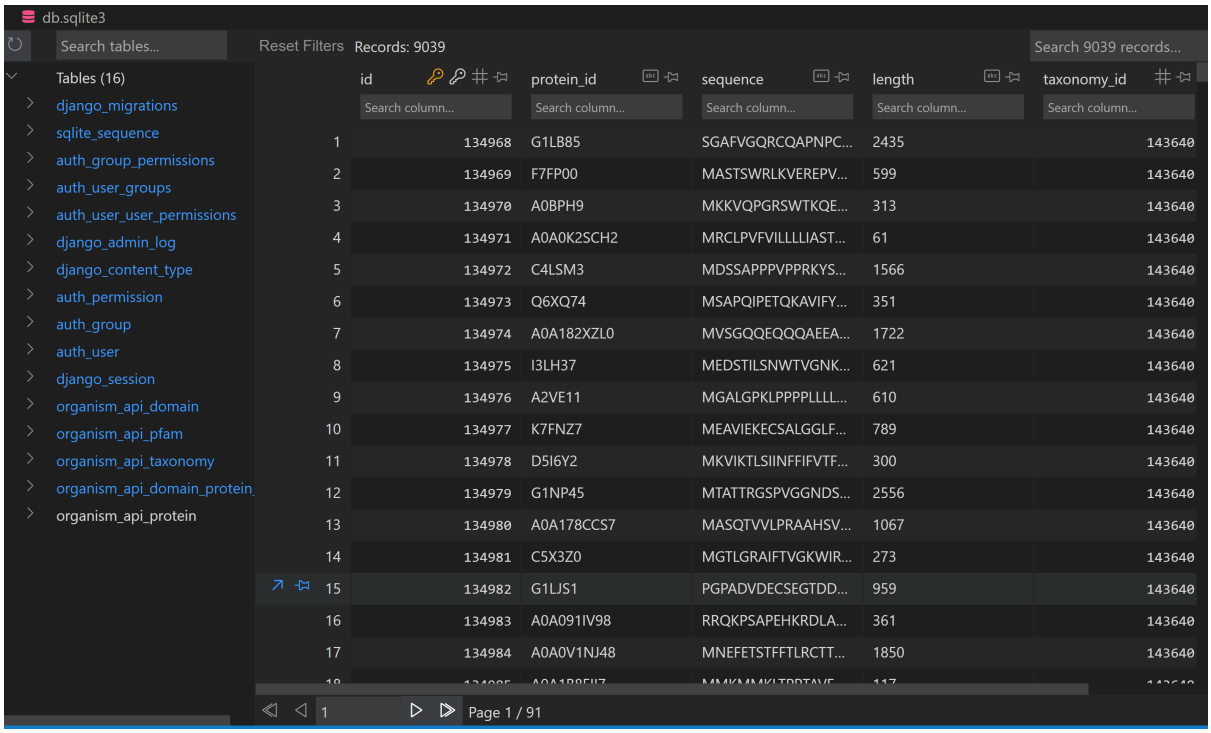
The screenshot below shows how data is successfully stored into the taxonomy table.



The screenshot below shows how data is successfully stored into the domain_protein_link table.



The screenshot below shows how data is successfully stored into the protein table.



This whole process is part of scripting and populating data from csv files into the tables.

b) correct use of form, validators and serialisation

In the setting.py folder, the apps are added and initialised so that they can be read later.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'organism_api',
    'rest_framework',
]
```

The serializer.py file is created to import the models. It stores data from the models in a format that can then be rendered to give the REST webpages. The screenshot below shows how data from the models are imported.

organism_api > serializers.py > ...

```
from rest_framework import serializers
from .models import Taxonomy
from .models import Pfam
from .models import Protein
from .models import Domain
```

For the screenshot below, it shows how the PfamSerializer() and TaxonomySerializer() is created by using the function ModelSerializer, which can be called from the REST framework.

For example, for the PfamSerializer, it uses the model Pfam to call the fields in line 14, which will be returned and mapped to a specific API either directly or indirectly, which will be explained later in the views.py and url.py files. The process is the same for the TaxonomySerializer().

```
7   # creating the pfam serializer
8   class PfamSerializer(serializers.ModelSerializer):
9
10      class Meta:
11          # serializer uses the model Pfam to call the fields
12          model = Pfam
13          # serializer will return the respective fields in the api
14          fields = ['domain_id', 'domain_description']
15
16  # creating the taxonomy serializer
17  class TaxonomySerializer(serializers.ModelSerializer):
18
19      class Meta:
20          # serializer uses the model Taxonomy to call the fields
21          model = Taxonomy
22          # serializer will return the respective fields in the api
23          fields = ['taxa_id', 'clade', 'genus', 'species']
24
```

In the screenshot below, it shows how the DomainSerializer() first calls the PfamSerializer() to retrieve the pfam_id. This allows it to be called in fields, after importing the rest of the field data from the Domain table. The whole DomainSerializer() can then be used to import the relevant data in the fields that are specified in order to render the corresponding API page given by the question paper.

The process is the same for the ProteinSerializer(), which calls the taxonomy and domains fields from the TaxonomySerialzer() and DomainSerializer() respectively, in addition to the other fields that can be imported directly from the Protein table.

```
25  # creating the Domain serializer
26  class DomainSerializer(serializers.ModelSerializer):
27      # calling the PfamSerializer to get the respective data for pfam_id
28      pfam_id = PfamSerializer()
29
30      class Meta:
31          # serializer uses the model Domain to call the fields
32          model = Domain
33          # serializer will return the respective fields in the api
34          fields = ['pfam_id', 'description', 'start', 'stop']
35
36  # creating the Protein serializer
37  class ProteinSerializer(serializers.ModelSerializer):
38      # calling the PfamSerializer and DomainSerializer to get the respective data for taxonomy and domains
39      taxonomy = TaxonomySerializer()
40      domains = DomainSerializer()
41
42      class Meta:
43          # serializer uses the model Protein to call the fields
44          model = Protein
45          # serializer will return the respective fields in the api
46          fields = ['protein_id', 'sequence', 'taxonomy', 'length', 'domains']
47
```

The ProteinSerializer is created by using the model Protein to call and store the data into the relevant fields which are the primary key “id” and “protein_id”.

The PfamsSerializer calls the pfam_id from the PfamSerializer() such that it can be called in the fields, in addition to the primary key ‘id’ which is from the Domain table.

```
48  # creating the Proteins serializer
49  class ProteinsSerializer(serializers.ModelSerializer):
50
51      class Meta:
52          # serializer uses the model Protein to call the fields
53          model = Protein
54          # serializer will return the respective fields in the api
55          fields = ['id','protein_id']
56
57  # creating the Pfams serializer
58  class PfamsSerializer(serializers.ModelSerializer):
59      # calling the PfamSerializer to get the respective data for pfam_id
60      pfam_id = PfamSerializer(many=True)
61
62      class Meta:
63          # serializer uses the model Domain to call the fields
64          model = Domain
65          # serializer will return the respective fields in the api
66          fields = ['id', 'pfam_id']
67
```

The following shows code for the CoverageSerializer(). I did my best and attempted to print the coverage but was unsuccessful, hence the code is commented out.

```
74
75  # class CoverageSerializer(serializers.ModelSerializer):
76
77  #     taxonomy = TaxonomySerializer()
78  #     domains = DomainSerializer()
79
80  #     for domains in Protein:
81  #         coverage =+ (Domain.start-Domain.stop)/Protein.length
82
83  #     class Meta:
84  #         model = Protein
85  #         fields = ['coverage']
86
```

In the next series of pages, I will show how I used the views.py file to render each API. It serves to map the data that was organized in the serializers.py file and map it to the url.py file in order to give a desired API when a url is called.

organism_api > views.py > ...

```python
from django.shortcuts import render
from django.http import JsonResponse
from rest_framework.decorators import api_view
from rest_framework.response import Response

from .serializers import PfamSerializer
from .serializers import ProteinSerializer
from .serializers import ProteinsSerializer
from .serializers import PfamsSerializer
# from .serializers import CoverageSerializer

from .models import Pfam
from .models import Taxonomy
from .models import Protein
from .models import Domain
# Create your views here.

```

The code below shows how an api overview page Is created and will be rendered, to provie easy instructions for the user on what to add to the url.

```python

@api_view(['GET'])
def apiOverview(request):
    api_urls = {
        'Protein': 'protein/',
        'Pfam': 'pfam/',
        'Proteins': 'proteins/',
        'Pfams': 'pfams/',
        'Coverage': 'coverage/'
        }
    return Response(api_urls)

```

The screenshot below shows the code to the function Protein_add. Since it is in POST format, it will allow users to post, and add a new record. Line 33 allows new records to be added to ProteinSerializer(), in the given format specified by it. Line 34 and 35 saves the record if it is valid, hence returing the newly serialized data as the response. The rendered API works.

```
28
29  # creates the api to allow post, which is to add a new record
30  @api_view(['POST'])
31  def Protein_add(request):
32      # allows new records to be added to ProteinSerializer
33      serializer = ProteinSerializer(data=request.data)
34      if serializer.is_valid():
35          serializer.save()
36      return Response(serializer.data)
37
```

The screenshot below shows how the Protein_views function is created. It is in GET format, hence it will retrieve data for the user. In this case, it takes in protein_id as the user input and matches it to the records in the Protein table which matches it by using the get() function. The ProteinSerializer() will then use the format saved to render the API in a certain sequence which is returned as the response. The rendered API works.

```
37
38  # creates the api to return the given protein seq
39  @api_view(['GET'])
40  def Protein_views(request, protein_id):
41      # get protein_id as user input in the url which corresponds to Protein table
42      protein = Protein.objects.get(protein_id=protein_id)
43      # check ProteinSerializer with the protein_id input
44      serializer = ProteinSerializer(protein, many=True)
45      return Response(serializer.data)
46  # error cos get returned more than 1 protein
47
```

The screenshot below shows how the Pfam_view and Pfam_views functions are created. The first is not required, but I did it for my own visualization, in order to see that the serializer is working correctly to retrieve all the data required in the format that matches the specification. You can use the url to check the whole list of domain and description available.

The second function is also a get function, to return a specific domain and description when the domain_id is added into the url as user input. In line 59, the get() function takes in the domain_id and matched it to the Pfam objects table, before rendering it using PfamSerializer() in a given format, which was specified before in serializers.py. The rendered API works.

```
48  # not necessary, to show all objects in Pfam table that corresponds to PfamSerializer
49  @api_view(['GET'])
50  def Pfam_view(request):
51      pfam = Pfam.objects.all()
52      serializer = PfamSerializer(pfam, many=True)
53      return Response(serializer.data)
54
55  # creates the api to return the domain and desc
56  @api_view(['GET'])
57  def Pfam_views(request,domain_id):
58      # get domain_id as user input in the url which corresponds to Pfam table
59      pfam = Pfam.objects.get(domain_id=domain_id)
60      # check PfamSerializer with the domain_id input
61      serializer = PfamSerializer(pfam, many=False)
62      return Response(serializer.data)
63  #works
```

The screenshot below shows how the Proteins_view and Proteins_views functions are created. The first is not required, but I did it for my own visualization, in order to see that the serializer is working correctly to retrieve all the data required in the format that matches the specification. You can use the url to check the whole list of protein objects available for all organisms.

The second function is also a get function, to return a all proteins for a given organism when the taxi_id_id is added into the url as user input. In line 77, the get() function takes in the taxa_id and matched it to the Taxonomy objects table, and saved it as the variable taxa_ids. This was then called in Protein table as the taxonomy field, and saved to protein variable. It was then rendered using ProteinsSerializer() in a given format, which was specified before in serializers.py. Unfortunately the code does not work and the API is not shown as I was unable to successfully call the taxa_id. However the first function works and successfully calls an API, hence I know the code is partially working.

```
64
65  # not necessayr, shows all protein objects for organisms
66  @api_view(['GET'])
67  def Proteins_view(request):
68      protein = Protein.objects.all()
69      serializer = ProteinsSerializer(protein, many=True)
70      return Response(serializer.data)
71  # works
72
73  # creates the api to return all proteins for given organism
74  @api_view(['GET'])
75  def Proteins_view(request, taxa_id):
76      # get taxa_id as user input in the url which corresponds to Protein table
77      taxa_ids  = Taxonomy.objects.get(taxa_id= taxa_id)
78      protein = Protein.objects.filter(taxonomy=taxa_ids)
79      # check ProteinsSerializer with the taxa_id input
80      serializer = ProteinsSerializer(protein, many=True)
81      return Response(serializer.data)
82
83
```

The screenshot below shows how the Pfams_view and Pfams_views functions are created. The first is not required, but I did it for my own visualization, in order to see that the serializer is working correctly to retrieve all the data required in the format that matches the specification. You can use the url to check the whole list of domain objects available for all organisms.

The second function is also a get function, to return a all domain for a given organism when the taxi_id_id is added into the url as user input. In line 96, the get() function takes in the taxa_id and matched it to the Domain objects table. It was then rendered using PfamsSerializer() in a given format, which was specified before in serializers.py. Unfortunately the code does not work and the API is not shown as I was unable to successfully call the taxa_id. However the first function works and successfully calls an API, hence I know the code is partially working.

```
83
84  # not necessary, shows all domain objects for organisms
85  @api_view(['GET'])
86  def Pfams_view(request):
87      domain = Domain.objects.all()
88      serializer = PfamsSerializer(domain, many=True)
89      return Response(serializer.data)
90  # object not iterable
91
92  # creates the api to return all domains for given organism
93  @api_view(['GET'])
94  def Pfams_view(request, taxa_id):
95      # get taxa_id as user input in the url which corresponds to Domain table
96      domain = Domain.objects.get(taxa_id=taxa_id)
97      # check PfamsSerializer with the taxa_id input
98      serializer = PfamsSerializer(domain, many=False)
99      return Response(serializer.data)
100
101
```

The screenshot below shows my attempt at creating the coverage api, which due to time constraints, was not completed.

```
103
104   # @api_view(['GET'])
105   # def Coverage_view(request):
106   #     protein = Protein.objects.all()
107   #     serializer = CoverageSerializer(protein, many=True)
108   #     return Response(serializer.data)
109
110
111   # @api_view(['GET'])
112   # def Coverage_view(request, protein_id):
113   #     protein = Protein.objects.get(protein_id=protein_id)
114   #     serializer = CoverageSerializer(protein, many=False)
115   #     return Response(serializer.data)
116
117
```

d) correct use of URL routing

The url.py page shows how the rest webpages are created by using urlpatterns. In the screenshot below, it shows how the api page is being initiallised.

```
16  from django.contrib import admin
17  from django.urls import path, include, re_path
18
19  urlpatterns = [
20      path('admin/', admin.site.urls),
21      path('api/', include('organism_api.urls')),
22
23  #    re_path(r'^', include('api.urls')),
24
25
26  ]
27
```

urls.py under the organism_api folder will then create the subsequent web pages. The screenshot below shows how the subsequent path maps to the views folder which will be explained later. <str:pk> shows how the path takes in a string value as input and a primary key in order to pull out the relevant information related to it and render it respectively, Each line corresponds to one url page, which is specified amd corresponds to the functions in the views.py folder which was explained above.

organism_api > urls.py > ...

```
1   from django.urls import path
2   from . import views
3
4   urlpatterns = [
5       path('', views.apiOverview, name = "api-overview"),
6
7       path('protein/', views.Protein_add, name="Protein"),
8       path('protein/<str:protein_id>/', views.Protein_views, name="Protein"),
9
10      path('pfam/', views.Pfam_view, name="Pfam"),
11      path('pfam/<str:domain_id>/', views.Pfam_views, name="Pfam"),
12
13      path('proteins/', views.Proteins_view, name="Proteins"),
14      path('proteins/<str:taxa_id>/', views.Proteins_views, name="Proteins"),
15
16      path('pfams/', views.Pfams_view, name="Pfams"),
17      path('pfams/<str:taxa_id>/', views.Pfams_views, name="Pfams"),
18
19      # path('coverage/', views.Coverage_view, name="Coverage"),
20      # path('coverage/<str:protein_id>/', views.Coverage_view, name="Coverage"),
21
22  ]
```

The views.py file will create either a get or post page that will give results when the primary key is added into the link, or allow users to add a new record. It imports from the serializers to render the page in a sequence that will match the REST requirements.

```python
from django.shortcuts import render
from django.http import JsonResponse
from rest_framework.decorators import api_view
from rest_framework.response import Response

from .serializers import PfamSerializer
# from .serializers import TaxonomySerializer
from .serializers import ProteinSerializer

from .models import Pfam
from .models import Taxonomy
from .models import Protein
# Create your views here.
```

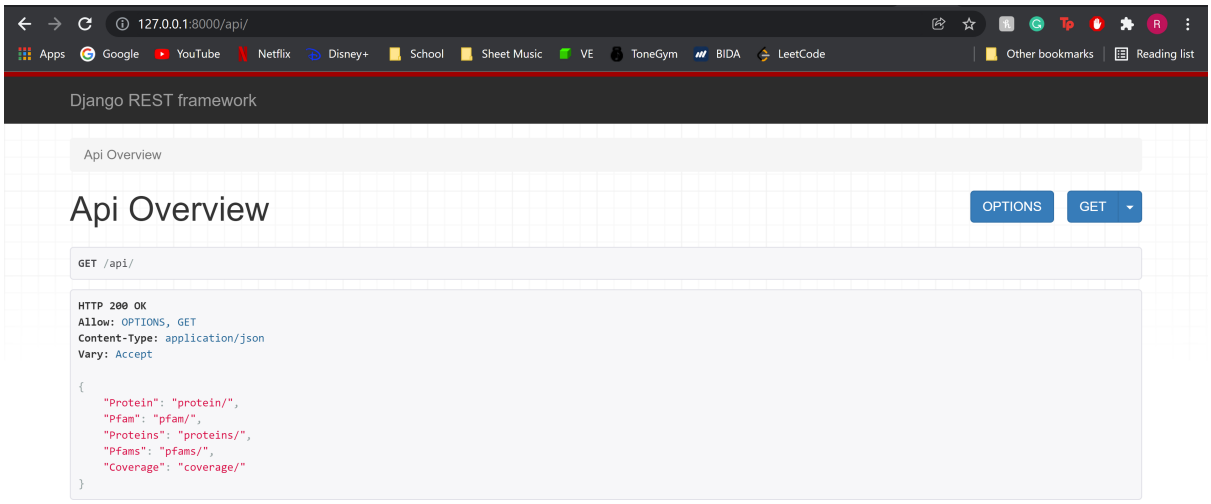
This screenshot below shows the api overview page, which explains which string should be added to the url in order to render the page for each get or post query.

```python
@api_view(['GET'])
def apiOverview(request):
    api_urls = {
        'Protein': 'protein/',
        'Pfam': 'pfam/',
        'Proteins': 'proteins/',
        'Pfams': 'pfams/',
        'Coverage': 'coverage/'
        }
    return Response(api_urls)
```

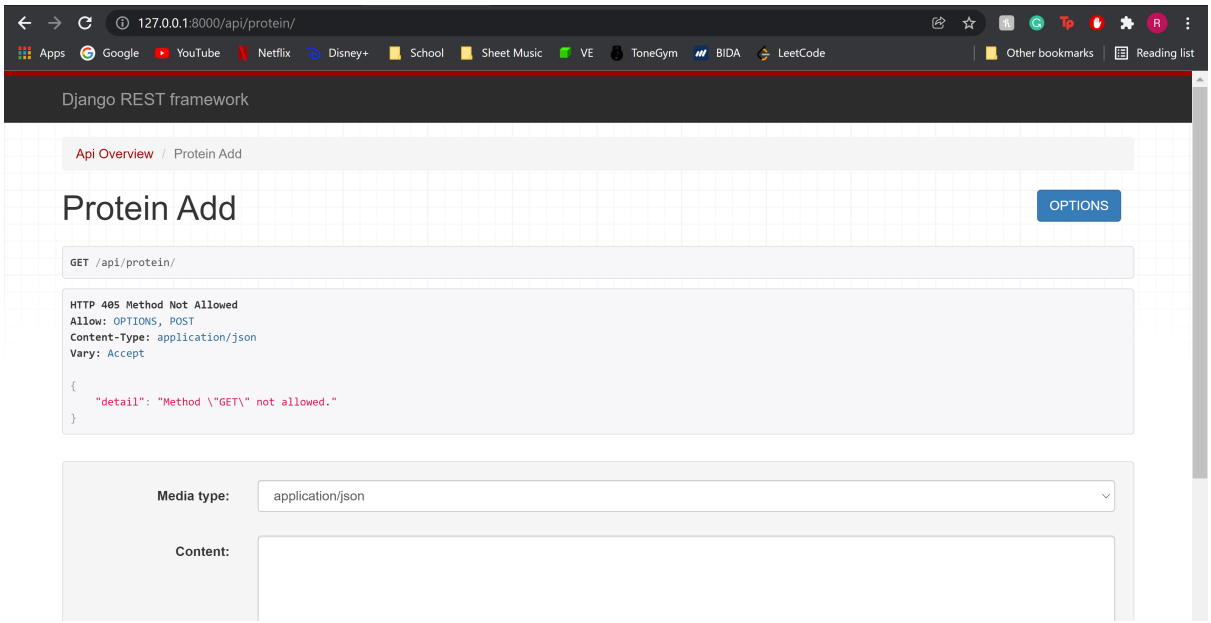
To view all the APIs, I used the command python manage.py runserver to start the server. I was able to successfully render 2 api pages, but the others were completed partially.

The following shows the rendered APIs.

The screenshot below shows the API overview homepage that I have created.

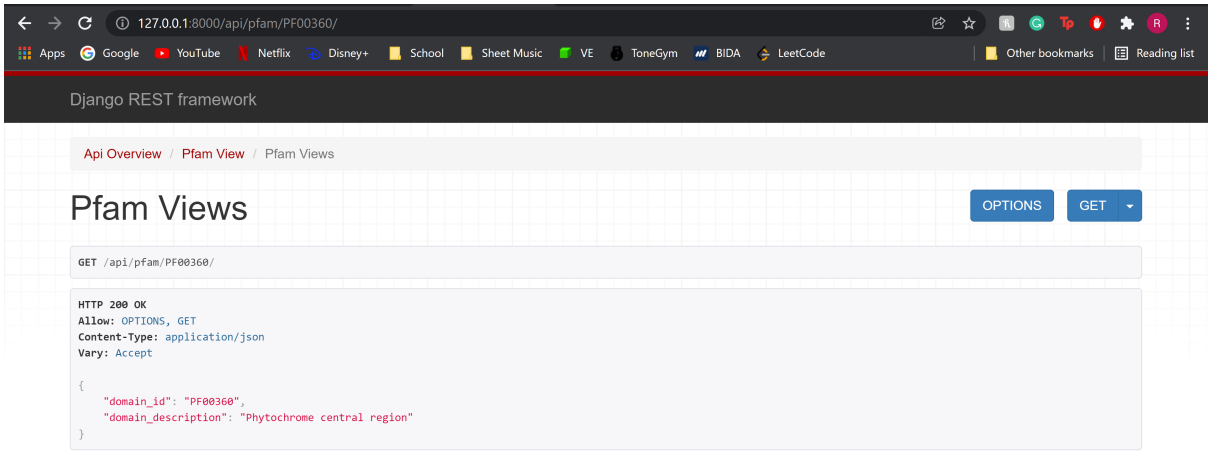


The screenshot below shows the API for protein.

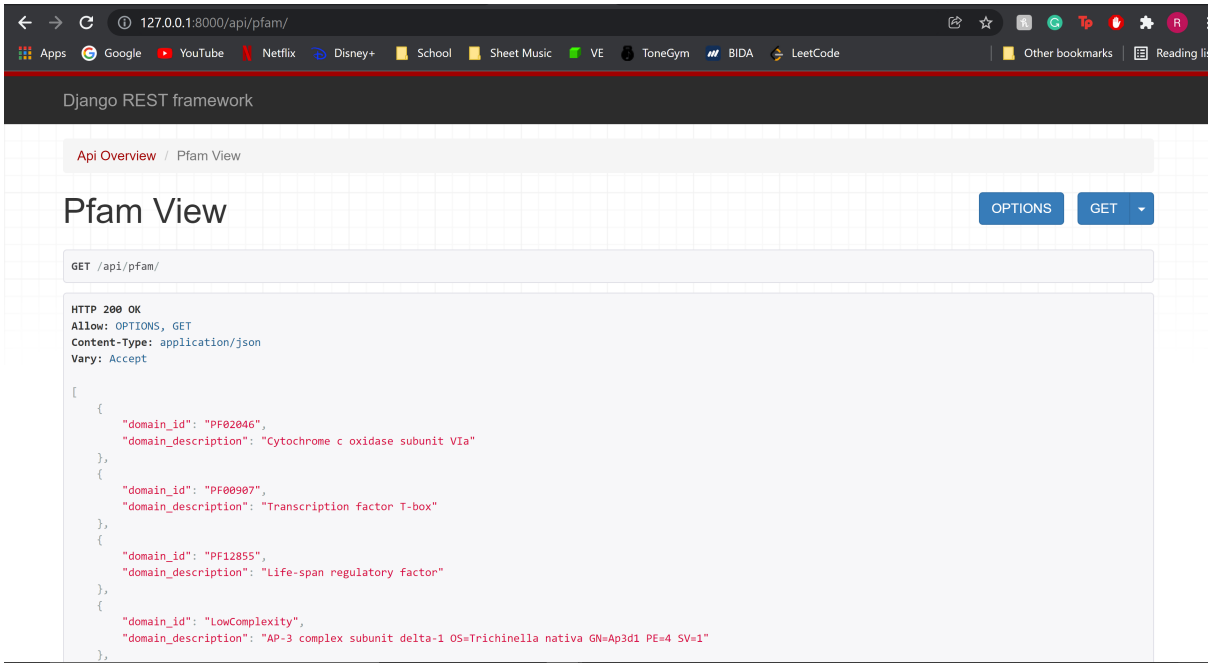


The API for http://127.0.0.1:8000/api/protein/A0A016S8J7/ does not work.

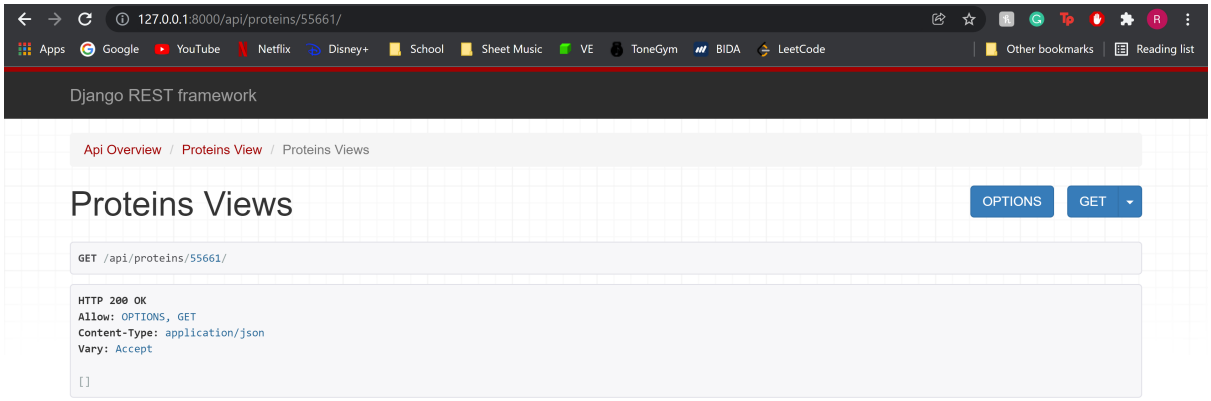
The API for http://127.0.0.1:8000/api/pfam/PF00360/ works.



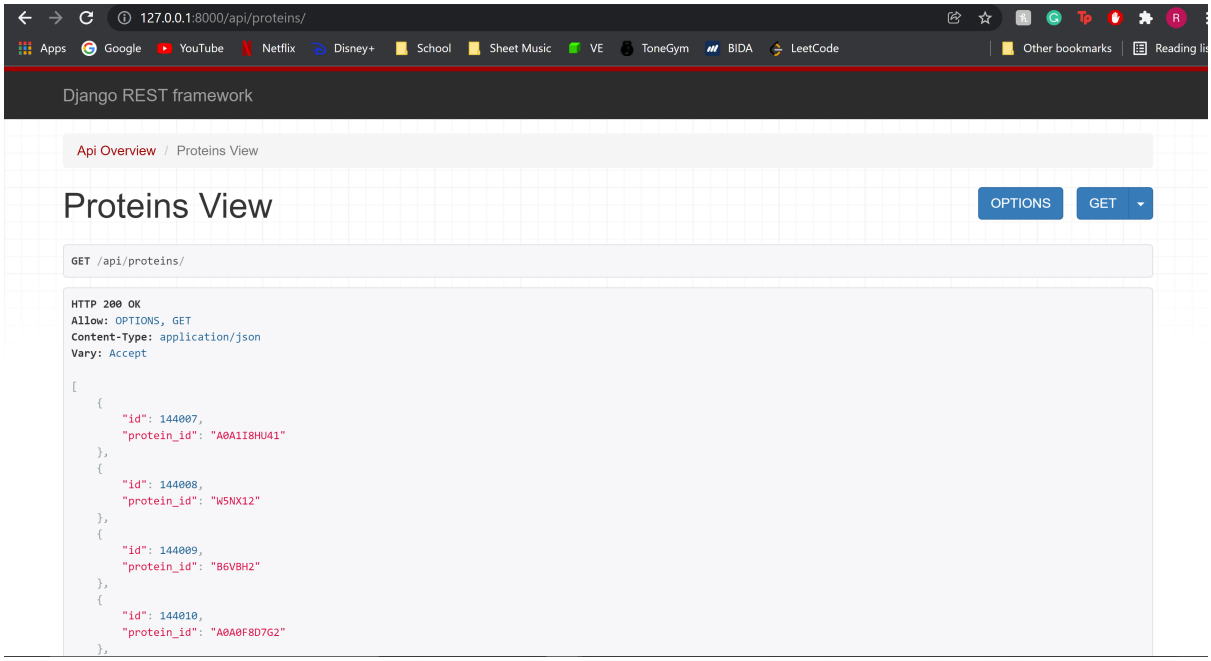
I created an additional overview page for all pfam objects as seen in the screenshot below



The API for http://127.0.0.1:8000/api/proteins/55661/ does not work as there is missing data due to some errors in population.



In hopes to get some method marks, I created a proteins overview api that calls all existing proteins in the database to show that the query works.



Like the previous API, the API for http://127.0.0.1:8000/api/pfams/55661/ does not work as there is missing data due to some errors in population.

Unfortunately, the API for http://127.0.0.1:8000/api/coverage/ does not work either.

e) appropriate use of unit testing

Next, I download and used pip install factory_boy in order to carry out with unit tests. I then created a new file called model_factories.py in order to store dummy data. The screenshots below shows how each function takes in dummy data by using its respective table and table elements.

organism_api > model_factories.py > ...

```python
import factory
from django.test import TestCase
from django.conf import settings
from django.core.files import File

from .models import *

# creating dummy data for Taxonomy_Factory by using Taxonomy
class Taxonomy_Factory(factory.django.DjangoModelFactory):
    taxa_id = 53326
    clade = "E"
    genus = "Ancylostoma"
    species = "ceylanicum"

    class Meta:
        model = Taxonomy

# creating dummy data for Pfam_Factory by using Pfam
class Pfam_Factory(factory.django.DjangoModelFactory):
    domain_id = "PF01650"
    domain_description = "PeptidaseC13family"

    class Meta:
        model = Pfam

# creating dummy data for Domain_Factory by using Domain
class Domain_Factory(factory.django.DjangoModelFactory):
    pfam_id = factory.SubFactory(Pfam_Factory)
    description = "Peptidase C13 legumain"
    start = 40
    stop = 94

    class Meta:
        model = Domain

# creating dummy data for Protein_Factory by using Protein
class Protein_Factory(factory.django.DjangoModelFactory):
    protein_id = "A0A016S8J7"
    sequence = "MVIGVGFLLVLFSSSVLGILNAGVQLRIEELFDTPGHTNNWAVLVCTSRFWFNYRHVSNVLALYHTVKRLGIPDSNIILMLAEDVPCNPRNPRPEAAVLSA"
    taxonomy = factory.SubFactory(Taxonomy_Factory)
    length = 101
    domains = factory.SubFactory(Domain_Factory)

    class Meta:
        model = Protein

# creating dummy data for Domain_protein_link_Factory by using Domain_protein_link
class Domain_protein_link_Factory(factory.django.DjangoModelFactory):
    domain = factory.SubFactory(Domain_Factory)
    protein = factory.SubFactory(Protein_Factory)

    class Meta:
        model = Domain_protein_link
```

Next, in my tests.py folder, I tested my APIs. The function Pfam_Test shows how I tested the api by using “PF01650” as the domain_id and test data, by using the Pfam_Factory that was specified in model_factories.py file.

organism_api > tests.py > Pfam_Test > test_PfamReturnSuccess

```python
from django.test import TestCase

# Create your tests here.

import json
from django.test import TestCase
from django.urls import reverse
from django.urls import reverse_lazy

from rest_framework.test import APIRequestFactory
from rest_framework.test import APITestCase

from .model_factories import *
from .serializers import *

class Pfam_Test(APITestCase):

    def test_PfamReturnSuccess(self):
        pfam_factory = Pfam_Factory.create()
        url = reverse('Pfam', kwargs={'domain_id': 'PF01650'})
        response = self.client.get(url)
        response.render()
        self.assertEqual(response.status_code, 200)

    class Meta:
        model = Taxonomy


```

I then used the command in command prompt python manage.py test to test the API. It was successful as shown in the screenshot below.

```
(myenv) C:\Users\FOO RUI QI\Documents\SIM\Advanced Web Development\awd_midterm\organism_data>python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.013s

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