JUNE 5, 2022

ADVANCED WEB DEVELOPMENT

REPORT FOR COURSEWORK 1(REST API)

CHIA YI QUAN
Student Number: 210220223

Contents

Introduction

The purpose of this report is to document down on the bioscience project. The report will be separated into 2 sections which is running of the projects and code explanation. The first section will cover setting up of the project, running migration and script, running of the project, running of test case and code explanation of the required api endpoint. The second section will explain the code in the scripts, models design, serializer and the api code.

Running Of The Projects

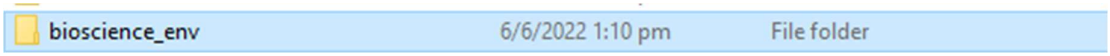
Setting Up Of The Project

Inside the folder that was submitted, there should be 2 items which is a requirements.txt and bioscience folder. Firstly, we have to setup the virtual environment before going to the migration. Open a terminal and navigate into the project folder(the folder where the requirements.txt and bioscience folder is in). You can create a new virtual environment with the following command.

py -m venv env_name -> Replace the env_name with the name you want.

```
C:\Users\yi quan\Desktop\year_3_sem_1\Web Development\cw>py -m venv bioscience_env
```

This will create a env folder in the directory.



In the terminal, for windows change the directory to the Scripts folder which in my case will be **cd bioscience_env/Scripts** and type in the following command

activate.bat

For unix system, change the directory to the bin folder which in my case will be **cd bioscience_env/bin** and type in the following command

source activate

If done successfully, the env name should appear at the front of the directory.

```
C:\Users\yi quan\Desktop\year_3_sem_1\Web Development\cw>cd bioscience_env/Scripts

C:\Users\yi quan\Desktop\year_3_sem_1\Web Development\cw\bioscience_env\Scripts>activate.bat

(bioscience_env) C:\Users\yi quan\Desktop\year_3_sem_1\Web Development\cw\bioscience_env\Scripts>
```

Now we have to change directory back to the project folder which is 2 directory above the current directory we are in, so in the terminal we can execute **cd ..** for 2 time. After changing the working directory to the project folder, we can run the following command to install the dependencies.

pip install -r ./requirements.txt

After running the command above you should see this screen

```
(bioscience_env) C:\Users\yi quan\Desktop\year_3_sem_1\Web Development\cw>pip install -r ./requirements.txt
Collecting asgiref==3.5.2
  Using cached asgiref-3.5.2-py3-none-any.whl (22 kB)
Collecting Django==4.0.4
  Using cached Django-4.0.4-py3-none-any.whl (8.0 MB)
Collecting djangorestframework==3.13.1
  Using cached djangorestframework-3.13.1-py3-none-any.whl (958 kB)
Collecting psycopg2==2.9.3
  Using cached psycopg2-2.9.3-cp39-cp39-win_amd64.whl (1.2 MB)
Collecting pytz==2022.1
  Using cached pytz-2022.1-py2.py3-none-any.whl (503 kB)
Collecting sqlparse==0.4.2
  Using cached sqlparse-0.4.2-py3-none-any.whl (42 kB)
Collecting tzdata==2022.1
  Using cached tzdata-2022.1-py2.py3-none-any.whl (339 kB)
Installing collected packages: pytz, tzdata, sqlparse, psycopg2, asgiref, Django, djangorestframework
Successfully installed Django-4.0.4 asgiref-3.5.2 djangorestframework-3.13.1 psycopg2-2.9.3 pytz-2022.1 sqlparse-0.4.2 t
zdata-2022.1
WARNING: You are using pip version 22.0.4; however, version 22.1.2 is available.
You should consider upgrading via the 'C:\Users\yi quan\Desktop\year_3_sem_1\Web Development\cw\bioscience_env\Scripts\p
ython.exe -m pip install --upgrade pip' command.
```

After installing the dependencies, we can continue to the next step but do not close the terminal yet.

Migration Of The Project

In the terminal(with the env activated), change directory into bioscience folder. After changing the directory to the folder, we can run the following command in the terminal to do a migration. For the python command it depend on the system global variable which can be python3 py etc..

python manage.py migrate

All the models will be migrated into the default database which is the sqlite.

```
(bioscience_env) C:\Users\yi quan\Desktop\year_3_sem_1\Web Development\cw>cd bioscience

(bioscience_env) C:\Users\yi quan\Desktop\year_3_sem_1\Web Development\cw\bioscience>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, organism, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying organism.0001_initial... OK
  Applying sessions.0001_initial... OK

(bioscience_env) C:\Users\yi quan\Desktop\year_3_sem_1\Web Development\cw\bioscience>
```

Next, we will run the script to import all the csv data into the database, I will explain all the scripts codes at the later part of the report but for now we will run this command to execute the scripts.

python ./scripts/populate_bioscience.py

It will take around 5 minutes if is on SSD and around 10 to 15 minutes for HDD for the scripts to execute finish. However, changing the database to postgres will improve the

migration speed. If all data have been inserted in successfully, the screen will appear as the following screenshot.

Running Of The Project

To start the project, simply just run the command below.

python manage.py runserver

Once the server started, the list of api is as listed:

[POST] http://localhost:8000/api/protein/

The structure to submit a new protein must be in this format

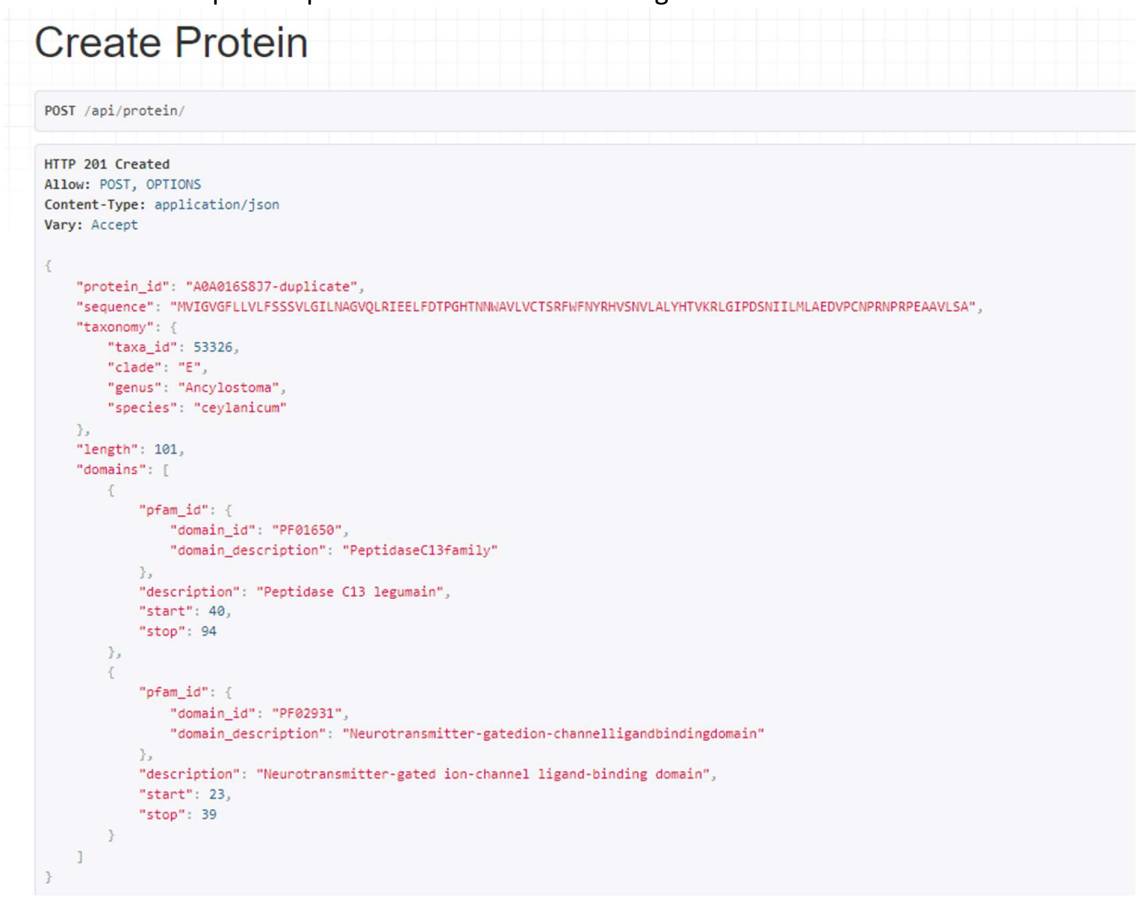

```
{
   "protein_id": "",
   "sequence": "",
   "taxonomy": {
      "taxa_id": null,
      "clade": "",
      "genus": "",
      "species": ""
   },
   "length": null,
   "domains": []
}
```


Example:


```
{
   "protein_id": "A0A016S8J7-duplicate",
   "sequence":
"MVIGVGFLLVLFSSSVLGILNAGVQLRIEELFDTPGHTNNWAVLVCTSRFWFNYRHVSNVLALYHTVKRL
GIPDSNIILMLAEDVPCNPRNPRPEAAVLSA",
   "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"
```

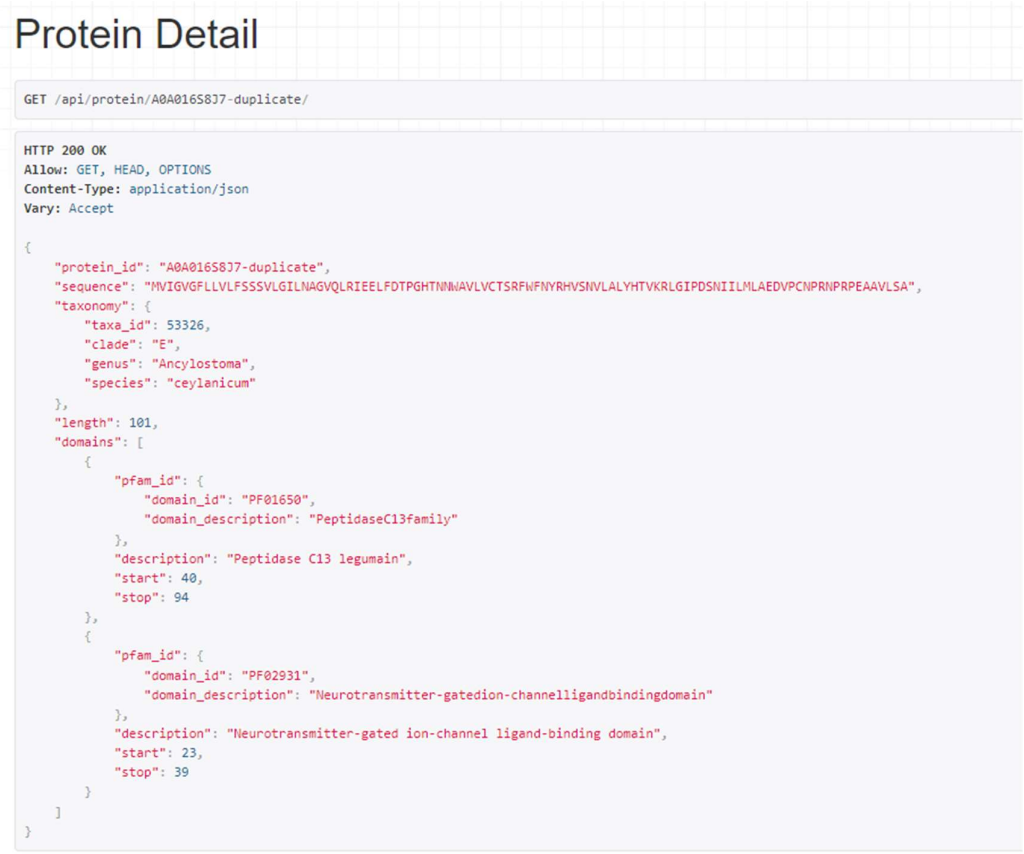
```
        },
        "description": "Neurotransmitter-gated ion-channel ligand-binding domain",
        "start": 23,
        "stop": 39
      }
    ]
}
```


The result of the post request will be like the following screenshot.



[GET] http://localhost:8000/api/protein/[PROTEIN_ID]
Example:
http://localhost:8000/api/protein/A0A016S8J7-duplicate/

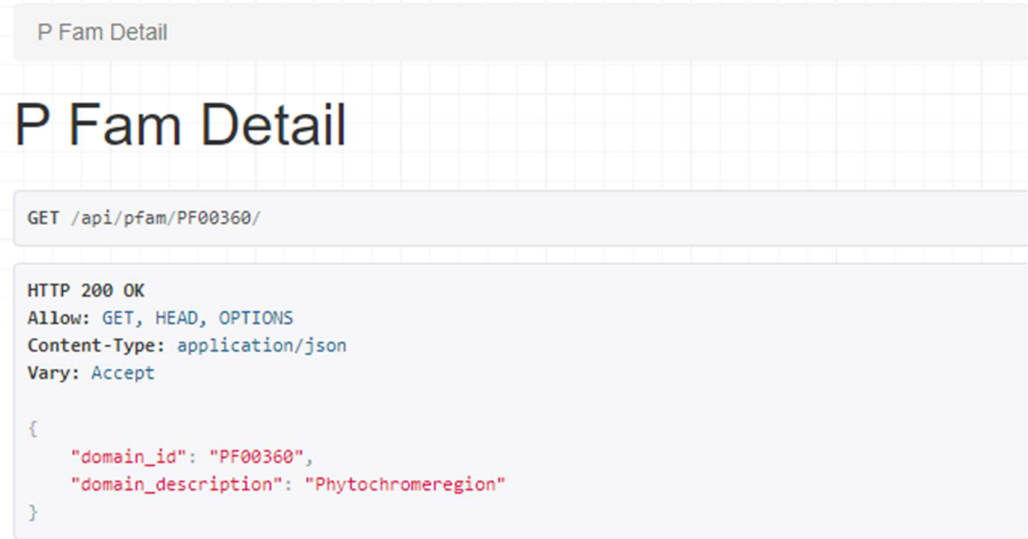
The result of the get request will be like the following screenshot.



[GET] http://localhost:8000/api/pfam/[PFAM_ID]

Example:
http://localhost:8000/api/pfam/PF00360/

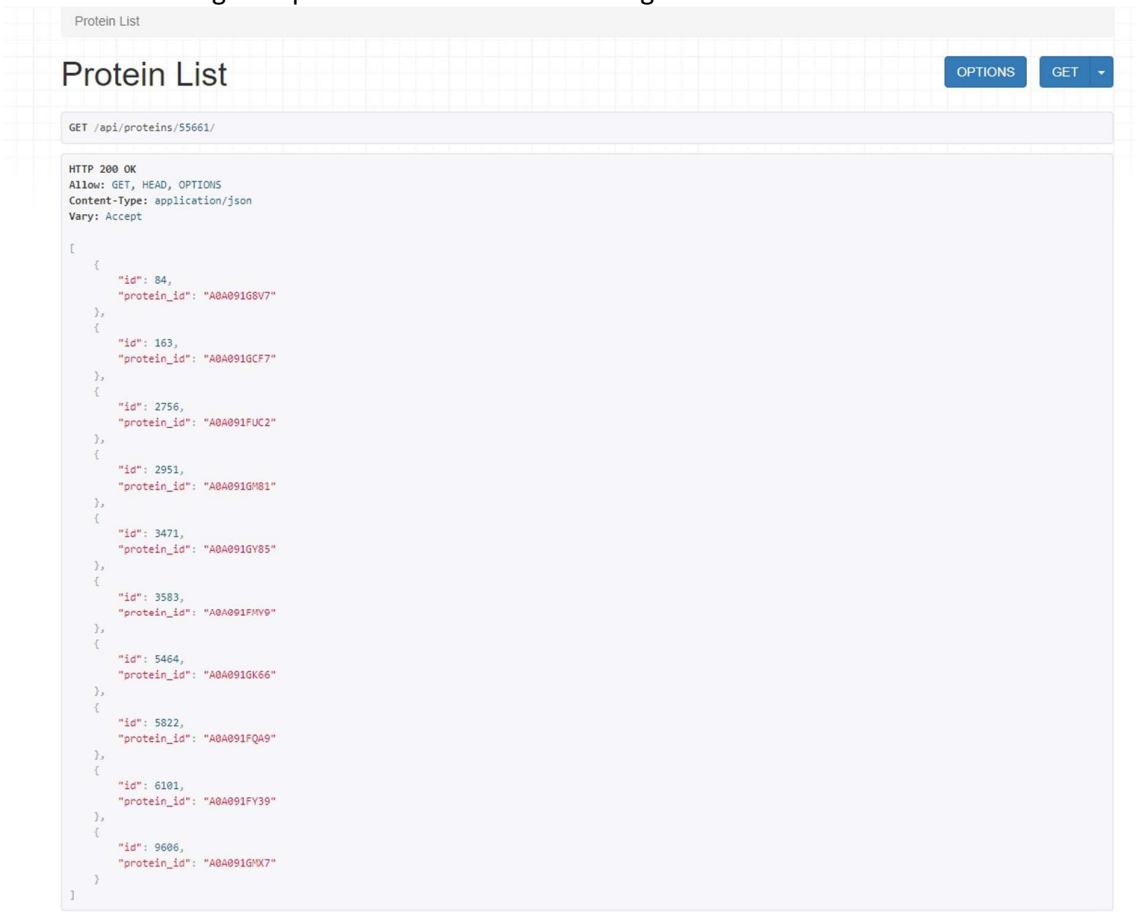
The result of the get request will be like the following screenshot.



[GET] http://localhost:8000/api/proteins/[TAXA_ID]

Example:
http://localhost:8000/api/proteins/55661/

The result of the get request will be like the following screenshot.



[GET] http://localhost:8000/api/pfams/[TAXA_ID]

Example:
http://127.0.0.1:8000/api/pfams/55661/

The result of the get request will be like the following screenshot.

Pfams List

Pfams List

OPTIONS GET

```
GET /api/pfams/55661/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
```


```
[
    {
        "id": 697,
        "pfam_id": {
            "domain_id": "mobidb-lite",
            "domain_description": "disorder prediction"
        }
    },
    {
        "id": 698,
        "pfam_id": {
            "domain_id": "mobidb-lite",
            "domain_description": "disorder prediction"
        }
    },
    {
        "id": 694,
        "pfam_id": {
            "domain_id": "PF16172",
            "domain_description": "DOCKN-terminus"
        }
    },
    {
        "id": 700,
        "pfam_id": {
            "domain_id": "PF00041",
            "domain_description": "FibronectintypeIIIdomain"
        }
    },
    {
        "id": 702,
        "pfam_id": {
            "domain_id": "PF14260",
            "domain_description": "C4-typezinc-fingerofDNApolymerasedelta"
        }
    },
    {
        "id": 691,
        "pfam_id": {
            "domain_id": "PF00307",
            "domain_description": "Calponinhomology(CH)domain"
        }
    },
    {
        "id": 699,
        "pfam_id": {
            "domain_id": "mobidb-lite",
            "domain_description": "disorder prediction"
        }
    },
    {
        "id": 692,
        "pfam_id": {
            "domain_id": "PF00415",
            "domain_description": "Regulatorofchromosomecondensation(RCC1)repeat"
        }
    },
    {
        "id": 696,
        "pfam_id": {
            "domain_id": "mobidb-lite",
            "domain_description": "disorder prediction"
        }
    },
    {
        "id": 693,
        "pfam_id": {
            "domain_id": "PF07648",
            "domain_description": "Kazal-typeserineproteaseinhibitordomain"
        }
    },
    {
        "id": 695,
        "pfam_id": {
            "domain_id": "mobidb-lite",
            "domain_description": "disorder prediction"
        }
    },
    {
        "id": 701,
        "pfam_id": {
            "domain_id": "PF02141",
            "domain_description": "DENN(AEX-3)domain"
        }
    }
]
```

[GET] http://localhost:8000/api/coverage/[PROTEIN_ID]

Example:
http://localhost:8000/api/coverage/A0A016S8J7/

The result of the get request will be like the following screenshot.

Running The Test Case

To run the test case, simply run the command.

python manage.py test

There are a total of 31 test case which will be like the following screenshot

Code Explanation

Models

```
You, 6 days ago | 1 author (You)
class Taxonomy(models.Model):
    taxa_id = models.IntegerField(null=False, blank=False)
    clade = models.CharField(
        max_length=256, null=False, blank=False)
    genus = models.CharField(
        max_length=256, null=False, blank=False)
    species = models.CharField(
        max_length=256, null=False, blank=False)
```

The taxonomy model consisted of taxa_id(integer field), clade(char field with max length 256), genus(char field with max length 256) and species(char field with max length 256). The taxa_id in this case is a “unique” key as there should not be 2 entry with same taxa_id with different properties. However, with the create protein api the unique constraint is not added in here because there can be multiple protein having the same taxonomy and the data won’t be able to pass the validation check.

```
class Pfam(models.Model):
    domain_id = models.CharField(
        max_length=256, null=False, blank=False)
    domain_description = models.CharField(
        max_length=256, null=False, blank=False)
```

The Pfam model consisted of domain_id(char field with max length 256) and domain_description(char field with max length 256). The domain_id in this case is a “unique” key as there should not be 2 entry with same domain_id with different domain description. However, with the create protein api the unique constraint is not added in here because there can be multiple domain having the same Pfam and the data won’t be able to pass the validation check.

```
class Domain(models.Model):
    pfam_id = models.ForeignKey(
        Pfam, on_delete=models.DO_NOTHING)
    description = models.CharField(
        max_length=256, null=False, blank=False)
    start = models.IntegerField(null=False, blank=False)
    stop = models.IntegerField(null=False, blank=False)
```

The Domain model consisted of pfam_id(foreign key referencing to Pfam model), description(char field with max length 256), start(integer field), stop(integer field).

```
class Protein(models.Model):
    protein_id = models.CharField(
        max_length=256, null=False, blank=False, unique=True)
    sequence = models.CharField(
        max_length=40000, null=True, blank=True)
    taxonomy = models.ForeignKey(Taxonomy, on_delete=models.DO_NOTHING)
    length = models.IntegerField(null=False, blank=False)
    domains = models.ManyToManyField(
        Domain, through='ProteinDomainLink', through_fields=('protein', 'domain'))
```

The Protein model consisted of protein_id(char field which is unique and have a max length 256), sequence(char field with max length 40000 and is nullable), taxonomy(foreign key referencing to taxonomy model), length(integer field) and domains(many to many field with Domain and is linked through ProteinDomainLink model with protein and domain field). The reason why protein_id is unique because we only want to have 1 unique record from the dataset as the duplicated data in the csv have different taxanomy and domains. The reason for ManyToManyField on domains is because a protein can have multiple domains.

```
class ProteinDomainLink(models.Model):
    protein = models.ForeignKey(Protein, on_delete=models.DO_NOTHING)
    domain = models.ForeignKey(Domain, on_delete=models.DO_NOTHING)
```

The ProteinDomainLink model consisted of protein(foreign key referencing Protein model) and domain(foreign key referencing Domain model).

Data Population Script

```
data_sequences_file = './scripts/assignment_data_sequences.csv'
data_set_file = './scripts/assignment_data_set.csv'
pfam_descriptions_file = './scripts/pfam_descriptions.csv'
protein = set()
taxonomy = defaultdict(set)
domain = defaultdict(list)
protein_sequence = defaultdict()
pfam = defaultdict()
```

```
with open(data_sequences_file) as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    for row in csv_reader:
        protein_sequence[row[0]] = row[1]
```

This portion of the code is to iterate through the assignment_data_sequence.csv and store each row of sequence into the protein_sequence variable with protein_id as the key and sequence as the value

```
with open(data_set_file) as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    for row in csv_reader:
        organism_name = []

        # for entry Cyprinus carpio 'Furong' x Carassius auratus red var.
        if " x " in row[3]:
            organism_name = row[3].split(' x ')
        # for entry with sp. and cf.
        elif ". " in row[3]:
            organism_name = row[3].split('. ')
            # append back the . because of the split removes it
            organism_name[0] = organism_name[0]+"."
        # for the rest of the entry that split by spaces
        else:
            organism_name = row[3].split(' ')

        taxonomy[row[1]].add(
            ( row[2], organism_name[0], organism_name[1]))
        domain[row[0]].append((row[4], row[5], row[6], row[7]))
        protein.add((row[0], row[1], row[-1]))
```

This portion of code iterate the assignment_data_set file. For each row, the organism_name will store the Genus and Species by splitting them either by " x " or ". " or " ". The reason for different kind of split is because there are different data with different formatting, so it is inaccurate to split everything by empty spaces. Each row is then stored in 3 different variables which is taxnomy, domain and protein. The taxonomy variable will store the

taxa_id(second column in csv) as the key and add a set of unique set clade(third column in csv), genus and species. Reasoning for using a set is because there is a lot of duplicated data in the csv which can be normalized. The domain variable will store the domain_id as the key and a list of tuple of description(fifth column in csv), domain_id(sixth column in csv), start(seventh column in csv), stop(eighth column in csv). There is a lot of duplicated description and domain_id in the data set however I did not normalize it because normalizing it the api result will be different from the rest_specification file. Lastly the protein variable store a set of tuple of protein_id(first column in csv), taxa_id(second column in csv) and length(last column in csv). The reason of using a set so that duplicated data will be removed.

```
with open(pfam_descriptions_file) as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    for row in csv_reader:
        pfam[row[0]]=row[1]
```

This portion of code iterate the pfam_descriptions file. For each row, the pfam variable will store the domain_id(first column in csv) and the domain_description(second column in csv) as the value.

```
ProteinDomainLink.objects.all().delete()
Domain.objects.all().delete()
Protein.objects.all().delete()
Taxonomy.objects.all().delete()
Pfam.objects.all().delete()
```

This portion of code is to delete all the entry in the table everytime the script runs.

```
taxonomy_rows = defaultdict()
pfam_rows = {}
domain_rows = defaultdict(list)
protein_rows = {}
```

```
for domain_id in pfam.keys():
    row = Pfam.objects.create(
        domain_id=domain_id, domain_description=pfam[domain_id])
    row.save()
    pfam_rows[domain_id] = row
print('Pfam inserted')
```

This portion of code iterate each key in the pfam variable that was created earlier on, once we have the domain_id, it will create the Pfam object and save it to the database. After the insertion is successful, the pfam_rows variable will store the domain_id as the key and the row inserted into the database as the value. The pfam_rows data will be used as foreign key in domain insertion.

```
for protein_id in domain.keys():
    for entry in domain[protein_id]:
        row = Domain.objects.create(
            description=entry[0], start=entry[2], stop=entry[3], pfam_id=pfam_rows[entry[1]])
        row.save()
        domain_rows[protein_id].append(row)
print('Domain inserted')
```

This portion of the code is to iterate each key in the domain variable and iterate each tuple in the list. Each domain will get the id of the pfam by using the domain_id and access the entry in pfam_rows variable. After the data is stored in the database, the row will be stored in domain_rows variable which will be used in protein insertion.

```
for taxa_id in taxonomy.keys():
    for entry in taxonomy[taxa_id]:
        row = Taxonomy.objects.create(
            taxa_id=taxa_id, clade=entry[0], genus=entry[1], species=entry[2])
        row.save()
        taxonomy_rows[taxa_id]=row
print('Taxonomy inserted')
```

This portion of the code is to iterate each key in taxonomy and iterate each set. Once the data is stored in the database, the row will be stored in taxonomy_rows variable which will be used in protein insertion.

```
for data in protein:
    protein_id = data[0]
    sequence = protein_sequence[protein_id] if protein_id in protein_sequence else None
    row = Protein.objects.create(
        protein_id=protein_id, sequence=sequence, taxonomy=taxonomy_rows[data[1]], length=data[2])
    for domain_object in domain_rows[protein_id]:
        row.domains.add(domain_object)
    row.save()
print('Protein inserted')
```

This portion of code will iterate the data in the protein variable. Because a protein can have no sequence so there is a check to see if the protein_id is inside the protein_sequence if is not then None will be assigned to the sequence variable. Once the protein object is created, we must iterate through the domain_rows with the protein_id key as a protein can have

multiple domains, after adding all the domains to the protein object, we save it to database and the populating of the csv have finished.

API And Serializers

[POST] api/protein/

```
class CreateProtein(mixins.CreateModelMixin,        You, 3 weeks
                    generics.GenericAPIView):
    serializer_class = ProteinSerializer

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)
```

This is the code that can be found in the api.py file. The CreateProtein class uses the ProteinSerializer to create a new protein.

```
class TaxonomySerializer(serializers.ModelSerializer):
    You, last week | 1 author (You)
    class Meta:
        model = Taxonomy
        fields = ['taxa_id', 'clade', 'genus', 'species']


You, 7 days ago | 1 author (You)
class PfamSerializer(serializers.ModelSerializer):
    You, 7 days ago | 1 author (You)
    class Meta:
        model = Pfam
        fields = ['domain_id', 'domain_description']


You, 7 days ago | 1 author (You)
class DomainSerializer(serializers.ModelSerializer):
    pfam_id = PfamSerializer()

    You, 7 days ago | 1 author (You)
    class Meta:
        model = Domain
        fields = ['pfam_id', 'description', 'start', 'stop']
```

This are the 3 serializers which can be found in the serializers.py file that is used in the ProteinSerializer. The TaxonomySerializer uses the Taxonomy model, PfamSerializer uses the Pfam model and DomainSerializer uses the Domain model.

```python
class ProteinSerializer(serializers.ModelSerializer):
    taxonomy = TaxonomySerializer()
    domains = DomainSerializer(many=True)

    You, 3 weeks ago | 1 author (You)
    class Meta:
        model = Protein
        fields = ['protein_id', 'sequence', 'taxonomy', 'length', 'domains']

    def create(self, validated_data):
        taxonomy_data = self.initial_data.get('taxonomy')
        domains_data = self.initial_data.get('domains')
        taxonomy = None
        domains_list = []       You, 3 weeks ago • Update web cw …

        if len(Taxonomy.objects.filter(taxa_id=taxonomy_data['taxa_id'])) > 0:
            taxonomy = Taxonomy.objects.filter(
                taxa_id=taxonomy_data['taxa_id'])[0]
        else:
            taxonomy = Taxonomy.objects.create(
                taxa_id=taxonomy_data['taxa_id'], clade=taxonomy_data['clade'], genus=taxonomy_data['genus'], species=taxonomy_data['species'])

        for domain in domains_data:
            pfam = Pfam.objects.filter(
                domain_id=domain['pfam_id']['domain_id'])

            if len(pfam) < 1:
                pfam = Pfam.objects.create(domain_id=domain['pfam_id']['domain_id'],
                                           domain_description=domain['pfam_id']['domain_description'])
            else:
                pfam = pfam[0]

            domain_obj = Domain.objects.create(
                pfam_id=pfam, description=domain['description'], start=domain['start'], stop=domain['stop'])
            domains_list.append(domain_obj)

        protein = Protein.objects.create(protein_id=validated_data['protein_id'],
                                         sequence=validated_data['sequence'],
                                         taxonomy=taxonomy,
                                         length=validated_data['length']
                                         )

        for domain_obj in domains_list:
            protein.domains.add(domain_obj)

        protein.save()
        return protein
```

This is the ProteinSerializer which can be found in the serializers.py file. Firstly, in the create function, the taxonomy data and domains data are being extracted from the post request body. After that we check if the existing taxonomy id is in the database if it is we will use the record that is in the database else create a new taxonomy in the database. The same check goes for pfam if there is existing pfam it will use that pfam else it will create a new pfam record. After the pfam check, a new domain will be created with the pfam as the foreign key and it is appended to the domain list. After that a new protein object is created and the domain list is being looped to add each domain into the protein.

api/protein/<str:protein_id>/

```python
class ProteinDetail(mixins.RetrieveModelMixin,
                    generics.GenericAPIView):
    queryset = Protein.objects.all()
    serializer_class = ProteinSerializer
    lookup_field = 'protein_id'

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

This is the code that can be found in the api.py file. The ProteinDetail class uses the ProteinSerializer to get the detail of a protein. The lookup_field is to search the result based on the protein_id field in the database and the url param.

```python
class TaxonomySerializer(serializers.ModelSerializer):
    You, last week | 1 author (You)
    class Meta:
        model = Taxonomy
        fields = ['taxa_id', 'clade', 'genus', 'species']


You, 7 days ago | 1 author (You)
class PfamSerializer(serializers.ModelSerializer):
    You, 7 days ago | 1 author (You)
    class Meta:
        model = Pfam
        fields = ['domain_id', 'domain_description']


You, 7 days ago | 1 author (You)
class DomainSerializer(serializers.ModelSerializer):
    pfam_id = PfamSerializer()

    You, 7 days ago | 1 author (You)
    class Meta:
        model = Domain
        fields = ['pfam_id', 'description', 'start', 'stop']
```

This are the 3 serializers which can be found in the serializers.py file that is used in the ProteinSerializer. The TaxonomySerializer uses the Taxonomy model, PfamSerializer uses the Pfam model and DomainSerializer uses the Domain model.

```python
class ProteinSerializer(serializers.ModelSerializer):
    taxonomy = TaxonomySerializer()
    domains = DomainSerializer(many=True)

    You, last week | 1 author (You)
    class Meta:
        model = Protein
        fields = ['protein_id', 'sequence', 'taxonomy', 'length', 'domains']
```

This is the ProteinSerializer that can be found in serializers.py file. It calls the TaxonomySerializer and DomainSerializer with the many=True parameter as it is expected that domains will return a list of result. It returns the protein_id, sequence, taxonomy, length and domains field from the Protein model.

api/pfam/<str: pfam_id >/

```
class PFamDetail(mixins.RetrieveModelMixin,
                 generics.GenericAPIView):
    queryset = Pfam.objects.all()
    serializer_class = PfamSerializer
    lookup_field = 'domain_id'
    lookup_url_kwarg = 'pfam_id'

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

This is the code that can be found in the api.py file. The PFamDetail class uses the PfamSerializer to find the pfam details with pfam_id. The lookup_field is the field to search in the database and the lookup_url_kwarg is the field to search in the url param.

```
class PfamSerializer(serializers.ModelSerializer):
    You, 7 days ago | 1 author (You)
    class Meta:
        model = Pfam
        fields = ['domain_id', 'domain_description']
```

This is PfamSerializer which can be found in the serializers.py file. It returns the domain_id and domain_description field from the Pfam model.

api/proteins/<int:taxa_id>/

```
class ProteinList(generics.ListAPIView):
    serializer_class = ListProteinSerializer

    def get_queryset(self):
        return Protein.objects.filter(taxonomy_id__in=Taxonomy.objects.filter(taxa_id=self.kwargs['taxa_id']))
```

This is the code that can be found in the api.py file. The ProteinList class uses the ListProteinSerializer to list protein_id with taxa_id. Because generics.ListAPIView is unlike generics.GenericAPIView as it won't take in lookup_field so there is a need create a get_queryset function to filter the taxa_id.

```
class ListProteinSerializer(serializers.ModelSerializer):
    You, 7 days ago | 1 author (You)
    class Meta:
        model = Protein
        fields = ['id', 'protein_id']
```

This is ListProteinSerializer which can be found in the serializers.py file. It returns the id(database generated) and protein_id field from the Protein model.

api/pfams/<int:taxa_id>/

```
class PfamsList(generics.ListAPIView):
    serializer_class = ListPfamsSerializer

    def get_queryset(self):
        taxonomy = Taxonomy.objects.filter(taxa_id=self.kwargs['taxa_id'])
        protein = Protein.objects.filter(taxonomy_id__in=taxonomy)
        return Domain.objects.filter(protein__in=protein)
```

This is the code that can be found in the api.py file. The PfamList class uses the ListPfamSerializer to list domain_id and domain_description with taxa_id. Because generics.ListAPIView is unlike generics.GenericAPIView as it won't take in lookup_field so there is a need to create a get_queryset function to filter the taxa_id.

```
class ListPfamsSerializer(serializers.ModelSerializer):
    pfam_id = PfamSerializer()

    You, 7 days ago | 1 author (You)
    class Meta:       You, 7 days ago • Update web cw …
        model = Domain
        fields = ['id', 'pfam_id']
```

This is ListPfamsSerializer which can be found in the serializers.py file. It returns the id(database generated in domain table) and pfam_id(domain_id and domain_description from Pfam) field from the Domain model.

api/coverage/<str:protein_id>/

```
@api_view(['GET'])
def coverage(request, protein_id):
    if request.method == 'GET':
        startTotal = 0
        stopTotal = 0
        protein = Protein.objects.filter(protein_id=protein_id)
        if len(protein) == 0:
            return Response({'status': 'protein not found'}, status=status.HTTP_404_NOT_FOUND)
        protein = protein[0]
        domains = Domain.objects.filter(protein=protein)

        for domain in domains:
            startTotal += domain.start
            stopTotal += domain.stop
        return Response({'coverage': (stopTotal-startTotal)/protein.length})
```

This is the code that can be found in the api.py file. The coverage function does not use generics to generate the response instead I use api_view as there is no need to create a model for this api. This function only accepts GET request and a protein_id. Firstly, it will check whether the protein_id exist in the protein table or not if it doesn't exist then 404 will

be returned with a status of protein not found. Next using the protein found which is guaranteed to be 1 record as explained in the models section and search it in the domain table. After retrieving the list of domains, it will iterate through the list and sum up the start and stop value. Lastly the response will be returned with a field coverage and the value which is derived by using the (stopTotal-startTotal)/the protein length.

Test Case

For test case I won't cover on model_factories.py but most of the code there are mock data and some data that will be used in tests.py file to easier compare the result.

tests.py

```
def newProteinFactory(data=default_data):
    ProteinFactory.create(protein_id=data['protein_id'], sequence=data['sequence'], taxonomy=TaxonomyFactory.create(taxa_id=data['taxonomy']['taxa_id'],
                                                                                                                    clade=data[
                                                                                                                        'taxonomy']['clade'],
                                                                                                                    genus=data[
                                                                                                                        'taxonomy']['genus'],
                                                                                                                    species=data['taxonomy']['species']),
                          length=data['length']).domains.set([DomainFactory.create(description=domain['description'],
                                                                                   start=domain['start'],
                                                                                   stop=domain['stop'],
                                                                                   pfam_id=PfamFactory.create(domain_id=domain['pfam_id']['domain_id'],
                                                                                                              domain_description=domain['pfam_id']['domain_description']))
                                                              for domain in data['domains']])


def setUp():
    newProteinFactory()
    newProteinFactory(mock_data_1)
    newProteinFactory(mock_data_2)


def tearDown():
    ProteinDomainLink.objects.all().delete()
    Domain.objects.all().delete()
    Protein.objects.all().delete()
    Taxonomy.objects.all().delete()
    Pfam.objects.all().delete()
```

This few function are helper function which will be used in most of the test case.

```python
class CreateProteinTest(APITestCase):
    url = reverse('create_protein_api')

    def tearDown(self):
        tearDown()

    def test_create_protein_success(self):
        response = self.client.post(self.url, default_data, format="json")
        response.render()
        self.assertEqual(response.status_code, 201)
        data = json.loads(response.content)
        self.assertDictEqual(data, default_data)
        self.assertEqual(Protein.objects.count(), 1)       You, 5 days ago • Add a
        protein = Protein.objects.filter(
            protein_id=default_data['protein_id'])[0]
        self.assertEqual(protein.protein_id, default_data['protein_id'])

    def test_create_protein_without_sequence_success(self):
        data_with_empty_sequence = {**default_data}
        data_with_empty_sequence['sequence'] = ""
        response = self.client.post(
            self.url, data_with_empty_sequence, format="json")
        response.render()
        self.assertEqual(response.status_code, 201)
        data = json.loads(response.content)
        self.assertDictEqual(data, data_with_empty_sequence)
        self.assertEqual(Protein.objects.count(), 1)
        protein = Protein.objects.filter(
            protein_id=default_data['protein_id'])[0]
        self.assertEqual(protein.protein_id, default_data['protein_id'])

    def test_create_protein_duplicate_protein_id_failure(self):
        self.client.post(self.url, default_data, format="json")
        response = self.client.post(self.url, default_data, format="json")
        response.render()
        self.assertEqual(response.status_code, 400)

    def test_create_protein_missing_protein_id_field_failure(self):
        data_without_protein_id = {**default_data}
        del data_without_protein_id['protein_id']
        response = self.client.post(
            self.url, data_without_protein_id, format="json")
        response.render()
        self.assertEqual(response.status_code, 400)
```

```python
def test_create_protein_missing_taxonomy_field_failure(self):
    data_without_taxonomy = {**default_data}
    del data_without_taxonomy['taxonomy']
    response = self.client.post(
        self.url, data_without_taxonomy, format="json")
    response.render()
    self.assertEqual(response.status_code, 400)

def test_create_protein_missing_length_field_failure(self):
    data_without_length = {**default_data}
    del data_without_length['length']
    response = self.client.post(
        self.url, data_without_length, format="json")
    response.render()
    self.assertEqual(response.status_code, 400)

def test_create_protein_missing_domains_field_failure(self):
    data_without_domains = {**default_data}
    del data_without_domains['domains']
    response = self.client.post(
        self.url, data_without_domains, format="json")
    response.render()
    self.assertEqual(response.status_code, 400)
```

This is the test case for create protein api. The first test case test_create_protein_success test whether creating of a new protein is successful by checking the database after the request. The second test case test_create_protein_without_sequence_success test whether if a protein without sequence can be created. The third test case test_create_protein_duplicate_protein_id_failure is to test if duplicated protein_id can be inserted. The fourth test case test_create_protein_missing_protein_id_field_failure is to test if the api will return 400 if protein_id is not provided. The fifth test case test_create_protein_missing_taxonomy_field_failure is to test if the api will return 400 if taxonomy is missing. The sixth test case test_create_protein_missing_length_field_failure is to test if the api return 400 if length is not provided. The last test case test_create_protein_missing_domains_field_failure is to test if the api return 400 if domains is not provided.

```
class ProteinTest(APITestCase):
    good_url = ''
    bad_url = ''

    def setUp(self):
        setUp()
        self.good_url = reverse('protein_detail_api', kwargs={
            'protein_id': 'W5N0U1'})
        self.bad_url = "/api/protein/123/"

    def tearDown(self):
        tearDown()

    def test_protein_detail_return_success(self):
        response = self.client.get(self.good_url, format='json')
        response.render()
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.content)
        self.assertDictEqual(data, default_data)

    def test_protein_detail_return_failure(self):
        response = self.client.get(self.bad_url, format='json')
        response.render()
        self.assertEqual(response.status_code, 404)
```

This is the test for protein detail api. The first test case test_protein_detail_return_success will test if the api return the data when it is found. The last test case test_protein_detail_return_failure will test if the api return 404 if protein_id is not found.

```
You, now | 1 author (You)
class PFamTest(APITestCase):
    good_url = reverse('pfam_detail_api', kwargs={
        'pfam_id': 'PF00520'})       You, now • Uncommitted changes
    bad_url = reverse('pfam_detail_api', kwargs={
        'pfam_id': 'A0A066X4Z8'})

    def setUp(self):
        setUp()

    def tearDown(self):
        tearDown()

    def test_pfam_detail_return_success(self):
        response = self.client.get(self.good_url, format='json')
        response.render()
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.content)
        self.assertDictEqual(data, default_data['domains'][0]['pfam_id'])

    def test_pfam_detail_return_failure(self):
        response = self.client.get(self.bad_url, format='json')
        response.render()
        self.assertEqual(response.status_code, 404)
```

This is the test for pfam detail api. The first test case test_pfam_detail_return_success which will test the if the pfam is being returned. The last test case test_pfam_detail_return_failure will test if the api return 404 if the pfam is not found.

```
class ProteinListTest(APITestCase):
    good_url = reverse('proteins_api', kwargs={
        'taxa_id': 7918})
    bad_url = reverse('proteins_api', kwargs={
        'taxa_id': 5000})

    def setUp(self):
        setUp()

    def tearDown(self):
        tearDown()

    def test_protein_list_return_success(self):
        response = self.client.get(self.good_url, format='json')
        response.render()
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.content)
        self.assertTrue(len(data) == 1)
        self.assertEqual(data[0]['protein_id'], default_data['protein_id'])

    def test_protein_list_empty_result_return_success(self):
        response = self.client.get(self.bad_url, format='json')
        response.render()
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.content)
        self.assertTrue(len(data) == 0)
```

This is the test for list of protein api. The first test case test_protein_list_return_success will return the list of protein_id if the taxa_id can be found. The last test case test_protein_list_empty_result_return_success will test if the api return an empty list if it cannot find any protein_id related to the taxa_id.

```
class PfamListTest(APITestCase):
    good_url = reverse('pfams_api', kwargs={
        'taxa_id': 7918})
    bad_url = reverse('pfams_api', kwargs={
        'taxa_id': 5000})

    def setUp(self):
        setUp()

    def tearDown(self):
        tearDown()

    def test_pfam_list_return_success(self):
        response = self.client.get(self.good_url, format='json')
        response.render()
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.content)
        self.assertTrue(len(data) == 1)
        self.assertDictEqual(data[0]['pfam_id'],
                             default_data['domains'][0]['pfam_id'])

    def test_pfam_list_empty_result_return_success(self):
        response = self.client.get(self.bad_url, format='json')
        response.render()
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.content)
        self.assertTrue(len(data) == 0)
```

This is the test for list of pfam api. The first case test_pfam_list_return_success will return the list of pfam if taxa_id is found. The last test case test_pfam_list_empty_result_return_success will test if the api return an empty list if the taxa_id is not found.

```
class CoverageTest(APITestCase):
    good_url = reverse('coverage_api', kwargs={
        'protein_id': 'W5N0U1'})
    bad_url = reverse('coverage_api', kwargs={
        'protein_id': 'invalid'})

    def setUp(self):
        setUp()

    def tearDown(self):
        tearDown()

    def test_coverage_return_success(self):
        response = self.client.get(self.good_url, format='json')
        response.render()
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.content)
        start = default_data['domains'][0]['start']
        stop = default_data['domains'][0]['stop']
        length = default_data['length']
        self.assertEqual(data['coverage'], (stop-start)/length)

    def test_coverage_return_failure(self):
        response = self.client.get(self.bad_url, format='json')
        response.render()
        self.assertEqual(response.status_code, 404)
```

This is the test for coverage api. The first test case test_coverage_return_success will return a response with a calculated coverage. The last test case test_coverage_return_failure will test the api in returning 404 if the protein_id is not found.

The remaining test case in the file is serializer test case which I won't go through as this report is getting too lengthy but the serializer test case basically test the column it returned.

Conclusion

To summarize this coursework, I have learnt a lot from Django and the thinking in normalizing the dataset provided. It was a fun journey for this coursework, and I am sorry for my bad English and some of the thing I might not explained it clearly. Lastly, thank you for your time for reading this lengthy report.