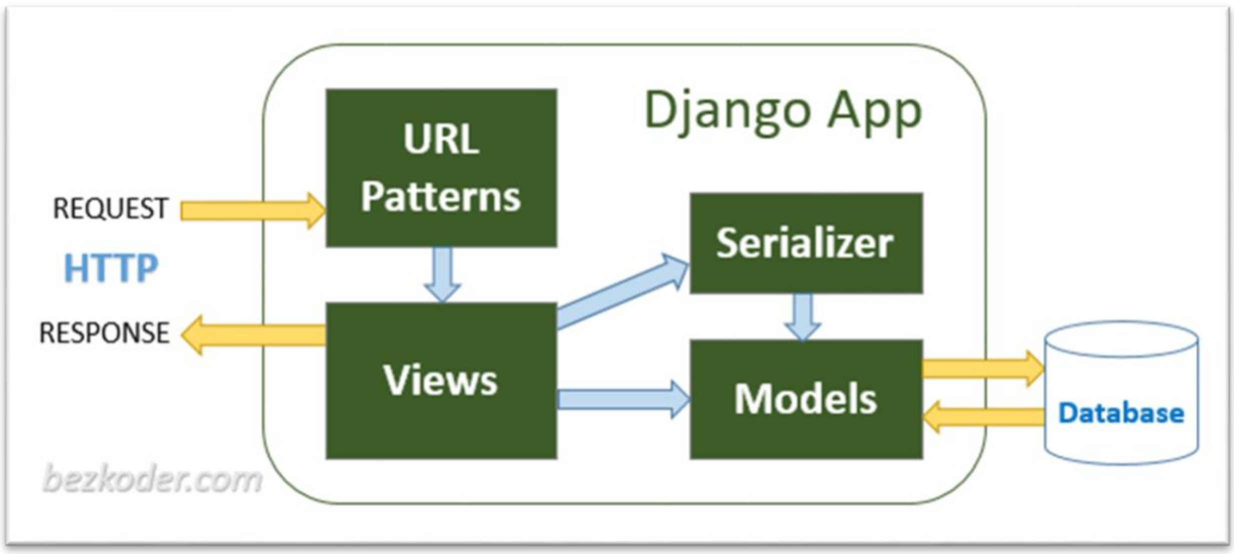
Midterm Report

Introduction

For this midterm, we are assigned to develop a RESTFUL API Web service, that returns JSON data from csv-populated database. The purpose of this RESTFUL API application is to allow the researchers to query data from the database as and when they need.

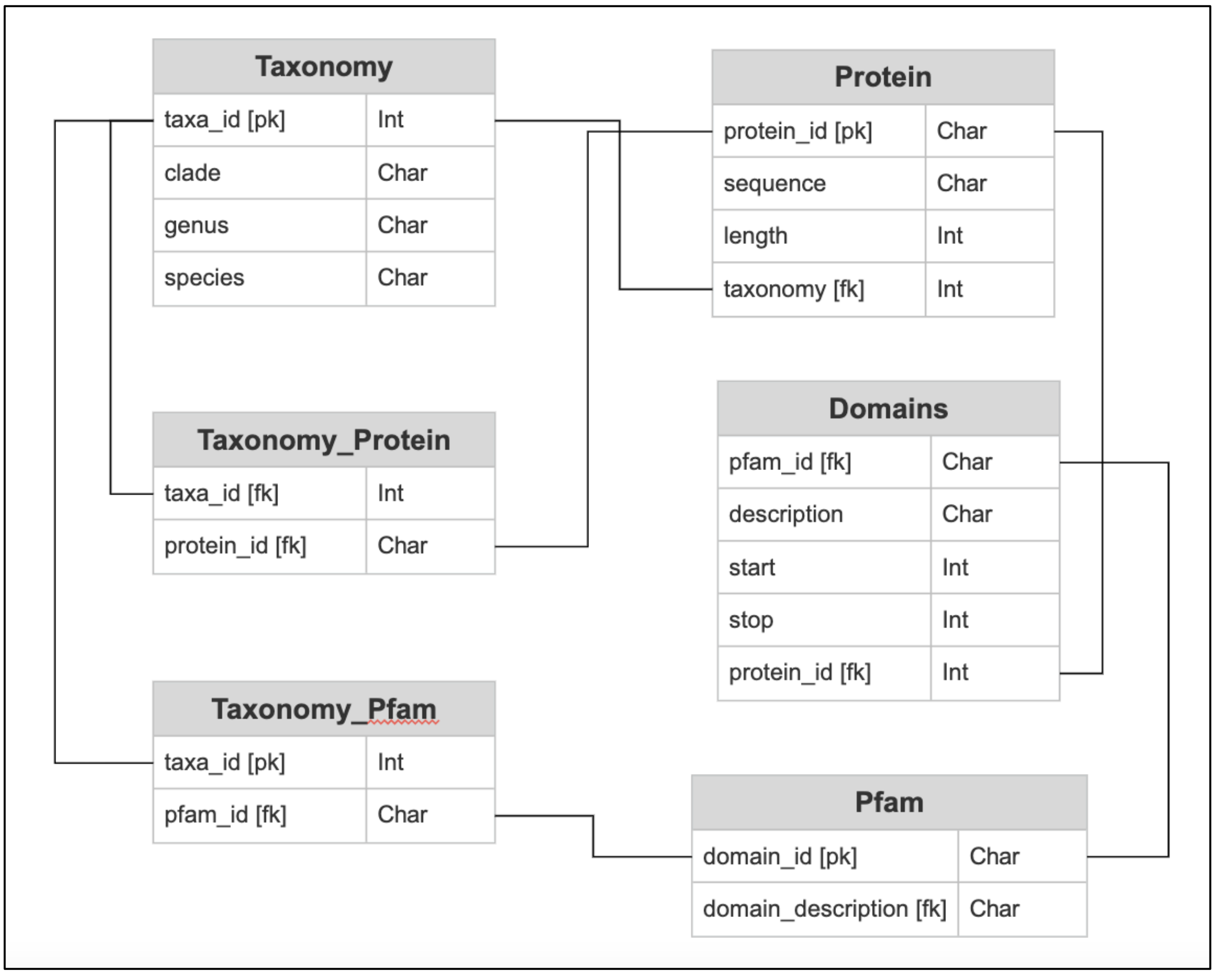


Source: https://www.bezkoder.com/django-rest-api/

The image above summarizes how the Django rest framework works. When the path from the url patterns is called, a request is made, and a response is returned. Models are used to define the tables in the database, and serializers allows data to be converted to json format for displaying on the api.

Tables/Models

I created 6 tables/models in models.py, to contain the data in the 3 csv files. They are: Taxonomy, Protein, Pfam, Domains, Taxonomy_Protein, Taxonomy_Pfam. Each of the tables contains relevant fields to store data, and are linked with foreign keys.



Since organism have many proteins:

Taxonomy --- Protein: One to many relationship

Since proteins have many domains:

Proteins --- Domains: One to many relationship

Since domains have one pfam domain_id:

Domains --- Pfam: One to One relationship

Taxonomy_Pfam is a linking table for Taxonomy and Pfam tables. It is used to retrieve the pfam's domain_id [pk]. This is to ensure that the domain_id and domain_description data can be retrieved when the api/pfams/[TAXA ID] is called.

Taxonomy_Protein is a linking table for Taxonomy and Proteins tables. It is used to retrieve the Protein's protein_id [pk]. This is to ensure that the protein_id data can be retrieved when the api/proteins/[TAXA ID] is called.

Taxonomy
taxa_id [Primary key]
calde
genus
species

Protein
protein_id [Primary key]
sequence
length
taxonomy [Foreign Key, links to Taxonomy]

Domains
pfam_id
description
start
stop
protein_id [Foreign Key, links to Protein]

Pfam
domain_id [Primary key]
domain_description

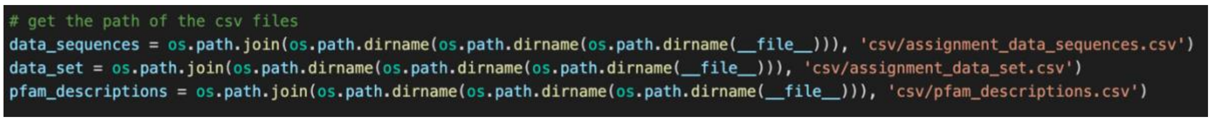
Taxonomy_Protein [link table]
taxa_id [Foreign Key, links to Taxonomy]
protein_id [Foreign Key, links to Protein]

Taxonomy_Pfam [link table]
taxa_id [Foreign Key, links to Taxonomy]
pfam_id [Foreign Key, links to Pfam]

Populating Database

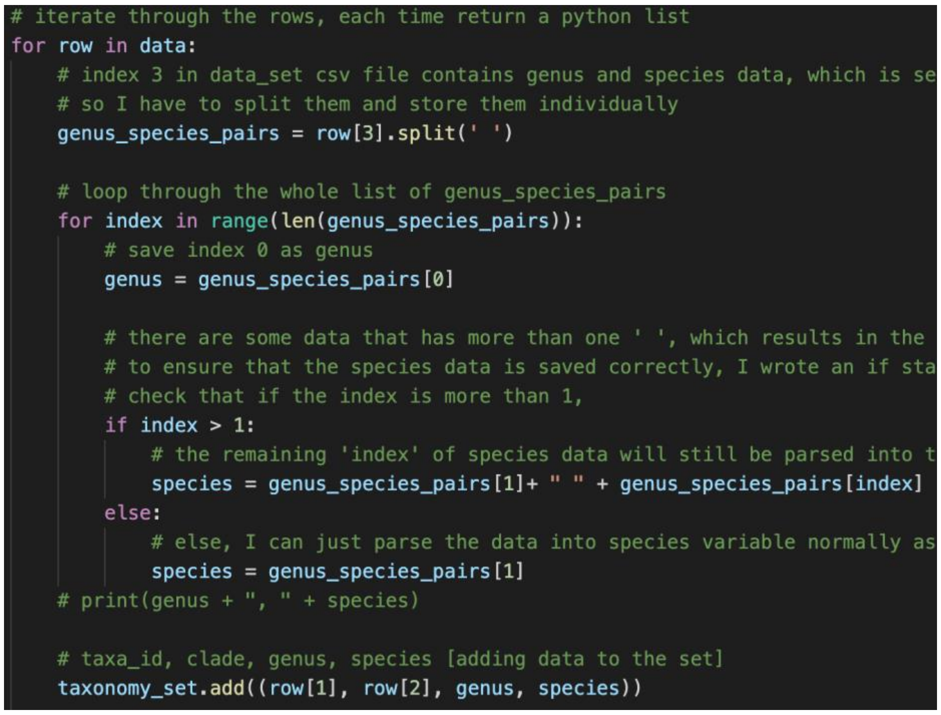
After setting up the models, I need to split the csv data up and populate them into the correct tables and fields.

This is done under scripts/populate_db.py file. Firstly, I initialised a variable to reference the csv paths. The csv files are places in a folder called csv, and contains the 3 csv files provided.



```python
# get the path of the csv files
data_sequences = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(__file__))), 'csv/assignment_data_sequences.csv')
data_set = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(__file__))), 'csv/assignment_data_set.csv')
pfam_descriptions = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(__file__))), 'csv/pfam_descriptions.csv')
```

The csv files are than read, split into parts, and saved in the different tables and fields.

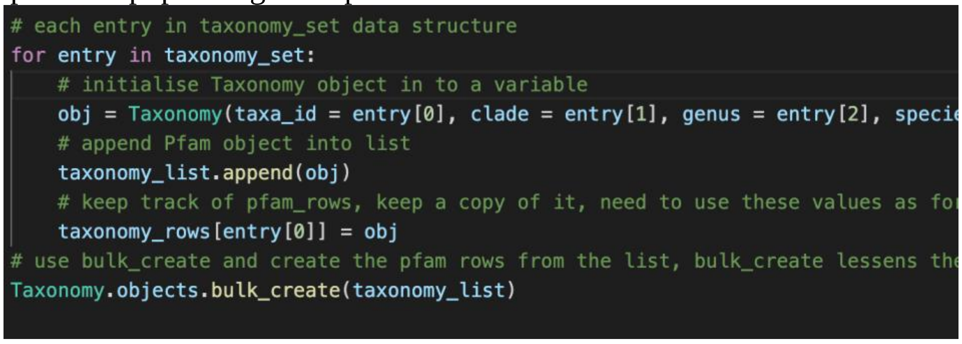


```python
# iterate through the rows, each time return a python list
for row in data:
    # index 3 in data_set csv file contains genus and species data, which is se
    # so I have to split them and store them individually
    genus_species_pairs = row[3].split(' ')

    # loop through the whole list of genus_species_pairs
    for index in range(len(genus_species_pairs)):
        # save index 0 as genus
        genus = genus_species_pairs[0]

        # there are some data that has more than one ' ', which results in the
        # to ensure that the species data is saved correctly, I wrote an if sta
        # check that if the index is more than 1,
        if index > 1:
            # the remaining 'index' of species data will still be parsed into t
            species = genus_species_pairs[1]+ " " + genus_species_pairs[index]
        else:
            # else, I can just parse the data into species variable normally as
            species = genus_species_pairs[1]
    # print(genus + ", " + species)

    # taxa_id, clade, genus, species [adding data to the set]
    taxonomy_set.add((row[1], row[2], genus, species))
```

The code snippet above is after the data set csv file is opened. It goes through each row in the csv file, and gets the data. For example, column 4 contains genus and species data, which is separated by a blank space. Since one of the api should return the genus and species data separately, I than created genus and species variables to save the data that has been split up unto them.

After looking through the data set, I realised that some rows that contained genus and species data had more than one space in them. An if-else statement is used to check that if there is more than 2 elements (more than 1 space), the rest of the data at the back are concatenated to the species variable. This ensures that no data is lost in the populating process.

After which, the taxa_id, clade, genus and species data are added to the taxonomy_set. Before creating the objects, the objects in all the tables are deleted to ensure that there are no residual data from previous populating attempts.



```
# each entry in taxonomy_set data structure
for entry in taxonomy_set:
    # initialise Taxonomy object in to a variable
    obj = Taxonomy(taxa_id = entry[0], clade = entry[1], genus = entry[2], specie
    # append Pfam object into list
    taxonomy_list.append(obj)
    # keep track of pfam_rows, keep a copy of it, need to use these values as fo
    taxonomy_rows[entry[0]] = obj
# use bulk_create and create the pfam rows from the list, bulk_create lessens the
Taxonomy.objects.bulk_create(taxonomy_list)
```

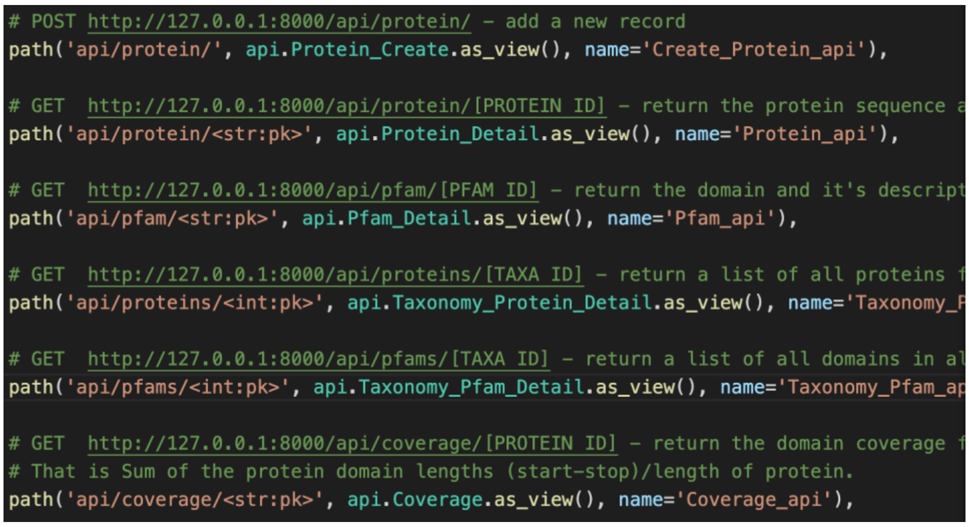
It is than time to create the objects. The method that was taught in Coursera is to create the objects each time it goes through an entry in the taxonomy_set. This results in taking huge amount of time to populate the database as the csv file grows bigger. To make this process more efficient, I look through the documents online and found another method, bulk create, which is more efficient for larger files. Firstly, I created a variable that stores the object instance with the fields. This object variable is than appended into a list, which is than used to bulk_create all the objects at once. This bulk_create process is significantly faster, taking about 25 seconds for populating all 3 files, as compared to the usual create method that takes 2.5minutes. I have retained the code that creates objects by each entry at the bottom.

This process is than created for each model. I have also included code that prints out the amount of time needed to populate the database.

To populate the database: python3 ./scripts/populate_db.py

URLS

With the database correctly setup and data populated in, I moved on to create a urls.py file in the genedata app directory. This file contains 6 paths for the api.

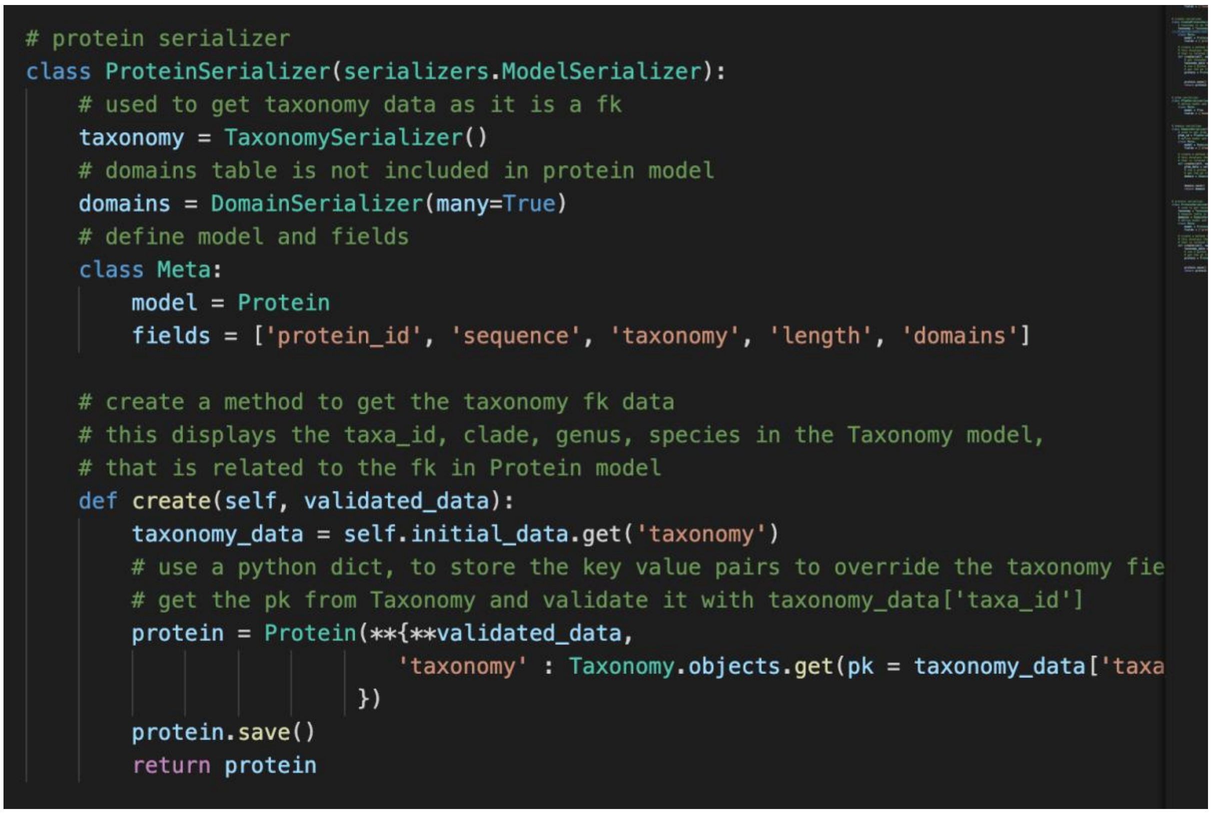


```
# POST http://127.0.0.1:8000/api/protein/ - add a new record
path('api/protein/', api.Protein_Create.as_view(), name='Create_Protein_api'),

# GET  http://127.0.0.1:8000/api/protein/[PROTEIN ID] - return the protein sequence a
path('api/protein/<str:pk>', api.Protein_Detail.as_view(), name='Protein_api'),

# GET  http://127.0.0.1:8000/api/pfam/[PFAM ID] - return the domain and it's descript
path('api/pfam/<str:pk>', api.Pfam_Detail.as_view(), name='Pfam_api'),

# GET  http://127.0.0.1:8000/api/proteins/[TAXA ID] - return a list of all proteins f
path('api/proteins/<int:pk>', api.Taxonomy_Protein_Detail.as_view(), name='Taxonomy_P

# GET  http://127.0.0.1:8000/api/pfams/[TAXA ID] - return a list of all domains in al
path('api/pfams/<int:pk>', api.Taxonomy_Pfam_Detail.as_view(), name='Taxonomy_Pfam_ap

# GET  http://127.0.0.1:8000/api/coverage/[PROTEIN ID] - return the domain coverage f
# That is Sum of the protein domain lengths (start-stop)/length of protein.
path('api/coverage/<str:pk>', api.Coverage.as_view(), name='Coverage_api'),
```

Each path includes a class name that is used to setup the api, as well as a name that will be used during testing in the later part. Other than the ‘api/protein/’ path, the other paths includes a primary key (either integer or string depending on the primary key data). When the path is visited, the primary key is used to determine which row of data to display.

Serializers

Rest_framework is used to create serializers. The serializer classes in serializers.py are used to display the api data as json.



```python
# protein serializer
class ProteinSerializer(serializers.ModelSerializer):
    # used to get taxonomy data as it is a fk
    taxonomy = TaxonomySerializer()
    # domains table is not included in protein model
    domains = DomainSerializer(many=True)
    # define model and fields
    class Meta:
        model = Protein
        fields = ['protein_id', 'sequence', 'taxonomy', 'length', 'domains']

    # create a method to get the taxonomy fk data
    # this displays the taxa_id, clade, genus, species in the Taxonomy model,
    # that is related to the fk in Protein model
    def create(self, validated_data):
        taxonomy_data = self.initial_data.get('taxonomy')
        # use a python dict, to store the key value pairs to override the taxonomy fie
        # get the pk from Taxonomy and validate it with taxonomy_data['taxa_id']
        protein = Protein(**{**validated_data,
                            'taxonomy' : Taxonomy.objects.get(pk = taxonomy_data['taxa
                           })
        protein.save()
        return protein
```

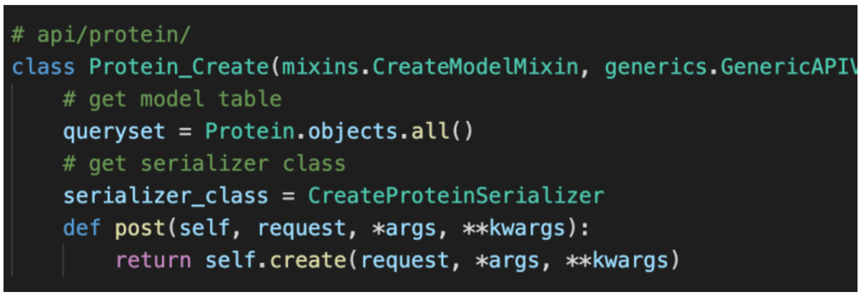
Each serializer class uses the serializers imported from rest_framework. They include a meta class within, that tells the api the model and fields. Some models includes foreign keys, and I have to create a method to get the foreign key data (reference the foreign key).

Variables are first used to save the serializer class that the foreign key is references from. In the create method, first have a variable that saves the initial foreign key data. Than, get the key from the referenced model, use a python dictionary to override the foreign key data. The data retrieved by validating the primary key of the referenced model, with the foreign key. If it matched, the data will then be retrieved.

This process is repeated for each model, expect for the 2 link table models.

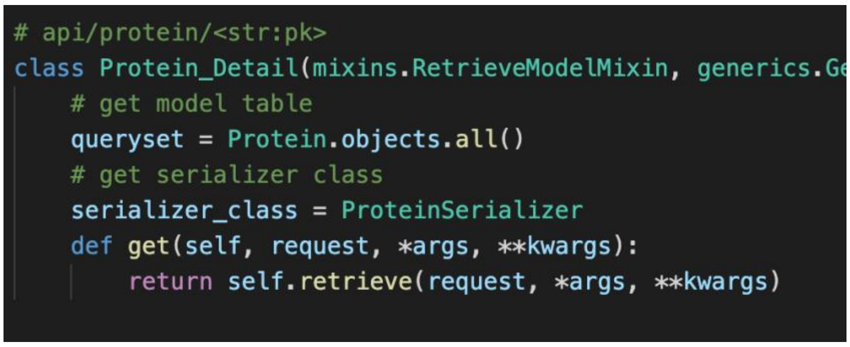
API

In the api.py file, mixins is used to create or retrieve data, and generics is used to display the api view. Using the rest_framework makes the whole process of creating a api application easier and more straightforward.



```
# api/protein/
class Protein_Create(mixins.CreateModelMixin, generics.GenericAPIV
    # get model table
    queryset = Protein.objects.all()
    # get serializer class
    serializer_class = CreateProteinSerializer
    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)
```

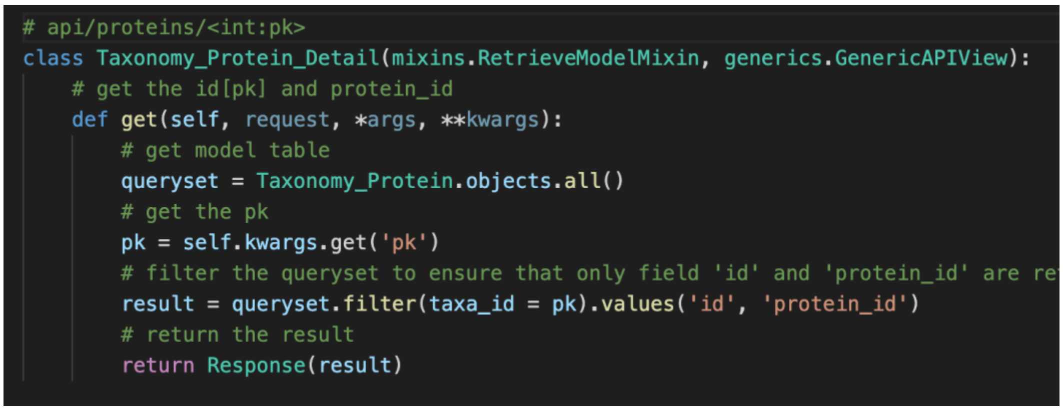
To create new protein and post it via the api interface, CreateModelMixin is used. The model is first defined and saved as a variable. The serializer class is also defined. After which, a post method is used to create the new protein as per the user request.



```
# api/protein/<str:pk>
class Protein_Detail(mixins.RetrieveModelMixin, generics.Ge
    # get model table
    queryset = Protein.objects.all()
    # get serializer class
    serializer_class = ProteinSerializer
    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)
```

Otherwise, the RetrieveModelMixin is used to get the data from the database and display it. Similarly, the model and serializer are defined and saved as variable. A get method is than used to get the data and display it.

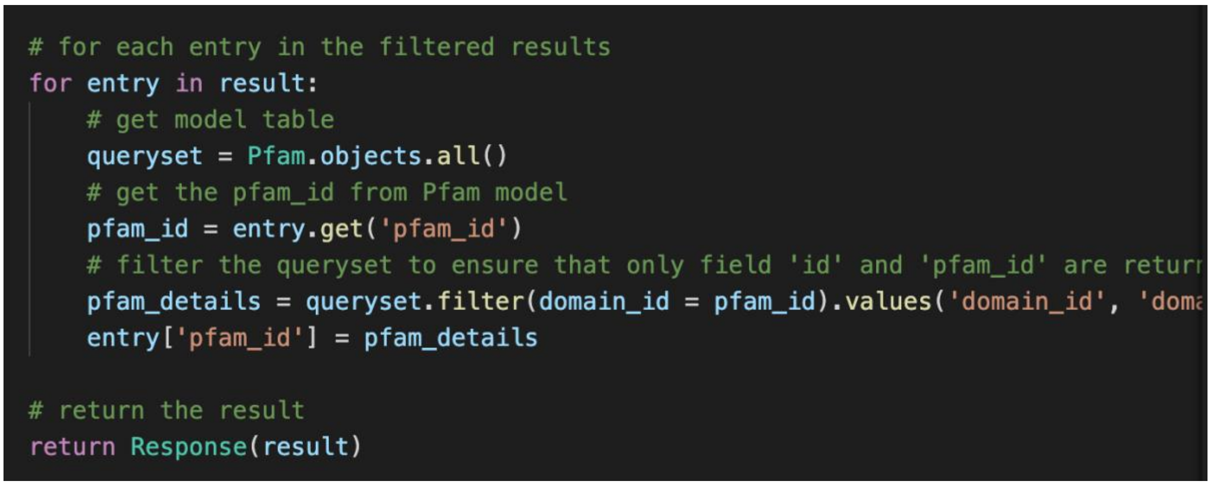
The 2 link tables doesn't use serializers, and data is returned as a response.



```
# api/proteins/<int:pk>
class Taxonomy_Protein_Detail(mixins.RetrieveModelMixin, generics.GenericAPIView):
    # get the id[pk] and protein_id
    def get(self, request, *args, **kwargs):
        # get model table
        queryset = Taxonomy_Protein.objects.all()
        # get the pk
        pk = self.kwargs.get('pk')
        # filter the queryset to ensure that only field 'id' and 'protein_id' are re
        result = queryset.filter(taxa_id = pk).values('id', 'protein_id')
        # return the result
        return Response(result)
```

I will be using the Taxonomy Protein link table to explain. A get method is created within the class, with the model saved as a variable. I than use self.kwargs.get to retrieve the primary

key of the model table. This primary key is used to filter the model table, to get the relevant row is retrieved. The field names are also parsed in to ensure that only the required fields are returned. The filtered result is returned as response. Now, the paths would correctly display the datas.
The Taxonomy Pfam link table is slightly more complicated, as within the pfam_id, I have to display the domain_id and domain_description.



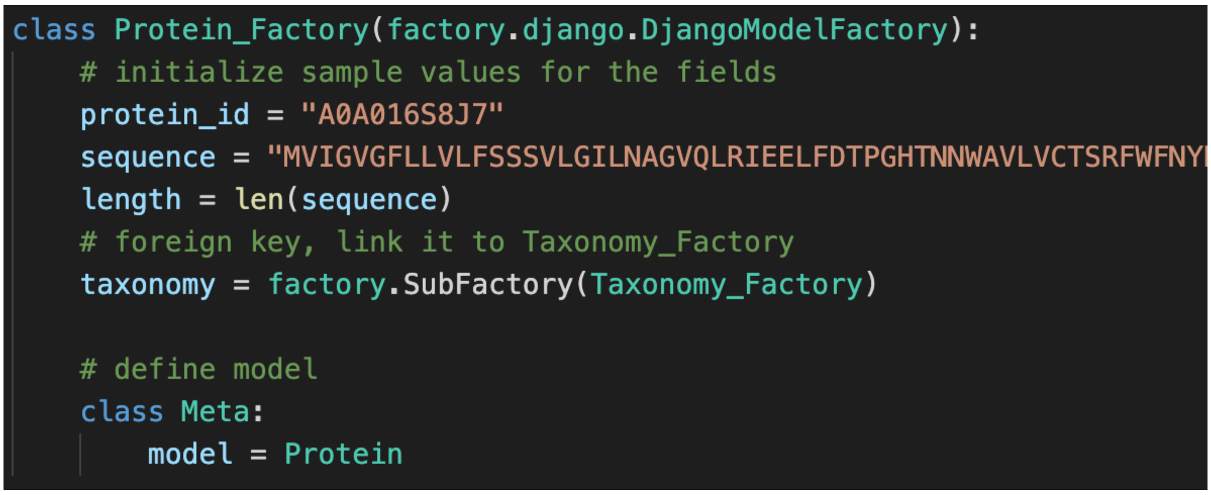
```
# for each entry in the filtered results
for entry in result:
    # get model table
    queryset = Pfam.objects.all()
    # get the pfam_id from Pfam model
    pfam_id = entry.get('pfam_id')
    # filter the queryset to ensure that only field 'id' and 'pfam_id' are retur
    pfam_details = queryset.filter(domain_id = pfam_id).values('domain_id', 'doma
    entry['pfam_id'] = pfam_details

# return the result
return Response(result)
```

With the filtered result that gets the id and pfam_id field, I used a for loop to go through each entry. The Pfam model is queried, filtered by comparing the domain_id and the pfam_id, to retrieve the domain_id and domain_description. These data are than returned as response, and displayed in the api interface when the pfams/pk path is called.

To run the api: python3 manage.py runserver

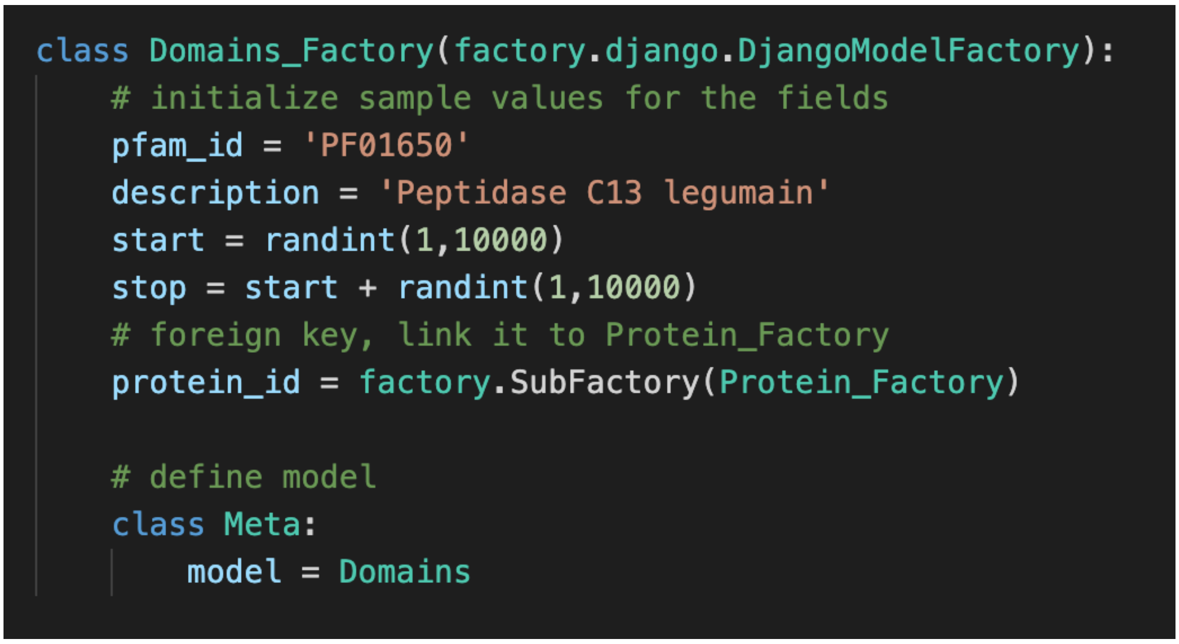
Model Factories

To ensure that the correct data is returned, model_factories.py is used to create sample data, for the unit testing to refer and check on. As per the number of models, 6 model factory classes are created.



```python
class Protein_Factory(factory.django.DjangoModelFactory):
    # initialize sample values for the fields
    protein_id = "A0A016S8J7"
    sequence = "MVIGVGFLLVLFSSSVLGILNAGVQLRIEELFDTPGHTNNWAVLVCTSRFWFNY
    length = len(sequence)
    # foreign key, link it to Taxonomy_Factory
    taxonomy = factory.SubFactory(Taxonomy_Factory)

    # define model
    class Meta:
        model = Protein
```

Each field is given a sample data. In the case of a foreign key, I used the subfactory method, to call the referenced factory. The model is defined in the meta class within the factory class.

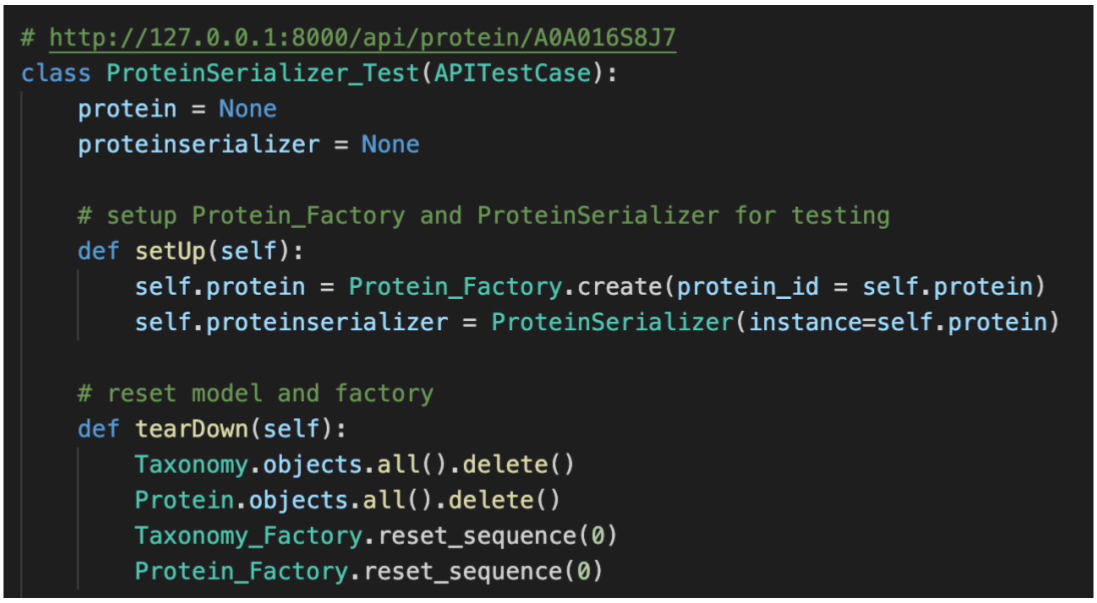


```python
class Domains_Factory(factory.django.DjangoModelFactory):
    # initialize sample values for the fields
    pfam_id = 'PF01650'
    description = 'Peptidase C13 legumain'
    start = randint(1,10000)
    stop = start + randint(1,10000)
    # foreign key, link it to Protein_Factory
    protein_id = factory.SubFactory(Protein_Factory)

    # define model
    class Meta:
        model = Domains
```

For the domains factory, the start and stop fields uses random integer ranging from 1 to 10000 (with the addition of 'start' data for stop field) for testing.

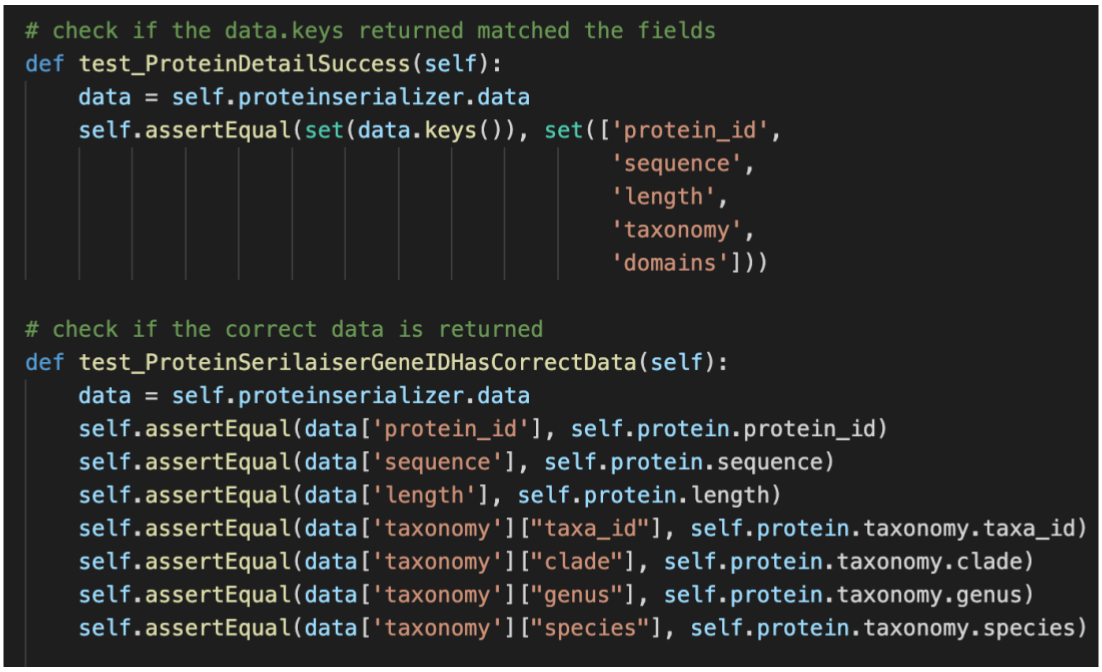
Unit Tests

The protein, pfam models have tests for both the serializers and api path. The 2 link tables, taxonomy_protein and taxonomy_domain only have tests for the api path.



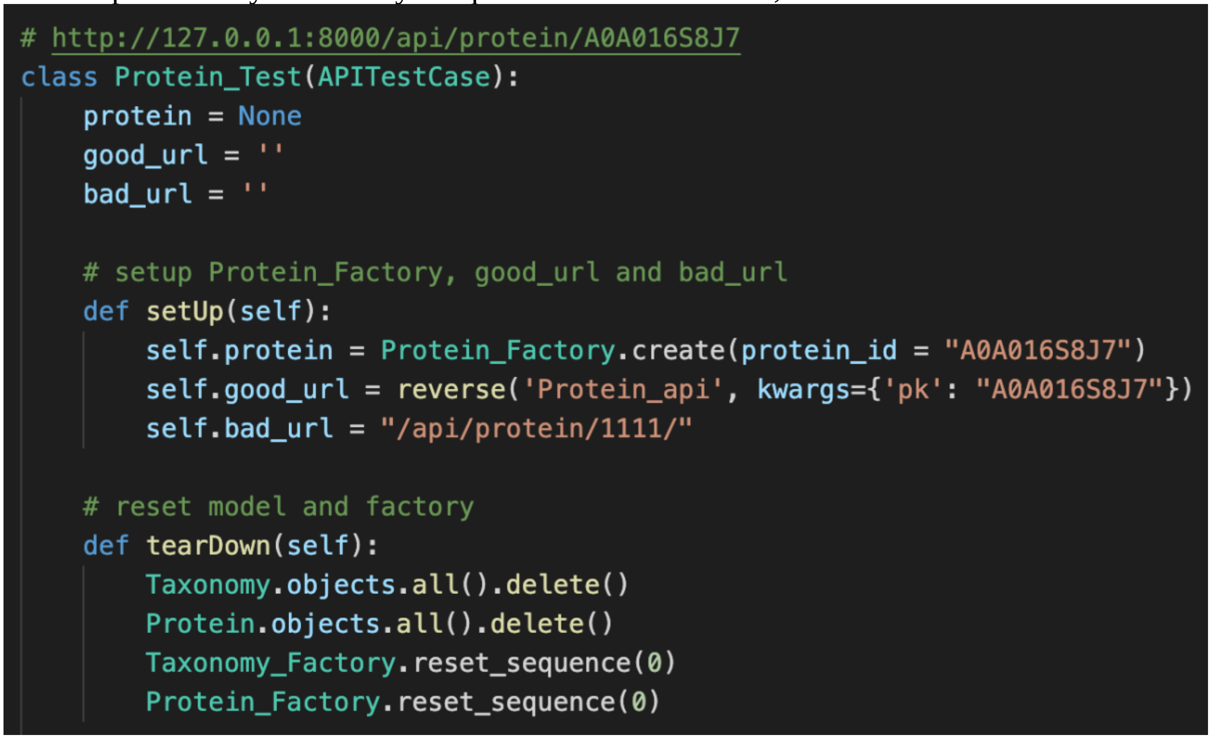
```python
# http://127.0.0.1:8000/api/protein/A0A016S8J7
class ProteinSerializer_Test(APITestCase):
    protein = None
    proteinserializer = None

    # setup Protein_Factory and ProteinSerializer for testing
    def setUp(self):
        self.protein = Protein_Factory.create(protein_id = self.protein)
        self.proteinserializer = ProteinSerializer(instance=self.protein)

    # reset model and factory
    def tearDown(self):
        Taxonomy.objects.all().delete()
        Protein.objects.all().delete()
        Taxonomy_Factory.reset_sequence(0)
        Protein_Factory.reset_sequence(0)
```

For the serializer test, a setup method is created to store the factory and serializer for testing. The teardown method is responsible for deleting and resetting the models and factories. This ensures that data is cleared after each test.



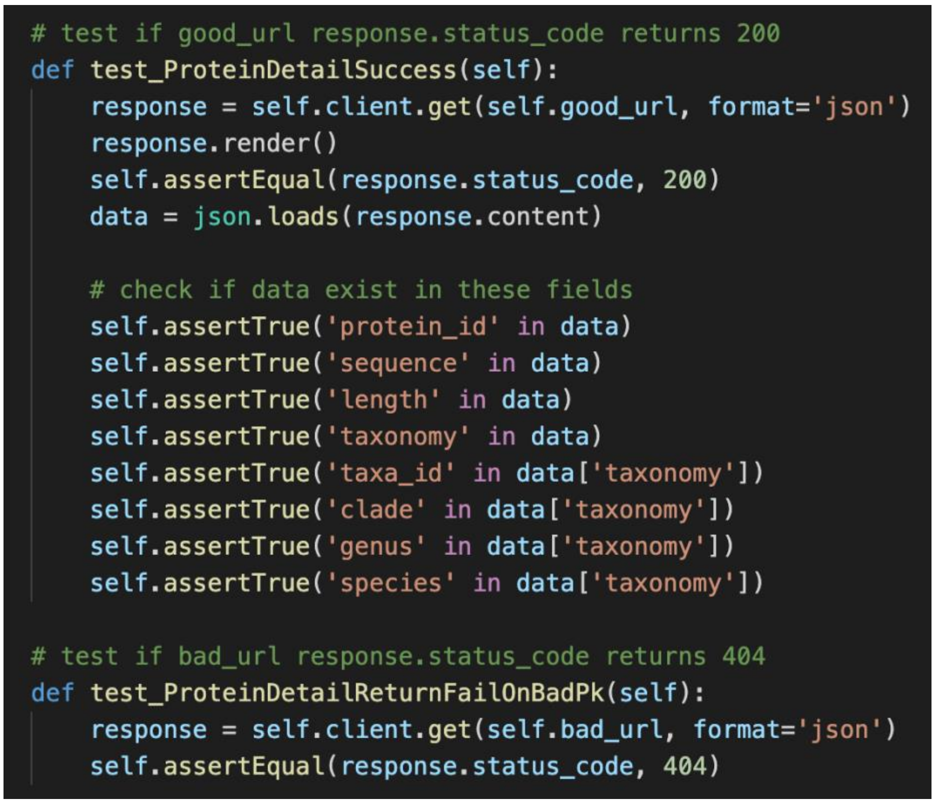
```python
# check if the data.keys returned matched the fields
def test_ProteinDetailSuccess(self):
    data = self.proteinserializer.data
    self.assertEqual(set(data.keys()), set(['protein_id',
                                            'sequence',
                                            'length',
                                            'taxonomy',
                                            'domains']))

# check if the correct data is returned
def test_ProteinSerilaiserGeneIDHasCorrectData(self):
    data = self.proteinserializer.data
    self.assertEqual(data['protein_id'], self.protein.protein_id)
    self.assertEqual(data['sequence'], self.protein.sequence)
    self.assertEqual(data['length'], self.protein.length)
    self.assertEqual(data['taxonomy']["taxa_id"], self.protein.taxonomy.taxa_id)
    self.assertEqual(data['taxonomy']["clade"], self.protein.taxonomy.clade)
    self.assertEqual(data['taxonomy']["genus"], self.protein.taxonomy.genus)
    self.assertEqual(data['taxonomy']["species"], self.protein.taxonomy.species)
```

Test methods are named with '_test' at the start so that Django knows which are testing methods to run. Firstly, ProteinDetailSuccess checks if the fields returned are the correct fields when the path is called. Next, ProteinSerialiserGeneIDHasCorrectData checks if each of the data in the fields returned is correct. It checks the data returned with the model factory data, whereby for both instances the protein_id is the same. assertEqual would return true if the data provided by the factory is equal to the data returned, and false otherwise.



```python
# http://127.0.0.1:8000/api/protein/A0A016S8J7
class Protein_Test(APITestCase):
    protein = None
    good_url = ''
    bad_url = ''

    # setup Protein_Factory, good_url and bad_url
    def setUp(self):
        self.protein = Protein_Factory.create(protein_id = "A0A016S8J7")
        self.good_url = reverse('Protein_api', kwargs={'pk': "A0A016S8J7"})
        self.bad_url = "/api/protein/1111/"

    # reset model and factory
    def tearDown(self):
        Taxonomy.objects.all().delete()
        Protein.objects.all().delete()
        Taxonomy_Factory.reset_sequence(0)
        Protein_Factory.reset_sequence(0)
```

As for the path test, good and bad url variables are initialized with empty string. The setup method parses in sample protein_id data to create the protein factory object. Django.urls reverse is used to get define a path with the correct protein_id parsed in, which is inserted into good_url variable. Bad_url is defined with random integer. Similar to the serializer test, a teardown method is used to reset model and factories.



```
# test if good_url response.status_code returns 200
def test_ProteinDetailSuccess(self):
    response = self.client.get(self.good_url, format='json')
    response.render()
    self.assertEqual(response.status_code, 200)
    data = json.loads(response.content)

    # check if data exist in these fields
    self.assertTrue('protein_id' in data)
    self.assertTrue('sequence' in data)
    self.assertTrue('length' in data)
    self.assertTrue('taxonomy' in data)
    self.assertTrue('taxa_id' in data['taxonomy'])
    self.assertTrue('clade' in data['taxonomy'])
    self.assertTrue('genus' in data['taxonomy'])
    self.assertTrue('species' in data['taxonomy'])

# test if bad_url response.status_code returns 404
def test_ProteinDetailReturnFailOnBadPk(self):
    response = self.client.get(self.bad_url, format='json')
    self.assertEqual(response.status_code, 404)
```

Protein detail success checks if json response is as expected and the fields exist in the data. Response is retrieved by parsing in the good_url and stating that the format retuned should be in json. This response is than rendered. I used assertEqual to check that if the response is status good, meaning its code is 200, it will return True. assertTrue is used to check if the fields in the response is within the data returned.

ProteinDetailReturnFailOnBadPk checks that if the bad_url is used to get the response, the status code should be 400.

To run the test: python3 manage.py test

Summary

With this, the database will be populated with the csv file and placed into models n fields correctly, and then displayed through the api interface as json format. Unit testing helps to ensure that the api is working as expected, and would return an error of assertion fail if any errors were to occur.