RESTful Web Service

BSc Computer Science CM3035 - Advanced Web Development Midterm Coursework

In an article written by TIME magazine (Bremmer, 2021) Singapore was listed as one of the countries that had a good response to the covid pandemic of 2020, to see how covid impacted the country during that time and whether that statement is justified I utilised a dataset containing various details regarding the spread of covid in Singapore between 2020-2022 (Chua, 2022). I utilised this dataset as it contains a large range of data regarding the covid from the number of positive cases to the transmission rate to the hospitalisation statistics. As I was more interested in how covid directly affected the country I focused more on the general data such as the number of cases and mortality rate with additional data regarding the hospitalisation statistics to see how the cases were dealt with.

The 6 endpoints implemented are as follows:

```
1. 'api/phase'
```

This endpoint is interesting as it lists the various phases that Singapore utilised to control the extent of social lockdown and isolation. Query utilises Django generic ListAPIView alongside phase table.

```
2. 'api/date'
```

This endpoint list the current range of data available in the database and allows users to know what data is available to them and where to add on from if they wish to. This was done by utilising a custom date serializer to ensure the correct data type was being returned along with additional functions to obtain the highest and lowest dates.

```
3. 'api/covid/<str:pk>'
```

This endpoint returns a row from the CovidStats table based on the date input by users. This allows users to find interesting statistics regarding the spread of covid from various dates. This was done by utilising nested serializers as well as djangos mixins class to handle args.

```
4. 'api/hospital/<str:pk>'
```

This endpoint returns additional data regarding the hospitalisation statistics due to covid based on the date input by users. Requires a custom serializer to obtain only the date from the covid row which is then nested in the hospital serializer in order to return the row desired – also uses Django mixins class to handle args.

```
5. 'api/list/<str:pk>'
```

Endpoint returns a combined list of both covid and hospitalisation data based on a date returned by users. Allows user to see the overall effect of covid on the country. Utilises nested serializers as well as djangos mixins class to handle user args.

```
6.'api/stats/<str:model>/<str:column>'
```

Endpoint returns statistics such as the minimum, maximum, sum and average of the values in a given column and table. This allows users to obtain cumulative data from the database regarding statistics they are interested in instead of having to calculate it themselves. Utilises additional functions such as aggregate to obtain various statistics as well as requires the handling of two user arguments ad the validation that the respective requested data exist.

R1: The application contains the basic functionality described in class

As the application is built using Django it utilizes the structures and functions Django provides that were also described in class.

a) Models and migrations

A model as defined by Django is meant to be the single definitive source for information regarding data (Django, Models, n.d.) in your database. As such there were utilised in my application to define the tables in my database.

To do this each class in my model is mapped to a table in my database that inherits from the Django model class.

Models.py:

```
class Phase(models.Model):
    phase_name = models.CharField(max_length=256, null=False, blank=False)
    def __str__(self):
        return self.phase_name

class CovidStats(models.Model):
    date_id = models.DateField(null=False, blank=False, primary_key = True)
    positive = models.IntegerField(null=False, blank = True)
    mortality = models.IntegerField(null=False, blank = True)
    vacc = models.ForeignKey(Vaccination, on_delete=models.DO_NOTHING)
    phase = models.ForeignKey(Phase, on_delete=models.DO_NOTHING)
```

Tables that are foreign keys must be declared before they are used as foreign keys. Hence, the structure of my code in my models.py follow this rule.

b) Forms, validators and serialisation

Forms & Validators

Django forms class describes a form and determines how it works and appears (Django, Working with forms, n.d.). Hence, I utilised it much like I would utilise HTML forms - to collect data.

By using forms I can help users add data into the database without having to use the api interface. In order to do this I utilised Django ModelForm that maps the fields in my model class to a form. This helps me not only create forms more easily but also helps validate the data

received by the form as because the form is built based on my models it helps to check that the data input into the form matches the data type as written in my models.

```
class CovidForm(ModelForm):
    class Meta:
        model = CovidStats
        fields = ['date_id','positive','mortality','vacc','phase']
```

Serializers

Serializers as defined by the Django Rest Framework guide are used to convert complex data into easily rendered data types such as json. They can also be used in reverse to convert data passed to it back into more complex types after validating it.

As such my serializers were written according to the data I wanted to convert to either pass on or to receive from users.

As majority of the serializers I used were intended to work with model instances, majority of my serializers inherited from Django ModelSerializer class except for my DateRangeSerializer which was used to ensure the data returned to the API were both dates.

serializers.py

```
# Serializer for date
class DateRangeSerializer(serializers.Serializer):
    earliest_date = serializers.DateField()
    latest_date = serializers.DateField()

# Return vacc_perc from vaccination table
class VaccSerializer(serializers.ModelSerializer):
    class Meta:
        model = Vaccination
        fields = ['vacc_perc']
```

Similar to my models serialized data from tables that were used as foreign keys in other serializers were declared.

```
# Return row form covidstats table
class CovidSerializer(serializers.ModelSerializer):
    phase = PhaseSerializer()
    vacc = VaccSerializer()
    class Meta:
        model = CovidStats
        fields = ['date_id','positive','mortality','vacc','phase']

        def create(self, validated_data):
```

```
            phase_data = self.initial_data.get('phase')
            vacc_data = self.initial_data.get('vacc')
            covid = CovidStats(**{**validated_data,
                            'phase': Phase.objects.get(pk = phase_data['id']),
                            'vacc': Vaccination.objects.get(pk=vacc_data['id'])
                             })
            covid.save()
            return covid
```

c) Django-rest-framework

Django rest framework utilises serializers, views and urls in order to create a simple way to create a RESTful API. The flow of logic follows as such: users request are sent as urls which get routed through the djangos url file to its respective view that utilises serializers to return the data requested.

In line with this logic my application does the same. However, in order to show clear separation between the view code used for my home application versus the view code used for my API endpoints and RESTful functions, API view codes are stored in api.py

As such my application logic follows the Django rest framework where user request go through my urls.py file where it is redirected to its respective view code in api.py which calls related serializers from serializers.py

d) URL routing

When a user makes a request Django runs through urlpatterns till it reaches a match to the users request. Hence, in order to route the user to the right URL i added paths for each of the various endpoints i had created as well as to my web application itself..

url.py

```
# Application URLS
urlpatterns = [
    path('',views.SPA, name='index'),
    path('api/phase', api.PhaseList.as_view(), name='phase-list'),
    path('api/covid/<str:pk>', api.CovidDetails.as_view(), name='covid-row'),
]
```

e) Unit testing

Unit testing was done with the help of the factory_boy package. In order to create my unit tests I first created dummy data that I could utilise in my tests. Similar to my models I defined the dummy data utilised as foreign keys before the data that calls it as seen in the following code.

```
# unit test dummy data
class PhaseFactory(factory.django.DjangoModelFactory):
     phase_name = "Phase 1"

     class Meta:
          model = Phase

class CovidFactory(factory.django.DjangoModelFactory):
     date_id = 2023-12-25
     positive = 0
     mortality = 0
     vacc = factory.SubFactory(VaccinationFactory)
     phase = factory.SubFactory(PhaseFactory)

     class Meta:
          model = CovidStats
```

Unit tests were then written for each function I wanted to test. Mainly two types of tests were carried out, testing of my serializers and testing of the endpoints.

Serializer unit tests were written to ensure that my serializers returned the data that i desired, whilst endpoint test covered page and correct data returns.

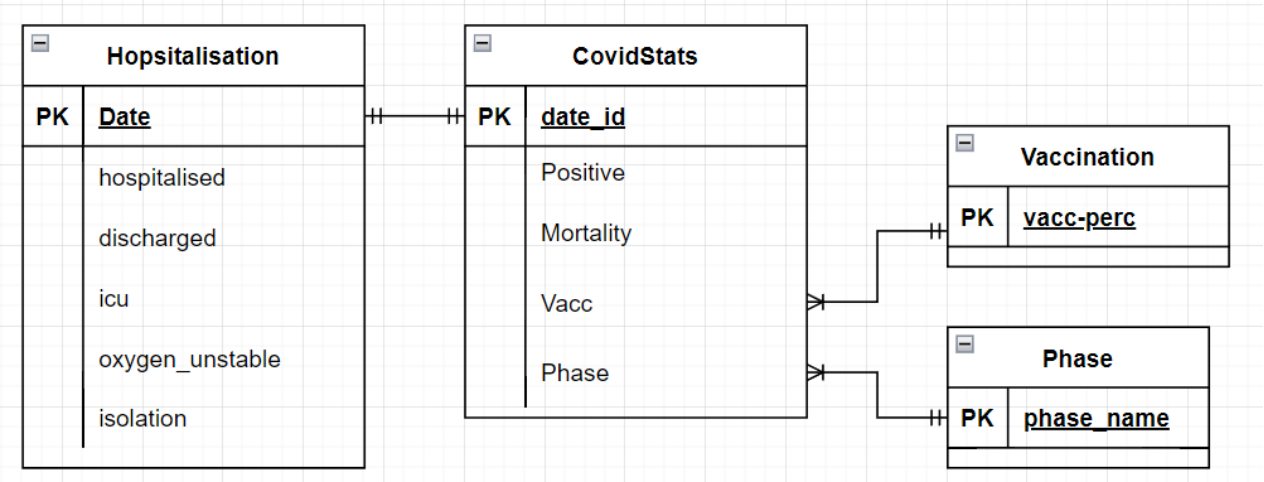
test.py

```
# serializer tests
    def test_covidSerializer(self):
        data = self.covidserializer.data
        self.assertEqual(set(data.keys()),
set({'date_id','positive','mortality','vacc','phase'}))

 # endpoint tests
    def test_covidReturnSuccess(self):
        response = self.client.get(self.covid_good_url, format='json')
        response.render()
        # ensure page is loaded
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.content) #load dummy data
        self.assertEqual(data['positive'],0)
        self.assertEqual(data['mortality'],0)
```

R2: Data model for the data

DATABASE ER DIAGRAM



As the main reason for my chosen data set was to investigate the spread of covid in Singapore based on the original data I created the following 4 tables with the relationship as shown in the above ER diagram.

The main table, CovidStats, comprises of the number of positive cases, the mortality rate due to covid, the percentage of population vaccination, and the current quarantine phase. This was because these were the most relevant statistics to my research reason. After normalisation I separated out the vaccination percentage and phases into separate tables due to the repetition.

A separate table, hospitalisation, was also then created to store statistics regarding the number of people hospitalised, discharged, in the icu, in oxygen dependent./unstable condition and in isolation. Which were statistics that I felt were interesting but not as relevant to the main research reason.

R3: Code for the required REST endpoints

In order to create better separation of logic, all code for the REST API endpoints were written in the api.py file separate from views.py which holds code for the Covid Web Application.

In order to create RESTful functions I utilised Djangos Mixin classes that utilises Django GenericAPIVIew for the core functionality whilst still allowing me to provide the other REST functions without excessive repetitive code.

api.py

```
# covid row in api
class CovidDetails(mixins.CreateModelMixin,
                   mixins.RetrieveModelMixin,
                   mixins.UpdateModelMixin,
                   mixins.DestroyModelMixin,
                   generics.GenericAPIView):
    queryset = CovidStats.objects.all()
    serializer_class = CovidSerializer

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

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

    def put(self,request,*args,**kwargs):
        return self.update(request,*args,**kwargs)

    def delete(self,request,*args,**kwargs):
        return self.destroy(request,*args,**kwargs)
```

R4: Method of bulk loading data

In order to bulk load the data into the database a populate script was written that opens the data file and reads the data into data structures which is then used to create rows with the data I wish to add to the database.

```
with open(data_file_2) as csv_file:
    reader_hospital = csv.reader(csv_file, delimiter=',')
    header = reader_hospital.__next__()
    for row in reader_hospital:
        hospitalisation[row[0]] = row[1:6]

for date, item in hospitalisation.items():
    row = Hospitalisation.objects.create(date = covid_rows[date], hospitalised =
      item[0],discharged = item[1], icu = item[2],oxygen_unstable = item[3],
      isolation = item[4])
    row.save()
```

How to unpack and run the application:

1. Download the zip file
2. Extract the zip file
3. Inside the folder create a virtual environment : python -m venv advanced_web_dev
4. Start virtual environment
5. Run the following commands:
    pip install -r requirements.txt
    pip install Django
    pip install djangorestframework
    pip install django-bootstrap4
    pip install factory_boy
6. cd midterm
7. python manage.py runserver
8. Access application at http://localhost:8000

Django-admin login instructions:

1. py manage.py runserver
2. click on the admin button or go to /admin
3. login with the following details:

username: admin
password: pass123

How to populate database:

1. cd midterm
2. py .\scripts\populate_api.py

How to run unit test:

1. cd midterm
2. py .\manage.py test

Data loading script: MID-TERM/midterm/scripts/populate_api.py

Packages and versions used:

asgiref==3.7.2
beautifulsoup4==4.12.2
bootstrap4==23.4
djangorestframework==3.14.0
factory-boy==3.3.0
Faker==22.0.0
pip==23.2.1
pytz==2023.3.post1
six==1.16.0
soupsieve==2.5
sqlparse==0.4.4
tzdata==2023.4

Development environment:
Operating system: windows 11
Python version: 3.12.0
Django version: 5.0

References

Bremmer, I. (2021, February 23). *The Best Global Responses to the COVID-19 Pandemic, 1 Year Later*. Retrieved from TIME: https://time.com/5851633/best-global-responses-covid-19/

Chua, H. X. (2022). *COVID-19 Singapore*. Retrieved from data.world: https://data.world/hxchua/covid-19-singapore

Django. (n.d.). *Models*. Retrieved from django: https://docs.djangoproject.com/en/3.0/topics/db/models/

Django. (n.d.). *Working with forms*. Retrieved from Django: https://docs.djangoproject.com/en/3.0/topics/forms/