Proj 2 Web Service
‘Plantronixs’ Plant Information System

Thomas Rhodes
459274825

Last Updated: April 17, 2022

Contents

Part A
Implementation

1 Web Service That Supports at Least Four Unique Get and Post Actions

Post actions for this application can be seen when creating a new User, Plant, Family, Genus, PlantInfo, SoilPreference, UserPlant, Edible And when signing into the admin panel.

1.1 Terminal Outputs for Post Requests:

Admin Login Post Request

```
[12/Apr/2022 23:22:23] "POST /admin/login/?next=/admin/ HTTP/1.1" 302 0
```

api_auth Login Post Request

```
[12/Apr/2022 23:21:36] "POST /api-auth/login/ HTTP/1.1" 302 0
```

Create User Post Request

```
[12/Apr/2022 23:35:09] "POST /admin/auth/user/add/ HTTP/1.1" 200 7574
```

Create genus Post Request

```
[12/Apr/2022 23:44:00] "POST /genus/ HTTP/1.1" 201 9640
```

Create Family Post Request

```
[12/Apr/2022 23:48:22] "POST /family/ HTTP/1.1" 201 9671
```

Create Plant Post Request

```
[12/Apr/2022 23:51:30] "POST /plants/ HTTP/1.1" 201 11497
```

Create PlantInfo Post Request

```
[13/Apr/2022 00:08:37] "POST /info/ HTTP/1.1" 201 13509
```

Create Soil Preference Post Request

```
[13/Apr/2022 02:46:26] "POST /soilpreference/ HTTP/1.1" 201 11164
```

Create User Plant Post Request

```
[13/Apr/2022 02:47:48] "POST /userplant/ HTTP/1.1" 201 10234
```

Create Edible Post Request

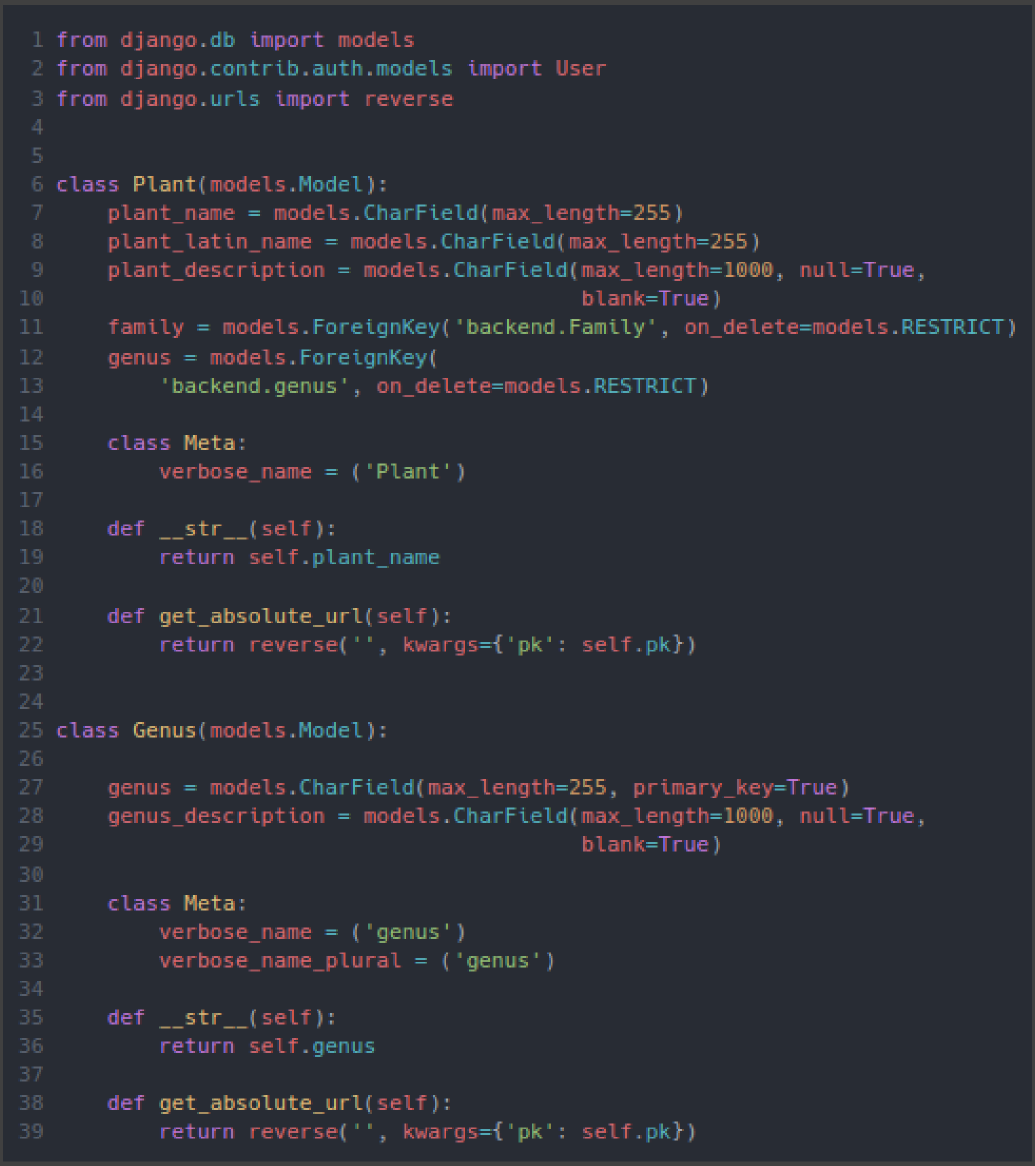
```
[13/Apr/2022 02:48:34] "POST /edible/ HTTP/1.1" 201 13914
```

2 Build a correctly formed database to implement the specifications of Proj1

2.1 Defining the Database Tables

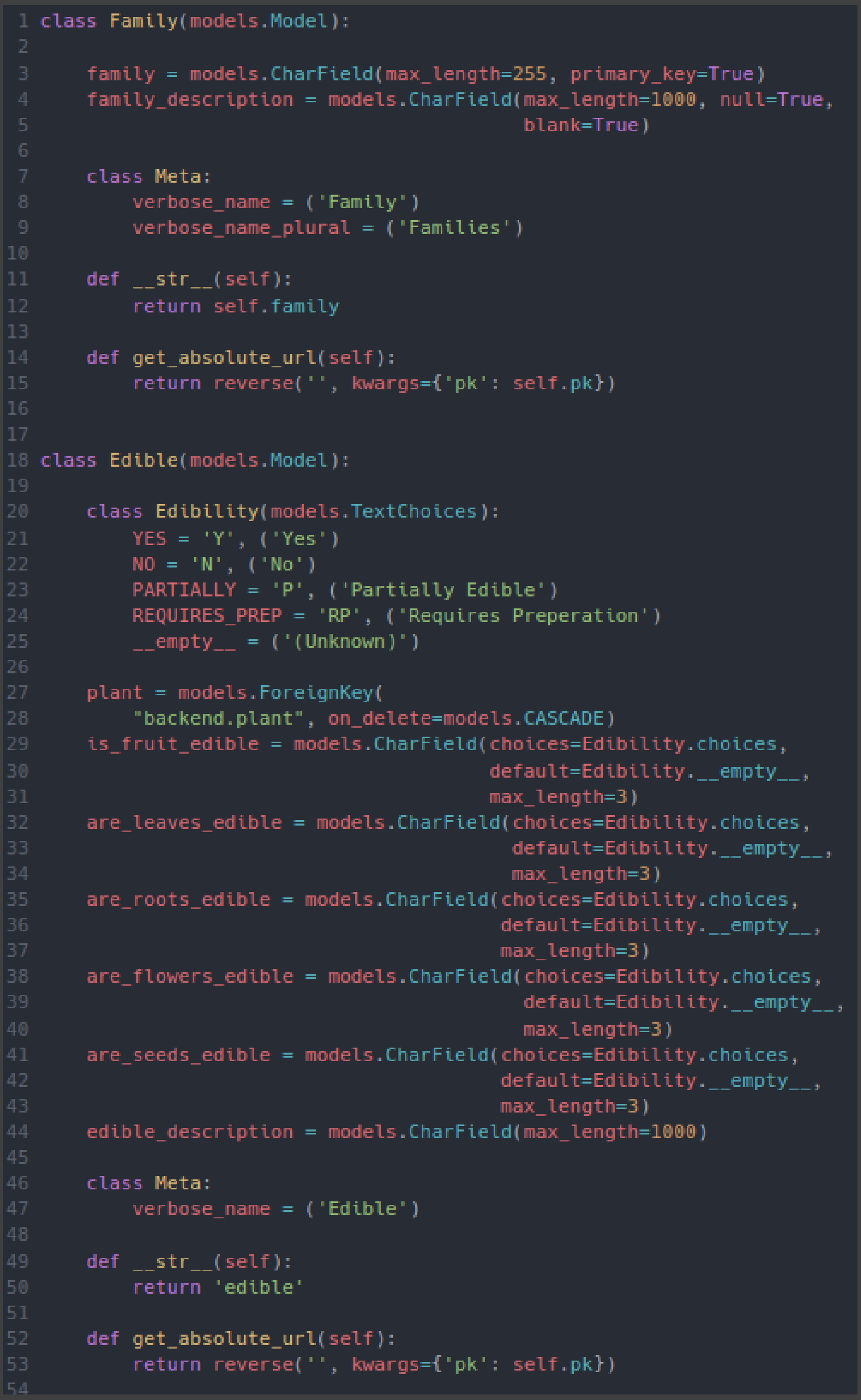
Screenshots below depict the code in 'models.py' Which is used to define the database for this project, excluding the user, groups, permissions, and ip_logging, tables which are created by Django and the ip_logging middleware respectively:

Fig. 1: Models.py 1/4



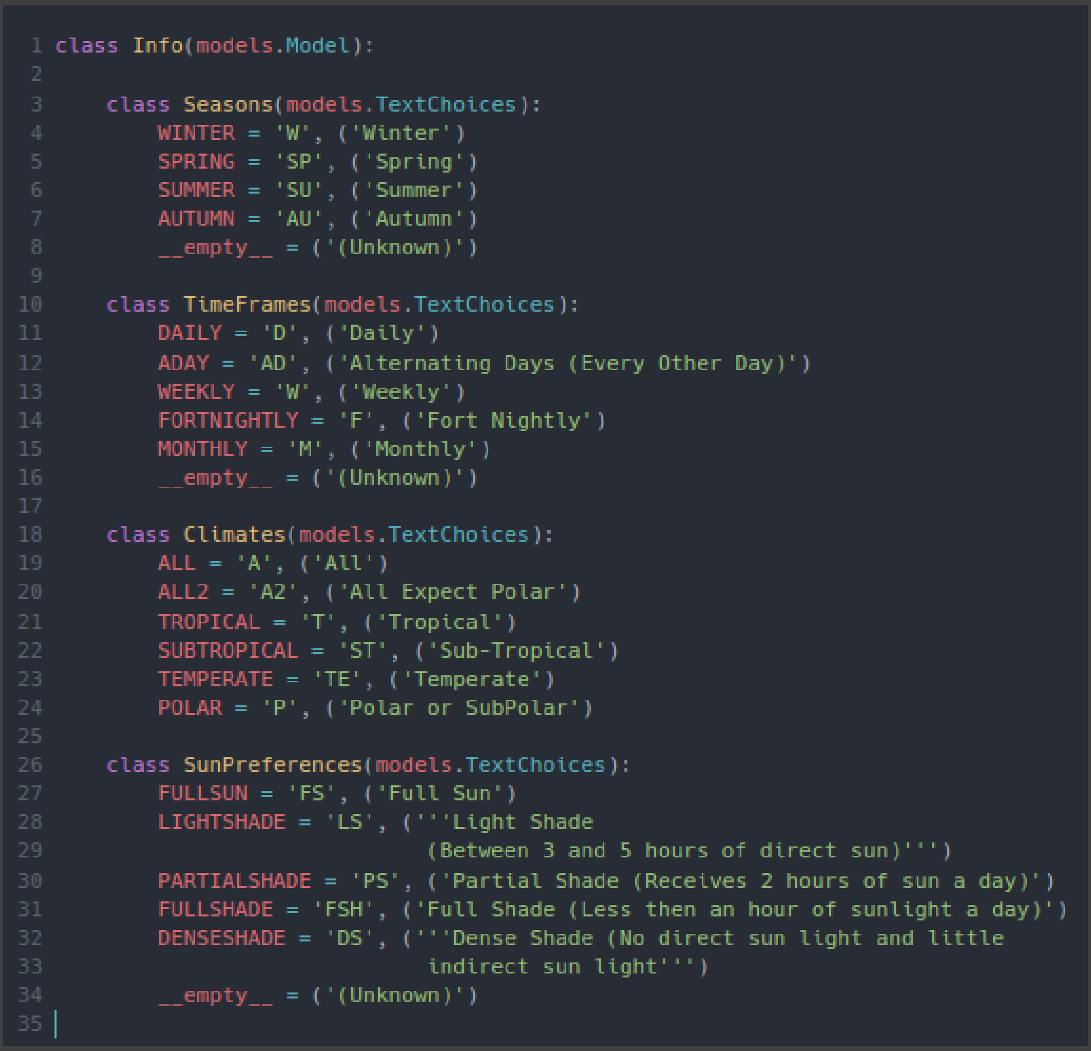
```python
from django.db import models
from django.contrib.auth.models import User
from django.urls import reverse


class Plant(models.Model):
    plant_name = models.CharField(max_length=255)
    plant_latin_name = models.CharField(max_length=255)
    plant_description = models.CharField(max_length=1000, null=True,
                                         blank=True)
    family = models.ForeignKey('backend.Family', on_delete=models.RESTRICT)
    genus = models.ForeignKey(
        'backend.genus', on_delete=models.RESTRICT)

    class Meta:
        verbose_name = ('Plant')

    def __str__(self):
        return self.plant_name

    def get_absolute_url(self):
        return reverse('', kwargs={'pk': self.pk})


class Genus(models.Model):

    genus = models.CharField(max_length=255, primary_key=True)
    genus_description = models.CharField(max_length=1000, null=True,
                                         blank=True)

    class Meta:
        verbose_name = ('genus')
        verbose_name_plural = ('genus')

    def __str__(self):
        return self.genus

    def get_absolute_url(self):
        return reverse('', kwargs={'pk': self.pk})
```

Fig. 2: Models.py 2/4



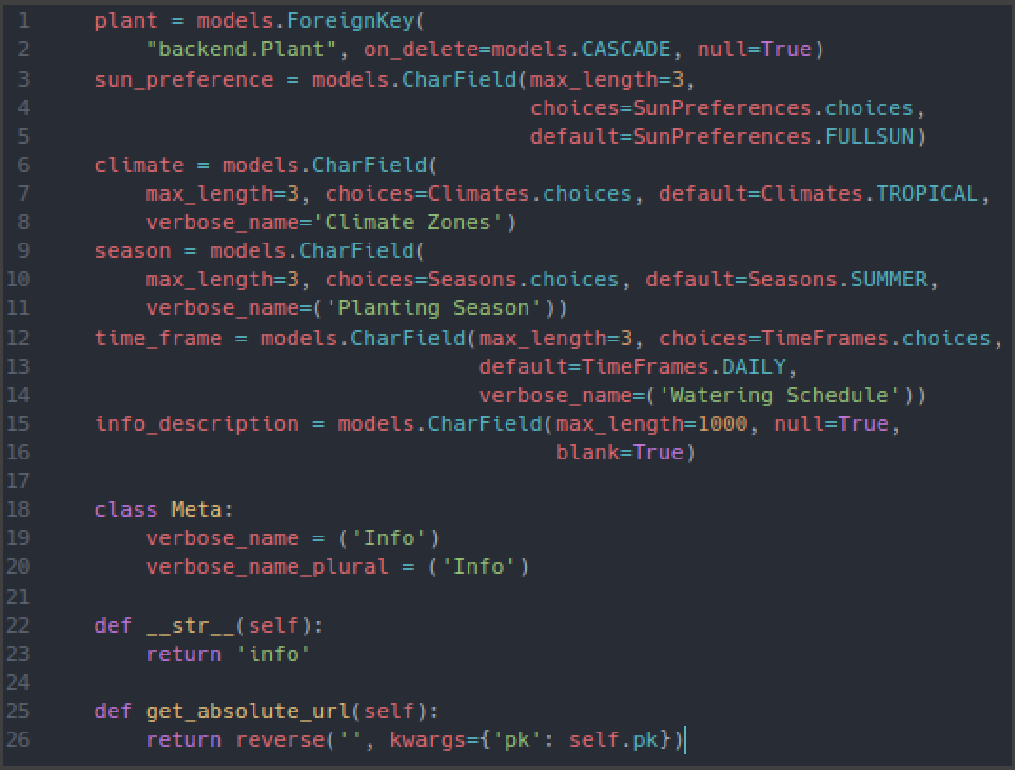
```python
class Family(models.Model):

    family = models.CharField(max_length=255, primary_key=True)
    family_description = models.CharField(max_length=1000, null=True,
                                          blank=True)

    class Meta:
        verbose_name = ('Family')
        verbose_name_plural = ('Families')

    def __str__(self):
        return self.family

    def get_absolute_url(self):
        return reverse('', kwargs={'pk': self.pk})


class Edible(models.Model):

    class Edibility(models.TextChoices):
        YES = 'Y', ('Yes')
        NO = 'N', ('No')
        PARTIALLY = 'P', ('Partially Edible')
        REQUIRES_PREP = 'RP', ('Requires Preperation')
        __empty__ = ('(Unknown)')

    plant = models.ForeignKey(
        "backend.plant", on_delete=models.CASCADE)
    is_fruit_edible = models.CharField(choices=Edibility.choices,
                                       default=Edibility.__empty__,
                                       max_length=3)
    are_leaves_edible = models.CharField(choices=Edibility.choices,
                                         default=Edibility.__empty__,
                                         max_length=3)
    are_roots_edible = models.CharField(choices=Edibility.choices,
                                        default=Edibility.__empty__,
                                        max_length=3)
    are_flowers_edible = models.CharField(choices=Edibility.choices,
                                          default=Edibility.__empty__,
                                          max_length=3)
    are_seeds_edible = models.CharField(choices=Edibility.choices,
                                        default=Edibility.__empty__,
                                        max_length=3)
    edible_description = models.CharField(max_length=1000)

    class Meta:
        verbose_name = ('Edible')

    def __str__(self):
        return 'edible'

    def get_absolute_url(self):
        return reverse('', kwargs={'pk': self.pk})
```

Fig. 3: Models.py 3/4 Part 1



```python
class Info(models.Model):

    class Seasons(models.TextChoices):
        WINTER = 'W', ('Winter')
        SPRING = 'SP', ('Spring')
        SUMMER = 'SU', ('Summer')
        AUTUMN = 'AU', ('Autumn')
        __empty__ = ('(Unknown)')

    class TimeFrames(models.TextChoices):
        DAILY = 'D', ('Daily')
        ADAY = 'AD', ('Alternating Days (Every Other Day)')
        WEEKLY = 'W', ('Weekly')
        FORTNIGHTLY = 'F', ('Fort Nightly')
        MONTHLY = 'M', ('Monthly')
        __empty__ = ('(Unknown)')

    class Climates(models.TextChoices):
        ALL = 'A', ('All')
        ALL2 = 'A2', ('All Expect Polar')
        TROPICAL = 'T', ('Tropical')
        SUBTROPICAL = 'ST', ('Sub-Tropical')
        TEMPERATE = 'TE', ('Temperate')
        POLAR = 'P', ('Polar or SubPolar')

    class SunPreferences(models.TextChoices):
        FULLSUN = 'FS', ('Full Sun')
        LIGHTSHADE = 'LS', ('''Light Shade
                             (Between 3 and 5 hours of direct sun)''')
        PARTIALSHADE = 'PS', ('Partial Shade (Receives 2 hours of sun a day)')
        FULLSHADE = 'FSH', ('Full Shade (Less then an hour of sunlight a day)')
        DENSESHADE = 'DS', ('''Dense Shade (No direct sun light and little
                             indirect sun light''')
        __empty__ = ('(Unknown)')

```

Fig. 4: Models.py 3/4 Part 2



```
    plant = models.ForeignKey(
        "backend.Plant", on_delete=models.CASCADE, null=True)
    sun_preference = models.CharField(max_length=3,
                                      choices=SunPreferences.choices,
                                      default=SunPreferences.FULLSUN)
    climate = models.CharField(
        max_length=3, choices=Climates.choices, default=Climates.TROPICAL,
        verbose_name='Climate Zones')
    season = models.CharField(
        max_length=3, choices=Seasons.choices, default=Seasons.SUMMER,
        verbose_name=('Planting Season'))
    time_frame = models.CharField(max_length=3, choices=TimeFrames.choices,
                                  default=TimeFrames.DAILY,
                                  verbose_name=('Watering Schedule'))
    info_description = models.CharField(max_length=1000, null=True,
                                        blank=True)

    class Meta:
        verbose_name = ('Info')
        verbose_name_plural = ('Info')

    def __str__(self):
        return 'info'

    def get_absolute_url(self):
        return reverse('', kwargs={'pk': self.pk})
```

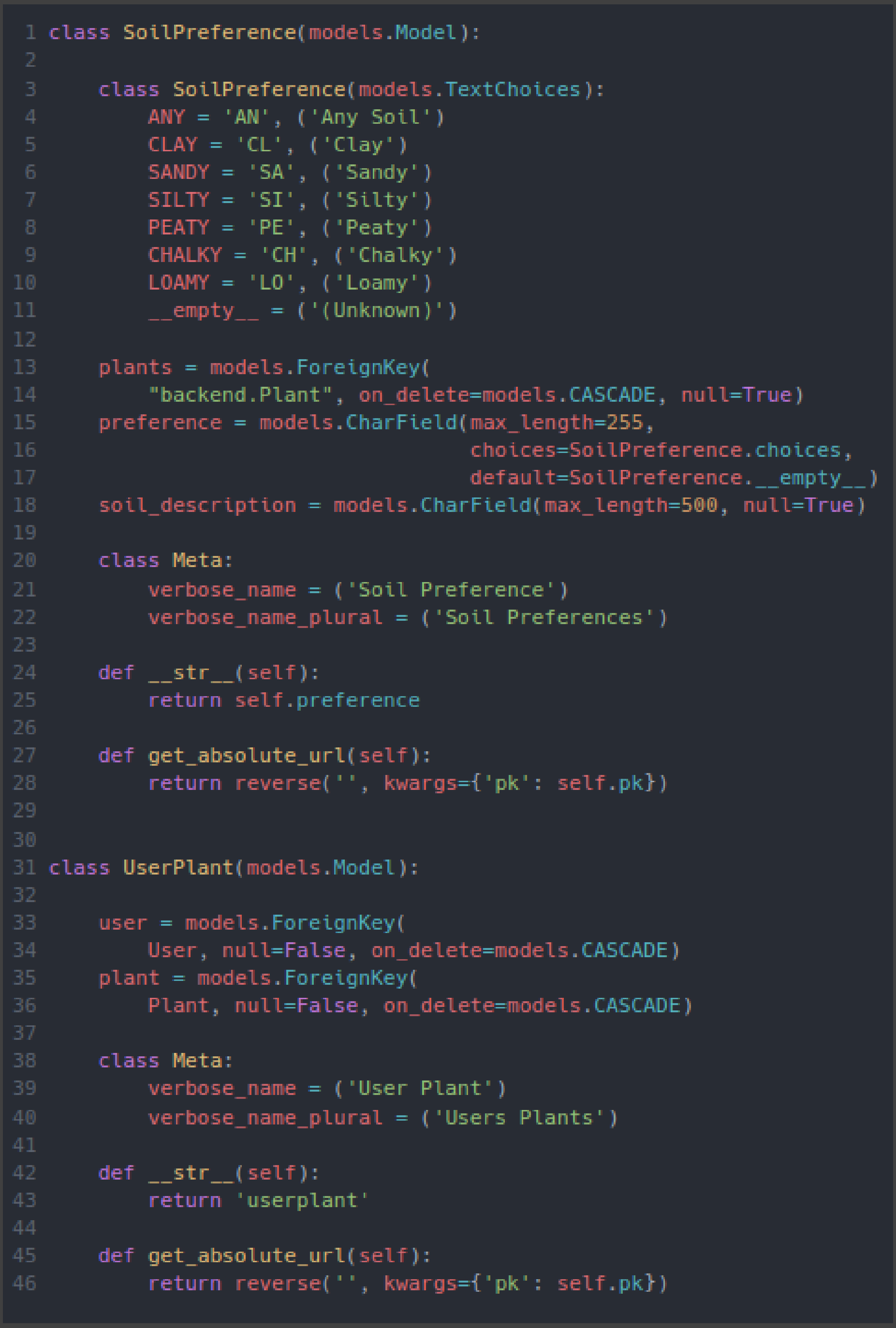


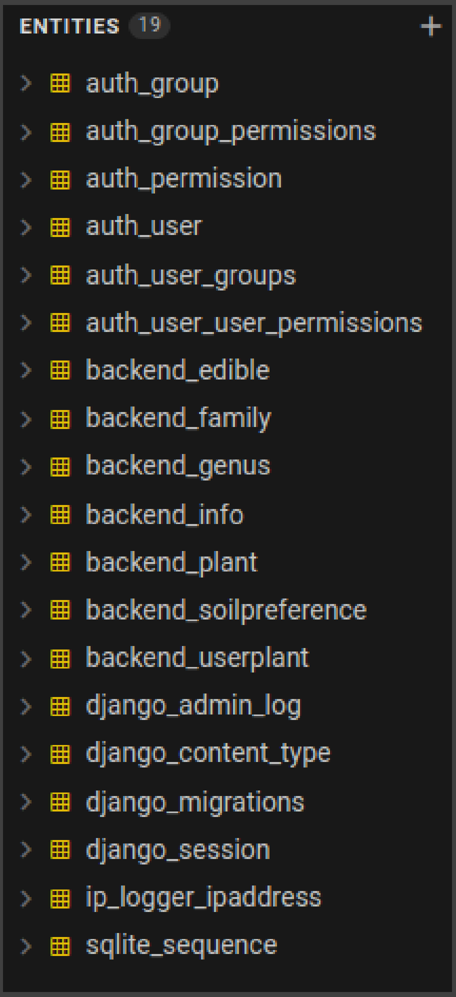
Fig. 5: Models.py 4/4

```python
class SoilPreference(models.Model):

    class SoilPreference(models.TextChoices):
        ANY = 'AN', ('Any Soil')
        CLAY = 'CL', ('Clay')
        SANDY = 'SA', ('Sandy')
        SILTY = 'SI', ('Silty')
        PEATY = 'PE', ('Peaty')
        CHALKY = 'CH', ('Chalky')
        LOAMY = 'LO', ('Loamy')
        __empty__ = ('(Unknown)')

    plants = models.ForeignKey(
        "backend.Plant", on_delete=models.CASCADE, null=True)
    preference = models.CharField(max_length=255,
                                  choices=SoilPreference.choices,
                                  default=SoilPreference.__empty__)
    soil_description = models.CharField(max_length=500, null=True)

    class Meta:
        verbose_name = ('Soil Preference')
        verbose_name_plural = ('Soil Preferences')

    def __str__(self):
        return self.preference

    def get_absolute_url(self):
        return reverse('', kwargs={'pk': self.pk})


class UserPlant(models.Model):

    user = models.ForeignKey(
        User, null=False, on_delete=models.CASCADE)
    plant = models.ForeignKey(
        Plant, null=False, on_delete=models.CASCADE)

    class Meta:
        verbose_name = ('User Plant')
        verbose_name_plural = ('Users Plants')

    def __str__(self):
        return 'userplant'

    def get_absolute_url(self):
        return reverse('', kwargs={'pk': self.pk})
```

2.2 Generated Database Tables

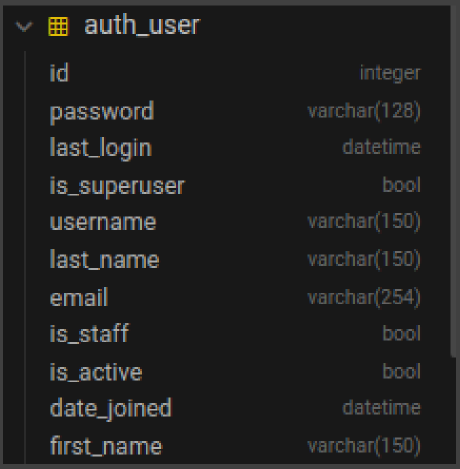
Using beekeeper studio, or with the use of the VSCode Extension 'SQLite', we are able to see all of the generated tables in the database including, all which are automatically created by django. Screenshots provided below:

Fig. 6: Table List and User_Auth Table:
8 of these tables are of relevancy to the specifications outlined in Proj1, These include auth_user, backend_edible, backend_family, backend_genus, backend_info, backend_plant, backend_soilpreference, backend_userplant, as shown in the right image of Fig. 6, and Fig. 7, Fig. 8, and Fig. 9 Below



ENTITIES 19

auth_group
auth_group_permissions
auth_permission
auth_user
auth_user_groups
auth_user_user_permissions
backend_edible
backend_family
backend_genus
backend_info
backend_plant
backend_soilpreference
backend_userplant
django_admin_log
django_content_type
django_migrations
django_session
ip_logger_ipaddress
sqlite_sequence



auth_user

id	integer
password	varchar(128)
last_login	datetime
is_superuser	bool
username	varchar(150)
last_name	varchar(150)
email	varchar(254)
is_staff	bool
is_active	bool
date_joined	datetime
first_name	varchar(150)

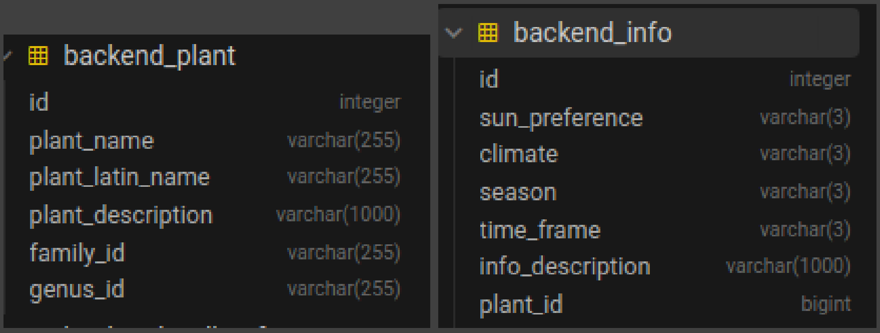


Fig. 7: Plant and Info Tables

backend_plant	
id	integer
plant_name	varchar(255)
plant_latin_name	varchar(255)
plant_description	varchar(1000)
family_id	varchar(255)
genus_id	varchar(255)

backend_info	
id	integer
sun_preference	varchar(3)
climate	varchar(3)
season	varchar(3)
time_frame	varchar(3)
info_description	varchar(1000)
plant_id	bigint

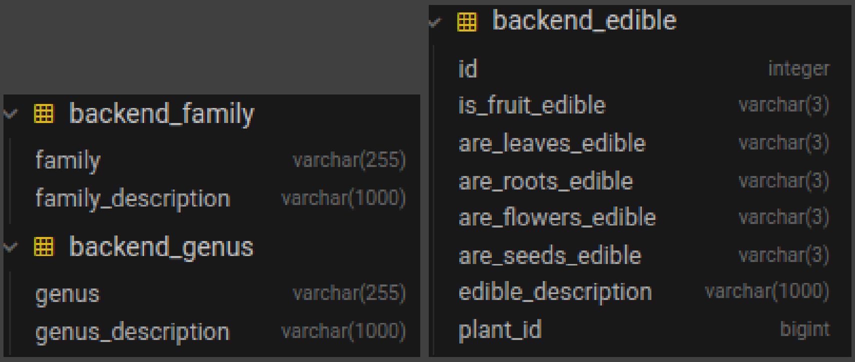


Fig. 8: Family, Genus and Edible Tables

backend_family	
family	varchar(255)
family_description	varchar(1000)

backend_genus	
genus	varchar(255)
genus_description	varchar(1000)

backend_edible	
id	integer
is_fruit_edible	varchar(3)
are_leaves_edible	varchar(3)
are_roots_edible	varchar(3)
are_flowers_edible	varchar(3)
are_seeds_edible	varchar(3)
edible_description	varchar(1000)
plant_id	bigint

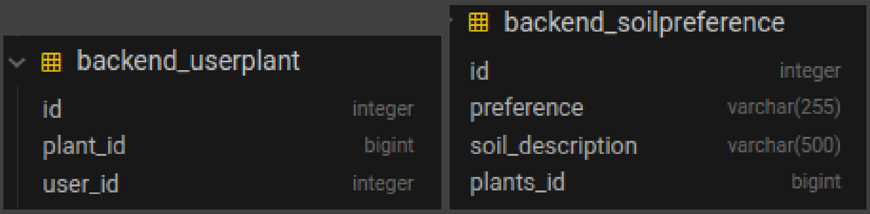


Fig. 9: Soil Preference and User Plant

backend_userplant	
id	integer
plant_id	bigint
user_id	integer

backend_soilpreference	
id	integer
preference	varchar(255)
soil_description	varchar(500)
plants_id	bigint

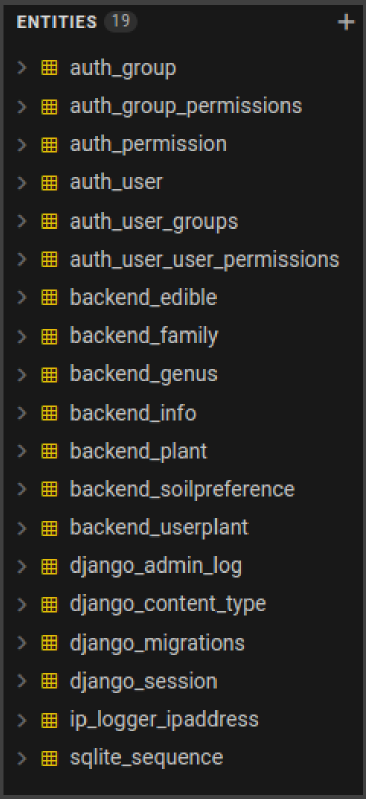
2.3 Integrating with a Back-end Database

Django comes pre setup with sqlite3, a lightweight portable relational database. Django has its own Object Relationship Mapper (ORM). Only after the following can a developer make or showcase any CRUD actions. once a developer has defined their models in the models.py, they need to run `python manage.py makemigrations`, to generate the migration files, Which are Django's way of propagating changes made to your models. After Django has finished making the migration files (Which are stored in the appfolder/migrations/ directory, which in our case is backend/migrations/) , you'll need to run `python manage.py migrate`, which creates the necessary tables or changes to your models in the database.

Apart from the code connected with a developer's own custom object as contained in their 'models.py' code, CRUD in Django is handled by the framework without explicit direction from the developer, thus we can test this right away. The only need at this point is that we are logged in as a user using Django's default authentication settings and that the application is functioning.

As mention above Django by default ships with sqlite3 preconfigured, which means our will live in our project folder for the time being. which is in 'Plantronics/b.sqlite3'

We can sign into the admin panel and add a 'Plant,' then delete that 'Plant,' to provide two examples of CRUD. To do this step by step, log into the admin panel and follow the steps in :



```
ENTITIES 19
auth_group
auth_group_permissions
auth_permission
auth_user
auth_user_groups
auth_user_user_permissions
backend_edible
backend_family
backend_genus
backend_info
backend_plant
backend_soilpreference
backend_userplant
django_admin_log
django_content_type
django_migrations
django_session
ip_logger_ipaddress
sqlite_sequence
```

Fig. 10: For clarity here is a Screenshot of the integrated backend database again as shown originally in Fig. 6

Fig. 11: Once logged in you should be presented by this screen, select the +add option next to plants in the backend list

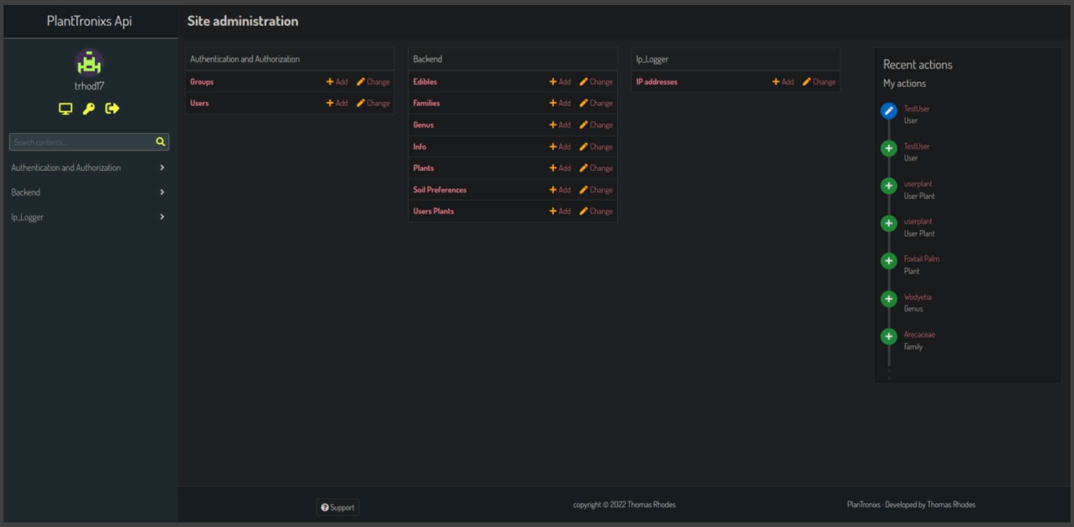
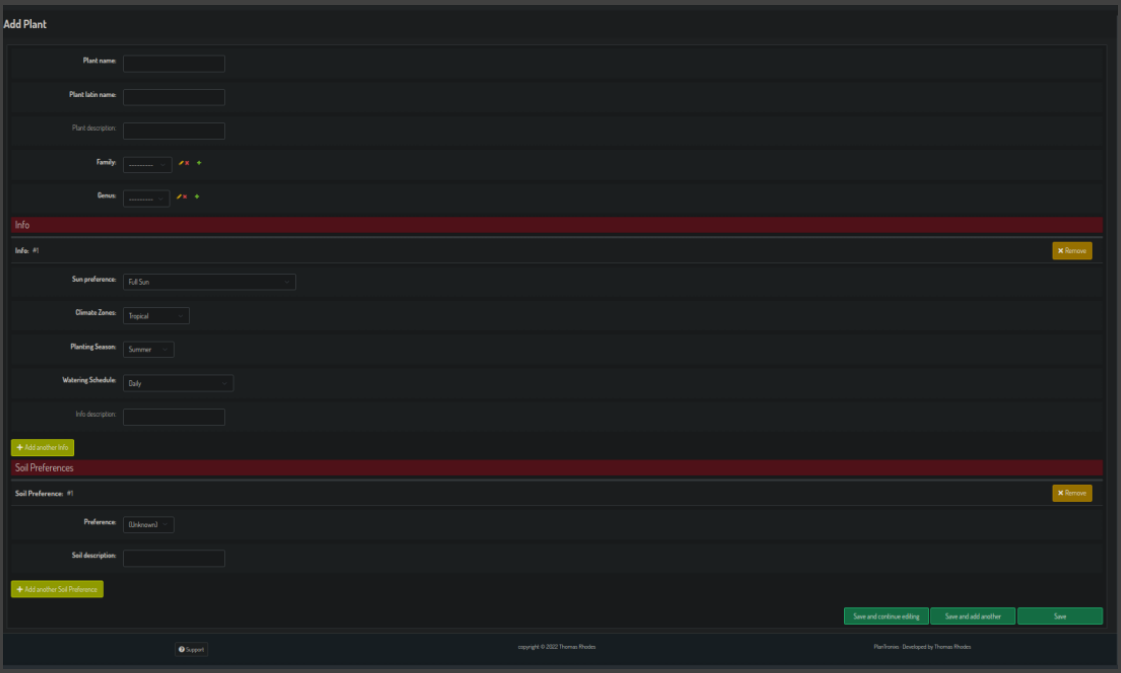


Fig. 12: You'll be presented by a screen that looks like this, click the two buttons label 'X Remove', we wont be needing those sections. fill out the Plant name, Plant Latin Name, Plant Description (can be left empty), select a family and genus, then select save at the bottom of the screen. Example of filled out form in Fig. 13 below.


Add Plant

Plant name:
Plant latin name:
Plant description:
Family: ---------
Genus: ---------

Info

Info: #1

Sun preference: Full Sun
Climate Zones: Tropical
Planting Season: Summer
Watering Schedule: Daily
Info description:

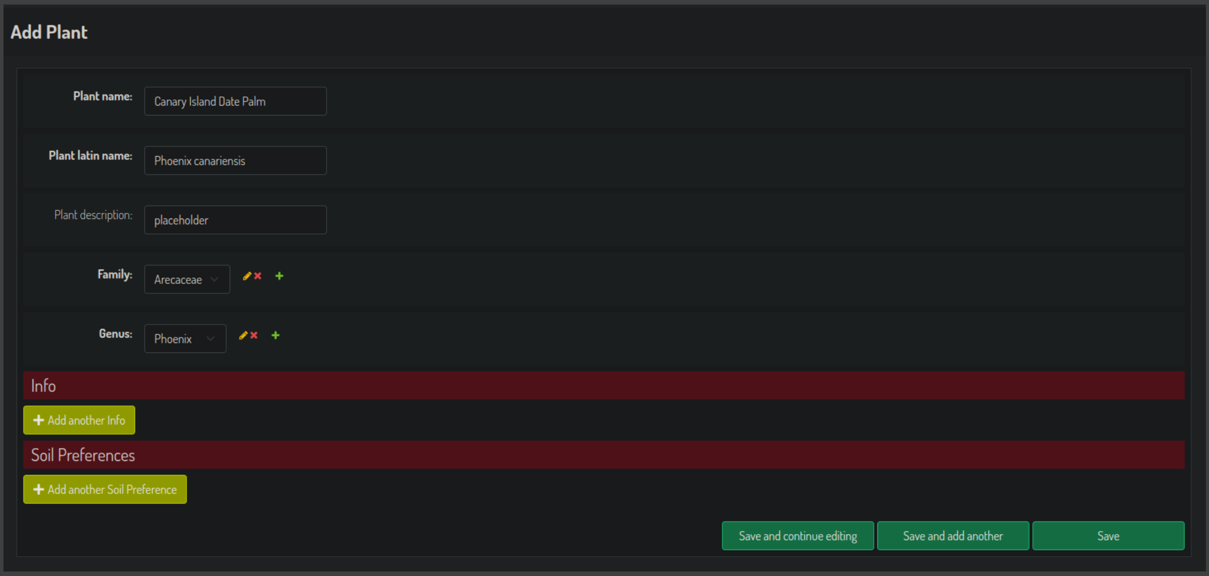
Soil Preferences

Soil Preference: #1

Preference: (Unknown)
Soil description:

✖ Remove
+ Add another Info
✖ Remove
+ Add another Soil Preference
Save and continue editing
Save and add another
Save
Support
copyright © 2022 Thomas Rhodes
PlanTronixs · Developed by Thomas Rhodes

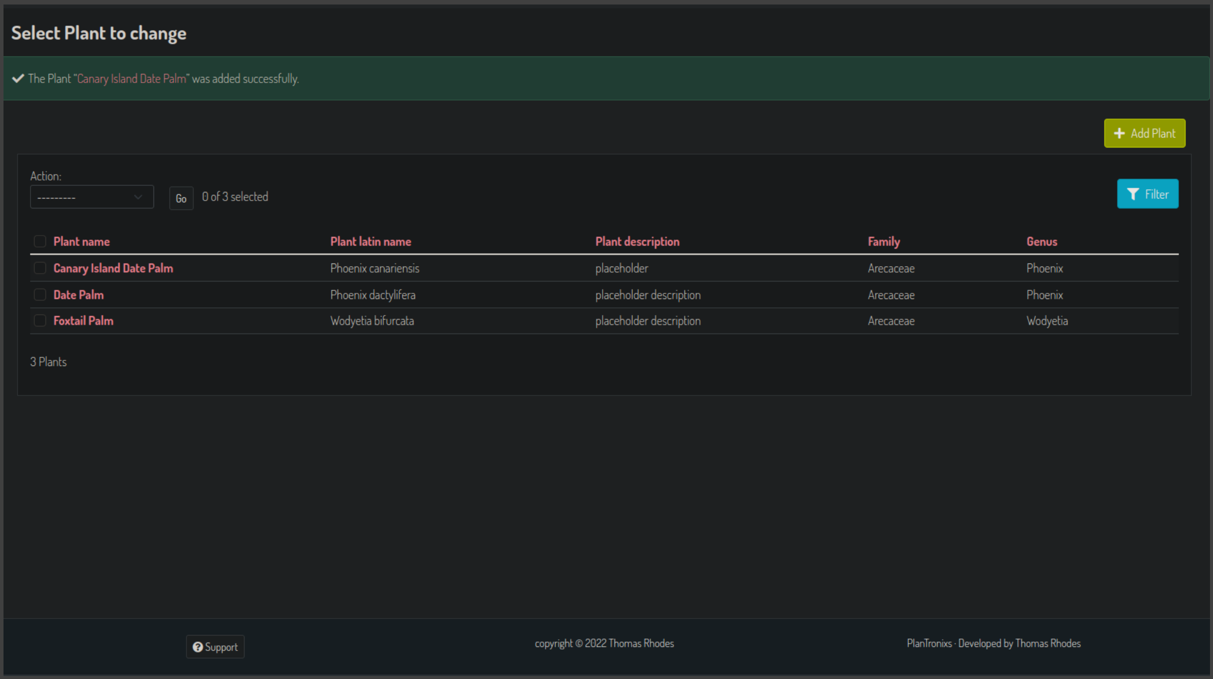
Fig. 13: Example of filled out form



Add Plant

Plant name:	Canary Island Date Palm
Plant latin name:	Phoenix canariensis
Plant description:	placeholder
Family:	Arecaceae
Genus:	Phoenix

Fig. 14: If it was created successfully it should take you to the following page, Select the plant you created. for this example the plant we created was called the Canary Island Date Palm. Go to Fig. 15



Select Plant to change

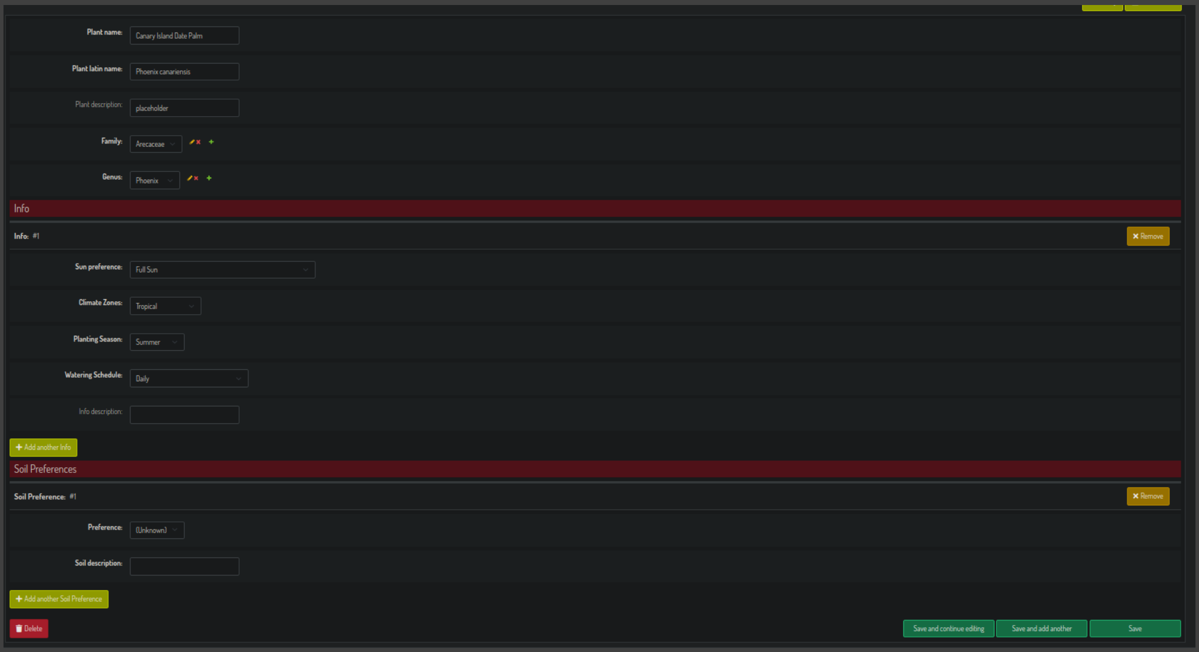
✓ The Plant "Canary Island Date Palm" was added successfully.

Plant name	Plant latin name	Plant description	Family	Genus
Canary Island Date Palm	Phoenix canariensis	placeholder	Arecaceae	Phoenix
Date Palm	Phoenix dactylifera	placeholder description	Arecaceae	Phoenix
Foxtail Palm	Wodyetia bifurcata	placeholder description	Arecaceae	Wodyetia

3 Plants

Support
copyright © 2022 Thomas Rhodes
PlanTronixs - Developed by Thomas Rhodes

Fig. 15: It should take you to a page like this were you can update, add info or soil preferences, or delete it. In this instance we are going to delete it so click on the delete button in the bottom left corner.



Plant name:	Canary Island Date Palm
Plant latin name:	Phoenix canariensis
Plant description:	placeholder
Family:	Arecaceae
Genus:	Phoenix

Info

Info: #1

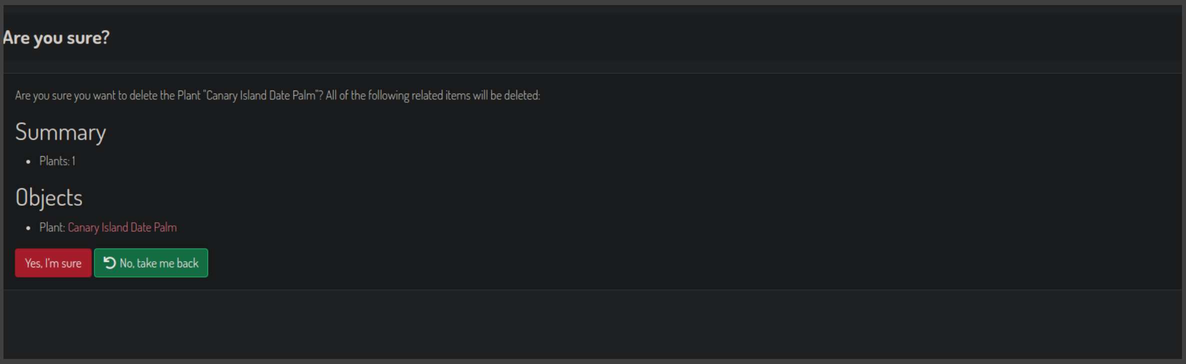
Sun preference:	Full Sun
Climate Zones:	Tropical
Planting Season:	Summer
Watering Schedule:	Daily
Info description:	

Soil Preferences

Soil Preference: #1

Preference:	(Unknown)
Soil description:	

Fig. 16: It will present you with the following confirmation. Press 'Yes, I'm Sure' to delete it or 'No, Take me back' to cancel this action.



Are you sure?

Are you sure you want to delete the Plant "Canary Island Date Palm"? All of the following related items will be deleted:

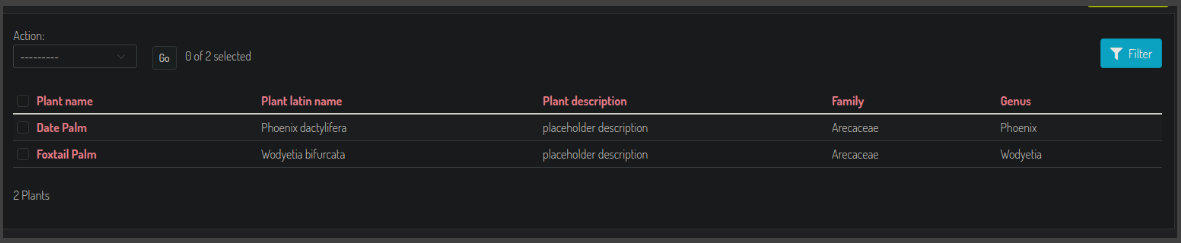
Summary

- Plants: 1

Objects

- Plant: Canary Island Date Palm

Fig. 17: Once its been deleted you'll be taken back to this page and the entry should be deleted successfully.



Action: --------- Go 0 of 2 selected

Plant name	Plant latin name	Plant description	Family	Genus
Date Palm	Phoenix dactylifera	placeholder description	Arecaceae	Phoenix
Foxtail Palm	Wodyetia bifurcata	placeholder description	Arecaceae	Wodyetia

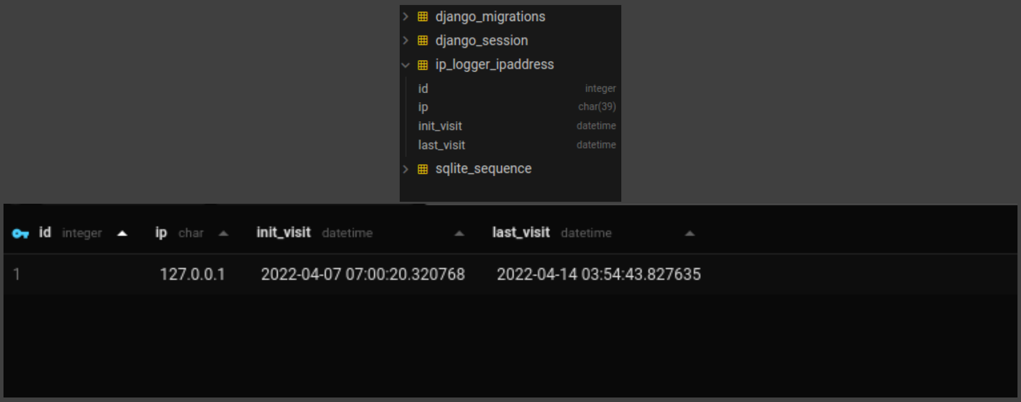
2 Plants

And thus concludes the integration of a backend database and showcasing 2 crud actions working on said database.

3 Logging Feature That Records IP, Session, Username, Usertype, Date, and Action for Each Request.

Logging IP addresses in Django is as easy as running `pip install django-ip-logger` and inserting 'ip_logger' into the installed_apps list and 'ip_logger.middleware.LogIPMiddleware' to the middleware list in plantronics/settings.py then creating and running migrations. This plug-in creates a table in the database called 'ip_logger_ipaddress' with four fields: id, ip, init_visit and last_visit.

Fig. 18: Database Table for IP Logging.



```
django_migrations
django_session
ip_logger_ipaddress
  id            integer
  ip            char(39)
  init_visit    datetime
  last_visit    datetime
sqlite_sequence
```

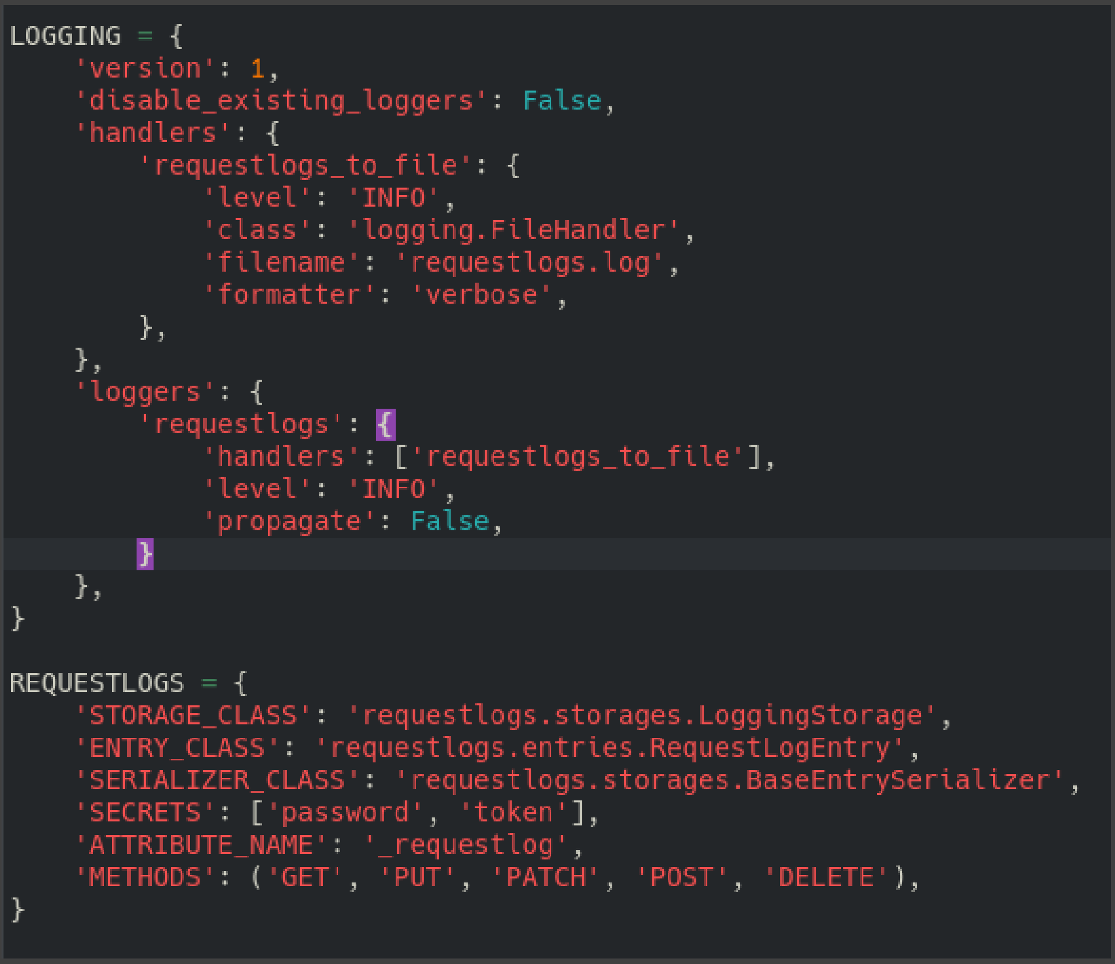
id integer	ip char	init_visit datetime	last_visit datetime
1	127.0.0.1	2022-04-07 07:00:20.320768	2022-04-14 03:54:43.827635

The justification for logging ip addresses this way as opposed to logging to the console is the persistent nature of databases and deemed necessary to have a separate table for IPs to quickly list all addresses that have made connections to the application. For more information please see the about the package django-ip-logger.

Out of the box, Django logs all requests to the console, for example when directing the browser to http://127.0.0.1:8000/plants/ while the server is running it will log `[14/Apr/2022 15:13:50] "GET /plants/ HTTP/1.1" 200 12345`
Django by default logs the Request/Action type, a date-time stamp, and the http status code. The session, username, and user type are not logged, to do this requires some configuration on our part in the settings.py file.

Starting at line 82 and finishing at roughly line 145 in 'plantronics/settings.py', is the configuration required to log out the remaining requirements as shown in the screenshot below.

Fig. 19: Logging Config



```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'requestlogs_to_file': {
            'level': 'INFO',
            'class': 'logging.FileHandler',
            'filename': 'requestlogs.log',
            'formatter': 'verbose',
        },
    },
    'loggers': {
        'requestlogs': {
            'handlers': ['requestlogs_to_file'],
            'level': 'INFO',
            'propagate': False,
        }
    },
}

REQUESTLOGS = {
    'STORAGE_CLASS': 'requestlogs.storages.LoggingStorage',
    'ENTRY_CLASS': 'requestlogs.entries.RequestLogEntry',
    'SERIALIZER_CLASS': 'requestlogs.storages.BaseEntrySerializer',
    'SECRETS': ['password', 'token'],
    'ATTRIBUTE_NAME': '_requestlog',
    'METHODS': ('GET', 'PUT', 'PATCH', 'POST', 'DELETE'),
}
```

A logger in Django is comprised of the logger itself, a handler which decides where to send the data e.g to the console or to a file, filters which further decide which data is to be sent and what is to be kept secret, and formatters which format how the output looks.

The logger in this instance is setup to write the logs to a .log file, each log in the file will look something like below:

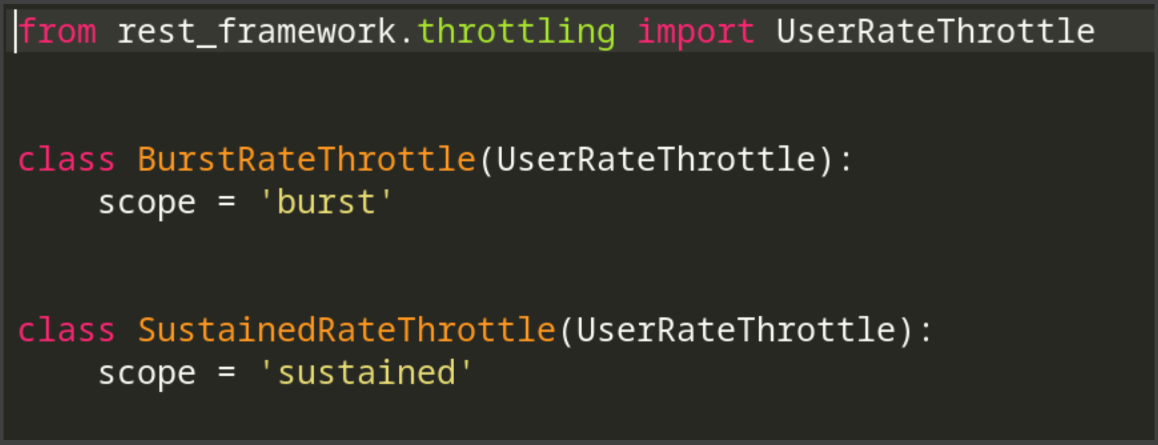
```
requestlogs INFO 2022-04-05 23:39:17,072 storages {'action_name':
None, 'execution_time': '00:00:00.005267', 'timestamp':
'2022-04-05T23:39:17.072009Z', 'ip_address': None, 'request':
OrderedDict([('method', 'GET'), ('full_path',
'/soilpreference/'), ('data', '{}'), ('query_params',
'{}'), ('request_headers', '{"HTTP_COOKIE":
"sessionid=dxur9ryb5i1lqylbla9wf9mqkhzh02w4"}')]),
'response': OrderedDict([('status_code', 200), ('data',
'[]')]), 'user': OrderedDict([('id', 2), ('username',
'test-admin')])}
```

Which is only missing the user-type, this is because Django identifies the user-type for each request automatically by comparing 'sessionid' values sent in the header of each request. The 'sessionid' is a value automatically given to logged in users by Django, which it compares against currently existing 'sessionids' to determine if the session is still valid and what permissions the user has. however the user id is logged which means you can search through the users for that id and to find the users type, is_staff or is_superuser or permissions/groups.

4 Rate Limit Session Requests to 1000 per 24 hours per session

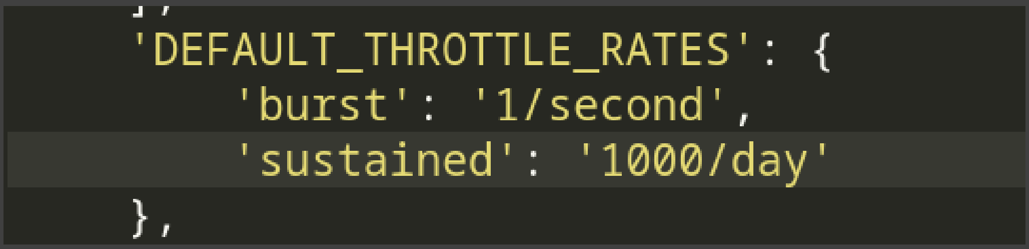
In most cases rate limiting in Django is as simple as running `pip install django-ratelimit` and configuring it, but in our case because we are using Django rest framework, which has rate limiting built in, there are two simple steps required to set it up

Fig. 20: We Create a file called 'throttles.py' and add the following code



```python
from rest_framework.throttling import UserRateThrottle


class BurstRateThrottle(UserRateThrottle):
    scope = 'burst'


class SustainedRateThrottle(UserRateThrottle):
    scope = 'sustained'
```

Fig. 21: Then in ‘settings.py’ we add the following to the Django rest framework config



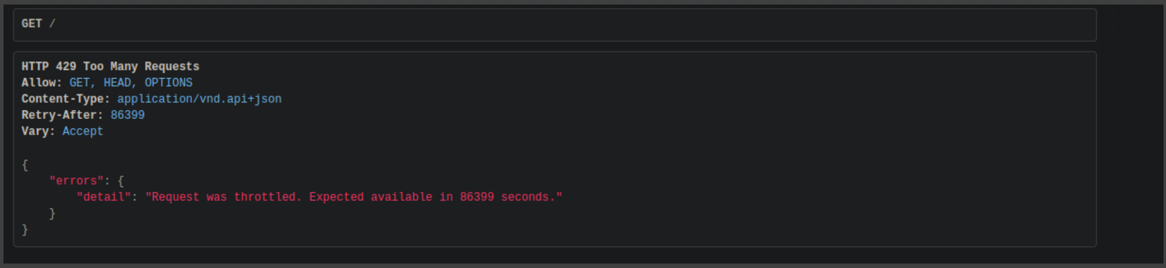
```
    ],
    'DEFAULT_THROTTLE_RATES': {
        'burst': '1/second',
        'sustained': '1000/day'
    },
```

To make sure its works, we can test by launching the server and reloading any page as fast as possible, in the console you should see messages like this

```
Too Many Requests: /
[16/Apr/2022 02:16:57] "GET / HTTP/1.1" 429 5413
```

and in the sake of our sanity when it comes to testing the sustained rate limiting we will change it to 5 per day rather than 1000 and you should see a message similar to the one above and get a message on the webpage similar to below:

Fig. 22: Rate Limited

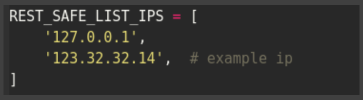


```
GET /

HTTP 429 Too Many Requests
Allow: GET, HEAD, OPTIONS
Content-Type: application/vnd.api+json
Retry-After: 86399
Vary: Accept

{
    "errors": {
        "detail": "Request was throttled. Expected available in 86399 seconds."
    }
}
```

5 Domain Locking Web Service to Whitelisted Referrers

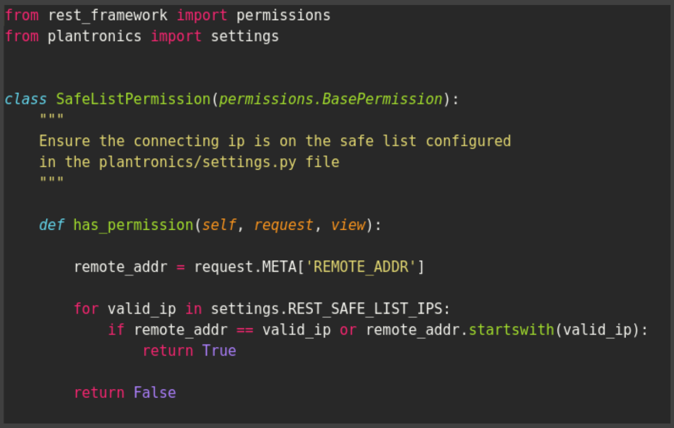
To enable and setup whitelisting in Django requires the following steps:

Fig. 23: First create a list in settings.py called REST_SAFE_LIST_IPS and add whatever ips you want to it:



```
REST_SAFE_LIST_IPS = [
    '127.0.0.1',
    '123.32.32.14',  # example ip
]
```

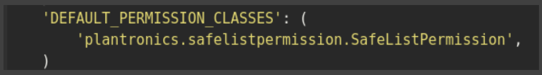
Fig. 24: In the same directory as the settings.py file create a new file called safelistpermissions.py and add in the following:



```
from rest_framework import permissions
from plantronics import settings


class SafeListPermission(permissions.BasePermission):
    """
    Ensure the connecting ip is on the safe list configured
    in the plantronics/settings.py file
    """

    def has_permission(self, request, view):

        remote_addr = request.META['REMOTE_ADDR']

        for valid_ip in settings.REST_SAFE_LIST_IPS:
            if remote_addr == valid_ip or remote_addr.startswith(valid_ip):
                return True

        return False
```

The first two lines import the Django Rest Frameworks permission function and import the settings.py file, this needs to be done provide rest_framework_permissions.BasePermission a custom subclass.

Fig. 25: At the bottom of the Django rest framework array in settings.py add in the following:

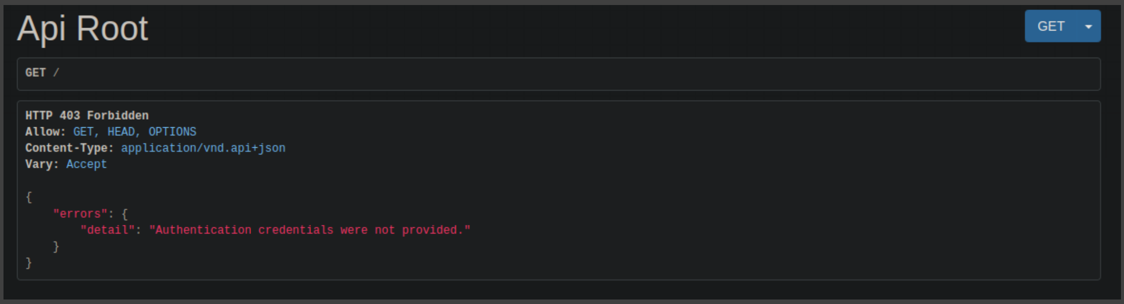


```
    'DEFAULT_PERMISSION_CLASSES': (
        'plantronics.safelistpermission.SafeListPermission',
    )
```

SafelistPermission is used to allow certain IPs defined in the settings.py::REST_SAFE_LIST_IPS access to the service.

If we comment out all of the IPs in REST_SAFE_LIST_IPS and try to connect to the web service well get the following message:

Fig. 26: It will force a false value and deny you access to the service



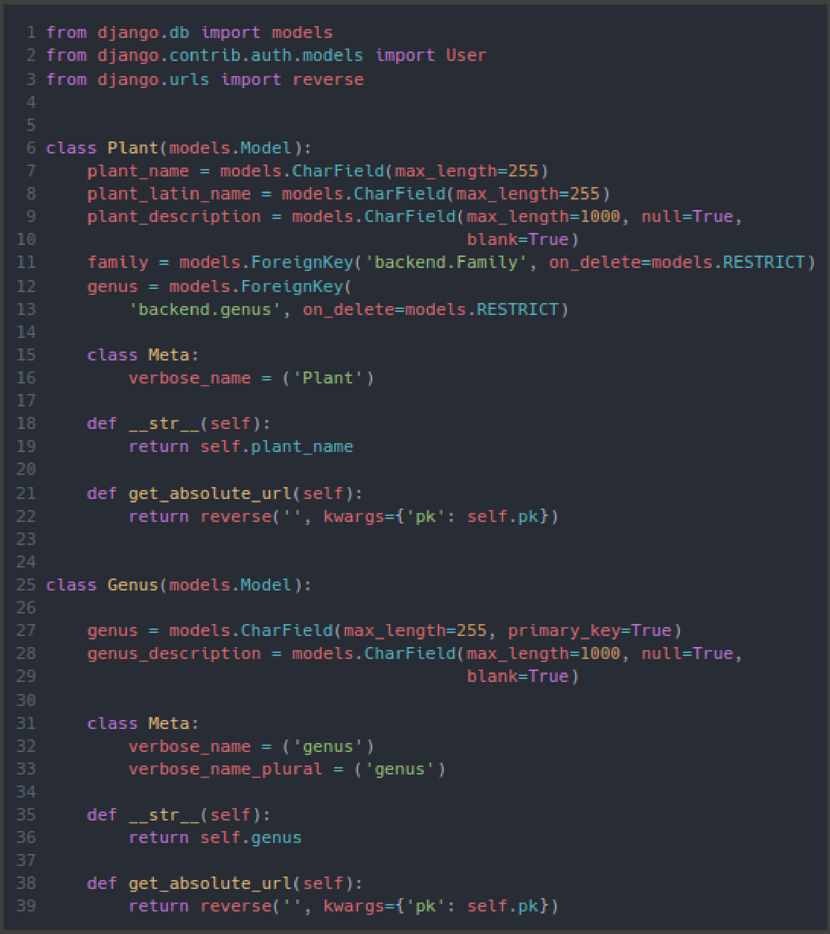
```
GET /

HTTP 403 Forbidden
Allow: GET, HEAD, OPTIONS
Content-Type: application/vnd.api+json
Vary: Accept

{
    "errors": {
        "detail": "Authentication credentials were not provided."
    }
}
```

Part B
Code Structure

6 Custom Written Database Option

In Django user defined entities/database objects are created in the ‘models.py’ file, Django uses a custom ORM for writing these objects, which comes preconfigured to translate what we see below into sqlite3 which is sent to and run on the database to setup or update the database tables/rows/columns/etc.

Fig. 27: models.py

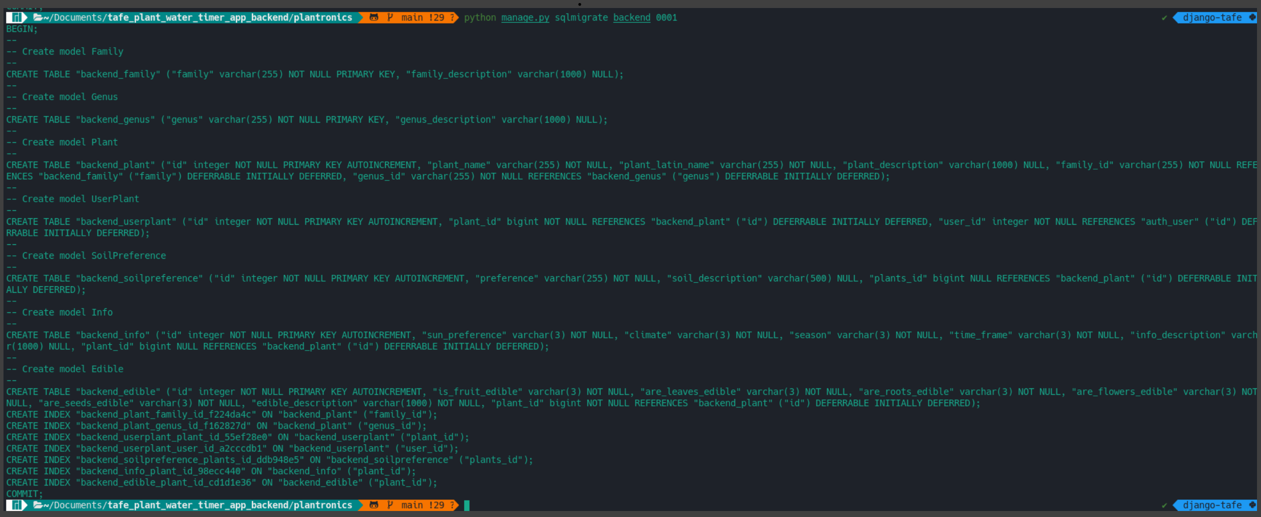


```python
from django.db import models
from django.contrib.auth.models import User
from django.urls import reverse


class Plant(models.Model):
    plant_name = models.CharField(max_length=255)
    plant_latin_name = models.CharField(max_length=255)
    plant_description = models.CharField(max_length=1000, null=True,
                                         blank=True)
    family = models.ForeignKey('backend.Family', on_delete=models.RESTRICT)
    genus = models.ForeignKey(
        'backend.genus', on_delete=models.RESTRICT)

    class Meta:
        verbose_name = ('Plant')

    def __str__(self):
        return self.plant_name

    def get_absolute_url(self):
        return reverse('', kwargs={'pk': self.pk})


class Genus(models.Model):

    genus = models.CharField(max_length=255, primary_key=True)
    genus_description = models.CharField(max_length=1000, null=True,
                                         blank=True)

    class Meta:
        verbose_name = ('genus')
        verbose_name_plural = ('genus')

    def __str__(self):
        return self.genus

    def get_absolute_url(self):
        return reverse('', kwargs={'pk': self.pk})
```

Django also has official support for PostgreSQL, MariaDB, MySQL, Oracle SQL, and Django also has many third party backends or engines as there sometimes called for using other databases.

After saving our models.py and running `python manage.py makemigrations` which creates a migration file with the changes the need to be made to either create or update the database, then `python manage.py migrate`, which creates and updates all the required tables/columns/etc. in your database.

Fig. 28: As shown here is the associated SQL for creating the SQLite 3 database for this project



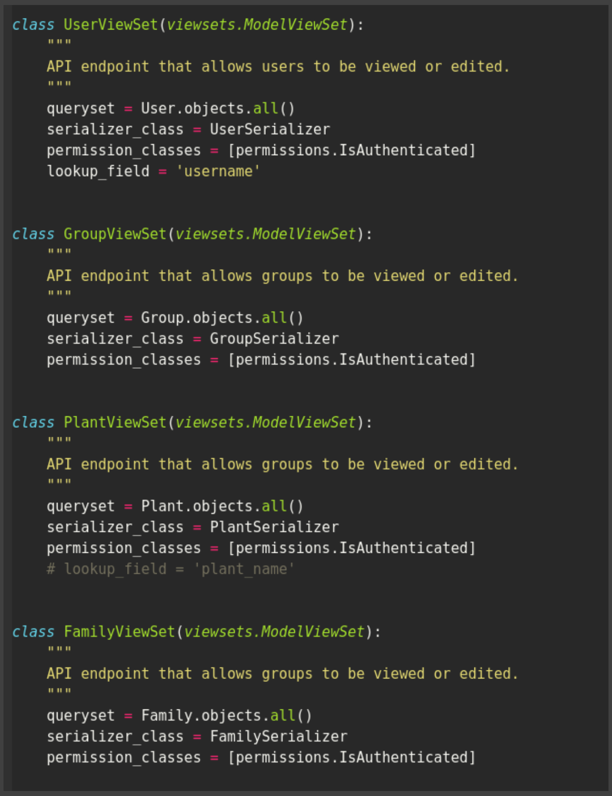
```
~/Documents/tafe_plant_water_timer_app_backend/plantronics  main !29 ? python manage.py sqlmigrate backend 0001          ✓ django-tafe
BEGIN;
--
-- Create model Family
--
CREATE TABLE "backend_family" ("family" varchar(255) NOT NULL PRIMARY KEY, "family_description" varchar(1000) NULL);
--
-- Create model Genus
--
CREATE TABLE "backend_genus" ("genus" varchar(255) NOT NULL PRIMARY KEY, "genus_description" varchar(1000) NULL);
--
-- Create model Plant
--
CREATE TABLE "backend_plant" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "plant_name" varchar(255) NOT NULL, "plant_latin_name" varchar(255) NOT NULL, "plant_description" varchar(1000) NULL, "family_id" varchar(255) NOT NULL REFE
ENCES "backend_family" ("family") DEFERRABLE INITIALLY DEFERRED, "genus_id" varchar(255) NOT NULL REFERENCES "backend_genus" ("genus") DEFERRABLE INITIALLY DEFERRED);
--
-- Create model UserPlant
--
CREATE TABLE "backend_userplant" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "plant_id" bigint NOT NULL REFERENCES "backend_plant" ("id") DEFERRABLE INITIALLY DEFERRED, "user_id" integer NOT NULL REFERENCES "auth_user" ("id") DEF
RRABLE INITIALLY DEFERRED);
--
-- Create model SoilPreference
--
CREATE TABLE "backend_soilpreference" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "preference" varchar(255) NOT NULL, "soil_description" varchar(500) NULL, "plants_id" bigint NULL REFERENCES "backend_plant" ("id") DEFERRABLE INIT
ALLY DEFERRED);
--
-- Create model Info
--
CREATE TABLE "backend_info" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "sun_preference" varchar(3) NOT NULL, "climate" varchar(3) NOT NULL, "season" varchar(3) NOT NULL, "time_frame" varchar(3) NOT NULL, "info_description" varch
r(1000) NULL, "plant_id" bigint NULL REFERENCES "backend_plant" ("id") DEFERRABLE INITIALLY DEFERRED);
--
-- Create model Edible
--
CREATE TABLE "backend_edible" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "is_fruit_edible" varchar(3) NOT NULL, "are_leaves_edible" varchar(3) NOT NULL, "are_roots_edible" varchar(3) NOT NULL, "are_flowers_edible" varchar(3) NOT
NULL, "are_seeds_edible" varchar(3) NOT NULL, "edible_description" varchar(1000) NOT NULL, "plant_id" bigint NOT NULL REFERENCES "backend_plant" ("id") DEFERRABLE INITIALLY DEFERRED);
CREATE INDEX "backend_plant_family_id_f224da4c" ON "backend_plant" ("family_id");
CREATE INDEX "backend_plant_genus_id_f162827d" ON "backend_plant" ("genus_id");
CREATE INDEX "backend_userplant_plant_id_55ef28e0" ON "backend_userplant" ("plant_id");
CREATE INDEX "backend_userplant_user_id_a2cccdb1" ON "backend_userplant" ("user_id");
CREATE INDEX "backend_soilpreference_plants_id_ddb948e5" ON "backend_soilpreference" ("plants_id");
CREATE INDEX "backend_info_plant_id_98ecc440" ON "backend_info" ("plant_id");
CREATE INDEX "backend_edible_plant_id_cd1d1e36" ON "backend_edible" ("plant_id");
COMMIT;
~/Documents/tafe_plant_water_timer_app_backend/plantronics  main !29 ?          ✓ django-tafe
```

7 use request, response and session objects rather then their superglobals

Unlike languages like PHP, Python and by extension Django have no such thing as a SuperGlobal, mainly because the Python devs dont want to support the use of this type of functionality, However global variables do exist, which are variables written outside of functions and can be called in other files by importing them e.g `from main import x`, which import the variable x from the file main.py.

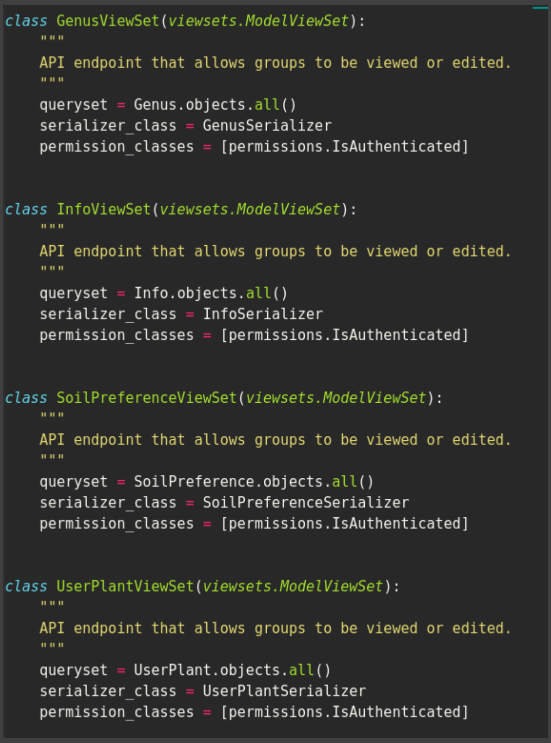
In Django request, response and session objects are handled by serializers, views, and django.auth.contrib which is its built-in authentication backend, transaction requests are handled in views.py.

Fig. 29: Views.py 1/3



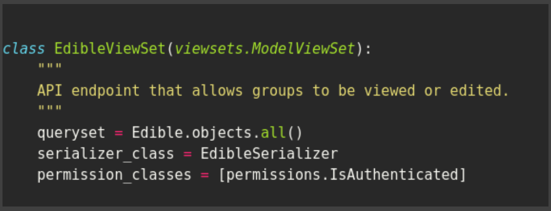
```python
class UserViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows users to be viewed or edited.
    """
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = [permissions.IsAuthenticated]
    lookup_field = 'username'


class GroupViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Group.objects.all()
    serializer_class = GroupSerializer
    permission_classes = [permissions.IsAuthenticated]


class PlantViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Plant.objects.all()
    serializer_class = PlantSerializer
    permission_classes = [permissions.IsAuthenticated]
    # lookup_field = 'plant_name'


class FamilyViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Family.objects.all()
    serializer_class = FamilySerializer
    permission_classes = [permissions.IsAuthenticated]
```

Fig. 30: Views.py 2/3



```
class GenusViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Genus.objects.all()
    serializer_class = GenusSerializer
    permission_classes = [permissions.IsAuthenticated]


class InfoViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Info.objects.all()
    serializer_class = InfoSerializer
    permission_classes = [permissions.IsAuthenticated]


class SoilPreferenceViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = SoilPreference.objects.all()
    serializer_class = SoilPreferenceSerializer
    permission_classes = [permissions.IsAuthenticated]


class UserPlantViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = UserPlant.objects.all()
    serializer_class = UserPlantSerializer
    permission_classes = [permissions.IsAuthenticated]
```

Fig. 31: Views.py 3/3

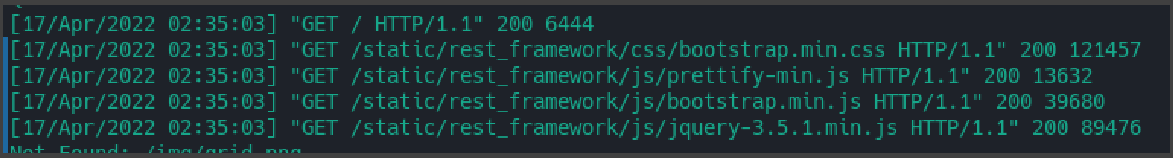


```
class EdibleViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Edible.objects.all()
    serializer_class = EdibleSerializer
    permission_classes = [permissions.IsAuthenticated]
```

8 Use at least three http response codes including 200

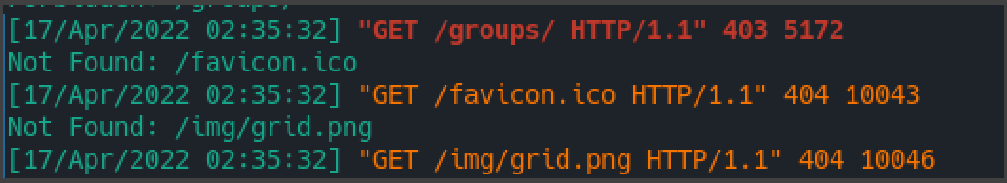
Django is designed to respond to all requests with all the http codes logged to the console, below are some screens of response logged to the console by Django:

Fig. 32: response codes



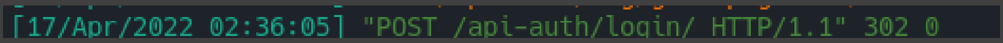
```
[17/Apr/2022 02:35:03] "GET / HTTP/1.1" 200 6444
[17/Apr/2022 02:35:03] "GET /static/rest_framework/css/bootstrap.min.css HTTP/1.1" 200 121457
[17/Apr/2022 02:35:03] "GET /static/rest_framework/js/prettify-min.js HTTP/1.1" 200 13632
[17/Apr/2022 02:35:03] "GET /static/rest_framework/js/bootstrap.min.js HTTP/1.1" 200 39680
[17/Apr/2022 02:35:03] "GET /static/rest_framework/js/jquery-3.5.1.min.js HTTP/1.1" 200 89476
Not Found: /img/grid.png
```

Fig. 33: response codes



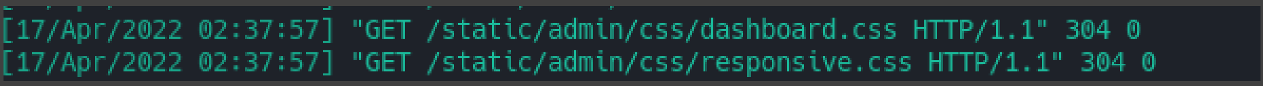
```
[17/Apr/2022 02:35:32] "GET /groups/ HTTP/1.1" 403 5172
Not Found: /favicon.ico
[17/Apr/2022 02:35:32] "GET /favicon.ico HTTP/1.1" 404 10043
Not Found: /img/grid.png
[17/Apr/2022 02:35:32] "GET /img/grid.png HTTP/1.1" 404 10046
```

Fig. 34: response codes



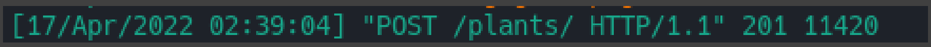
```
[17/Apr/2022 02:36:05] "POST /api-auth/login/ HTTP/1.1" 302 0
```

Fig. 35: response codes



```
[17/Apr/2022 02:37:57] "GET /static/admin/css/dashboard.css HTTP/1.1" 304 0
[17/Apr/2022 02:37:57] "GET /static/admin/css/responsive.css HTTP/1.1" 304 0
```

Fig. 36: response codes



```
[17/Apr/2022 02:39:04] "POST /plants/ HTTP/1.1" 201 11420
```

9 Response Data

9.1 Validation of GET and POST data before it is serialized or processed by an object

9.1.1 POST Requests

POST data such as Users, Plants, Family, Genus, etc, the data must contain all fields present in the model including fields which are not required which can be left empty, for example Family consists of 2 user defined fields, family and family_description, family is required and the post request will fail if its left empty by family_description isn't so it can be left empty when submitting the request without any consequences. both of the aforementioned user defined fields are both varchar fields if the POST request tries to submit data that doesn't conform to the fields format the request will not be accepted with a response stating malformed object data or something similar. some data can only by changed or updated by users who have either is_staff or is_superuser, which is defined in Django's user model as booleans, set to True.

9.1.2 GET Requests

Depending on what your trying to GET, GET requests are handled by various different means, if you trying to access the admin panel your request will be denied unless you login or are logged in as a staff user or superuser which is handled by Django's inbuilt auth system, other pages such as the Django Rest Api pages are by default accessibly by anyone but are configured in this instance to only allow registered users access to these pages.

9.1.3 Additional Validation

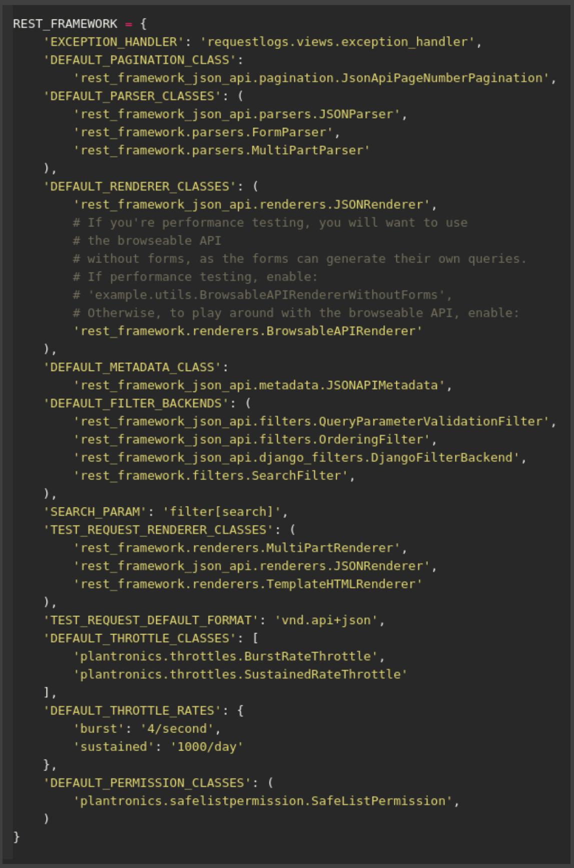
Additional validation is handled by various functions of Django and Django Rest Framework which are listed below:

```
        'rest_framework_json_api.filters.QueryParameterValidationFilter'
        'django.contrib.auth.password_validation.UserAttributeSimilarityValidator'
        'django.contrib.auth.password_validation.MinimumLengthValidator'
        'django.contrib.auth.password_validation.CommonPasswordValidator'
        'django.contrib.auth.password_validation.NumericPasswordValidator'
        'django.contrib.auth.middleware.AuthenticationMiddleware'
        'django.contrib.sessions.middleware.SessionMiddleware'
        'django.middleware.security.SecurityMiddleware'
```

9.2 Response Format

Into order to turn our app into a JSON churning machine, we must make some changes to the Django Rest Framework Config in settings.py as shown below:

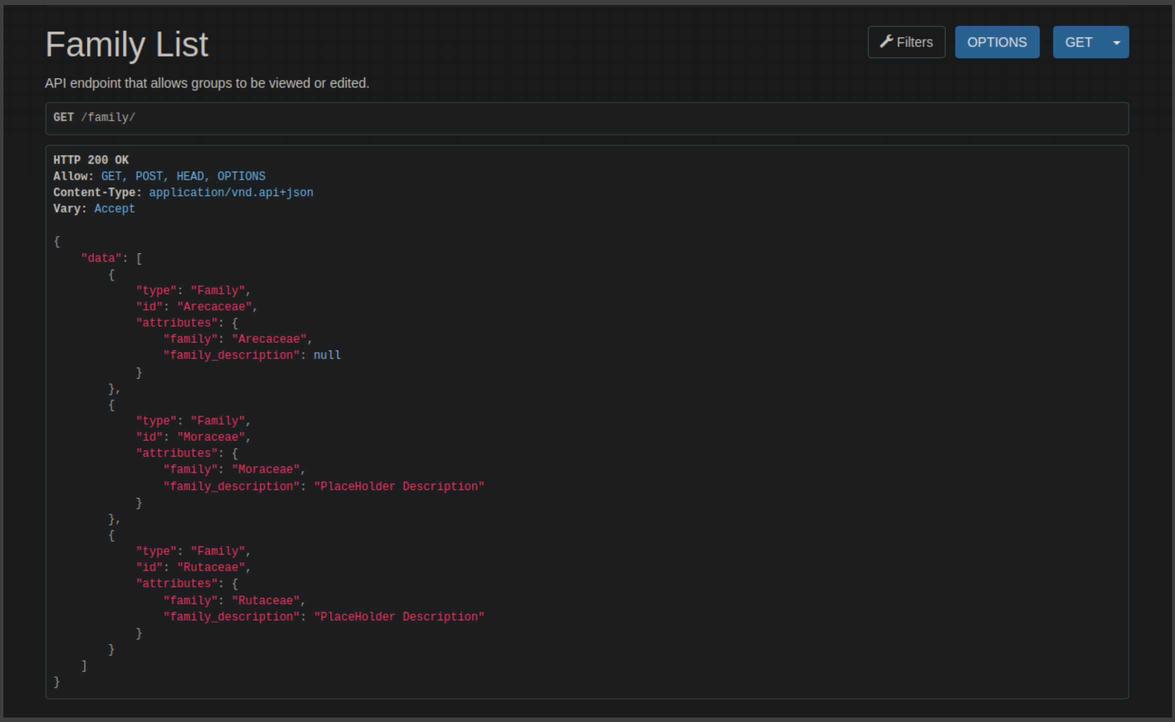
Fig. 37: Django Rest Framework Config



```
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER': 'requestlogs.views.exception_handler',
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework_json_api.pagination.JsonApiPageNumberPagination',
    'DEFAULT_PARSER_CLASSES': (
        'rest_framework_json_api.parsers.JSONParser',
        'rest_framework.parsers.FormParser',
        'rest_framework.parsers.MultiPartParser'
    ),
    'DEFAULT_RENDERER_CLASSES': (
        'rest_framework_json_api.renderers.JSONRenderer',
        # If you're performance testing, you will want to use
        # the browseable API
        # without forms, as the forms can generate their own queries.
        # If performance testing, enable:
        # 'example.utils.BrowsableAPIRendererWithoutForms',
        # Otherwise, to play around with the browseable API, enable:
        'rest_framework.renderers.BrowsableAPIRenderer'
    ),
    'DEFAULT_METADATA_CLASS':
        'rest_framework_json_api.metadata.JSONAPIMetadata',
    'DEFAULT_FILTER_BACKENDS': (
        'rest_framework_json_api.filters.QueryParameterValidationFilter',
        'rest_framework_json_api.filters.OrderingFilter',
        'rest_framework_json_api.django_filters.DjangoFilterBackend',
        'rest_framework.filters.SearchFilter',
    ),
    'SEARCH_PARAM': 'filter[search]',
    'TEST_REQUEST_RENDERER_CLASSES': (
        'rest_framework.renderers.MultiPartRenderer',
        'rest_framework_json_api.renderers.JSONRenderer',
        'rest_framework.renderers.TemplateHTMLRenderer'
    ),
    'TEST_REQUEST_DEFAULT_FORMAT': 'vnd.api+json',
    'DEFAULT_THROTTLE_CLASSES': [
        'plantronics.throttles.BurstRateThrottle',
        'plantronics.throttles.SustainedRateThrottle'
    ],
    'DEFAULT_THROTTLE_RATES': {
        'burst': '4/second',
        'sustained': '1000/day'
    },
    'DEFAULT_PERMISSION_CLASSES': (
        'plantronics.safelistpermission.SafeListPermission',
    )
}
```

once we have done that going to any of the api pages will present us with the following.

Fig. 38: Family Api Page



Family List

API endpoint that allows groups to be viewed or edited.

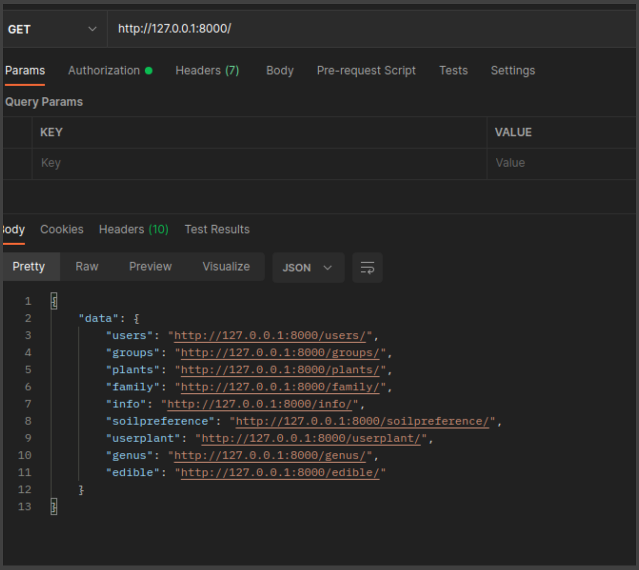
```
GET /family/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/vnd.api+json
Vary: Accept

{
    "data": [
        {
            "type": "Family",
            "id": "Arecaceae",
            "attributes": {
                "family": "Arecaceae",
                "family_description": null
            }
        },
        {
            "type": "Family",
            "id": "Moraceae",
            "attributes": {
                "family": "Moraceae",
                "family_description": "PlaceHolder Description"
            }
        },
        {
            "type": "Family",
            "id": "Rutaceae",
            "attributes": {
                "family": "Rutaceae",
                "family_description": "PlaceHolder Description"
            }
        }
    ]
}
```

In the above the content-type is automatically set as vnd.api+json, however as this page will never be seen by users we must test this in the same way a user will interacted with it, well closer to how a user will interact with it anyway, by using a piece of software known as Postman to send a get requests to the api.

As is displayed in the previous images the data is displayed in the JSON format, this format will be used by the frontend and server to send and receive data to/from each other.

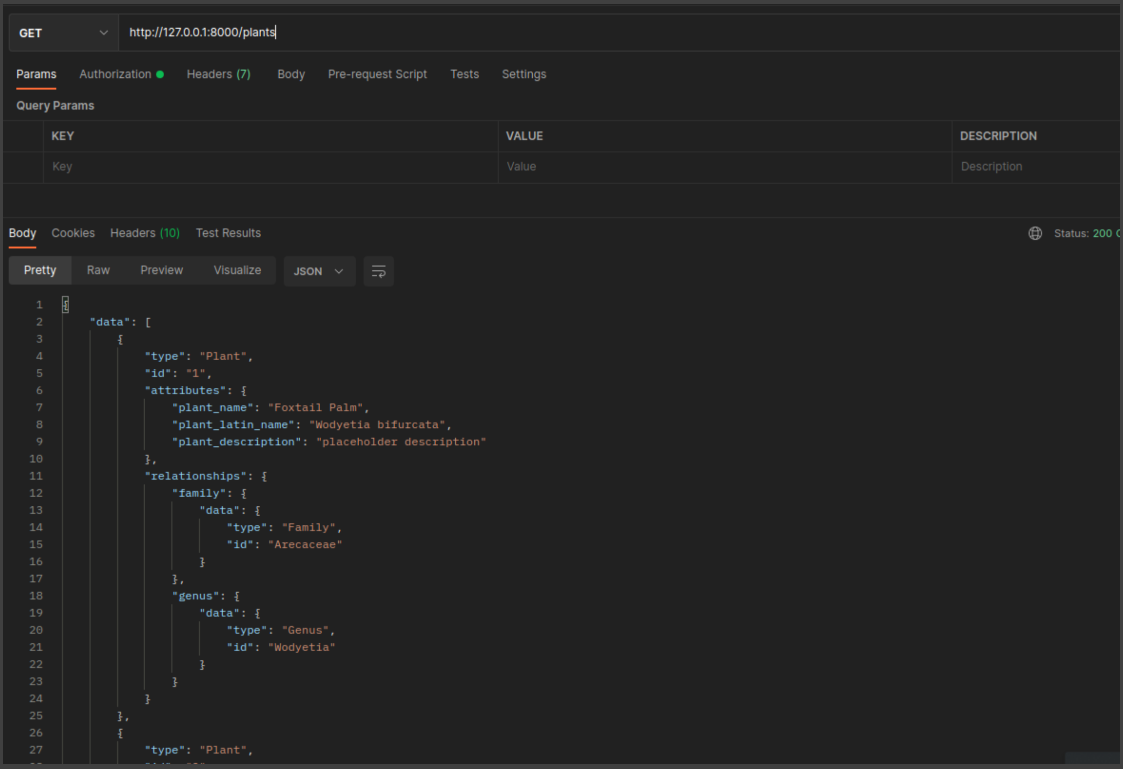
Fig. 39: Get request, to the home page



GET http://127.0.0.1:8000/

```
{
    "data": {
        "users": "http://127.0.0.1:8000/users/",
        "groups": "http://127.0.0.1:8000/groups/",
        "plants": "http://127.0.0.1:8000/plants/",
        "family": "http://127.0.0.1:8000/family/",
        "info": "http://127.0.0.1:8000/info/",
        "soilpreference": "http://127.0.0.1:8000/soilpreference/",
        "userplant": "http://127.0.0.1:8000/userplant/",
        "genus": "http://127.0.0.1:8000/genus/",
        "edible": "http://127.0.0.1:8000/edible/"
    }
}
```

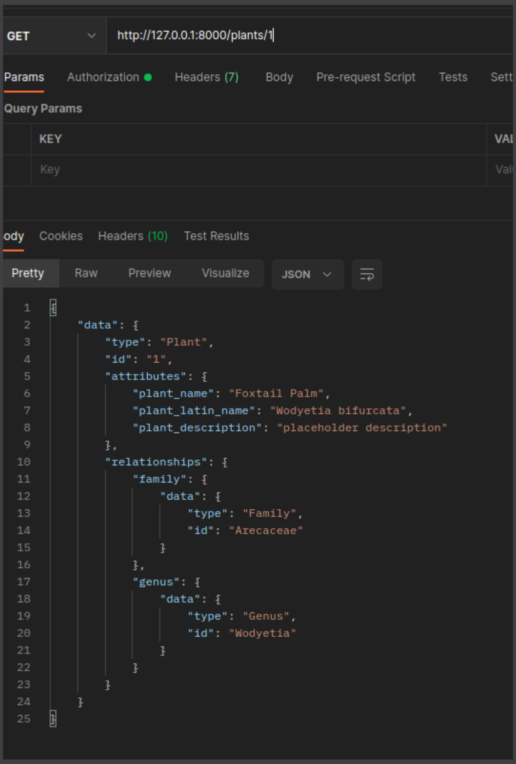
Fig. 40: Get request, to the plant page



GET http://127.0.0.1:8000/plants

```
{
    "data": [
        {
            "type": "Plant",
            "id": "1",
            "attributes": {
                "plant_name": "Foxtail Palm",
                "plant_latin_name": "Wodyetia bifurcata",
                "plant_description": "placeholder description"
            },
            "relationships": {
                "family": {
                    "data": {
                        "type": "Family",
                        "id": "Arecaceae"
                    }
                },
                "genus": {
                    "data": {
                        "type": "Genus",
                        "id": "Wodyetia"
                    }
                }
            }
        },
        {
            "type": "Plant",
```

Fig. 41: Get request, to the plant/1 to get the first plant



```
{
    "data": {
        "type": "Plant",
        "id": "1",
        "attributes": {
            "plant_name": "Foxtail Palm",
            "plant_latin_name": "Wodyetia bifurcata",
            "plant_description": "placeholder description"
        },
        "relationships": {
            "family": {
                "data": {
                    "type": "Family",
                    "id": "Arecaceae"
                }
            },
            "genus": {
                "data": {
                    "type": "Genus",
                    "id": "Wodyetia"
                }
            }
        }
    }
}
```

Part C
Code Comments

10 Explain why you created the database and session objects in that particular location

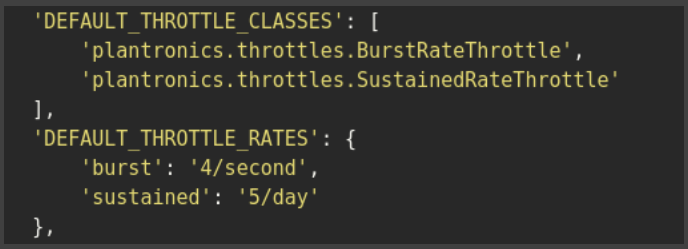
The database models were instantiated in models.py as is defined/required by Django, models.py is located in the backend folder. The reason for this is that Django follows the MVC folder structure, and Django likes to separate the main program which in this case is plantronics and its child applications which in this case is backend. Django prefers and insists developers create new apps rather then working in the main program directory because it allows for better readability and separation of sometimes fundamentally different parts of an program.

The models.py file in our application is where Django requires we build our database objects, The reason for this is that Django is preconfigured to use code from certain preconfigured locations such as models.py. Another maybe better way to explain this is that Django is a semi opinionated framework, which means it assumes some things like where are database objects are stored, but some things like whats shown to the user is left up to us. This is true for the database type as well, because by default Django is setup to use SQLite which makes sense in development and would most likely be changed in later closer to deployment.

Sessions and Session objects are handled by a middleware application called 'django.contrib.sessionsMIDDLEWARE' which needs to be Enabled to work, the session objects are stored in a table in the database called django_session.

11 Explain the arithmetic behind one of your rate-limiting codes

Fig. 42: Our Rate Limiters



```
'DEFAULT_THROTTLE_CLASSES': [
    'plantronics.throttles.BurstRateThrottle',
    'plantronics.throttles.SustainedRateThrottle'
],
'DEFAULT_THROTTLE_RATES': {
    'burst': '4/second',
    'sustained': '5/day'
},
```

We technically have two rate limiters, burst and sustained, burst allows each unique request origin to make 1 request for every second passed, which means for each connected ip address there is a counter and when the counter is greater then one in a give second, any additional requests are from that ip are rejected until the block is lifted for that ip which usually happens after one to two seconds.

For sustained rate limiting, in the code is specified as 1000/day, meaning for every unique request origin no more then 1000 requests can be made in a twenty four hour period as measured in Universal Basic Time (UTC), so once a unique address as surpassed that limit it will be blocked for one UTC day before being allow to send more requests.

Simply put once a IP address surpasses the defined limit for which each request made is recorded, they will no longer be able to make any more request until there cool down period is over.

12 Note where you're testing to see if a session already exists and what you'll do if it does

In Django sessions are stored in the database in a table called django_sessions, in Django SESSION_EXPIRE_AT_BROWSER_CLOSE is set to false, which means Django will store the sessions in the database until they and the cookie saved on the client expire, every time a unique request its made whether from a new browser or with a different header or set of headers it will create a new session. If the header of the browser connecting to the site contains the session cookie it will resume interacting with the site as a persistent user as long as the cookie has not expired and self destructed on either end determined by SESSION_COOKIE_AGE, this is handled by Django's session middleware.

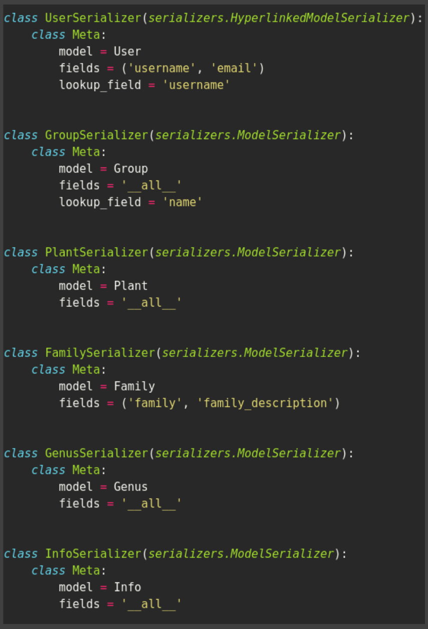
For further information refer to: Django csrf.py github.

13 Describe the code structure that verifies all GET/POST structures

Before each and every GET and Post request is processed, a UID or unique identifier is either sent in the request header or a new UID is set which itself is a cookie, which is stored in Django's Auth table and the users browsers. This cookie is automatically compare by Django's authentication middleware, which is used to track user interactions and validating which user has done what.

Each field has a associated data type and max length, for example the info table has a field called sun_preference which has a max length of 3 characters meaning it can only accept a string of 3 or less characters, the characters can be alphabetical, numerical or other special characters as long as its a string, the purpose of the 3 character limit is that, its purpose is to accept a key which maps to a choice provided in the model, if someone trys to post a new info data with the sun_preference set to a invalid choice the post request would be denied.

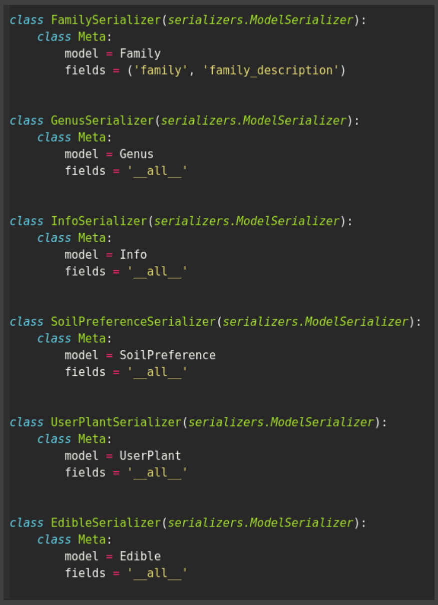
Fig. 43: Serializers



```
class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ('username', 'email')
        lookup_field = 'username'


class GroupSerializer(serializers.ModelSerializer):
    class Meta:
        model = Group
        fields = '__all__'
        lookup_field = 'name'


class PlantSerializer(serializers.ModelSerializer):
    class Meta:
        model = Plant
        fields = '__all__'


class FamilySerializer(serializers.ModelSerializer):
    class Meta:
        model = Family
        fields = ('family', 'family_description')


class GenusSerializer(serializers.ModelSerializer):
    class Meta:
        model = Genus
        fields = '__all__'


class InfoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Info
        fields = '__all__'
```

A serializers role is to specify the metadata for what a entity is and fields associate with it so that the data received may be translated from JSON to Python code and vice versa, serializers also allow developers to set a list of fields which are contained inside manually, but Django has the option '__all__' which is used to display all fields in a get request, but for example if you were to make a GET request for info with soilpreference of any which dosent exist in info, it will return a malformed json error, which also provides another layer of security and validation.

Fig. 44: Serializers



```
class FamilySerializer(serializers.ModelSerializer):
    class Meta:
        model = Family
        fields = ('family', 'family_description')


class GenusSerializer(serializers.ModelSerializer):
    class Meta:
        model = Genus
        fields = '__all__'


class InfoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Info
        fields = '__all__'


class SoilPreferenceSerializer(serializers.ModelSerializer):
    class Meta:
        model = SoilPreference
        fields = '__all__'


class UserPlantSerializer(serializers.ModelSerializer):
    class Meta:
        model = UserPlant
        fields = '__all__'


class EdibleSerializer(serializers.ModelSerializer):
    class Meta:
        model = Edible
        fields = '__all__'
```

Django's default auth model handles the necessary checks to determine if the user has the correct privileges in order to do certain actions.

14 Write a README file that explains how to setup and configure this service

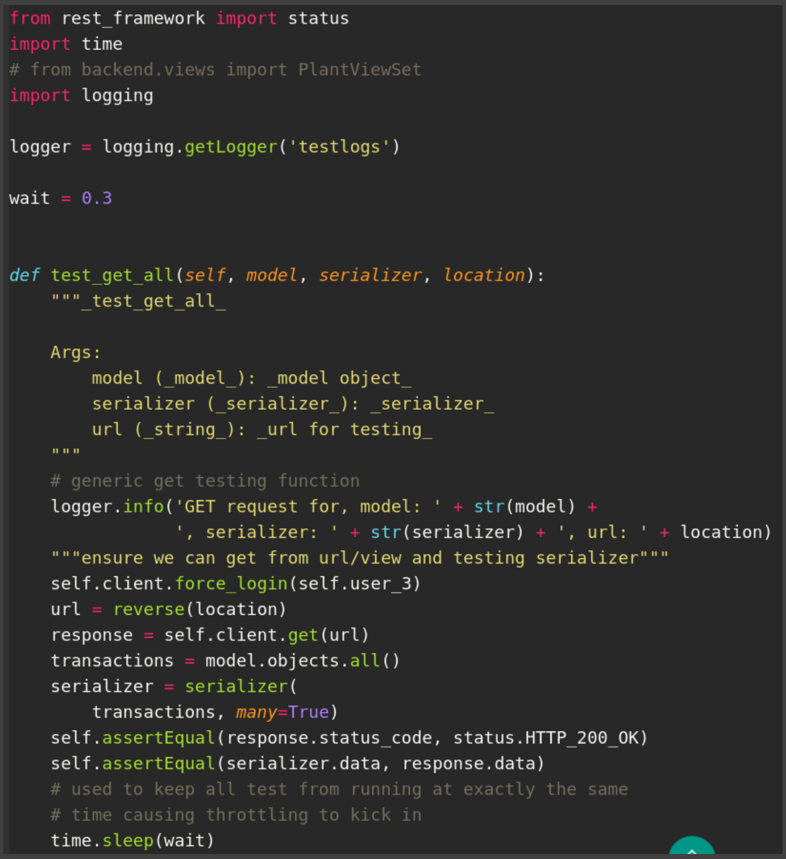
See Attached README.pdf in the same folder as this documentation

15 As part of unit testing, create a test script that interacts with the web service to test all known GET and POST requests.

There are eight total tests with each test containing three subtests, one post test, one get all test, and one get specific test. Which in total comes to 24 tests, in order to run all those test without getting rate limited there are two options, option 1 is to increase the burst rate limit to 4 or more per second, the second option which i chose to go with uses the python time library to wait around 0.4 seconds before preceding to the next test.

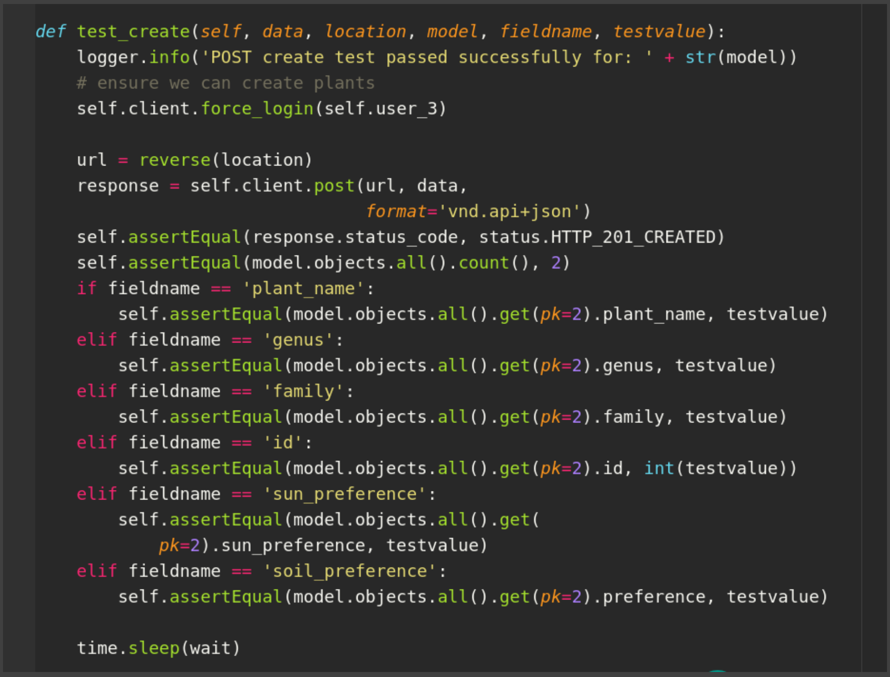
Test Cases

Fig. 45: Test Cases 1/3



```python
from rest_framework import status
import time
# from backend.views import PlantViewSet
import logging

logger = logging.getLogger('testlogs')

wait = 0.3


def test_get_all(self, model, serializer, location):
    """_test_get_all_

    Args:
        model (_model_): _model object_
        serializer (_serializer_): _serializer_
        url (_string_): _url for testing_
    """
    # generic get testing function
    logger.info('GET request for, model: ' + str(model) +
                ', serializer: ' + str(serializer) + ', url: ' + location)
    """ensure we can get from url/view and testing serializer"""
    self.client.force_login(self.user_3)
    url = reverse(location)
    response = self.client.get(url)
    transactions = model.objects.all()
    serializer = serializer(
        transactions, many=True)
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertEqual(serializer.data, response.data)
    # used to keep all test from running at exactly the same
    # time causing throttling to kick in
    time.sleep(wait)
```

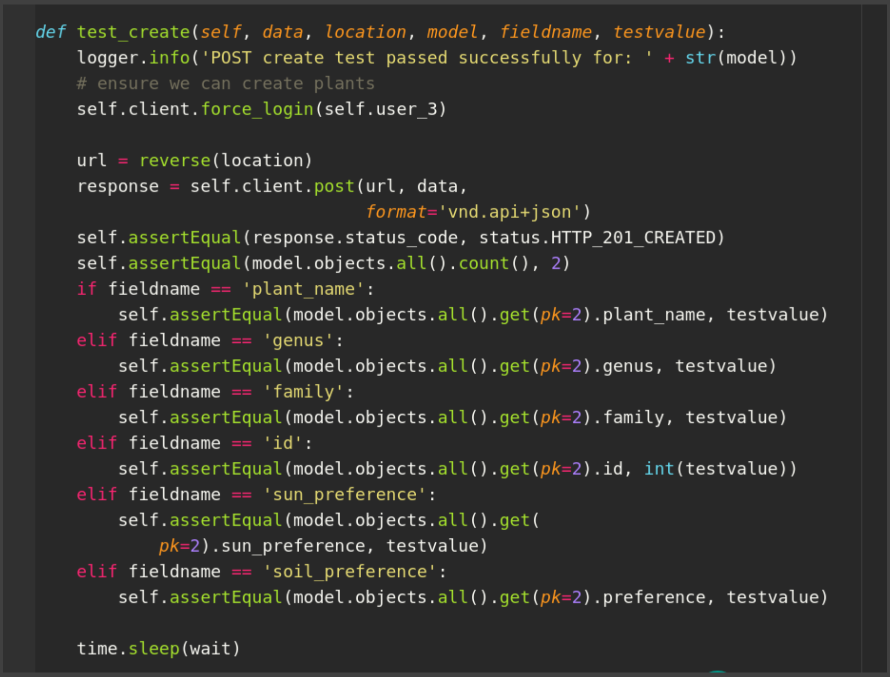
Fig. 46: Test Cases 3/3



```python
def test_create(self, data, location, model, fieldname, testvalue):
    logger.info('POST create test passed successfully for: ' + str(model))
    # ensure we can create plants
    self.client.force_login(self.user_3)

    url = reverse(location)
    response = self.client.post(url, data,
                                format='vnd.api+json')
    self.assertEqual(response.status_code, status.HTTP_201_CREATED)
    self.assertEqual(model.objects.all().count(), 2)
    if fieldname == 'plant_name':
        self.assertEqual(model.objects.all().get(pk=2).plant_name, testvalue)
    elif fieldname == 'genus':
        self.assertEqual(model.objects.all().get(pk=2).genus, testvalue)
    elif fieldname == 'family':
        self.assertEqual(model.objects.all().get(pk=2).family, testvalue)
    elif fieldname == 'id':
        self.assertEqual(model.objects.all().get(pk=2).id, int(testvalue))
    elif fieldname == 'sun_preference':
        self.assertEqual(model.objects.all().get(
            pk=2).sun_preference, testvalue)
    elif fieldname == 'soil_preference':
        self.assertEqual(model.objects.all().get(pk=2).preference, testvalue)

    time.sleep(wait)
```

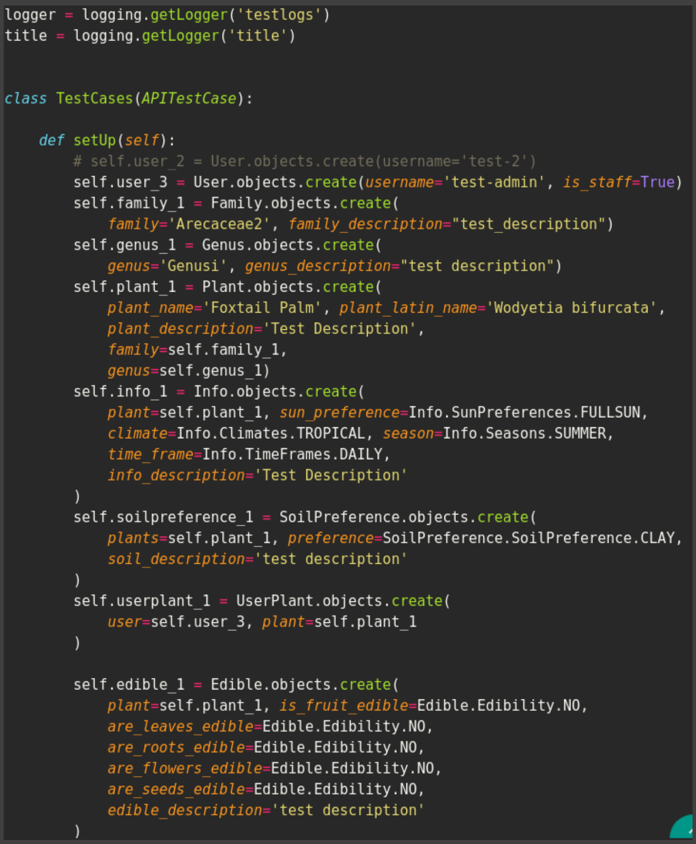
Tests

Fig. 47: Test Cases 3/3



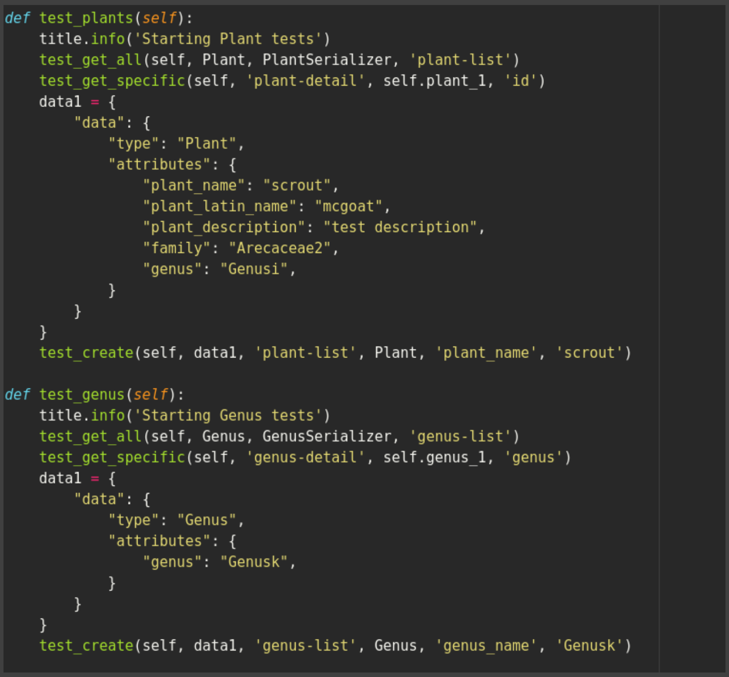
```python
def test_create(self, data, location, model, fieldname, testvalue):
    logger.info('POST create test passed successfully for: ' + str(model))
    # ensure we can create plants
    self.client.force_login(self.user_3)

    url = reverse(location)
    response = self.client.post(url, data,
                                format='vnd.api+json')
    self.assertEqual(response.status_code, status.HTTP_201_CREATED)
    self.assertEqual(model.objects.all().count(), 2)
    if fieldname == 'plant_name':
        self.assertEqual(model.objects.all().get(pk=2).plant_name, testvalue)
    elif fieldname == 'genus':
        self.assertEqual(model.objects.all().get(pk=2).genus, testvalue)
    elif fieldname == 'family':
        self.assertEqual(model.objects.all().get(pk=2).family, testvalue)
    elif fieldname == 'id':
        self.assertEqual(model.objects.all().get(pk=2).id, int(testvalue))
    elif fieldname == 'sun_preference':
        self.assertEqual(model.objects.all().get(
            pk=2).sun_preference, testvalue)
    elif fieldname == 'soil_preference':
        self.assertEqual(model.objects.all().get(pk=2).preference, testvalue)

    time.sleep(wait)
```

Fig. 48: Tests 1/5



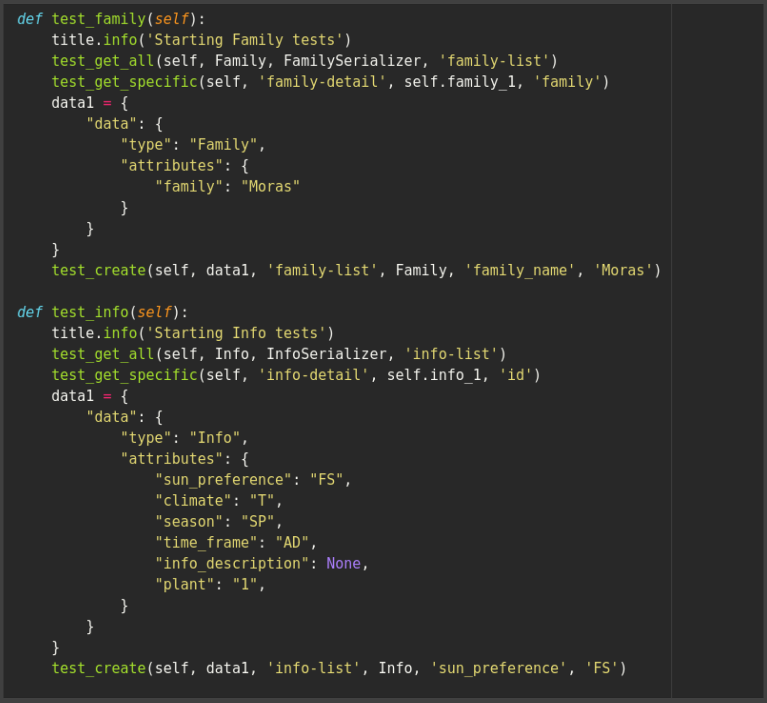
```
logger = logging.getLogger('testlogs')
title = logging.getLogger('title')


class TestCases(APITestCase):

    def setUp(self):
        # self.user_2 = User.objects.create(username='test-2')
        self.user_3 = User.objects.create(username='test-admin', is_staff=True)
        self.family_1 = Family.objects.create(
            family='Arecaceae2', family_description="test_description")
        self.genus_1 = Genus.objects.create(
            genus='Genusi', genus_description="test description")
        self.plant_1 = Plant.objects.create(
            plant_name='Foxtail Palm', plant_latin_name='Wodyetia bifurcata',
            plant_description='Test Description',
            family=self.family_1,
            genus=self.genus_1)
        self.info_1 = Info.objects.create(
            plant=self.plant_1, sun_preference=Info.SunPreferences.FULLSUN,
            climate=Info.Climates.TROPICAL, season=Info.Seasons.SUMMER,
            time_frame=Info.TimeFrames.DAILY,
            info_description='Test Description'
        )
        self.soilpreference_1 = SoilPreference.objects.create(
            plants=self.plant_1, preference=SoilPreference.SoilPreference.CLAY,
            soil_description='test description'
        )
        self.userplant_1 = UserPlant.objects.create(
            user=self.user_3, plant=self.plant_1
        )

        self.edible_1 = Edible.objects.create(
            plant=self.plant_1, is_fruit_edible=Edible.Edibility.NO,
            are_leaves_edible=Edible.Edibility.NO,
            are_roots_edible=Edible.Edibility.NO,
            are_flowers_edible=Edible.Edibility.NO,
            are_seeds_edible=Edible.Edibility.NO,
            edible_description='test description'
        )
```

Fig. 49: Tests 2/5



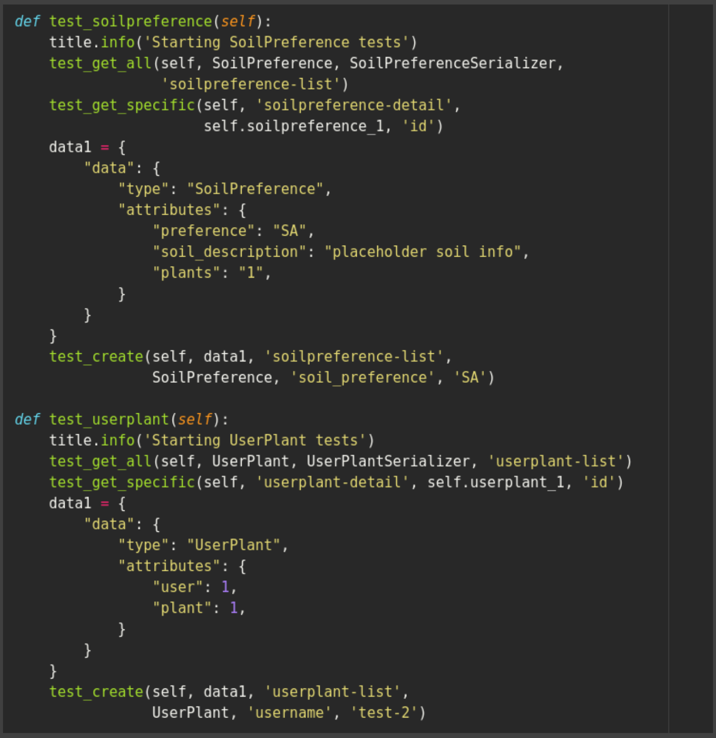
```python
def test_plants(self):
    title.info('Starting Plant tests')
    test_get_all(self, Plant, PlantSerializer, 'plant-list')
    test_get_specific(self, 'plant-detail', self.plant_1, 'id')
    data1 = {
        "data": {
            "type": "Plant",
            "attributes": {
                "plant_name": "scrout",
                "plant_latin_name": "mcgoat",
                "plant_description": "test description",
                "family": "Arecaceae2",
                "genus": "Genusi",
            }
        }
    }
    test_create(self, data1, 'plant-list', Plant, 'plant_name', 'scrout')

def test_genus(self):
    title.info('Starting Genus tests')
    test_get_all(self, Genus, GenusSerializer, 'genus-list')
    test_get_specific(self, 'genus-detail', self.genus_1, 'genus')
    data1 = {
        "data": {
            "type": "Genus",
            "attributes": {
                "genus": "Genusk",
            }
        }
    }
    test_create(self, data1, 'genus-list', Genus, 'genus_name', 'Genusk')
```

Fig. 50: Tests 3/5



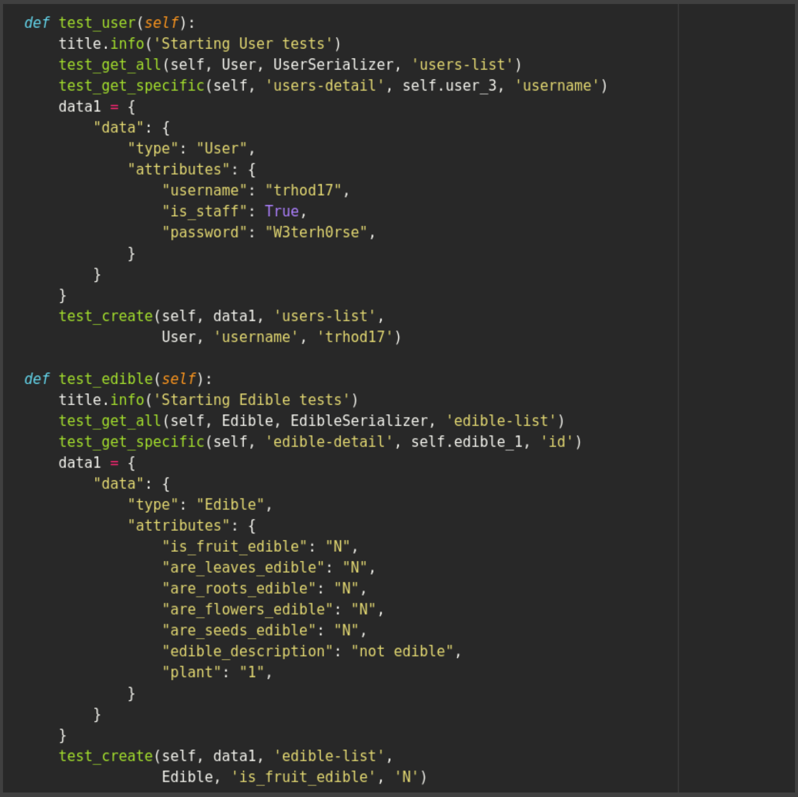
```python
def test_family(self):
    title.info('Starting Family tests')
    test_get_all(self, Family, FamilySerializer, 'family-list')
    test_get_specific(self, 'family-detail', self.family_1, 'family')
    data1 = {
        "data": {
            "type": "Family",
            "attributes": {
                "family": "Moras"
            }
        }
    }
    test_create(self, data1, 'family-list', Family, 'family_name', 'Moras')

def test_info(self):
    title.info('Starting Info tests')
    test_get_all(self, Info, InfoSerializer, 'info-list')
    test_get_specific(self, 'info-detail', self.info_1, 'id')
    data1 = {
        "data": {
            "type": "Info",
            "attributes": {
                "sun_preference": "FS",
                "climate": "T",
                "season": "SP",
                "time_frame": "AD",
                "info_description": None,
                "plant": "1",
            }
        }
    }
    test_create(self, data1, 'info-list', Info, 'sun_preference', 'FS')
```

Fig. 51: Tests 4/5



```
def test_soilpreference(self):
    title.info('Starting SoilPreference tests')
    test_get_all(self, SoilPreference, SoilPreferenceSerializer,
                 'soilpreference-list')
    test_get_specific(self, 'soilpreference-detail',
                      self.soilpreference_1, 'id')
    data1 = {
        "data": {
            "type": "SoilPreference",
            "attributes": {
                "preference": "SA",
                "soil_description": "placeholder soil info",
                "plants": "1",
            }
        }
    }
    test_create(self, data1, 'soilpreference-list',
                SoilPreference, 'soil_preference', 'SA')

def test_userplant(self):
    title.info('Starting UserPlant tests')
    test_get_all(self, UserPlant, UserPlantSerializer, 'userplant-list')
    test_get_specific(self, 'userplant-detail', self.userplant_1, 'id')
    data1 = {
        "data": {
            "type": "UserPlant",
            "attributes": {
                "user": 1,
                "plant": 1,
            }
        }
    }
    test_create(self, data1, 'userplant-list',
                UserPlant, 'username', 'test-2')
```

Fig. 52: Tests 5/5



```python
def test_user(self):
    title.info('Starting User tests')
    test_get_all(self, User, UserSerializer, 'users-list')
    test_get_specific(self, 'users-detail', self.user_3, 'username')
    data1 = {
        "data": {
            "type": "User",
            "attributes": {
                "username": "trhod17",
                "is_staff": True,
                "password": "W3terh0rse",
            }
        }
    }
    test_create(self, data1, 'users-list',
                User, 'username', 'trhod17')

def test_edible(self):
    title.info('Starting Edible tests')
    test_get_all(self, Edible, EdibleSerializer, 'edible-list')
    test_get_specific(self, 'edible-detail', self.edible_1, 'id')
    data1 = {
        "data": {
            "type": "Edible",
            "attributes": {
                "is_fruit_edible": "N",
                "are_leaves_edible": "N",
                "are_roots_edible": "N",
                "are_flowers_edible": "N",
                "are_seeds_edible": "N",
                "edible_description": "not edible",
                "plant": "1",
            }
        }
    }
    test_create(self, data1, 'edible-list',
                Edible, 'is_fruit_edible', 'N')
```