# **Guide on Django Basics**

***Note: everything you see here (and in every other word document we have included) is done automatically by our single shell script— you don’t need to read any of this-- but there’s still some useful information here on how to edit and manipulate and make use of the code once it’s been created (just as an example: since I forgot to mention elsewhere-- this terminal command allows you to test your API out once you’ve got it all set up: python manage.py test).***

A typical Django Rest API is basically just a folder containing a bunch of files organized in a specific structure (mostly made up of .py files within various folders) such that the many python files scattered throughout all interact with one another to perform various tasks. These tasks all relate to taking and receiving large sums of information in a form that only python code understands (python objects) and relaying it to and from a website-- and in doing so turns this data into a form of information that’s essentially going to be understood by most any device (and which can relatively easily and painlessly be transformed back into code to be used again by python): JavaScript, as you’d expect-- or more specifically, we call this type of notation JSON, wherein we transform Python objects into JavaScript objects –and then then back into Python. We choose this form for one reason (as far as I can tell)-- JavaScript is easy/simple/ubiquitous (relatively speaking) and functions like a sort of lingua franca for any random computer that might want to use our API regardless of the type, age or operating system of their machine--JSON lets them interact/communicate and transfer information back and forth.

You don’t really need to know anything about JavaScript, or anything else beyond what I’ve just told you—*as well as the following information I am about to give you regarding which websites to use in case of questions*: if you run into a problem/question here are the steps you should take to attempt to resolve it—in order, with most of your time spent with the earlier options:

1) here is how almost every problem/question you get will start: you get your API running but some URL on your website just results in a big yellow box showing a long complicated error message-- the optimal solution you should attempt first is **simply copying and pasting that error info into google, and seeing what pops up in stackexchange as a result** (I really don’t like doing this, but this is generally what is expected of you)-- if that fails then 2) check out their main website if easy solutions fail: <http://www.cdrf.co/> ……..and in absolute emergencies if you’ve tried those two things sufficiently 3) consult the original github code for django directly: <https://github.com/django/django>

...and finally, if you have no other work/questions for the time being this is what you should spend some time on-- whenever you have a free moment: 4) whenever you have free time-- mess around with this: <https://learngitbranching.js.org/> ...that’s what will help you develop the skills you’ll need to properly utilize github such that you’ll be able to do fun stuff related to creating multiple versions and branches so that you’ll be able to make use of (and you’ll *need* to for the final project) to divide up any large project into a bunch smaller tasks-- and then reincorporate code altered/developed during these tasks back into your main project. It’s relatively straightforward and shouldn’t take you too long to get through all of it.

Whenever you’re making a *specific* module (1 .py file) or group of modules as part of an individual lab assignment for either comp525 or comp705, this is how they want you to do it:

1. Create Design Document- for each method/function lab asks you to make-- give the name of the function and then the following: Design Section (gives created code explanation structured like comments), Docstring (the triple quotation marks thing), Test Cases (give 3 test cases for each and their return values)
2. Activate your virtual environment: in our case, we called ours EnvironV1, and store it in our Workspace directory, so we just use this (as mentioned earlier, above): source ~/workspace/environV1/bin/activate

*Note: here (below) are the steps that should be taken to create and control a new virtual environment, if you have not already done so prior to this-- if you already know how to set up a virtual environment, you can skip this section:*

**making a new virtual environment:**

Here is the process by which you would **create a new virtual environment** when you need one-- though you won’t need to do this very often-- just any time we want to have a unique, specific, controlled number of packages different from any of your other virtual environments:

1) Make a virtual environment: python3 -m venv environV1   
2) Active with source: source environV1/bin/activate (from your current location this would be: source ../environV1/bin/activate)   
  
Optional 2.5) Double check it's working by checking all the --version on the pip's and the python's   
  
2.6-2.9) We want any python package installation we do while first-venv is activated to be CONFINED to the specific bin   
directory of the given virtual environment-- in THIS case then all packages would be stored in: environV1/bin....... to do this   
we'd obviously use pip install <package>   
  
3) deactivate virtual environment with: deactivate   
  
4) delete whole entire virtual environment recursively when we're done with it using rm -rf first-venv   
       (Note: by making sure we stay INSIDE our virtual-envs directory when we do that, we can make it less likely we will    
accidentally delete far more than intended with that trick)

1. go to the URL you’re given that gives you invitation for the lab or homework in questions (usually in email or ppt canvas messages)-- then just accept everything to get access, copy/paste the URL from git repository you’re brought to into terminal (using ctrl+shift+c and ctrl+shift+c or simply with right click to select the copy and paste options)-- that way you can just use: **git clone long\_url\_name\_goes\_here**
2. For visualizing/editing our files: open folder containing code in text editor Atom in right hand window (it’s nice to have the terminal open as well, beside it-- **tree** is useful to have, for visualizing stuff in the terminal-- use sudo apt-get tree or whatever you need to type to get it then you can see all of the various directories/sub-directories beneath your location by just typing tree into the terminal)
   1. Note (as it relates to preferred text editors): The number one cause of all of the problems you are ever going to face is not due to anything complex or difficult-- it’s***inconsistent indentation varying between using tabs to indent and using spaces****.* This happens because any time you grab code from various other parts of your code-- or from online sources (which you are frequently instructed to do) there is a tendency for the code you grabbed to be transferred into a format that uses multiple spaces for indentation instead of tabs. If even a single line is inconsistent, none of the code will work. The simplest way to fix this is just with a generic text editor like “kate”-- atom text editor unfortunately seems to have no way of even recognizing that there is a difference between the two forms of indentation in how the code is displayed-- but is perfectly content to stubbornly stone-wall you with errors despite this. Therefore, if this crops up, immediately x out of atom, enter kate file\_name.py into terminal-- change the indentation type to hard tabulation at the bottom of the text editor-- and then just use the old ctrl+f to search for the multi-space combo that means tab, and replace them with *actual* tabs. Then save it, x out of kate immediately, and start atom again. It should start atom in the same exact location you had when you had earlier. Crude. Messy. Ugly. Frequent. Frustrating. Time-wasting. If I knew of a solution that was better than the one I just gave you I would but I don’t. Just giving you the heads up on what to expect.
3. Here’s how to test your Python code directly in Bash using terminal commands:

...enter into terminal to start: **python**

...now that you’re in python, import what you need (namely the code): **import h3problems.problems**

...optional: you can **import importlib** as well if you want the be able to use **importlib.reload** to change and re-test code without needing to close/re-open python. After this step, just use tab a bunch to make all the remaining steps go quicker.

...if your python code whose functions your testing are inside a class you’ll need to create an object of that class-- I call mine ob1:

**ob1=h3problems.problems.Problems()**  
...then create a function-- again use tab to make this quicker: fun1=ob1. --and then just hit tabs to see what you choose from, and quickly auto-complete it out from a single letter to get something like this in the terminal:

**fun1=ob1.keep\_count(‘example test case goes in here')**  
...then just use this to see what you got: **fun1**

...and then you can continue to mess around or ctrl+d out when done

1. Unit test:
   1. a reminder: to implement an individual test simply use the following(entered into terminal): **python -m unittest -v tests/test\_needs\_sorting.py**
   2. ...or simply: **python -m unittest tests/test\_needs\_sorting.py**
   3. To test ALL the files within a folder, use discover when you enter your terminal command-- like so: **python -m unittest discover tests**
   4. or for more verbose details use this: **python -m unittest discover -v tests**
      * **...that last one is all you really need to do all the tests in one command**
   5. If not already created, make a test folder with the necessary testing modules (.py) that have testing classes (so you can test them all in one step)
   6. create unique tests with unique names based on the example test given— just follow that naming format convention and structure-- so the names just have to start with “test.”
2. Verify coding style

terminal command: **pycodestyle some\_file.py**

1. Apply version control to Github
   1. **git status**
   2. **git add (file name).py**
   3. **git status**
   4. **git commit -m “implemented (function name) in (file name).py”  OR**

**git commit -m “added three test cases to (file name).py”**

* 1. **git push**

Simpler, Better, Clearer, and More Explicit Starting Procedure for Setting Up and Coding Django REST API:

*Note: just treat the term 'bucketlist' whenever I mention it like a generic placeholder term-- you can put anything you want in there based on the situation—just stay consistent for a given project with what you call it.*

*However, just make sure when you ctrl+f out of that situation that you first replace the plural (bucketlists with an s) before the singulars so you dont accidentally miss a bunch of random s's lying around after a replacement spree.*

**Ok here’s how to start (you may know a lot of these early steps already—just mentioning everything for consistency’s sake):**

1) activate virtual environment

2) set up your main folder to be the root of the project called whatever you want: **mkdir xxx-peanuts-project.**

3) START VERSION CONTROL:

3a) **git init**

...to turn project into a git repo

3b) copy a specific file over using **cp .gitignore** file from some other random python project-- there's one you can copy in the rest-api directory next door, so once you've copied it over just

3c) use the appropriate git commands to add it and you're on your way:

**git add .gitignore**

**git commit -m 'we created the project root directory and added in .gitignore to get ourselves started'**

4) create a text file that lists the installation requirements needed to make this code run properly like so:

**touch requirements.txt**

**pip freeze > requirements.txt**

5) Use startproject to create our first important subdirectory in any API-- call it whatever-- I call it config by convention: I create config package with this command (don't forget the **dot** at the end!):

**django-admin startproject config .**

6) Apply Migrations:

**python manage.py migrate**

a) when changes get made you want to implement use this command-- in all subsequent cases, you’ll want to use the makemigrations command just before the migrate command like so:

**python manage.py makemigrations**

**python manage.py migrate**

7.1) ...and when we're ready to start running this thing we use this:

**python manage.py runserver**

7.2) ...then just plug whatever pops up on screen into a url like so-- http://127.0.0.1:8000 becomes

**enter this into url box in your normal browser like firefox: http://127.0.0.1:8000/admin/** and at that point once you've typed that in you should be prompted to enter username/password to get past that part of the website-- which leads us to 7.3 below

7.3) ...and once we've got it all up and running we can create a superuser to get past the login screen on our new website by using this:

**python manage.py createsuperuser**

8) ...then we just want to do a slight alteration to make sure to add (just shove it into the python list use a text editor like you see below) **rest\_framework** to **INSTALLED\_APPS** list in **config/settings.py**

...so that it INSTALLED\_APPS list will have 'rest\_framework' somewhere on it-- sort of looking like this-ish:

INSTALLED\_APPS = [

'rest\_framework',

etc, etc, etc...--just shove the new installed app in there as you create it, whatever you call it

]

9) **git add, commit and check for statuses** before moving on to the next step

10) last step before moving onto coding-- create our ***really*** important starting package-- this one’s the one the only one that we’re really ever going to be directly messing around in, to any significant degree-- there’s one other file: urls.py that oftentimes exists in both the config directory and another one we’ll place in this one-- both matter to us-- but other than that one file-- this folder is going to contain essentially everything that we care about:

**python manage.py startapp api**

...then add that final package name to our list of INSTALLED\_APPS in the list as well, and add it to that INSTALLED\_APPS list that's located in **config/settings.py**

***...and now we're finally ready to start coding:***

Everything you see below is essentially the basic sort of generic setup we’re essentially going to carry out any time we want to create an API-- these are the steps after the initial setup I just gave all the steps/instructions for-- once that’s done, we’ll need put the rest in directly, by mainly creating and altering the python code in the various files within the directory we created with startapp just a moment ago-- which we called api (as well as that one urls.py we will be setting up within the config folder) .py files

Here are the generic default files/folders we need to have setup within the api directory-- some are in here automatically—any that aren’t here you can just create normally with touch or nano: **admin.py  apps.py  \_\_init\_\_.py  migrations  models.py  \_\_pycache\_\_  serializers.py  tests.py  urls.py  views.py**  
...and here are the ones that will be messing around with *directly* (by altering the python code itself): **admin.py  apps.py  models.py  serializers.py  tests.py  urls.py  views.py**

--but you never really do anything with apps.py after creating it so don’t worry about that.

***So—after that initial setup here’s what you need to know: start gaining understanding and becoming more comfortable with those seven files (since you’ll typically see some version of these in any typical Django Rest API you come across).***

# >>initial setup ends, **actual coding** starts now<<

Once all of that setup is finished, we're ready to start our first step of the second half of this process: the actual CODING-- which starts out writing some tests into the tests.py folder of our api app.

-----------------------

1) write/run tests:

write some tests in the tests.py folder of our api app

here's an example of something basic that you can plug in:

# /api/tests.py

Running tests that you’ve created there is as simple as entering this into the terminal: **python manage.py test**

Here’s an example from another file (that works) of what you might see inside a tests.py file:

**import** unittest   
**from** django.test **import** TestCase   
**from** django.test **import** Client   
  
**# Create your tests here.**   
**from** django.urls **import** reverse   
**from** rest\_framework.test **import** APITestCase, APIClient   
**from** rest\_framework.views **import** status   
  
**from** .models **import** User   
**from** .serializers **import** UserSerializer

**class** BaseViewTest(APITestCase):   
   client = APIClient()   
  
   @staticmethod   
   **def** **create\_user**(first\_name="**", last\_name="", email="", username="", password="**"):   
       **if** first\_name != "**" and last\_name != "" and email != "" and username != "" and password != "**":   
           User.objects.create(first\_name=first\_name, last\_name=last\_name, email=email, username=username, password=password)   
  
   **def** **setUp**(self):   
       **# add test data**   
       self.create\_user(**"Billy"**, **"Blackenthrope"**, **"something777@mail.com"**, **"pixel\_lad69"**, **"eato\_guess!!!"**)   
       self.create\_user(**"Jill"**, **"Blacklorne"**, **"something1@mail.com"**, **"pixel\_lad\_abc999"**, **"easo\_guess!!!"**)   
       self.create\_user(**"Willy"**, **"Blackman"**, **"something23@mail.com"**, **"pixel\_lad779"**, **"easy\_to\_gu!!!"**)   
       self.create\_user(**"Timmy"**, **"Blackenthropenstork"**, **"something@mail.com"**, **"pixel\_lady420"**, **"easy!!!"**)   
  
  
**class** GetAllUsersTest(BaseViewTest):   
  
   **def** **test\_get\_all\_users**(self):   
       response = self.client.get(   
           reverse(**"user-all"**, kwargs={**'version'**: **'v1'**})   
       )   
  
       expected = User.objects.all()   
       serialized = UserSerializer(expected, many=**True**)   
       self.assertEqual(response.data, serialized.data)   
  
       self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

All you really need to understand about that is that the BaseViewTest class is creating our data that we’re going to test by using the create\_user function to grab it out of the setUp function-- as the create\_user function is called over and over again, for each series of strings we test. This is then taken by the GetAllUsersTest class and put into one of the functions within it to be tested-- for the test you see here, we would have ONE test run, because we only have ONE function. Adding functions here increases the number of tests.

The only things to bear in mind is the consistency of ‘v1’ in this and other files, the consistency of ‘user-all’ or whatever name you choose.

Test results would look like this for 1 function like we have:

python manage.py test

Creating test database for alias 'default'...   
System check identified no issues (0 silenced).   
.   
----------------------------------------------------------------------   
Ran 1 test in 0.023s   
  
OK   
Destroying test database for alias 'default'...

-----------------------

2) define our models:

    ...edit this file-- /api/models.py --and thereby create a blank model class there which will basically just look like this:

# /api/models.py

from django.db import models

class Bucketlist(models.Model):

    pass

-----------------------

3) use this terminal command:

python manage.py test

...to MAKE SURE we're getting error messages.  that's how we know we're on the right track.

-----------------------

4) Now we're going to define some fields that'll represent our table fields in our database-- by adding some code to further altering our models.py file into this (filling in that empty models class that we had created earlier in step 2 of coding):

# api/models.py

from django.db import models

class Bucketlist(models.Model):

    """This class represents the bucketlist model."""

    name = models.CharField(max\_length=255, blank=False, unique=True)

    date\_created = models.DateTimeField(auto\_now\_add=True)

    date\_modified = models.DateTimeField(auto\_now=True)

    def \_\_str\_\_(self):

       """Return a human readable representation of the model instance."""

       return "{}".format(self.name)

-----------------------

5) we then migrate to propagate those changes with the following terminal command:

python manage.py makemigrations

...and then apply those migrations to the database like so:

python manage.py migrate

-----------------------

6) then you can run that test one last time-- same as you did earlier:

python manage.py test

...and hopefully this time it will pass (since it's supposed to after this)-- yay! It works!

-----------------------

7) Serializers: these will take the complex querysets out from the DB to turn it into human-readable/usable data like JSON or XML.  Obviously, we also need a DE-serializer, in order to transfer that data BACK to the database when we're done with it.

...we'll use a ModelSerializer class to make this whole process slightly more efficient if we can.  Here's the code we want in serializers.py:

# api/serializers.py

from rest\_framework import serializers

from .models import Bucketlist

class BucketlistSerializer(serializers.ModelSerializer):

    """Serializer to map the Model instance into JSON format."""

    class Meta:

       """Meta class to map serializer's fields with the model fields."""

       model = Bucketlist

       fields = ('id', 'name', 'date\_created', 'date\_modified')

       read\_only\_fields = ('date\_created', 'date\_modified')

-----------------------

8) views:

8a) first create our tests (BEFORE creating the things that we're GOING to test) within api/tests.py:

...for us it'll look something like this:

# api/tests.py

# Add these imports at the top

from rest\_framework.test import APIClient

from rest\_framework import status

from django.core.urlresolvers import reverse

# Define this after the ModelTestCase

class ViewTestCase(TestCase):

    """Test suite for the api views."""

    def setUp(self):

       """Define the test client and other test variables."""

       self.client = APIClient()

       self.bucketlist\_data = {'name': 'Go to Ibiza'}

       self.response = self.client.post(

           reverse('create'),

           self.bucketlist\_data,

           format="json")

    def test\_api\_can\_create\_a\_bucketlist(self):

       """Test the api has bucket creation capability."""

       self.assertEqual(self.response.status\_code, status.HTTP\_201\_CREATED)

-----------------------

...Note: if possible, before creating the thing that we are about to test you need to perform a test NOW, to MAKE SURE YOUR CODE FAILS BEFORE checking to see if it works when you're done making the thing you're planning on testing with it later on.

-----------------------

8b) ...once you've made sure your test fails, you can start actually constructing the thing it's designed to test for: the views and the URLs that will actually be handling our POST request-- so get started on finally getting something written into api/views.py:

# api/views.py

from rest\_framework import generics

from .serializers import BucketlistSerializer

from .models import Bucketlist

class CreateView(generics.ListCreateAPIView):

    """This class defines the create behavior of our rest api."""

    queryset = Bucketlist.objects.all()

    serializer\_class = BucketlistSerializer

    def perform\_create(self, serializer):

       """Save the post data when creating a new bucketlist."""

       serializer.save()

...this views.py here is basically in charge of deciding how data gets moved, stored, and most importantly, how it's displayed.  ListCreateAPIView is just designed to show us a sort of a generic view, and which provides means of handling

2 of the 4 big guys: GET and POST.

-----------------------

9) URLs: for our purposes these are basically just an interface with the outside world-- if people want to interact with our web API, they'll need to use these URLs to do it.

9a) So first we define our URL patterns-- within a new file we create within api called urls.py which will look like this:

# api/urls.py

from django.conf.urls import url, include

from rest\_framework.urlpatterns import format\_suffix\_patterns

from .views import CreateView

urlpatterns = {

    url(r'^bucketlists/$', CreateView.as\_view(), name="create"),

}

urlpatterns = format\_suffix\_patterns(urlpatterns)

-----------------------

9b) Of note here is that what SOME people refer to as djangorest, I refer to here as config.  So our final bit of code to make this thing complete is getting put into the urls python module that's already in the config folder.  This is what it should look like (after you're done plugging this in):

# config/urls.py

# This is the main urls.py. It shouldn’t be mistaken for the urls.py in the api directory

from django.conf.urls import url, include

urlpatterns = [

    url(r'^admin/', admin.site.urls),

    url(r'^', include('api.urls')) # Add this line

]

10) DONE!!! ...hopefully?  If all goes as planned you should now simply be able to use this like you would any other standard django rest API. Just type the usual thing in to get it running:

python manage.py runserver

...and grab the url development server from whatever gets output onto your console (just like usual).  The only slight difference here you need to be aware of it that this url is currently set up to be used with 'bucketlists'-- therefore, you'll need to use that in the place of where you'd normally put something like admin-- you'll therefore need to use something like this to get to the main website:

http://127.0.0.1:8000/bucketlists

*The 7 Django .py files: UNDERSTANDING HOW THE PYTHON CODE IN THESE 7 GENERIC PYTHON FILES (WHICH YOU WILL SEE IN ESSENTIALLY ANY DJANGO REST API YOU MESS WITH) ACTUALLY INTER-RELATE AND INTER-CONNECT WITH ONE ANOTHER*

*[Code From Reflections Comp705 Proj—this is the simplest example you’ll find of what to expect within these 7 files—* ***again: all these steps are taken care of automatically by our shell script—****this is just for reference for those who may be interested. These 7 files are the ones that you would normally have to take care of manually by hand—they aren’t generated automatically, normally. Obviously though, our shell script will take care of all of this for you]*

**6 python files within api/:**

1. **admin.py:**

from django.contrib import admin

from .models import User

admin.site.register(User)

1. **models.py:**

from django.db import models

class User(models.Model):

first\_name = models.CharField(max\_length=255, null=False)

last\_name = models.CharField(max\_length=255, null=False)

email = models.CharField(max\_length=255, null=False)

username = models.CharField(max\_length=255, null=False)

password = models.CharField(max\_length=255, null=False)

def \_\_str\_\_(self):

return "{} - {}".format(self.first\_name, self.last\_name, self.email,

self.username, self.password)

1. **serializers.py:**

from rest\_framework import serializers

from .models import User

class UserSerializer(serializers.ModelSerializer):

class Meta:

model = User

fields = ("first\_name", "last\_name", "email", "username", "password")

1. **tests.py:**

import unittest

from django.test import TestCase

from django.test import Client

# Create your tests here.

from django.urls import reverse

from rest\_framework.test import APITestCase, APIClient

from rest\_framework.views import status

from .models import User

from .serializers import UserSerializer

#first\_name

#last\_name

#email

#username

#password

class BaseViewTest(APITestCase):

client = APIClient()

@staticmethod

def create\_user(first\_name="", last\_name="", email="", username="", password=""):

if first\_name != "" and last\_name != "" and email != "" and username != "" and password != "":

User.objects.create(first\_name=first\_name, last\_name=last\_name, email=email, username=username, password=password)

def setUp(self):

# add test data

self.create\_user("Billy", "Blackenthrope", "something777@mail.com", "pixel\_lad69", "eato\_guess!!!")

self.create\_user("Jill", "Blacklorne", "something1@mail.com", "pixel\_lad\_abc999", "easo\_guess!!!")

self.create\_user("Willy", "Blackman", "something23@mail.com", "pixel\_lad779", "easy\_to\_gu!!!")

self.create\_user("Timmy", "Blackenthropenstork", "something@mail.com", "pixel\_lady420", "easy!!!")

class GetAllUsersTest(BaseViewTest):

def test\_get\_all\_users(self):

response = self.client.get(

reverse("user-all", kwargs={'version': 'v1'})

)

expected = User.objects.all()

serialized = UserSerializer(expected, many=True)

self.assertEqual(response.data, serialized.data)

self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

def test\_get\_all\_usersb2(self):

response = self.client.get(

reverse("user-all", kwargs={'version': 'v1'})

)

expected = User.objects.all()

serialized = UserSerializer(expected, many=True)

self.assertEqual(response.data, serialized.data)

self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

1. **urls.py:**

from django.urls import path

from .views import UserView

urlpatterns = [

path('user/', UserView.as\_view(), name="user-all"),

]

1. **views.py:**

from .models import User

from .serializers import UserSerializer

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import status

class UserView(APIView):

"""

This class handles GET and POST actions for the Song resource.

GET - Retrieves a list of all users

POST - Creates a new song

"""

def get(self, request, version, format=None):

user = User.objects.all()

serializer = UserSerializer(user, many=True)

return Response(serializer.data)

**...plus 1 more in config/urls.py makes 7 files we care about in total.**

1. **config/urls.py:**

"""config URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:

https://docs.djangoproject.com/en/2.1/topics/http/urls/

Examples:

Function views

1. Add an import: from my\_app import views

2. Add a URL to urlpatterns: path('', views.home, name='home')

Class-based views

1. Add an import: from other\_app.views import Home

2. Add a URL to urlpatterns: path('', Home.as\_view(), name='home')

Including another URLconf

1. Import the include() function: from django.urls import include, path

2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))

"""

from django.contrib import admin

from django.urls import path, include, re\_path

urlpatterns = [

path('admin/', admin.site.urls),

re\_path('api/(?P<version>(v1|v2))', include('reflections.urls'))

]