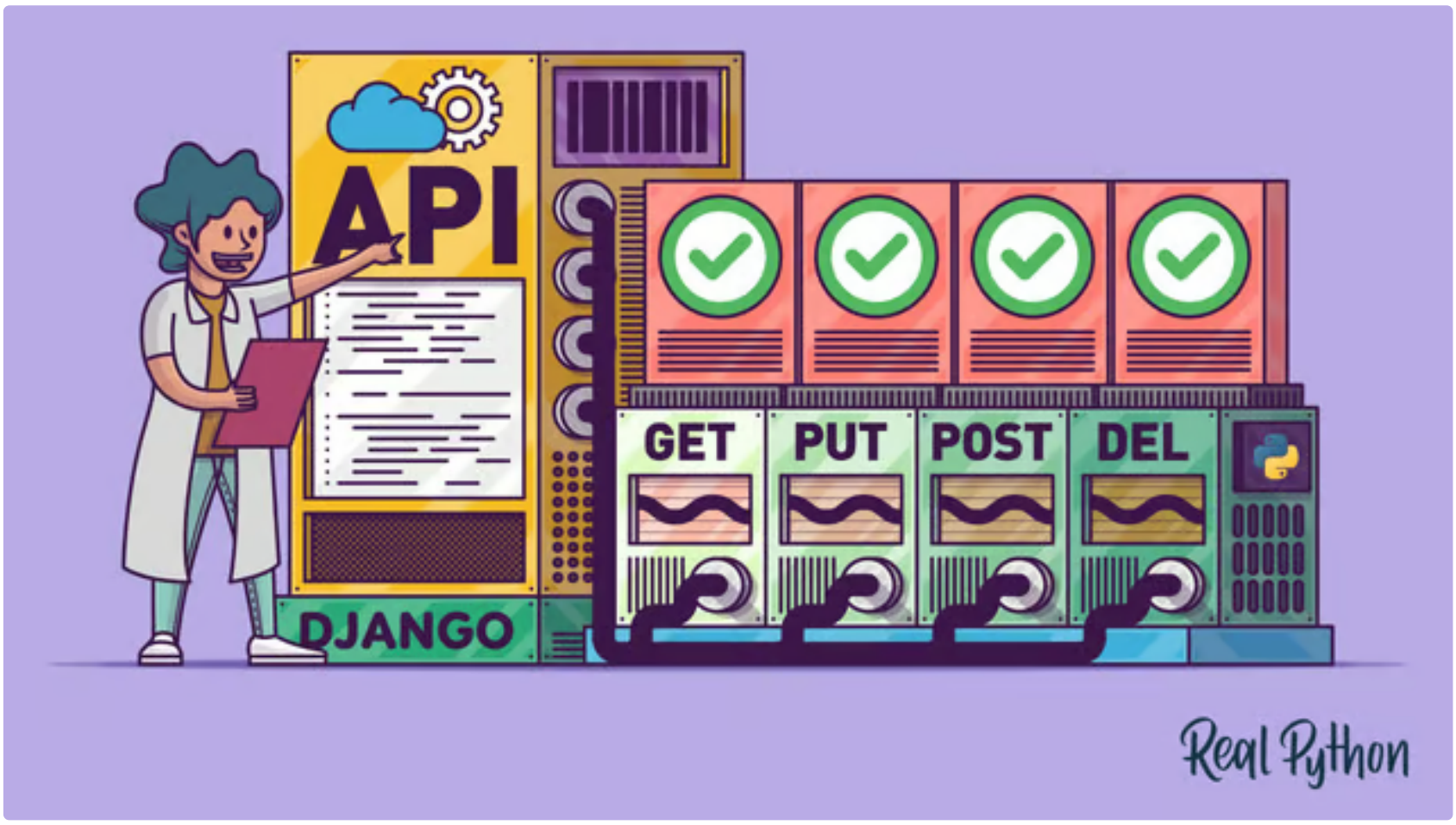

Test Driven Development of a Django RESTful API

by Real Python May 02, 2017 21 Comments

advanced api django testing web-dev

Mark as Completed

Tweet Share Email

Table of Contents

This post walks through the process of developing a CRUD-based RESTful API with Django and Django REST Framework, which is used for rapidly building RESTful APIs based on Django models.

This application uses:

- Python v3.6.0
- Django v1.11.0
- Django REST Framework v3.6.2
- Postgres v9.6.1
- Psycopg2 v2.7.1

> **NOTE:** Check out the third Real Python course for a more in-depth tutorial on Django REST Framework.

Objectives

By the end of this tutorial you will be able to…

1. Discuss the benefits of using Django REST Framework for bootstrapping the development of a RESTful API
2. Validate model querysets using serializers
3. Appreciate Django REST Framework's Browsable API feature for a cleaner and well-documented version of your APIs
4. Practice test-driven development

Why Django REST Framework?

Django REST Framework (REST Framework) provides a number of powerful features out-of-the-box that go well with idiomatic Django, including:

1. Browsable API: Documents your API with a human-friendly HTML output, providing a beautiful form-like interface for submitting data to resources and fetching from them using the standard HTTP methods.
2. Auth Support: REST Framework has rich support for various authentication protocols along with permissions and throttling policies which can be configured on a per-view basis.
3. Serializers: Serializers are an elegant way of validating model querysets/instances and converting them to native Python datatypes that can be easily rendered into JSON and XML.
4. Throttling: Throttling is way to determine whether a request is authorized or not and can be integrated with different permissions. It is generally used for rate limiting API requests from a single user.

Plus, the documentation is easy to read and full of examples. If you're building a RESTful API where you have a one-to-one relationship between your API endpoints and your models, then REST Framework is the way to go.

Django Project Setup

Create and activate a virtualenv:

Shell

```
$ mkdir django-puppy-store
$ cd django-puppy-store
$ python3.6 -m venv env
$ source env/bin/activate
```

Install Django and set up a new project:

Shell

```
(env)$ pip install django==1.11.0
(env)$ django-admin startproject puppy_store
```

Your current project structure should look like this:

```
└── puppy_store
    ├── manage.py
    └── puppy_store
        ├── __init__.py
        ├── settings.py
        ├── urls.py
        └── wsgi.py
```

Django App and REST Framework Setup

Start by creating the `puppies` app and installing REST Framework inside your virtualenv:

Shell

```
(env)$ cd puppy_store
(env)$ python manage.py startapp puppies
(env)$ pip install djangorestframework==3.6.2
```

Now we need to configure our Django project to make use of REST Framework.

First, add the `puppies` app and `rest_framework` to the `INSTALLED_APPS` section within *puppy_store/puppy_store/settings.py*:

Python

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'puppies',
    'rest_framework'
]
```

Next, define global settings for REST Framework in a single dictionary, again, in the *settings.py* file:

Python

```
REST_FRAMEWORK = {
    # Use Django's standard `django.contrib.auth` permissions,
    # or allow read-only access for unauthenticated users.
    'DEFAULT_PERMISSION_CLASSES': [],
    'TEST_REQUEST_DEFAULT_FORMAT': 'json'
}
```

This allows unrestricted access to the API and sets the default test format to JSON for all requests.

> **NOTE:** Unrestricted access is fine for local development, but in a production environment you may need to restrict access to certain endpoints. Make sure to update this. Review the docs for more information.

Your current project structure should now look like:

```
└── puppy_store
    ├── manage.py
    ├── puppies
    │   ├── __init__.py
    │   ├── admin.py
    │   ├── apps.py
    │   ├── migrations
    │   │   └── __init__.py
    │   ├── models.py
    │   ├── tests.py
    │   └── views.py
    └── puppy_store
        ├── __init__.py
        ├── settings.py
        ├── urls.py
        └── wsgi.py
```


Database and Model Setup

Let's set up the Postgres database and apply all the migrations to it.

> **NOTE**: Feel free to swap out Postgres for the relational database of your choice!

Once you have a working Postgres server on your system, open the Postgres interactive shell and create the database:

SQL

```
$ psql
# CREATE DATABASE puppy_store_drf;
CREATE DATABASE
# \q
```

Install psycopg2 so that we can interact with the Postgres server via Python:

Shell

```
(env)$ pip install psycopg2==2.7.1
```

Update the database configuration in *settings.py*, adding the appropriate username and password:

Python

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'puppy_store_drf',
        'USER': '<your-user>',
        'PASSWORD': '<your-password>',
        'HOST': '127.0.0.1',
        'PORT': '5432'
    }
}
```

Next, define a puppy model with some basic attributes in *django-puppy-store/puppy_store/puppies/models.py*:

Python

```python
from django.db import models


class Puppy(models.Model):
    """
    Puppy Model
    Defines the attributes of a puppy
    """
    name = models.CharField(max_length=255)
    age = models.IntegerField()
    breed = models.CharField(max_length=255)
    color = models.CharField(max_length=255)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def get_breed(self):
        return self.name + ' belongs to ' + self.breed + ' breed.'

    def __repr__(self):
        return self.name + ' is added.'
```

Now apply the migration:

Shell

```
(env)$ python manage.py makemigrations
(env)$ python manage.py migrate
```

Sanity Check

Hop into `psql` again and verify that the `puppies_puppy` has been created:

SQL

```
$ psql
# \c puppy_store_drf
You are now connected to database "puppy_store_drf".
puppy_store_drf=# \dt
                      List of relations
 Schema |             Name              | Type  |     Owner
--------+-------------------------------+-------+----------------
 public | auth_group                    | table | michael.herman
 public | auth_group_permissions        | table | michael.herman
 public | auth_permission               | table | michael.herman
 public | auth_user                     | table | michael.herman
 public | auth_user_groups              | table | michael.herman
 public | auth_user_user_permissions    | table | michael.herman
 public | django_admin_log              | table | michael.herman
 public | django_content_type           | table | michael.herman
 public | django_migrations             | table | michael.herman
 public | django_session                | table | michael.herman
 public | puppies_puppy                 | table | michael.herman
(11 rows)
```

> **NOTE:** You can run `\d+ puppies_puppy` if you want to look at the actual table details.

Before moving on, let's write a quick unit test for the Puppy model.

Add the following code to a new file called *test_models.py* in a new folder called "tests" within "django-puppy-store/puppy_store/puppies":

Python

```python
from django.test import TestCase
from ..models import Puppy


class PuppyTest(TestCase):
    """ Test module for Puppy model """

    def setUp(self):
        Puppy.objects.create(
            name='Casper', age=3, breed='Bull Dog', color='Black')
        Puppy.objects.create(
            name='Muffin', age=1, breed='Gradane', color='Brown')

    def test_puppy_breed(self):
        puppy_casper = Puppy.objects.get(name='Casper')
        puppy_muffin = Puppy.objects.get(name='Muffin')
        self.assertEqual(
            puppy_casper.get_breed(), "Casper belongs to Bull Dog breed.")
        self.assertEqual(
            puppy_muffin.get_breed(), "Muffin belongs to Gradane breed.")
```

In the above test, we added dummy entries into our puppy table via the `setUp()` method from `django.test.TestCase` and asserted that the `get_breed()` method returned the correct string.

Add an *__init__.py* file to “tests” and remove the *tests.py* file from “django-puppy-store/puppy_store/puppies”.

Let’s run our first test:

Shell

```
(env)$ python manage.py test
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.007s

OK
Destroying test database for alias 'default'...
```

Great! Our first unit test has passed!

Serializers

Before moving on to creating the actual API, let’s define a serializer for our Puppy model which validates the model querysets and produces Pythonic data types to work with.

Add the following snippet to *django-puppy-store/puppy_store/puppies/serializers.py*:

Python

```python
from rest_framework import serializers
from .models import Puppy


class PuppySerializer(serializers.ModelSerializer):
    class Meta:
        model = Puppy
        fields = ('name', 'age', 'breed', 'color', 'created_at', 'updated_at')
```

In the above snippet we defined a `ModelSerializer` for our puppy model, validating all the mentioned fields. In short, if you have a one-to-one relationship between your API endpoints and your models - which you probably should if you're creating a RESTful API - then you can use a ModelSerializer to create a Serializer.

With our database in place, we can now start building the RESTful API…

RESTful Structure

In a RESTful API, endpoints (URLs) define the structure of the API and how end users access data from our application using the HTTP methods - GET, POST, PUT, DELETE. Endpoints should be logically organized around *collections* and *elements*, both of which are resources.

In our case, we have one single resource, `puppies`, so we will use the following URLS - `/puppies/` and `/puppies/<id>` for collections and elements, respectively:

Endpoint	HTTP Method	CRUD Method	Result
`puppies`	GET	READ	Get all puppies
`puppies/:id`	GET	READ	Get a single puppy
`puppies`	POST	CREATE	Add a single puppy
`puppies/:id`	PUT	UPDATE	Update a single puppy
`puppies/:id`	DELETE	DELETE	Delete a single puppy

Routes and Testing (TDD)

We will be taking a test-first approach rather than a thorough test-driven approach, wherein we will be going through the following process:

- add a unit test, just enough code to fail
- then update the code to make it pass the test.

Once the test passes, start over with the same process for the new test.

Begin by creating a new file, *django-puppy-store/puppy_store/puppies/tests/test_views.py*, to hold all the tests for our views and create a new test client for our app:

Python

```
import json
from rest_framework import status
from django.test import TestCase, Client
from django.urls import reverse
from ..models import Puppy
from ..serializers import PuppySerializer


# initialize the APIClient app
client = Client()
```

Before starting with all the API routes, let's first create a skeleton of all view functions that return empty responses and map them with their appropriate URLs within the *django-puppy-store/puppy_store/puppies/views.py* file:

Python

```python
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from .models import Puppy
from .serializers import PuppySerializer


@api_view(['GET', 'DELETE', 'PUT'])
def get_delete_update_puppy(request, pk):
    try:
        puppy = Puppy.objects.get(pk=pk)
    except Puppy.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    # get details of a single puppy
    if request.method == 'GET':
        return Response({})
    # delete a single puppy
    elif request.method == 'DELETE':
        return Response({})
    # update details of a single puppy
    elif request.method == 'PUT':
        return Response({})


@api_view(['GET', 'POST'])
def get_post_puppies(request):
    # get all puppies
    if request.method == 'GET':
        return Response({})
    # insert a new record for a puppy
    elif request.method == 'POST':
        return Response({})
```

Create the respective URLs to match the views in *django-puppy-store/puppy_store/puppies/urls.py*:

Python

```python
from django.conf.urls import url
from . import views


urlpatterns = [
    url(
        r'^api/v1/puppies/(?P<pk>[0-9]+)$',
        views.get_delete_update_puppy,
        name='get_delete_update_puppy'
    ),
    url(
        r'^api/v1/puppies/$',
        views.get_post_puppies,
        name='get_post_puppies'
    )
]
```

Update *django-puppy-store/puppy_store/puppy_store/urls.py* as well:

Python

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^', include('puppies.urls')),
    url(
        r'^api-auth/',
        include('rest_framework.urls', namespace='rest_framework')
    ),
    url(r'^admin/', admin.site.urls),
]
```


Browsable API

With all routes now wired up with the view functions, let's open up REST Framework's Browsable API interface and verify whether all the URLs are working as expected.

First, fire up the development server:

Shell

```
(env)$ python manage.py runserver
```

Make sure to comment out all the attributes in `REST_FRAMEWORK` section of our `settings.py` file, to bypass login. Now visit `http://localhost:8000/api/v1/puppies`

You will see an interactive HTML layout for the API response. Similarly we can test the other URLs and verify all URLs are working perfectly fine.

Let's start with our unit tests for each route.

Routes

GET ALL

Start with a test to verify the fetched records:

Python

```python
class GetAllPuppiesTest(TestCase):
    """ Test module for GET all puppies API """

    def setUp(self):
        Puppy.objects.create(
            name='Casper', age=3, breed='Bull Dog', color='Black')
        Puppy.objects.create(
            name='Muffin', age=1, breed='Gradane', color='Brown')
        Puppy.objects.create(
            name='Rambo', age=2, breed='Labrador', color='Black')
        Puppy.objects.create(
            name='Ricky', age=6, breed='Labrador', color='Brown')

    def test_get_all_puppies(self):
        # get API response
        response = client.get(reverse('get_post_puppies'))
        # get data from db
        puppies = Puppy.objects.all()
        serializer = PuppySerializer(puppies, many=True)
        self.assertEqual(response.data, serializer.data)
        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

Run the test. You should see the following error:

Shell

```
self.assertEqual(response.data, serializer.data)
AssertionError: {} != [OrderedDict([('name', 'Casper'), ('age',[687 chars])])]
```

Update the view to get the test to pass.

Python

```python
@api_view(['GET', 'POST'])
def get_post_puppies(request):
    # get all puppies
    if request.method == 'GET':
        puppies = Puppy.objects.all()
        serializer = PuppySerializer(puppies, many=True)
        return Response(serializer.data)
    # insert a new record for a puppy
    elif request.method == 'POST':
        return Response({})
```

Here, we get all the records for puppies and validate each using the `PuppySerializer`.

Run the tests to ensure they all pass:

Shell

```
Ran 2 tests in 0.072s

OK
```

GET Single

Fetching a single puppy involves two test cases:

1. Get valid puppy - e.g., the puppy exists
2. Get invalid puppy - e.g., the puppy does not exists

Add the tests:

Python

```
class GetSinglePuppyTest(TestCase):
    """ Test module for GET single puppy API """

    def setUp(self):
        self.casper = Puppy.objects.create(
            name='Casper', age=3, breed='Bull Dog', color='Black')
        self.muffin = Puppy.objects.create(
            name='Muffin', age=1, breed='Gradane', color='Brown')
        self.rambo = Puppy.objects.create(
            name='Rambo', age=2, breed='Labrador', color='Black')
        self.ricky = Puppy.objects.create(
            name='Ricky', age=6, breed='Labrador', color='Brown')

    def test_get_valid_single_puppy(self):
        response = client.get(
            reverse('get_delete_update_puppy', kwargs={'pk': self.rambo.pk}))
        puppy = Puppy.objects.get(pk=self.rambo.pk)
        serializer = PuppySerializer(puppy)
        self.assertEqual(response.data, serializer.data)
        self.assertEqual(response.status_code, status.HTTP_200_OK)

    def test_get_invalid_single_puppy(self):
        response = client.get(
            reverse('get_delete_update_puppy', kwargs={'pk': 30}))
        self.assertEqual(response.status_code, status.HTTP_404_NOT_FOUND)
```

Run the tests. You should see the following error:

Shell

```
self.assertEqual(response.data, serializer.data)
AssertionError: {} != {'name': 'Rambo', 'age': 2, 'breed': 'Labr[109 chars]26Z'}
```

Update the view:

Python

```
@api_view(['GET', 'UPDATE', 'DELETE'])
def get_delete_update_puppy(request, pk):
    try:
        puppy = Puppy.objects.get(pk=pk)
    except Puppy.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    # get details of a single puppy
    if request.method == 'GET':
        serializer = PuppySerializer(puppy)
        return Response(serializer.data)
```

In the above snippet, we get the puppy using an ID. Run the tests to ensure they all pass.

POST

Inserting a new record involves two cases as well:

1. Inserting a valid puppy
2. Inserting a invalid puppy

First, write tests for it:

Python

```
class CreateNewPuppyTest(TestCase):
    """ Test module for inserting a new puppy """

    def setUp(self):
        self.valid_payload = {
            'name': 'Muffin',
            'age': 4,
            'breed': 'Pamerion',
            'color': 'White'
        }
        self.invalid_payload = {
            'name': '',
            'age': 4,
            'breed': 'Pamerion',
            'color': 'White'
        }

    def test_create_valid_puppy(self):
        response = client.post(
            reverse('get_post_puppies'),
            data=json.dumps(self.valid_payload),
            content_type='application/json'
        )
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)

    def test_create_invalid_puppy(self):
        response = client.post(
            reverse('get_post_puppies'),
            data=json.dumps(self.invalid_payload),
            content_type='application/json'
        )
        self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
```

Run the tests. You should see two failures:

Shell

```
self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
AssertionError: 200 != 400

self.assertEqual(response.status_code, status.HTTP_201_CREATED)
AssertionError: 200 != 201
```

Again, update the view to get the tests to pass:

Python

```
@api_view(['GET', 'POST'])
def get_post_puppies(request):
    # get all puppies
    if request.method == 'GET':
        puppies = Puppy.objects.all()
        serializer = PuppySerializer(puppies, many=True)
        return Response(serializer.data)
    # insert a new record for a puppy
    if request.method == 'POST':
        data = {
            'name': request.data.get('name'),
            'age': int(request.data.get('age')),
            'breed': request.data.get('breed'),
            'color': request.data.get('color')
        }
        serializer = PuppySerializer(data=data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Here, we inserted a new record by serializing and validating the request data before inserting to the database.

Run the tests again to ensure they pass.

You can also test this out with the Browsable API. Fire up the development server again, and navigate to http://localhost:8000/api/v1/puppies/. Then, within the POST form, submit the following as `application/json`:

JSON

```
{
    "name": "Muffin",
    "age": 4,
    "breed": "Pamerion",
    "color": "White"
}
```

Be sure the GET ALL and Get Single work as well.

PUT

Start with a test to update a record. Similar to adding a record, we again need to test for both valid and invalid updates:

Python

```
class UpdateSinglePuppyTest(TestCase):
    """ Test module for updating an existing puppy record """

    def setUp(self):
        self.casper = Puppy.objects.create(
            name='Casper', age=3, breed='Bull Dog', color='Black')
        self.muffin = Puppy.objects.create(
            name='Muffy', age=1, breed='Gradane', color='Brown')
        self.valid_payload = {
            'name': 'Muffy',
            'age': 2,
            'breed': 'Labrador',
            'color': 'Black'
        }
        self.invalid_payload = {
            'name': '',
            'age': 4,
            'breed': 'Pamerion',
            'color': 'White'
        }

    def test_valid_update_puppy(self):
        response = client.put(
            reverse('get_delete_update_puppy', kwargs={'pk': self.muffin.pk}),
            data=json.dumps(self.valid_payload),
            content_type='application/json'
        )
        self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)

    def test_invalid_update_puppy(self):
        response = client.put(
            reverse('get_delete_update_puppy', kwargs={'pk': self.muffin.pk}),
            data=json.dumps(self.invalid_payload),
            content_type='application/json')
        self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
```

Run the tests.

Shell

```
self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
AssertionError: 405 != 400

self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)
AssertionError: 405 != 204
```

Update the view:

Python

```python
@api_view(['GET', 'DELETE', 'PUT'])
def get_delete_update_puppy(request, pk):
    try:
        puppy = Puppy.objects.get(pk=pk)
    except Puppy.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    # get details of a single puppy
    if request.method == 'GET':
        serializer = PuppySerializer(puppy)
        return Response(serializer.data)

    # update details of a single puppy
    if request.method == 'PUT':
        serializer = PuppySerializer(puppy, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_204_NO_CONTENT)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    # delete a single puppy
    elif request.method == 'DELETE':
        return Response({})
```

In the above snippet, similar to an insert, we serialize and validate the request data and then respond appropriately.

Run the tests again to ensure that all the tests pass.

DELETE

To delete a single record, an ID is required:

Python

```python
class DeleteSinglePuppyTest(TestCase):
    """ Test module for deleting an existing puppy record """

    def setUp(self):
        self.casper = Puppy.objects.create(
            name='Casper', age=3, breed='Bull Dog', color='Black')
        self.muffin = Puppy.objects.create(
            name='Muffy', age=1, breed='Gradane', color='Brown')

    def test_valid_delete_puppy(self):
        response = client.delete(
            reverse('get_delete_update_puppy', kwargs={'pk': self.muffin.pk}))
        self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)

    def test_invalid_delete_puppy(self):
        response = client.delete(
            reverse('get_delete_update_puppy', kwargs={'pk': 30}))
        self.assertEqual(response.status_code, status.HTTP_404_NOT_FOUND)
```

Run the tests. You should see:

Shell

```
self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)
AssertionError: 200 != 204
```

Update the view:

Python

```python
@api_view(['GET', 'DELETE', 'PUT'])
def get_delete_update_puppy(request, pk):
    try:
        puppy = Puppy.objects.get(pk=pk)
    except Puppy.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    # get details of a single puppy
    if request.method == 'GET':
        serializer = PuppySerializer(puppy)
        return Response(serializer.data)

    # update details of a single puppy
    if request.method == 'PUT':
        serializer = PuppySerializer(puppy, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_204_NO_CONTENT)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    # delete a single puppy
    if request.method == 'DELETE':
        puppy.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

Run the tests again. Make sure all of them pass. Make sure to test out the UPDATE and DELETE functionality within the Browsable API as well!

Conclusion and Next Steps

In this tutorial, we went through the process of creating a RESTful API using Django REST Framework with a test-first approach.

> **Free Bonus: Click here to download a copy of the "REST API Examples" Guide** and get a hands-on introduction to Python + REST API principles with actionable examples.

What's next? To make our RESTful API robust and secure, we can implement permissions and throttling for a production environment to allow restricted access on the basis of authentication credentials and rate limiting to avoid any sort of DDoS attack. Also, don't forget to prevent the Browsable API from being accessible in a production environment.

Feel free to share your comments, questions, or tips in the comments below. The full code can be found in the django-puppy-store repository.