Learning Objectives

- **Define unit tests**
- **Create tests with APIClient**
- **Define the importance of the `setup` method in tests**
- **Write tests for `GET`, `PUT`, and `POST` requests**
- **Write a test to evaluate authentication**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Intro

Testing Django Rest Framework with Mocks

Intro

Welcome to Course 3 of Advanced Django. We'll continue building the Blango application, with a focus on Django Rest Framework still. We'll start by looking at how to test your DRF endpoints and then look at some DRF advanced features and plugins. We'll finish the course by implementing an interactive JavaScript frontend using React. Let's get started with testing.

Testing & Django

Unit testing in general

Writing tests for our code is something we all should be doing, and many of us say that we're doing. But for many teams and individuals it's a "do it later" thing. Before we talk about unit testing with Django and DRF, we'll give a brief introduction to unit testing in general, just to be sure you're up to speed.

Unit testing is the process of writing automated tests for your code. You might write your code to perform in a certain way, but you can't be sure it's working right until you verify that the output is correct for the given inputs. There are many ways you can write unit tests, but we'll stick with the built-in Django tools.

Unit tests are normally grouped into *test cases*, represented by a `TestCase` subclass. Each method in the class might test a different aspect of particular functionality, or be written for particular input. The method name must start with `test_`, so you can use objected-oriented principles to call helper methods or use inheritance, provided the other method's name does not start with `test_`.

The actual validation of results is done using *assertions*. `TestCase` provides a number of assertion methods that can be called for various checks. For example, `assertEqual()`, `assertNotEqual()`, `assertTrue()`, `assertFalse()`, `assertIn()` (for example, check if an item is in a list or dictionary), `assertNotIn()` (the opposite), and more. If you can't find an assertion you need, you can always fall back to writing a test and wrapping it in `assertTrue()`.

Here's a basic example that showcases the fundamentals. We've implemented an `is_even` function that returns `True` if an even number is passed in, or `False` for an odd number.

```
def is_even(n):
    return n % 2 == 0
```

Here's how we might test it:

```
from django.test import TestCase

class IsEvenTestCase(TestCase):
    def test_even_number(self):
        self.assertTrue(is_even(4))

    def test_odd_number(self):
        self.assertFalse(is_even(3))
```

While this is a simple example, you can see how we're grouping tests using `TestCase` (all the tests for `is_even` are in one class), how we split the test methods up (one for the "true" result and one for the "false"), and how we're using assertions.

▼ **TestCase Class**

The `TestCase` class also has methods that can be implemented to set up the environment before and after each test. For example, you could insert data into the database before each test. We'll cover some of these later.

Writing unit tests can get a lot more complicated, especially when you're trying to figure out how to "trick" Python into simulating reality during a test. Regardless, that's enough about fundamentals. Now, let's look at some Django specific testing.

Testing & Django

Hopefully as part of previous Django projects you've worked on, you've had a chance to write some tests. If not, here's a brief look at one aspect of Django testing that we're going to focus on: the test client.

To make requests to your Django views in a test, Django provides a test client class (`django.test.Client`). It's special in that it doesn't require the Django development server to be running. It actually makes its requests directly through the framework to the views. This is usually pretty good as it makes the tests faster by avoiding HTTP overheads. However some parts of your project might not work as expected. For instance, there are usually issues with static file and media serving.

The auto graders for the coding assessments throughout the Advanced Django courses are actually Django unit tests that make use of the test client. Here's some example code showing it in action:

```
from django.test import TestCase

class ExampleTestCase(TestCase):
    def test_invalid_login(self):
        bad_resp = self.client.post(
            "/accounts/login/", {"username": "bad", "password":
        "bad"}
        )
        self.assertEqual(bad_resp.status_code, 200)
        self.assertIn(b"Please enter a correct username and
        password", bad_resp.content)


    def test_valid_login(self):
        good_resp = self.client.post(
            "/accounts/login/", {"username": "testuser",
        "password": "password"}
        )


        self.assertEqual(good_resp.status_code, 302)
        self.assertEqual(good_resp.headers["Location"],
        "/accounts/profile/")
```

To explain this code a bit, the `django.test.TestCase` class instantiates a `django.test.Client` when it is run, and assigns this to its `client` attribute. You could of course instantiate more clients on your own if you wanted.

The `Client` instance is then used to make two `POST` requests. The first, with a bad username and password. In this test, we're expecting the login to fail. If it does, we should expect a `200` `status_code` and some error text in the response `content`.

In the second test, we expect the login to succeed. If it doesn't succeed, we'd expect a `302` status code (indicating a redirect), and then a `Location`(to redirect to) in the response headers to the profile page.

Now that we have seen how to test in Django, with the test client, how does that apply to Django Rest Framework?

We can use the standard Django test Client to test our API, we just need to add some additional parameters when making requests.

Making `GET` requests actually works pretty much as you expect. You can decode the response's `content` using the `json()` method and then see if it matches what you expect. For example, to test our `Post` lists endpoint returns a list, you could do something like this:

```
class RestTestCase(TestCase):
    def test_post_list(self):
        resp = self.client.get("/api/v1/posts/")
        data = resp.json()
        self.assertIsInstance(data, list)
```

Also note the use of a new assertion method, `assertIsInstance()`.

To make requests that send data in the body, we need to encode the data to JSON (using `json.dumps()` or similar), and then set the `content_type` of the request to `application/json`. If we need to use authorization, we set the `Authorization` HTTP header with the `HTTP_AUTHORIZATION` keyword argument.

```
class RestTestCase(TestCase):
    def test_post_create(self):
        resp = self.client.post(
            "/api/v1/posts/",
            json.dumps({"content": "Post Content", "slug":
        "post-slug", ...}),
            HTTP_AUTHORIZATION="Token abc1234def567",
            content_type="application/json",
        )
        data = resp.json()
        self.assertEqual(data["slug"], "post-slug")
```

This method of testing works, but DRF includes some helpers to make it a bit easier.

▼ **RequestFactory Class**

Django can also be tested using the `RequestFactory` class. This class can generate `Request` objects that can be directly passed to views to test them. This means you don't have to go through the Django URL mapping, and allows you to write tests without hard-coding URLs. It also allows you to treat view functions/classes as "black boxes" which you can test with exactly known inputs and outputs. The official Django documentation for `RequestFactory` is available here. DRF provides a subclass of `RequestFactory` called `APIRequestFactory`, which is documented here. We'd recommend coming back to `RequestFactory` after you're familiar with testing using the test client.

Testing Gets with APIClient

Testing `GET`s with `APIClient`

Instead of using Django's standard client, DRF provides a client with a similar interface we can use in its place. It's called `APIClient`, importable from `rest_framework.test`. It has a few extra features that cut down the amount of code to write when testing APIs.

We'll also be introducing the `setUp()` method. This is called before each test method is called, and can be used for setting up the test class or inserting data into the test database. Django uses an in-memory SQLite database that's cleared before each test is run, so you don't have to worry about the tests messing with the data in your database. You know what state it will be in which makes your test much easier to compose.

Here's an example of the use of `setUp()` on `TestCase`. We use the `setUp()` method to insert test users and `Post`s in the database. It also sets the `client` attribute to an instance of DRF's `APIClient`.

Then, we fetch all the `Post`s from the API in the `test_post_list()` method using the `get()` method on the client, extracting the data using the `json()` method. We verify that two `Post`s are returned in the list, and that each of their attributes match what's in the database.

```
class PostApiTestCase(TestCase):
    def setUp(self):
        self.u1 = get_user_model().objects.create_user(
            email="test@example.com", password="password"
        )

        self.u2 = get_user_model().objects.create_user(
            email="test2@example.com", password="password2"
        )

        posts = [
            Post.objects.create(
                author=self.u1,
                published_at=timezone.now(),
                title="Post 1 Title",
                slug="post-1-slug",
                summary="Post 1 Summary",
                content="Post 1 Content",
            ),
            Post.objects.create(
                author=self.u2,
```

```
                published_at=timezone.now(),
                title="Post 2 Title",
                slug="post-2-slug",
                summary="Post 2 Summary",
                content="Post 2 Content",
            ),
        ]

        # let us look up the post info by ID
        self.post_lookup = {p.id: p for p in posts}

        # override test client
        self.client = APIClient()
        self.token = Token.objects.create(user=self.u1)

    def test_post_list(self):
        resp = self.client.get("/api/v1/posts/")
        data = resp.json()
        self.assertEqual(len(data), 2)

        for post_dict in data:
            post_obj = self.post_lookup[post_dict["id"]]
            self.assertEqual(post_obj.title, post_dict["title"])
            self.assertEqual(post_obj.slug, post_dict["slug"])
            self.assertEqual(post_obj.summary,
        post_dict["summary"])
            self.assertEqual(post_obj.content,
        post_dict["content"])
            self.assertTrue(

        post_dict["author"].endswith(f"/api/v1/users/{post_obj.author.email}"

            )
            self.assertEqual(
                post_obj.published_at,
                datetime.fromisoformat(
                    post_dict["published_at"].replace("Z",
        "+00:00")
                ),
            )
```

Look at how we're testing the author. Since we've made it a `HyperlinkedRelatedField` it will come back as a URL. The URL actually has the hostname `testserver` as we're executing against a mock server, so the URL will be something like `http://testserver/api/v1/users/test@example.com`. We might sometime, in the future, want to execute against a real server, which would mean a different URL would be generated. In order to not have to change our test, we can just check that the URL `endswith()` the path we expect, so the hostname doesn't matter.

Authenticating with APIClient

Authenticating with `APIClient`

The next thing we'd want to test are requests that send data, like `POST`s or `PUT`s. However, as you know, we have to authenticate before we can send these kinds of requests to our API. If we were to write a test using these methods without first authenticating, we'd just get `401 Unauthorized` status responses back.

There are a couple of ways we can authenticate in our tests. Since the `APIClient` inherits from the base `TestClient`, we can use the `login()` method to log our client in. This stores the login information in the session, so this method only works if you have `SessionAuthentication` enabled for DRF (which we do).

Here's how we would log in, then make a request:

```
class PostApiTestCase(TestCase):
    def test_post_creation(self):
        self.client.login(email="test@example.com",
        password="test@example.com")
        self.client.post("/api/v1/posts/", {"content": ...})
```

Depending on what you're testing, you could choose to login in the `setUp()` method. Your client would be authenticated and ready to go in each test method.

However, you might want to test that your API is configured to return forbidden or unauthorized responses if a user isn't logged in. You have two options for this:

- You can manually `login()` in methods that should be authenticated.
- Add the `login()` call to `setUp()`, then Call `logout()` on the client in the methods that you need to test unauthorized responses for:

```
class PostApiTestCase(TestCase):
    def setUp(self):
        self.client.login(email="test@example.com",
        password="test@example.com")


    def test_post_creation_unauthorized(self):
        # log out to test response on unauthorized clients
        self.client.logout()
        resp = self.client.post("/api/v1/posts/", {"content":
        ...})
        self.assertEqual(resp.status_code, 401)
```

The method you choose will depend on how many authorized vs. unauthorized tests you have – make the default the one that leads you to writing less code!

As well as using `login()` for sessions authentication we can also use basic or token authentication, by using the `client.credentials()` method. This allows us to set HTTP headers that will be included on each request.

For basic authorization, the username and password are base-64 encoded and then included in the `Authorization` HTTP header with the prefix `Basic`.

Here's how we would generate the header for one of our test users:

```
import base64

credentials =
        base64.b64encode("test@example.com:password".encode("ascii"))


auth_header = "Basic " + credentials.decode("ascii")
```

Then we could set the credentials in the `setUp()` method. We can apply them to the `Authorization` header with the `HTTP_AUTHORIZATION` keyword argument, passed to the client's `credentials()` method:

```
class PostApiTestCase(TestCase):
    def setUp(self):
        # assume we have the auth_header variable in scope
        self.client.credentials(HTTP_AUTHORIZATION=auth_header)
```

To unset credentials, we just call `credentials()` with no arguments. Once again, it's up to you to decide if you want to set up the credentials in the `setUp()` method and then unset them by calling `credentials()` with no arguments, or if you only want to log in with `credentials()` in some of your methods.

Finally let's look at authenticating with a token.

First, we'll start by assuming you already have a token, we'll discuss different ways of getting one in a moment. We can use the `credentials()` method in a similar way, except this time `HTTP_AUTHORIZATION` will be a string starting with `Token`.

For example:

```
token =
        "4510a3fdd351d2a35059e9724fa4bdbb643f4325cd8f6696298f80efbaf4d2
        c9"

auth_header = "Token " + token
client.credentials(HTTP_AUTHORIZATION=auth_header)
```

You could get a token by posting to an API authentication URL that you've set up (for example, `/api/v1/token-auth/` in our case). But, that means your tests are dependent on a view that you may or may not have configured correctly.

When testing, you should try to remove as many unknowns as possible. If, for some reason, your authentication endpoint has been misconfigured this would mean all your tests that required authorization would fail. It could take you a long time to track down the reason. Instead, we should get a token in some other manner, and have a separate test just for our token authentication endpoint.

The better way of getting a token to use in tests is by generating one for the user and saving it to the database, then referring to it directly from the model instance. We can save this reference on our test class and then use it in our tests. Our `setUp` method would be modified like so:

```
class PostApiTestCase(TestCase):
    def setUp(self):
        self.u1 = get_user_model().objects.create_user(
            email="test@example.com", password="password"
        )

        self.client = APIClient()
        token = Token.objects.create(user=self.u1)
        self.client.credentials(HTTP_AUTHORIZATION='Token ' +
token.key)
```

All our API requests will then use token authentication as the user `test@example.com`.

Now, although you've seen them used a few times in the past few examples, let's talk a little about sending data.

Sending Data with APIClient

Sending data with `APIClient`

There isn't too much to say about the `post()` and `put()` methods on `APIClient`. You can pass data as a dictionary and the client will automatically take care of the encoding for you. Here's a short example test that creates a `Post` using the API, then fetches it from the database and checks that it matches the data that was posted. While it's not shown here, the `setUp()` method instantiated the client and authenticated with the `credentials()` method. The other test method logs out the client (by calling `credentials()` with no argument) and then checks that a `401` status code is returned.

```
class PostApiTestCase(TestCase):
    def test_unauthenticated_post_create(self):
        # unset credentials so we are an anonymous user
        self.client.credentials()
        resp = self.client.post("/api/v1/posts/", {})
        self.assertEqual(resp.status_code, 401)

    def test_post_create(self):
        post_dict = {
            "title": "Test Post",
            "slug": "test-post-3",
            "summary": "Test Summary",
            "content": "Test Content",
            "author":
        "http://testserver/api/v1/users/test@example.com",
            "published_at": "2021-01-10T09:00:00Z"
        }
        resp = self.client.post("/api/v1/posts/", post_dict)

        post_id = resp.json()["id"]
        post = Post.objects.get(id=post_id)
        self.assertEqual(post.title, post_dict["title"])
        self.assertEqual(post.slug, post_dict["slug"])
        self.assertEqual(post.summary, post_dict["summary"])
        self.assertEqual(post.content, post_dict["content"])
        self.assertEqual(post.author, self.u1)
        self.assertEqual(post.published_at, datetime(2021, 1,
        10, 9, 0, 0, tzinfo=UTC))
```

Try It Out

Try It Out

Let's set up a few tests for Blango. Start by creating a file in the `blog` app called `test_post_api.py`. Add these imports to the file:

```
from datetime import datetime

from django.contrib.auth import get_user_model
from django.test import TestCase
from django.utils import timezone
from pytz import UTC
from rest_framework.authtoken.models import Token
from rest_framework.test import APIClient

from blog.models import Post
```

There's a few of them, but we'll be using them all.

Next, define a class called `PostApiTestCase` which inherits from `TestCase`:

```
class PostApiTestCase(TestCase):
```

We'll go through the methods to implement one by one, and talk about what they do, starting with `setUp`. This is called before each `test_` method and does the following:

- Creates two test users and assigns them to `self.u1` and `self.u2`.
- Creates two `Post` objects. It creates a dictionary with a mapping between each post's ID and the object so we can look up the `Post` by ID later (`post_lookup`).
- Replaces the Django Test Client instance with an `APIClient` instance.
- Inserts a `Token` object into the database (which generates a `key` for authentication). The `Token` is for the `u1` user.
- Sets the `credentials()` of the `APIClient` client to use the token in the HTTP `Authorization` header.

Here's the `setUp()` method – add it to your `PostApiTestCase` class.

```
    def setUp(self):
        self.u1 = get_user_model().objects.create_user(
            email="test@example.com", password="password"
        )

        self.u2 = get_user_model().objects.create_user(
            email="test2@example.com", password="password2"
        )

        posts = [
            Post.objects.create(
                author=self.u1,
                published_at=timezone.now(),
                title="Post 1 Title",
                slug="post-1-slug",
                summary="Post 1 Summary",
                content="Post 1 Content",
            ),
            Post.objects.create(
                author=self.u2,
                published_at=timezone.now(),
                title="Post 2 Title",
                slug="post-2-slug",
                summary="Post 2 Summary",
                content="Post 2 Content",
            ),
        ]

        # let us look up the post info by ID
        self.post_lookup = {p.id: p for p in posts}

        # override test client
        self.client = APIClient()
        token = Token.objects.create(user=self.u1)
        self.client.credentials(HTTP_AUTHORIZATION="Token " +
        token.key)
```

Next, here's the `test_post_list()` method. We already explained earlier how the test works. It queries the `Post` objects that were inserted in `setUp()`, using the API, and then checks that their data matches what we expect.

Add the `test_post_list()` method to your `PostApiTestCase` class:

```
    def test_post_list(self):
        resp = self.client.get("/api/v1/posts/")
        data = resp.json()
        self.assertEqual(len(data), 2)

        for post_dict in data:
            post_obj = self.post_lookup[post_dict["id"]]
            self.assertEqual(post_obj.title, post_dict["title"])
            self.assertEqual(post_obj.slug, post_dict["slug"])
            self.assertEqual(post_obj.summary,
        post_dict["summary"])
            self.assertEqual(post_obj.content,
        post_dict["content"])
            self.assertTrue(

        post_dict["author"].endswith(f"/api/v1/users/{post_obj.author.email}"


            )
            self.assertEqual(
                post_obj.published_at,
                datetime.strptime(
                    post_dict["published_at"], "%Y-%m-
        %dT%H:%M:%S.%fZ"
                ).replace(tzinfo=UTC),
            )
```

Next is a method to test what happens if an unauthenticated user tries to create a `Post`. In order to simulate an unauthenticated user we call `credentials()` on the client with no arguments, which removes the saved `Authorization` header. We expect our API to respond with a `401 Unauthorized` HTTP status code in that case. We also expect that no new `Post` object is created, so we check that there are still 2 in the database. Add this method:

```
    def test_unauthenticated_post_create(self):
        # unset credentials so we are an anonymous user
        self.client.credentials()
        post_dict = {
            "title": "Test Post",
            "slug": "test-post-3",
            "summary": "Test Summary",
            "content": "Test Content",
            "author":
        "http://testserver/api/v1/users/test@example.com",
            "published_at": "2021-01-10T09:00:00Z",
        }
        resp = self.client.post("/api/v1/posts/", post_dict)
        self.assertEqual(resp.status_code, 401)
        self.assertEqual(Post.objects.all().count(), 2)
```

The final test method creates a `Post` through the API, then queries the database for it using the `id` that was returned. It then checks that the data in the database matches what was posted.

```
    def test_post_create(self):
        post_dict = {
            "title": "Test Post",
            "slug": "test-post-3",
            "summary": "Test Summary",
            "content": "Test Content",
            "author":
        "http://testserver/api/v1/users/test@example.com",
            "published_at": "2021-01-10T09:00:00Z",
        }
        resp = self.client.post("/api/v1/posts/", post_dict)
        post_id = resp.json()["id"]
        post = Post.objects.get(pk=post_id)
        self.assertEqual(post.title, post_dict["title"])
        self.assertEqual(post.slug, post_dict["slug"])
        self.assertEqual(post.summary, post_dict["summary"])
        self.assertEqual(post.content, post_dict["content"])
        self.assertEqual(post.author, self.u1)
        self.assertEqual(post.published_at, datetime(2021, 1,
        10, 9, 0, 0, tzinfo=UTC))
```

If you're wondering about the order in which the tests are run, it doesn't matter. The `setUp()` method is run before each test so we know the client is always logged in at the start, and the database is empty except for the objects we create during `setUp()`.

Save the file and then run the Django test using the `manage.py` script with the `test` argument. You should see output like this:

```
$ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...
----------------------------------------------------------------
        ------
Ran 3 tests in 0.359s

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

Wrap Up

You might have noticed that writing tests can take a lot of code. We've tested just a couple of aspects of our API and it's much, much longer than the API code itself. But this will give you an idea of how to write tests, and you can decide for yourself how thorough your automated testing needs to be for your application.

In the next section, we're going to look at how to write tests that run against a real HTTP server.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish testing DRF with mocks"
```

- Push to GitHub:

```
git push
```