Learning Objectives

- **Identify some of the benefits of writing tests with `RequestClient`**
- **Write basic authentication and token authentication tests with `RequestClient`**

Clone Blango

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.

RequestClient

`RequestsClient`

Sometimes you might want to run automated tests against a real HTTP server. This could be the Django Dev Server, or a production server like Nginx or Apache.

One reason to do this is to test against a staging environment. You might have some test data that's just too big for the in-memory SQLite database used by Django's testing mechanism. In this case you could have a script that resets a production database (like PostgreSQL or MySQL) and then runs automated tests against it.

Or, perhaps you want to test a REST API that's not written with Django. We can actually write automated tests for any REST API using these techniques.

The class that DRF provides is `rest_framework.test.RequestsClient`. It has the same interface as the `APIClient`, except it takes the full URL rather than just a path. It relies on the third-party *requests* library, which is pretty ubiquitous. You probably already have it installed as a dependency of some of the other libraries we've been using, otherwise it can be installed with `pip`.

In order to demonstrate it, we'll use it in conjunction with the `django.test.LiveServerTestCase`. This works similarly to the `django.test.TestCase` that we used previously, except that it starts up an instance of the Django Dev Server running on `localhost`. It listens on a random port, so the `live_server_url` attribute is set so we can look up what address the Dev Server instance has.

In this short example, we're creating four `Tag` objects and saving them to the database, in the `setUp()` method. Then, we have a test method called `test_tag_list()` which fetches the `Tag` objects from the API and checks that they match what we created.

It's similar to the test we've already written for the `Post` objects, but notice that we're using the `live_server_url` attribute to build the full URL that the `RequestsClient` instance will request.

```
from django.test import LiveServerTestCase
from rest_framework.test import RequestsClient

from blog.models import Tag


class TagApiTestCase(LiveServerTestCase):
    def setUp(self):
        self.tag_values = {"tag1", "tag2", "tag3", "tag4"}
        for t in self.tag_values:
            Tag.objects.create(value=t)
        self.client = RequestsClient()

    def test_tag_list(self):
        resp = self.client.get(self.live_server_url +
        "/api/v1/tags/")
        self.assertEqual(resp.status_code, 200)
        data = resp.json()
        self.assertEqual(len(data), 4)
        self.assertEqual(self.tag_values, {t["value"] for t in
        data})
```

Another important point to note is that we're "cheating" a little bit here for the sake of the demonstration. We have access to the database in the `TestCase`, so we can create objects using the normal methods.

If we were testing a service running elsewhere we'd need to create test objects in some other way, perhaps through the API itself. However, if you do this you'd need to assume that your API was working in order to create the test objects – a bit of a catch 22.

Authentication with RequestClient

Authentication with `RequestsClient`

We can make requests that need authentication with `RequestsClient` as well. To use basic authentication, we can use the `requests.auth.HTTPBasicAuth` class which wraps the base-64 encoding process and sets up the header. After instantiating `HTTPBasicAuth` we apply it by setting it as the `auth` attribute on the client.

Here's how we could implement a test that creates a new `Tag`, authenticating the client with basic authentication. While we haven't shown it here, we create the `testuser@example.com` `User` in the `setUp()` method.

```
from requests.auth import HTTPBasicAuth


class TagApiTestCase(LiveServerTestCase):
    # existing methods omitted

    def test_tag_create_basic_auth(self):
        self.client.auth = HTTPBasicAuth("testuser@example.com",
        "password")
        resp = self.client.post(
            self.live_server_url + "/api/v1/tags/", {"value":
        "tag5"}
        )
        self.assertEqual(resp.status_code, 201)
        self.assertEqual(Tag.objects.all().count(), 5)
```

To use token authentication, we could create a `Token` object in the database. Or, we could request a token from the API, assuming that the endpoint works. If you're working with a remote API, then this will be your only option.

Whichever method you use to get the token, you'll need to set the `Authorization` HTTP header. The `RequestsClient` class has a writable `headers` dictionary attribute, which can be used to define the HTTP headers of the request.

Here's another test that creates a `Tag`, this time fetching a token from the `token-auth` endpoint (the test user was created in the `setUp()` method).

We then build the authorization header and set it to `self.headers["Authorization"]`.

```
from requests.auth import HTTPBasicAuth


class TagApiTestCase(LiveServerTestCase):
    # existing methods omitted

        def test_tag_create_token_auth(self):
        token_resp = self.client.post(
            self.live_server_url + "/api/v1/token-auth/",
            {"username": "testuser@example.com", "password":
        "password"},
        )
        self.client.headers["Authorization"] = "Token " +
        token_resp.json()["token"]


        resp = self.client.post(
            self.live_server_url + "/api/v1/tags/", {"value":
        "tag5"}
        )
        self.assertEqual(resp.status_code, 201)
        self.assertEqual(Tag.objects.all().count(), 5)
```

Try It Out

Try It Out

Go ahead and create a file called `test_tag_api.py` inside the `blog` directory. Copy and paste this code into it:

```
from django.test import LiveServerTestCase
from requests.auth import HTTPBasicAuth
from rest_framework.test import RequestsClient

from django.contrib.auth import get_user_model
from blog.models import Tag


class TagApiTestCase(LiveServerTestCase):
    def setUp(self):
        get_user_model().objects.create_user(
            email="testuser@example.com", password="password"
        )

        self.tag_values = {"tag1", "tag2", "tag3", "tag4"}
        for t in self.tag_values:
            Tag.objects.create(value=t)
        self.client = RequestsClient()

    def test_tag_list(self):
        resp = self.client.get(self.live_server_url +
        "/api/v1/tags/")
        self.assertEqual(resp.status_code, 200)
        data = resp.json()
        self.assertEqual(len(data), 4)
        self.assertEqual(self.tag_values, {t["value"] for t in
        data})

    def test_tag_create_basic_auth(self):
        self.client.auth = HTTPBasicAuth("testuser@example.com",
        "password")
        resp = self.client.post(
            self.live_server_url + "/api/v1/tags/", {"value":
        "tag5"}
        )
        self.assertEqual(resp.status_code, 201)
        self.assertEqual(Tag.objects.all().count(), 5)
```

```
    def test_tag_create_token_auth(self):
        token_resp = self.client.post(
            self.live_server_url + "/api/v1/token-auth/",
            {"username": "testuser@example.com", "password":
        "password"},
        )
        self.client.headers["Authorization"] = "Token " +
        token_resp.json()["token"]

        resp = self.client.post(
            self.live_server_url + "/api/v1/tags/", {"value":
        "tag5"}
        )
        self.assertEqual(resp.status_code, 201)
        self.assertEqual(Tag.objects.all().count(), 5)
```

We've run through each of these methods in detail already, but to summarize:

- The `TagApiTestCase` inherits from `LiveServerTestCase`, which means a Django Dev Server will be started when the test case is run.
- `setUp()` creates a testing user, inserts some test `Tag` objects into the database, then sets `self.client` to an instance of `RequestClient`.
- `test_tag_list()` uses `self.client` to fetch a list of `Tags` using the API, then verifies that they match what's in the database.
- `test_tag_create_basic_auth()` authenticates by setting `client.auth` to an instance of `HTTPBasicAuth`. It then creates a new `Tag` and tests that the number of tags in the DB has increased.
- `test_tag_create_token_auth()` performs the same test as `test_tag_create_basic_auth()`, but authenticates with a token instead of basic authentication. It fetches the token by using the `token-auth` endpoint.

After saving the code above, you can run these tests (and the ones that you wrote in the last section) with the `manage.py test` command.

```
$  python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
......
----------------------------------------------------------------------
        ------
Ran 6 tests in 0.856s

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

▼ **Session Authentication**

As an aside, what about session authentication with the `RequestClient`? It is

possible, but requires manual cookie management. Most of the time, if you're going to be testing or using an API you won't be doing so with session based authentication. Using the session for authentication is normally only used for GUIs in browsers as a nicety for developers and not so much for actual API usage. For this reason we won't be covering it, but in short, you'd need to authenticate to the Django login view, save the cookie that's returned, and then send it back in any future requests in the `Cookie` HTTP header.

That's all we're for testing Django Rest Framework. In the next module, we're going to look at some advanced DRF concepts, starting with caching.

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 requests"
```

- Push to GitHub:

```
git push
```