Learning Objectives

- **Explain the importance of authentication**
- **Define session authentication**
- **Add session authentication to Blango**
- **Contrast basic authentication from session authentication**
- **Authenticate with Postman, your username, and your password**
- **Identify the benefits of token authentication**
- **Add token authentication to Blango**

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

Authentication

Intro

One aspect that has been distinctly lacking from our API views so far is authentication - anyone can perform any action on the API. Obviously that is not ideal! Especially since one of the pillars of Django is its security.

When dealing with user access to the API, there are actually two pieces: *authentication*, which is *who* you are; and *permissions*, which is what you are allowed to do.

We'll look at authentication first, and three methods that Django Rest Framework provides for us to authenticate.

DRF's authentication is pluggable, so different classes for authenticating a user can be turned on and off by updating the `settings.py` file. The first two authentication methods are the default ones, so we won't look at how to customize them just yet.

Authenticating with the Session

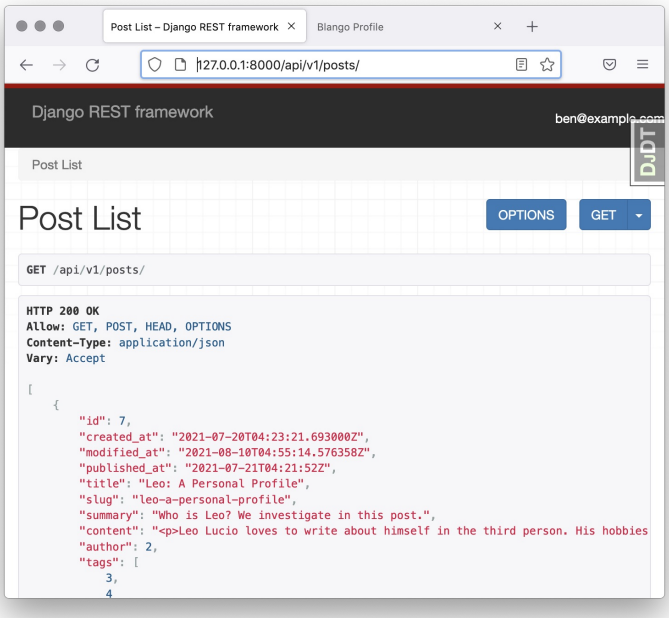
Authenticating with the session

The standard way of authenticating in Django, is to log in with a username and password using a form. This user information is then stored in the session, on the backend. The session is identified by a cookie sent by the browser.

Using session authentication is not ideal for use with an API. With a REST API it's useful for the client to be able to tell who the current user is, by looking at the request. This information isn't that obvious when looking at a session identifier (a random string) which can't be related to a user. In addition, the session might store extra information in the backend which can alter the response we get, so doing the same response more than once might unexpectedly return different data that is retrieved from the session. And finally, most HTTP clients designed for API usages aren't set up to store cookie data so this is something you'd have to manage manually.

However, when testing the API with the DRF browser GUI, session authentication works quite well, because it automatically works with forms and the browser will store the cookie automatically.

As we mentioned, session authentication is enabled by default, so we don't have to do anything to start using it. In fact, you might have been unknowingly using it already. If you visit one of our Blango DRF URLs (post list or detail), and you're logged in, you'll see your email address in the top right corner of the page.



logged in django rest framework

We already have two ways of logging into Blango, either with the Django Admin page at `/admin/` or with the standard user login page at `/accounts/login/`.

If we didn't already have login pages set up, DRF provides one we can use. We just need to include the `rest_framework.urls` URL patterns and DRF does the work.

We can do that by adding the following to the end of our `blog/api/urls.py` file:

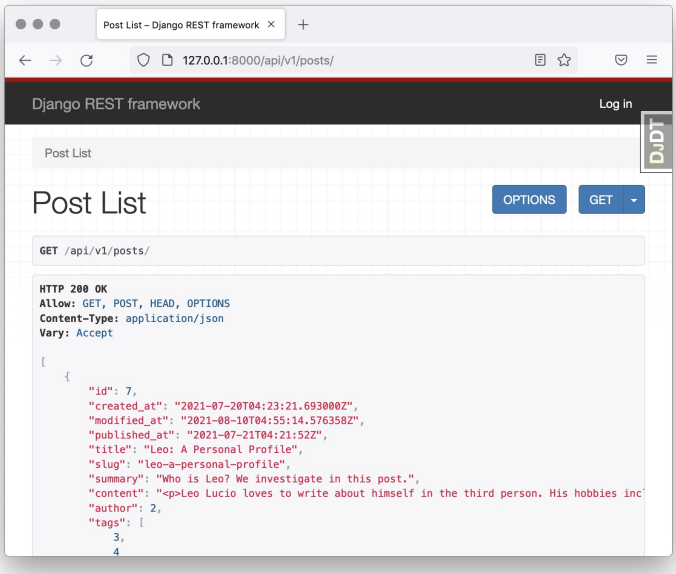
```
urlpatterns += [
    path("auth/", include("rest_framework.urls")),
]
```

The path can actually be anything we want, DRF uses the Django URL reversing functionality to check if its views are set up.

We also need to first import `include` from `django.urls` too:

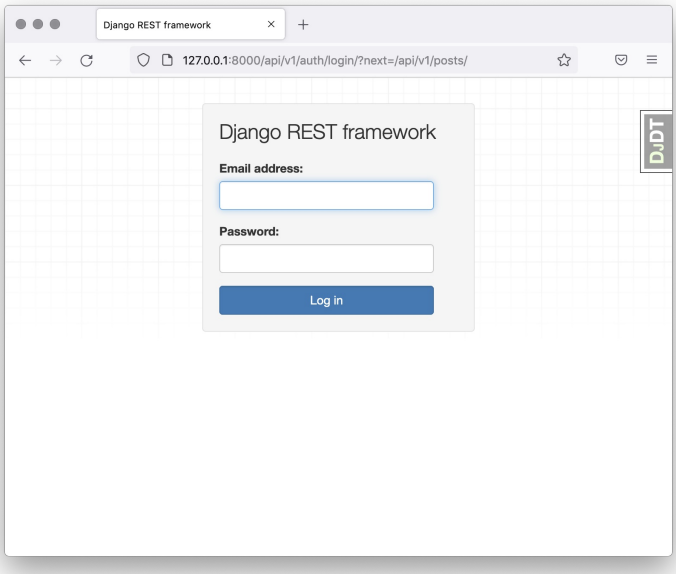
```
from django.urls import path, include
```

Then, DRF will detect that you have the login URL configured and automatically add a link to log in in the top right corner of a DRF view page.



log in link

After clicking the link you'll see the DRF login page.



django rest framework login page

Logging in here should behave the same as logging in with any of our other forms. You'll get redirected back to the DRF page you came from, and see your email address in the top right corner, indicating that you're logged in.

Try it out

Let's add the DRF auth URLs to Blango, so our API users have an easy way to find a login page.

Follow the instructions just above to add the import of `include` and extra `urlpatterns` to `blog/api/urls.py`.

Then, load up one of the Blango DRF API views. You should see a **Log in** link in the top right corner. Or, if you're already logged in, a dropdown menu with a **Log out** link.

View Blog

Next we'll look at basic authentication, which is more useful for REST APIs.

Basic Authentication

Basic authentication

Basic authentication is more useful than session authentication when it comes to REST APIs. Although it does have some drawbacks, which we'll cover when we discuss token authentication. Basic authentication involves sending the username and password with each request. These are joined by a `:`, then base-64 encoded, and sent in the `Authorization` header. The username *username* and password *password* would be sent like this:

```
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

`dXNlcm5hbWU6cGFzc3dvcmQ=` being the string `username:password` encoded by base-64.

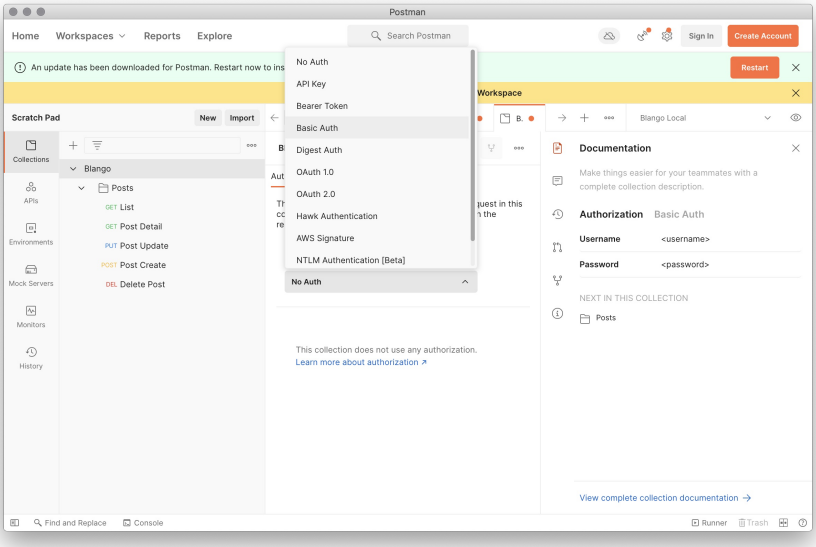
We can easily set up Postman to use Basic authentication. Let's do that now.

Try it out

You can now configure Postman to use basic authentication. Because this is a new assignment, your Codio hostname has changed. Be sure to look up the new hostname and update/save your `base_url` variable. Open Postman, and select the **Blango** collection from the collections list (not the **Posts** folder or any of the requests in the folder). Then, on the right the **Authorization** tab should be selected, but click on it if it's not.

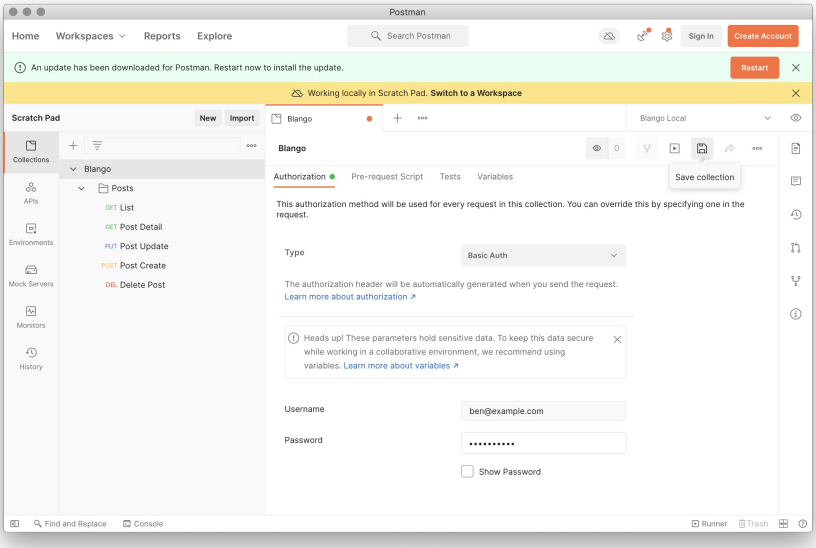
▼ **Finding your Codio host**
In the Codio menu bar, click "Project" => "Box Info". This will open a tab with your Codio host name in it.

From the *Type* dropdown menu select **Basic Auth**



Click here to see a larger version of the image

You'll then see *Username* and *Password* fields. Enter your email address as the username, and your password. Then, click the **Save Collection** button.



Click here to see a larger version of the image

You can test that it's working by making a request using Postman. Everything should work the same as it did previously. If you've put in an incorrect username (email address) or password, you'll get a response with a `403` status code and an error response body:

```
{"detail":"Invalid username/password."}
```

Even though none of our API views require authentication, DRF still automatically validates credentials for us, if provided.

To provide HTTP basic auth using the `requests` framework, we can use the `requests.auth.HTTPBasicAuth` class. Be sure to start the Python shell before continuing.

Open a Second Terminal

We need a second terminal to run the Python shell in addition to the currently running dev server. In the Codio menu bar, click "Tools" => "Terminal" to open a new tab with another terminal. Start the Python shell by entering the following command:

```
python3 blango/manage.py shell
```

After the shell is running, import `requests` and then import `HTTPBasicAuth`.

```
In [1]: import requests

In [2]: from requests.auth import HTTPBasicAuth
```

info

Finding the Correct URL

The URL you need to enter depends on the domain name of your Codio box. In the Codio menu bar, click "Project" => "Box Info". This will open a tab with your Codio domain name in it. Be sure you are appending `-8000` so you are looking at port 8000. Then add `/api/v1/posts/` so you are looking at the API endpoint. Here is an example of a correct URL:

```
https://martin-volume-8000.codio.io/api/v1/posts/
```

Once you have your correct URL, make a request with incorrect information.

```
In [3]: requests.get("https://martin-volume-
        8000.codio.io/api/v1/posts/",
        auth=HTTPBasicAuth("user@example.com", "badpassword"))
```

Because the authentication information is incorrect, you should see a 403 error.

```
DEBUG 2021-09-16 15:03:55,310 connectionpool 1794
        139885228320576 Starting new HTTPS connection (1):
        martin-volume-8000.codio.io
DEBUG 2021-09-16 15:03:55,492 connectionpool 1794
        139885228320576 https://martin-volume-8000.codio.io:443
        "GET /api/v1/posts/ HTTP/1.1" 403 39
Out[3]: <Response [403]>
```

Now make a request with correct information.

```
In [4]: requests.get("https://martin-volume-
        8000.codio.io/api/v1/posts/",
        auth=HTTPBasicAuth("patrick@example.com", "password"))
```

Using the correct authentication information should return a response of 200.

```
DEBUG 2021-09-16 15:09:09,875 connectionpool 1994
        139741725628224 Starting new HTTPS connection (1):
        martin-volume-8000.codio.io
DEBUG 2021-09-16 15:09:10,303 connectionpool 1994
        139741725628224 https://martin-volume-8000.codio.io:443
        "GET /api/v1/posts/ HTTP/1.1" 200 6071
Out[4]: <Response [200]>
```

We'll cover the use of `requests` to connect to third-party APIs in the fourth course.

Basic authentication has some drawbacks. If a user has one user account but wants to access it from many API clients, they all share credentials. This means if a user wants to access an API with a third-party service that they might not fully trust, they have to give that service their username and password. If that service were run by someone malicious they could take that username and password and log into our system with full access.

On top of that, the password is easily decoded from the `Authorization` header, so could be intercepted in transport (although this is less of a problem with the prevalence of HTTPS).

On the backend, we don't have an easy way to figure out which specific service is making API requests as they all share the same credentials. This also means we can't specifically rate-limit or disable access to a particular bad client/service without completely blocking that user.

By using a token for authentication we can mitigate these issues.

Token Authentication

Token Authentication

Token authentication works by passing a token (a long, random string) in the `Authorization` header. The backend server can then map the token back to a user and authenticate them without a username and password. This means that:

- We can provide different tokens to different API clients, and potentially untrusted clients don't get our username and password.
- Tokens can be revoked to remove access to a particular client, leaving others intact and other clients unaffected.
- Our username and password are not sent in every request.

DRF's built-in token system doesn't solve all these issues, as it only allows one token per user, but in Course 3 we'll look at JSON Web Tokens which are more flexible.

When using a token, the `Authorization` HTTP header looks like this:

```
Authorization: Token <token>
```

For example, the token `fccdf7307189644e9fee624224d3471d870a7b829f5c23d0297f15e34c41b974` would be sent like:

```
Authorization: Token
fccdf7307189644e9fee624224d3471d870a7b829f5c23d0297f15e34c41b974
```

The keyword `Token` describes what kind of authorization is being used. You might also see the word `Bearer` being used.

To use token authentication in DRF we need to add token authorization as an installed app and the authentication classes in the `settings.py` file.

`rest_framework.authtoken` must be added to `INSTALLED_APPS`.

Then, we need to add a `REST_FRAMEWORK` setting, like this:

```
REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": [
        "rest_framework.authentication.BasicAuthentication",
        "rest_framework.authentication.SessionAuthentication",
        "rest_framework.authentication.TokenAuthentication",
    ]
}
```

The `REST_FRAMEWORK` attribute is where all DRF settings can be configured. For now we're just setting authentication classes. `BasicAuthentication` and `SessionAuthentication` are the two classes included by default. We need to include them here so they can continue to be used for authentication.

After making these changes, `manage.py migrate` needs to be run, because DRF stores the tokens in the database.

Creating tokens for users

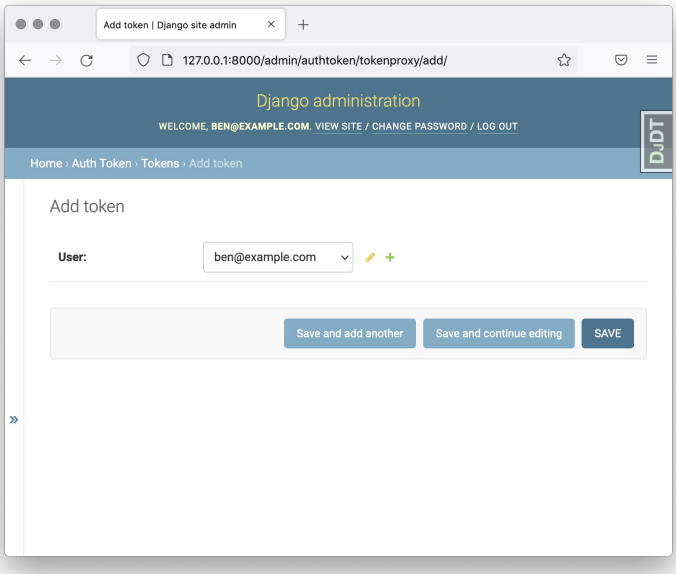
DRF tokens are stored in the database, as a `Token` model object, so we have a few options for creating them.

The first method is through Python code: just import the `Token` model from `rest_framework.authtoken.models` and `create()` a new instance. The token "key" (the string sent in the `Authorization` header) is automatically generated on create.

```
In [1]: from blango_auth.models import User

In [2]: from rest_framework.authtoken.models import Token

In [3]: u = User.objects.get(pk=1)

In [4]: u.email
Out[4]: 'ben@example.com'

In [5]: t = Token.objects.create(user=u)

In [6]: t.key
Out[6]: '81082614a73f331b122ba93dbdb5951b44cf21d4'
```

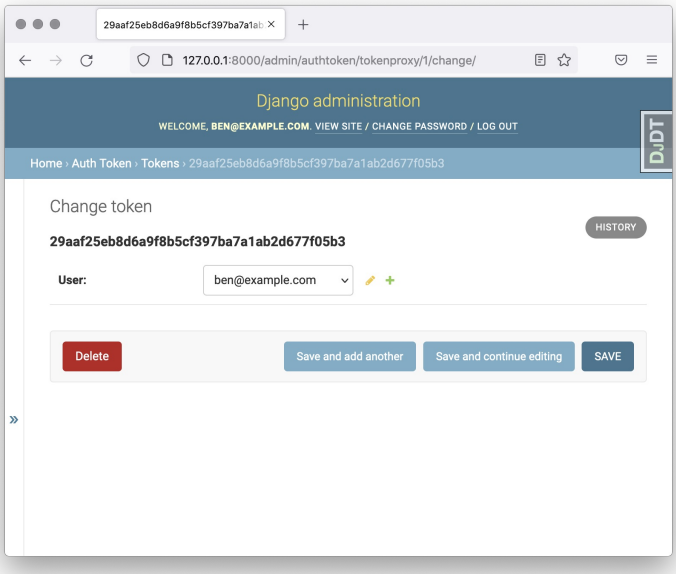
So in our case, we would authenticate as user `1` with the token `81082614a73f331b122ba93dbdb5951b44cf21d4`.

We could also create a token through Django Admin. We just choose to create a new `Token` object, and select the user.



create token through django admin

Then after saving, we'll see the `key` for that user's `Token`.



token key in django admin

The next method of creating (or retrieving) a token is through `manage.py`, with the `drf_create_token` command:

```
$ python manage.py drf_create_token ben@example.com
Generated token 81082614a73f331b122ba93dbdb5951b44cf21d4 for
        user ben@example.com
```

Notice that it has retrieved the same key as before, rather than creating a new one.

Finally, DRF provides a view we can use to get or retrieve a token for a user: `rest_framework.authtoken.views.obtain_auth_token`. To set this up we just need to add a URL for it. For example, in Blango's `api/urls.py` file, add the import:

```
from rest_framework.authtoken import views
```

Then, map a URL to the view:

```
urlpatterns += [
    path("auth/", include("rest_framework.urls")),
    path("token-auth/", views.obtain_auth_token)
]
```

▼ **URL Prefix**
Remember the prefix on our API URLs: the full path to the `obtain_auth_token` view is `/api/v1/token-auth`.

Then, POSTing a JSON object containing a `username` and `password` will return a JSON object with a `token`, like so:

```
{"token":"81082614a73f331b122ba93dbdb5951b44cf21d4"}
```

Try It Out

Try it out

We'll get the token authentication set up in Blango now. Start by opening the `settings.py` file and add the `REST_FRAMEWORK` setting, currently just containing authentication classes settings.

```
REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": [
        "rest_framework.authentication.BasicAuthentication",
        "rest_framework.authentication.SessionAuthentication",
        "rest_framework.authentication.TokenAuthentication",
    ]
}
```

Then, add `rest_framework.authtoken` to the list of `INSTALLED_APPS`.

You'll then need to run `manage.py migrate` to add the `Token` model to the database.

As we've seen, there are several ways to set up a token for a user, but we'll try out creating one using the API itself. This means setting up the `obtain_auth_token` view.

Open `api/urls.py` and update the `urlpatterns` appending portion of the code to look like this:

Open api/urls.py

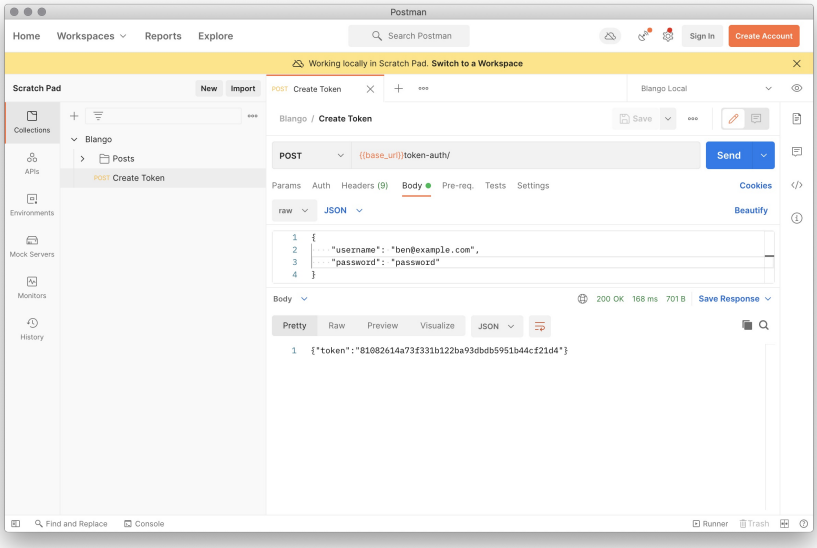
```
urlpatterns += [
    path("auth/", include("rest_framework.urls")),
    path("token-auth/", views.obtain_auth_token)
]
```

Don't forget, you need to import `views` as well.

```
from rest_framework.authtoken import views
```

Now, you can try it out in Postman. Create a new `POST` request inside the *Blango* collection. The URL should be `{{base_url}}token-auth/`. Create a JSON request body with `username` (the user's email address) and `password`.

Try out the request, and you should get a response with a token.



Click here to see a larger version of the image

How can we use this token in Postman? Select the Blango collection on the left, and then the Authorization tab. Select the type **API Key**. The values to use are:

- *Key*: `Authorization`
- *Value*: `Token <your token>`, e.g. `Token 81082614a73f331b122ba93dbdb5951b44cf21d4`
- *Add to*: **Header**

You should still be able to make requests as you did before. You can validate that the token authentication is working by trying to put in an incorrect token. In this case, your requests will return responses like this:

```
{"detail":"Invalid token."}
```

The response will have a `401 Unauthorized` HTTP status.

To pass the token in the `Authorization` header using the `requests` library, we pass it in the `headers` dictionary:

```
requests.get("http://127.0.0.1:8000/api/v1/posts/", headers=
        {"Authorization": "Token
        81082614a73f331b122ba93dbdb5951b44cf21d4"})
<Response [200]>
```

Applying authentication classes to views

While we won't use this feature, it's also possible to override the default authentication classes on a per-view basis. Set `authentication_classes` to a list of authentication classes to allow. For example:

```
from rest_framework.authentication import SessionAuthentication

class PostList(generics.ListCreateAPIView):
    authentication_classes = [SessionAuthentication]
    # other methods/attributes truncated for brevity
```

If we did this, then `PostList` would only be accessible to those authenticated by the session, while other views would fall back to the default authentication settings.

In the next section we'll look at adding some permissions to our DRF API views.

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 authentication"
```

- Push to GitHub:

```
git push
```