



## Django REST Framework – Part 2

Mario Alviano

# Outline

- Consume the API
- Authentication
- Authorization
- Tests
- Validators

# Consume the API: a Javascript example

```
<html>
<head>
  <script>
    fetch('http://localhost:8000/api/v1/')
      .then(response => response.json())
      .then(data => {
        let ul = document.createElement('ul');
        data.forEach(record => {
          let li = document.createElement('li');
          li.innerHTML = record.title;
          ul.appendChild(li);
        });
        let content = document.getElementById('content');
        content.innerHTML = '';
        content.appendChild(ul);
      });
  </script>
</head>
<body>
  <h1>Posts</h1>
  <div id='content'>
    Loading content...
  </div>
</body>
</html>
```

Create index.html in an empty directory

```
settings.py
62 CORS_ORIGIN_WHITELIST = [
63     'http://localhost:8000', # dev server
64     # add deploy server
65     'http://localhost:8001', # add front-end server
```

Allow requests from  
front-end server

```
$ python3 -m http.server 8001
Serving HTTP on 0.0.0.0 port 8001 (http://0.0.0.0:8001/) ...
```

Start the front-end server

Visit <http://localhost:8001>

## Posts

Loading content...

When data are fetched...



## Posts

- First post
- Second post

Plain Javascript is likely the worst option

Prefer some library or framework:  
Svelte, Vue.js, jQuery, React, Angular

Do requests from Python or Java and consume the API  
within a TUI or GUI

# Consuming the API: a Python example

```
def fetch_posts():
    res = requests.get(url=f'{api_server}/')
    if res.status_code != 200:
        return None
    return res.json()

def show_posts(posts):
    def sep():
        print('-'*60)

    fmt = '{:4}\t{:50}'

    print()
    sep()
    print('ALL POSTS FROM THE BLOG')
    sep()
    print(fmt.format('ID', 'TITLE'))
    sep()
    for post in posts:
        print(fmt.format(post['id'], post['title']))
    sep()
    print()
```

```
import requests

api_server = 'http://localhost:8000/api/v1'

def main():
    welcome()
    posts = fetch_posts()
    if posts is None:
        error_message()
    show_posts(posts)
    goodbye()
```

```
$ python3 app.py
===== Blog TUI =====
= Because we love the '80s so much! =
=====

-----
ALL POSTS FROM THE BLOG
-----
ID      TITLE
-----
  1     First post
  3     Second post
-----

It was nice to have your here. Have a nice day!
```

# Session-based authentication

- Stateful approach
- The client sends the initial credentials
- The server stores in the session object that the user is authenticated
- The client stores the session ID (usually, cookie)
- Session ID is sent on all requests
- After log out, the session is destroyed on both parts

Let's keep session-based authentication for the browsable API

# Token-based authentication

- Stateless approach
- The client sends the initial credentials
- The server generates a unique token
- The client stores the token (cookie or local storage or anywhere a setting like this can be safely stored; it's more like a key for using Google Maps API)
- The client sends the token with each request

By default, Django generates a token for each user and store it in the database

Alternatives are usually based on JWT and OAuth2 and don't need to store anything

# What we need

- Add URLs for session-based authentication
- Setup token-based authentication
  - `rest_framework.authtoken`
- Setup login, logout and reset endpoints
  - `dj-rest-auth`
- Setup registration endpoints
  - `django-allauth`

Must be installed

Must be installed



```
33 INSTALLED_APPS = [  
34     'django.contrib.admin',  
35     'django.contrib.auth',  
36     'django.contrib.contenttypes',  
37     'django.contrib.sessions',  
38     'django.contrib.messages',  
39     'django.contrib.staticfiles',  
40     'django.contrib.sites',  
41  
42     'rest_framework',  
43     'rest_framework.authtoken',  
44     'allauth',  
45     'allauth.account',  
46     'dj_rest_auth',  
47     'dj_rest_auth.registration',  
48     'corsheaders',  
49  
50     'posts.apps.PostsConfig',  
51 ]
```

Needed by allauth, allows to handle multiple sites in the same project

Add authorization apps

```
53 REST_FRAMEWORK = {  
54     'DEFAULT_AUTHENTICATION_CLASSES': [  
55         'rest_framework.authentication.SessionAuthentication',  
56         'rest_framework.authentication.TokenAuthentication',  
57     ],  
58     'DEFAULT_SCHEMA_CLASS': 'rest_framework.schemas.coreapi.AutoSchema',  
59 }
```

Support session and token authentication

```
150 ACCOUNT_EMAIL_REQUIRED = True  
151 ACCOUNT_EMAIL_VERIFICATION = "none"  
152  
153 SITE_ID = 1
```

Disable email verification for simplicity  
We have only one site, so let's assign ID 1

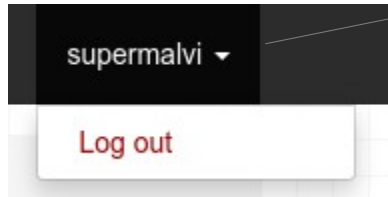
```
blog_api/urls.py x
24 urlpatterns = [
25     path('admin-IMinewINTANG/', admin.site.urls),
26     path('api-auth/', include('rest_framework.urls')),
27     path('docs/', include_docs_urls(title=API_TITLE, description=API_DESCRIPTION)),
28     path('schema/', get_schema_view(title=API_TITLE)),
29     path('api/v1/posts/', include('posts.urls')),
30     path('api/v1/auth/', include('dj_rest_auth.urls')),
31     path('api/v1/auth/registration/', include('dj_rest_auth.registration.urls')),
32 ]
```

Session-based authentication for the browsable API  
It's out of our REST API

Avoid ambiguities, add a prefix to posts URLs

Token-based authentication as part of our REST API

Let's migrate the database and start our project



The browsable API now supports log in and log out operations

### **URLs for token-based authentication**

<http://127.0.0.1:8000/api/v1/auth/login/>

<http://127.0.0.1:8000/api/v1/auth/logout/>

<http://127.0.0.1:8000/api/v1/auth/password/reset/>

<http://127.0.0.1:8000/api/v1/auth/password/reset/confirm/>

<http://127.0.0.1:8000/api/v1/auth/password/change/>

# Authorization

- Authentication deals with **who are you**
- Authorization deals with **what can you do**
- Restrict the default permission to admin only
- Authorize based on a per-view or per-object policy
- Take advantage of Django permissions and groups

```
settings.py x
53 REST_FRAMEWORK = {
54     'DEFAULT_AUTHENTICATION_CLASSES': [
55         'rest_framework.authentication.SessionAuthentication',
56         'rest_framework.authentication.TokenAuthentication',
57     ],
58     'DEFAULT_PERMISSION_CLASSES': [
59         'rest_framework.permissions.IsAdminUser',
60     ],
61     'DEFAULT_SCHEMA_CLASS': 'rest_framework.schemas.coreapi.AutoSchema',
62 }
```

Only admin is authorized

Default policy must be restrictive

Avoid to accidentally leave unauthorized usages of your API

```
views.py x
19 class PostViewSet(viewsets.ModelViewSet):
20     permission_classes = [permissions.IsAuthenticatedOrReadOnly]
21     queryset = Post.objects.all()
22     serializer_class = PostSerializer
```

Specify different permissions on views

In this case everyone can read posts, but only authenticated users can change them

Django REST framework Log in

Post List / Post Instance

## Post Instance

OPTIONS GET ▾

GET /api/v1/1/

HTTP 200 OK  
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS  
Content-Type: application/json  
Vary: Accept

```
{
  "id": 1,
  "author": 1,
  "title": "First post",
  "body": "Some text here",
  "created_at": "2020-11-16T00:34:28.904278Z"
}
```

No delete button is shown in the browsable API

No form is shown in the browsable API

Let's restrict change to authors

Let's restrict read to authenticated users

```
views.py x
19 class PostViewSet(viewsets.ModelViewSet):
20     # permission_classes = [permissions.IsAuthenticatedOrReadOnly]
21     permission_classes = [IsAuthorOrReadOnly, IsAuthenticated]
22     queryset = Post.objects.all()
23     serializer_class = PostSerializer
```

Add this file  
to the posts app

IsAuthorOrReadOnly is written by us

Let's add permissions.py to our app

```
permissions.py x
1 from rest_framework import permissions
2
3
4 class IsAuthorOrReadOnly(permissions.BasePermission):
5     def has_permission(self, request, view):
6         return True
7
8     def has_object_permission(self, request, view, obj):
9         if request.method in permissions.SAFE_METHODS:
10            return True
11            return obj.author == request.user
```

Do we have permission on the view?

This is the default implementation,  
we can also omit it

Do we have permission on this object  
provided by the view?

## **Be aware of what we are doing**

We are just experimenting!

In a real setting, authors of posts  
would be fixed to `request.user`

That said, we would implement the POST view  
with a different serializer not including field `author`



Can we restrict the view in a way that users can only read their own posts?

Yes! The view must filter the objects in the queryset

```
views.py x
26 class PostByAuthorViewSet(viewsets.ModelViewSet):
27     permission_classes = [IsAuthenticated]
28     serializer_class = PostSerializer
29
30     def get_queryset(self):
31         return Post.objects.filter(author=self.request.user)
```

Let's add a new view  
(or, in a real setting,  
modify the existing one)

We can filter the queryset  
by defining this method

```
posts/urls.py x
13 router = SimpleRouter()
14 router.register('by-author', PostByAuthorViewSet, basename='posts-by-author')
15 router.register('', PostViewSet, basename='posts')
```

If you opted for a new view,  
add URLs for it

Django provides a rich set of permissions for users and groups

Let's add a **post\_editors** group with all permissions on posts

The screenshot shows the Django administration interface for the 'post\_editors' group. The page is titled 'Change group' and includes a sidebar with navigation links for ACCOUNTS, AUTH TOKEN, AUTHENTICATION AND AUTHORIZATION, POSTS, and SITES. The 'Groups' link is highlighted. The main content area shows the group name 'post\_editors' and a list of permissions. The 'Available permissions' list includes various actions on different models, such as 'Can add email address', 'Can change email address', 'Can delete email address', 'Can view email address', 'Can add email confirmation', 'Can change email confirmation', 'Can delete email confirmation', 'Can view email confirmation', 'Can add log entry', 'Can change log entry', 'Can delete log entry', 'Can view log entry', and 'Can add social account'. The 'Chosen permissions' list includes 'posts | post | Can add post', 'posts | post | Can change post', 'posts | post | Can delete post', and 'posts | post | Can view post'. At the bottom of the page, there are buttons for 'Delete', 'Save and add another', 'Save and continue editing', and 'SAVE'.

# Add a user to the post\_editors group

Also add another user with only **posts | post | Can view post** permission

The screenshot shows the Django administration interface. The left sidebar contains a menu with categories: ACCOUNTS (Email addresses + Add), AUTH TOKEN (Tokens + Add), AUTHENTICATION AND AUTHORIZATION (Groups + Add, Users + Add), POSTS (Posts + Add), and SITES (Sites + Add). The 'Users' link is highlighted in yellow. The main content area is titled 'Change user' and shows the profile for a user named 'malvi'. The 'Personal info' section includes fields for First name, Last name, and Email address (malvi@example.com). The 'Permissions' section has three checkboxes: 'Active' (checked), 'Staff status' (unchecked), and 'Superuser status' (unchecked). At the bottom, the 'Groups' section shows an 'Available groups' search box and a 'Chosen groups' list containing 'post\_editors'.

Django administration

Home · Authentication and Authorization · Users · malvi

### Change user

**Username:**   
Required. 150 characters or fewer. Letters, digits and @/./+/-/\_ only.

**Password:** **algorithm:** pbkdf2\_sha256 **iterations:** 216000 **salt:** CSrIU8\*\*\*\*\* **hash:** 8EnVE3\*\*\*\*\*  
Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using [this form](#).

#### Personal info

**First name:**

**Last name:**

**Email address:**

#### Permissions

**Active**  
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

**Staff status**  
Designates whether the user can log into this admin site.

**Superuser status**  
Designates that this user has all permissions without explicitly assigning them.

**Groups:**

**Available groups** ⓘ

**Chosen groups** ⓘ +

- post\_editors

## Role-based authorization

views.py ×

```
34 class PostEditorViewSet(viewsets.ModelViewSet):  
35     permission_classes = [IsPostEditor]  
36     queryset = Post.objects.all()  
37     serializer_class = PostSerializer
```

posts/permissions.py ×

```
14 class IsPostEditor(permissions.BasePermission):  
15     def has_permission(self, request, view):  
16         return request.user.groups.filter(name='post_editors').exists()
```

posts/urls.py ×

```
15 router.register('editor', PostEditorViewSet, basename='post-editors')
```

Authorize views to members of specific groups

The group represents a role

## Permission-based authorization

```
views.py x
40 class PostPermissionViewSet(viewsets.ModelViewSet):
41     permission_classes = [IsPermittedOnPost]
42     queryset = Post.objects.all()
43     serializer_class = PostSerializer
```

```
posts/urls.py x
```

```
16 router.register('perm', PostPermissionViewSet, basename='post-perm')
```

```
posts/permissions.py x
```

```
17 method2perm = {
18     'POST': 'add',
19     'PUT': 'change',
20     'PATCH': 'change',
21     'DELETE': 'delete',
22     'GET': 'view',
23     'HEAD': 'view',
24     'OPTIONS': 'view',
25 }
26
27
28 class IsPermittedOnPost(permissions.BasePermission):
29     def has_permission(self, request, view):
30         if request.method not in method2perm:
31             return False
32         return request.user.has_perm(f'posts.{method2perm[request.method]}_post')
```

Authorize views to users  
with specific permissions

Note that users inherit  
permissions of their groups

# Consuming the API: the Python example reloaded

```
def login():
    username = input('Username: ')
    password = getpass.getpass('Password: ')

    res = requests.post(url=f'{api_server}/auth/login/', data={'username': username, 'password': password})
    if res.status_code != 200:
        return None
    json = res.json()
    return json['key']

def logout(key):
    res = requests.post(url=f'{api_server}/auth/logout/', headers={'Authorization': f'Token {key}'})
    if res.status_code == 200:
        print('Logged out!')
    else:
        print('Log out failed')
    print()

def fetch_posts(key):
    res = requests.get(url=f'{api_server}/posts/', headers={'Authorization': f'Token {key}'})
    if res.status_code != 200:
        return None
    return res.json()
```

```
import getpass
import requests

api_server = 'http://localhost:8000/api/v1'

def main():
    welcome()
    key = login()
    if key is None:
        error_message()
    posts = fetch_posts(key)
    if posts is None:
        error_message()
    show_posts(posts)
    logout(key)
    goodbye()
```

# Tests

- Test first, code later
- We have a lot of code, but no tests
- Just because we are learning Django, we first focused on code
- Time to fix this and add a few tests
- From this point, we should prefer a TDD approach

## Setup

Install pytest-django and mixer

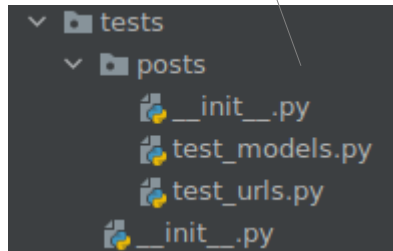
Set pytest as test tool

Add a run configuration for pytest in path tests

Create test files for models.py and urls.py

It's possible to test also views.py, but more tedious and somehow implicit in test\_urls.py

Since we keep tests in directory tests, we can also remove tests.py in the app module



```
pytest.ini x
1 [pytest]
2 DJANGO_SETTINGS_MODULE = blog_api.settings
3
```

Add pytest.ini in the root of the project and point to settings.py (unless you want to specify different settings for testing)



## If everything works, let's write our first test

```
test_models.py x
1 import pytest
2 from django.core.exceptions import ValidationError
3 from mixer.backend.django import mixer
4
5
6 def test_post_title_of_length_51_raises_exception(db):
7     post = mixer.blend('posts.Post', title='A'*51)
8     with pytest.raises(ValidationError) as err:
9         post.full_clean()
10     assert 'at most 50 characters' in '\n'.join(err.value.messages)
```

**db** is a **fixture**, an object we can use in our tests, in this case to access the temporary database used by tests

**mixer** creates objects from our models, with random values for non-provided fields

Call **full\_clean** on an object of our model to validate it

We can also check the message of the exception, but I would not encourage this practice

This test passes, it had to be written before the code!

## Let's write a failing test, then fix it

```
test_models.py x
13 def test_post_title_not_capitalized_raises_exception(db):
14     post = mixer.blend('posts.Post', title='my wrong title')
15     with pytest.raises(ValidationError):
16         post.full_clean()
```

```
validators.py x
1 from django.core.exceptions import ValidationError
2
3
4 def validate_title(value: str) -> None:
5     if len(value) == 0:
6         raise ValidationError('Title must not be empty')
7     if not value[0].isupper():
8         raise ValidationError('Title must be capitalized')
```

```
models.py x
8 class Post(models.Model):
9     author = models.ForeignKey(get_user_model(), on_delete=models.CASCADE)
10    title = models.CharField(max_length=50, validators=[validate_title])
11    body = models.TextField(validators=[RegexValidator(r'^[\w\s.:\;\'"]*$')])
12    created_at = models.DateTimeField(auto_now_add=True)
13    updated_at = models.DateTimeField(auto_now=True)
```

This test fails because we currently allow any text of 50 chars or less

We can define validators as functions  
Let's add validators.py to our app

A list of validators can be specified for each field of the model

Django provides many common validators

## Test URLs, mainly for permissions

We can define our own fixtures, for example to populate the DB

We use APIClient to simulate an API consumer

```
test_urls.py x
42 ▶ def test_post_anon_user_get_nothing():
43     path = reverse('posts-list')
44     client = get_client()
45     response = client.get(path)
46     assert response.status_code == HTTP_403_FORBIDDEN
47     assert contains(response, 'detail', 'credentials were not provided')
48
49
50 ▶ def test_post_user_get_list(posts):
51     path = reverse('posts-list')
52     user = mixer.blend(get_user_model())
53     client = get_client(user)
54     response = client.get(path)
55     assert response.status_code == HTTP_200_OK
56     obj = parse(response)
57     assert len(obj) == len(posts)
```

Method get() to perform a GET request

```
test_urls.py x
13     @pytest.fixture()
14     def posts(db):
15         return [
16             mixer.blend('posts.Post'),
17             mixer.blend('posts.Post'),
18             mixer.blend('posts.Post'),
19         ]
20
21
22 def get_client(user=None):
23     res = APIClient()
24     if user is not None:
25         res.force_login(user)
26     return res
27
28
29 def parse(response):
30     response.render()
31     content = response.content.decode()
32     return json.loads(content)
33
34
35 def contains(response, key, value):
36     obj = parse(response)
37     if key not in obj:
38         return False
39     return value in obj[key]
```

Arguments for the URL  
are specified in this dictionary

```
test_urls.py x
60 ▶ def test_post_retrieve_a_single_post(posts):
61     path = reverse('posts-detail', kwargs={'pk': posts[0].pk})
62     client = get_client(posts[0].author)
63     response = client.get(path)
64     assert response.status_code == HTTP_200_OK
65     obj = parse(response)
66     assert obj['title'] == posts[0].title
```

# Questions

