UNIVERSITÀ DELLA CALABRIA
il Campus per eccellenza

Django REST Framework – Part 2

Mario Alviano

Outline

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

Consume the API: a Javascript example

```html
<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

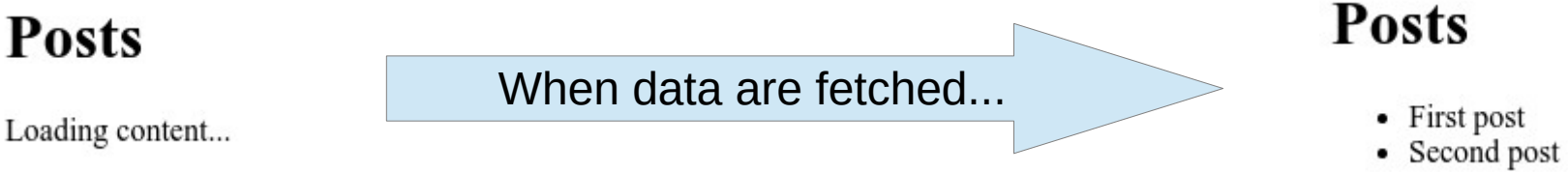
```
62  CORS_ORIGIN_WHITELIST = [
63      'http://localhost:8000', # dev server
64      # add deploy server
65      'http://localhost:8001', # add front-end server
```

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



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

```python
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()
```

```python
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()
```

```
$ 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

settings.py

```
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

```
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  ]
```

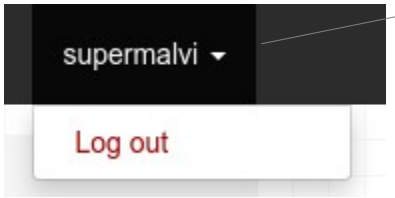
Session-based authentication for the browsable API

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

```
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

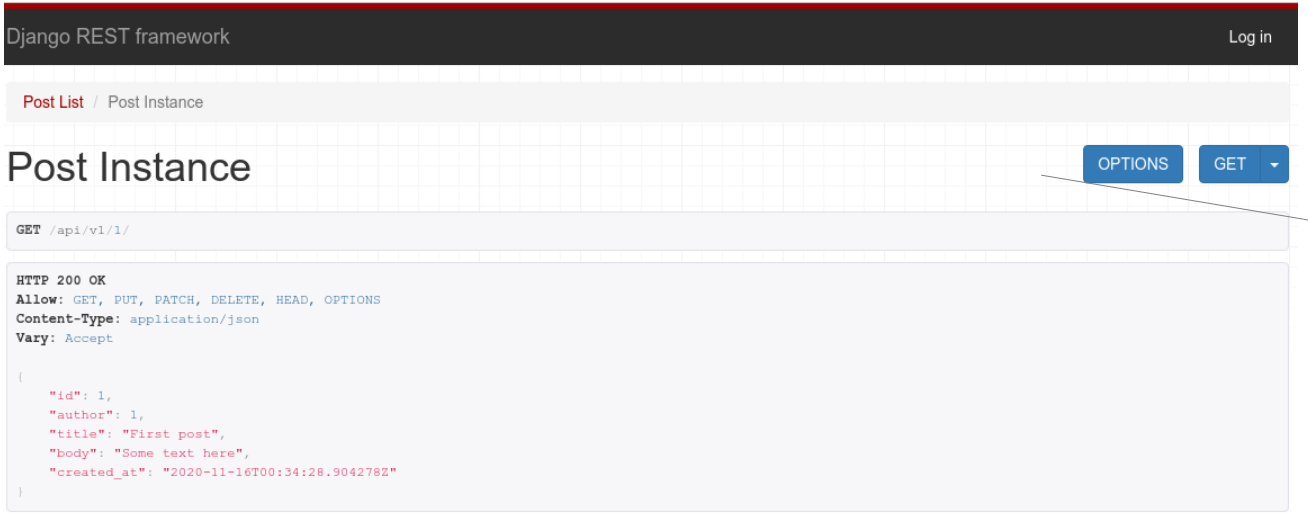
Avoid to accidentally leave unauthorized usages of your API

views.py

```python
class PostViewSet(viewsets.ModelViewSet):
    permission_classes = [permissions.IsAuthenticatedOrReadOnly]
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

Specify different permissions on views

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



No delete button is shown
in the browsable API

No form is shown in the browsable API

views.py

```python
class PostViewSet(viewsets.ModelViewSet):
    # permission_classes = [permissions.IsAuthenticatedOrReadOnly]
    permission_classes = [IsAuthorOrReadOnly, IsAuthenticated]
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

Let’s restrict change to authors

Let’s restrict read to authenticated users

Add this file
to the posts app

permissions.py

```python
from rest_framework import permissions


class IsAuthorOrReadOnly(permissions.BasePermission):
    def has_permission(self, request, view):
        return True

    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True
        return obj.author == request.user
```

IsAuthorOrReadOnly is written by us

Let’s add permissions.py to our app

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

```python
class PostByAuthorViewSet(viewsets.ModelViewSet):
    permission_classes = [IsAuthenticated]
    serializer_class = PostSerializer

    def get_queryset(self):
        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

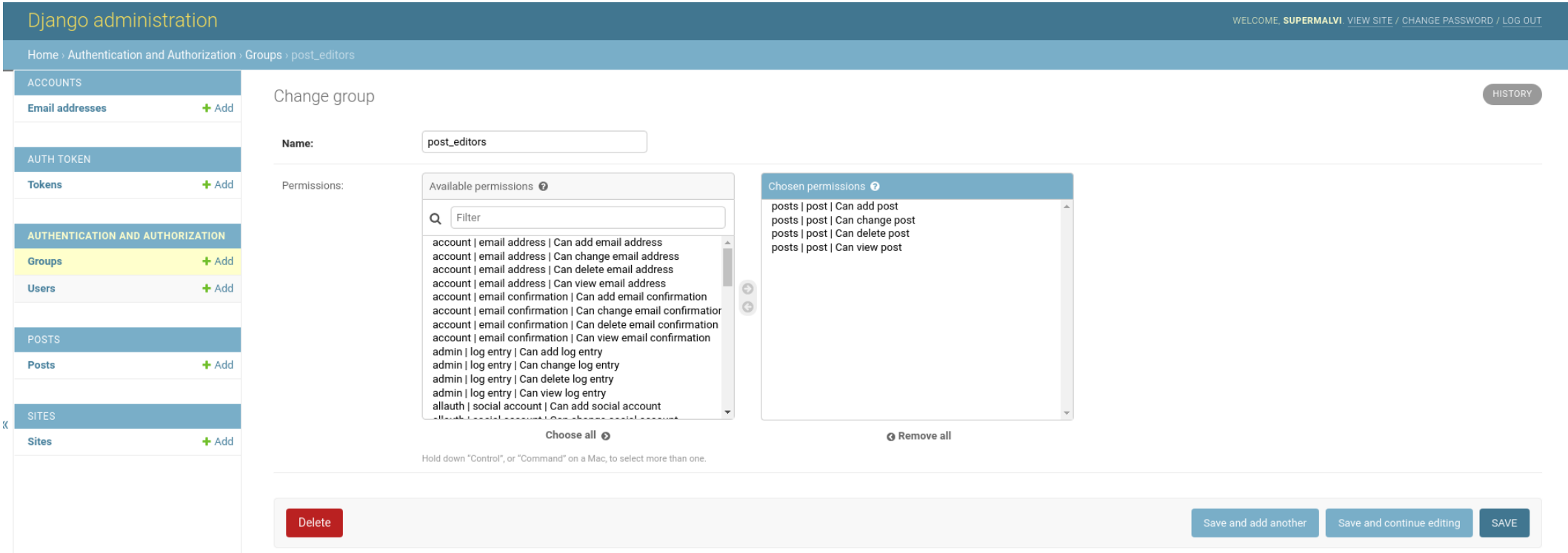
posts/urls.py

```python
router = SimpleRouter()
router.register('by-author', PostByAuthorViewSet, basename='posts-by-author')
router.register('', PostViewSet, basename='posts')
```

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

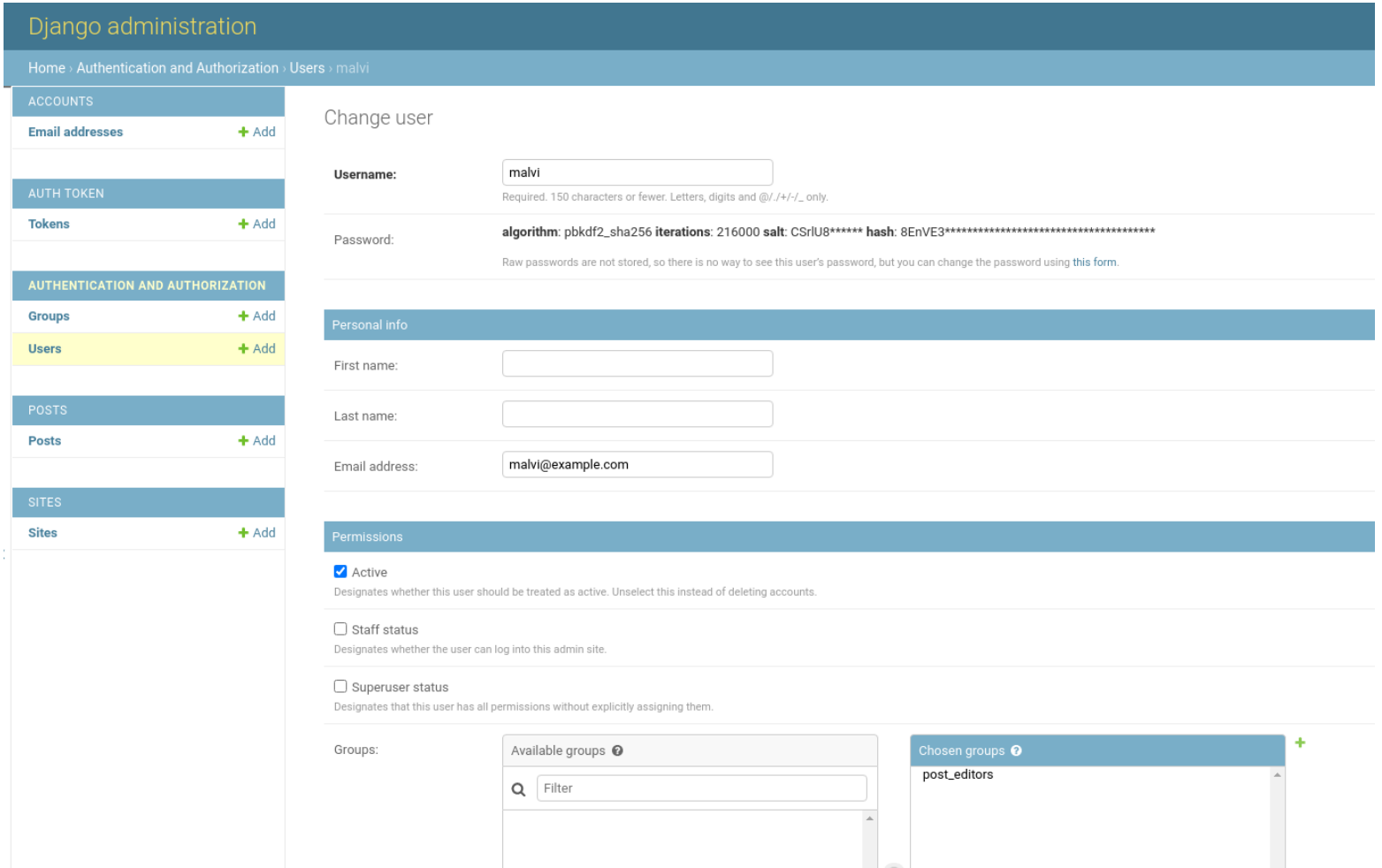
Django provides a rich set of permissions for users and groups

Let’s add a **post_editors** group with all permissions on posts



Add a user to the post_editors group

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



Role-based authorization

views.py

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

posts/permissions.py

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

posts/urls.py

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

Authorize views to members of specific groups

The group represents a role

Permission-based authorization

views.py

```python
class PostPermissionViewSet(viewsets.ModelViewSet):
    permission_classes = [IsPermittedOnPost]
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

posts/urls.py

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

posts/permissions.py

```python
method2perm = {
    'POST': 'add',
    'PUT': 'change',
    'PATCH': 'change',
    'DELETE': 'delete',
    'GET': 'view',
    'HEAD': 'view',
    'OPTIONS': 'view',
}


class IsPermittedOnPost(permissions.BasePermission):
    def has_permission(self, request, view):
        if request.method not in method2perm:
            return False
        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

```python
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()


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()
```

Tests

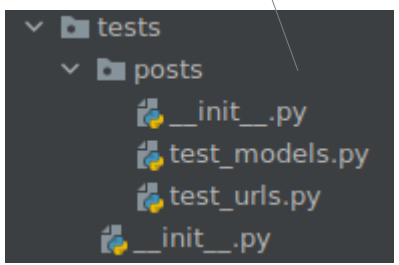
- Test first, code later
- We have a lot of code, but no tests
- Just because we are learning Django, we first focused on code
- Time to fix this and add a few tests
- From this point, we should prefer a TDD approach

Setup

Install pytest-django and mixer

Set pytest as test tool

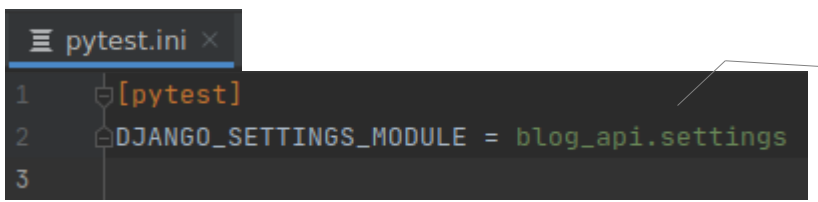
Add a run configuration for pytest in path tests



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



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

```python
import pytest
from django.core.exceptions import ValidationError
from mixer.backend.django import mixer


def test_post_title_of_length_51_raises_exception(db):
    post = mixer.blend('posts.Post', title='A'*51)
    with pytest.raises(ValidationError) as err:
        post.full_clean()
    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

```
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()
```

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

validators.py

```
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')
```

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

models.py

```
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)
```

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

```python
@pytest.fixture()
def posts(db):
    return [
        mixer.blend('posts.Post'),
        mixer.blend('posts.Post'),
        mixer.blend('posts.Post'),
    ]


def get_client(user=None):
    res = APIClient()
    if user is not None:
        res.force_login(user)
    return res


def parse(response):
    response.render()
    content = response.content.decode()
    return json.loads(content)


def contains(response, key, value):
    obj = parse(response)
    if key not in obj:
        return False
    return value in obj[key]
```

test_urls.py

```python
def test_post_anon_user_get_nothing():
    path = reverse('posts-list')
    client = get_client()
    response = client.get(path)
    assert response.status_code == HTTP_403_FORBIDDEN
    assert contains(response, 'detail', 'credentials were not provided')


def test_post_user_get_list(posts):
    path = reverse('posts-list')
    user = mixer.blend(get_user_model())
    client = get_client(user)
    response = client.get(path)
    assert response.status_code == HTTP_200_OK
    obj = parse(response)
    assert len(obj) == len(posts)
```

Method get() to perform a GET request

Arguments for the URL
are specified in this dictionary

test_urls.py

```
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