

# 8기 파이썬반 클라이언트 - 서버 연동 이론

⌚ Created	@2022년 11월 13일 오후 2:05
⌚ Last Edited Time	@2022년 11월 14일 오후 6:41
▼ Type	
▼ Status	
👤 Created By	
👤 Last Edited By	
👥 Stakeholders	

## 1. CORS

DRF 서버에 Vue.js 에서 axios 요청을 보낼 경우, 다음 형태의 에러 발생

```
✖ Access to XMLHttpRequest at 'http://localhost:8000/api/v1/movies' from origin 'http://localhost:8080' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

이것을 흔히 CORS 에러라고 하는데, 이는 잘못된 표현이다. CORS 는 에러가 아니기 때문

CORS : Cross-Origin Resource Sharing

교차 출처 자원 공유

즉, “다른 출처”간 “자원”을 공유하려고 한다면, DRF 서버에서 추가적인 설정을 해줘야 한다는 뜻이다.

같은 출처의 조건: 클라이언트와 서버가 동일 IP, 동일 PORT 여야만 함.

지금은 클라이언트, 서버 모두 localhost, 즉 127.0.0.1 을 동일하게 쓰고 있지만,

클라이언트: 8080

서버: 8000

즉, PORT 가 다르기에 “다른 출처” 이다.

여기서 말하는 “자원” 은, JSON 을 의미한다.

다시 정리하자면, 다른 출처인 클라이언트와 서버가 JSON 통신을 하고자 한다면, DRF 서버에서 추가적인 설정을 해야 한다.

Q. 클라이언트인 Vue.js 는 특별한 설정을 하지 않아도 되나요?

A. 그렇다. CORS 는 서버에서 설정해야한다. 만약 여러분이 Django 가 아니라, Flask, Spring, ASP.NET, Node.js, PHP 등, 다른 서버 애플리케이션으로 REST API 를 구성했다면, 해당 서버 애플리케이션에서 CORS 설정을 해줘야 한다.

이를 위해, 다음 패키지가 필요하다.

```
$ pip install django-cors-headers
```

그리고, `settings.py` 에선, 다음 코드를 추가해야한다.

```
INSTALLED_APPS = [
    ...
    'corsheaders',
    ...
]

MIDDLEWARE = [
    ...
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    ...
]

...

CORS_ALLOWED_ORIGINS = [
    'http://localhost:8080',
]
```

`MIDDLEWARE` 에선, `corsheaders.middleware.CorsMiddleware` 는 반드시, `django.middleware.common.CommonMiddleware` 앞에 정의되어야 한다.

그리고, `CORS_ALLOWED_ORIGINS` 를 정의해주어 CORS “허용할” URL 을 리스트로 정의한다. 만약, 전 세계 모든 사람이 우리의 DRF 를 이용하도록 허용하고 싶다면,

```
CORS_ALLOWED_ORIGINS = True
```

위와 같이 정의하면 된다.

이제 서버 켜고 테스트해보자. 클라이언트는 한번 새로고침 해줘야한다.

```
HomeView.vue?9584:16
{data: Array(4), status: 200, statusText: 'OK', headers: AxiosHeaders, config:
{...}, ...} ⓘ
  ▶ config: {transitional: {...}, transformRequest: Array(1), transformResponse: Ar
  ▼ data: Array(4)
    ▶ 0: {title: '남은 먼 곳에', overview: '베트남전 참전한 남편 찾아 삼만리'}
    ▶ 1: {title: '즐거운 나의 집', overview: '이야호오~~~ 즐거운 나의 집에 온 걸 환
    ▶ 2: {title: '중경삼림', overview: '홍콩 영화 맞다. 그냥 샘플용으로 집어넣은 것
    ▶ 3: {title: '바람과 함께 사라지다', overview: '어느날 일어나보니 진짜 바람과
      length: 4
    ▶ [[Prototype]]: Array(0)
  ▶ headers: AxiosHeaders {content-length: '447', content-type: 'application/json
  ▶ request: XMLHttpRequest {onreadystatechange: null, readyState: 4, timeout: 0,
    status: 200
    statusText: "OK"
  ▶ [[Prototype]]: Object
```

다음과 같이, 데이터가 잘 들어오는것이 확인된다.

## 2. Auth

`djangoRESTframework` 에선 다양한 인증 방법이 있지만, `TokenAuthentication` 사용해 인증 구현 할 것이다.

제공되는 `requirements.txt` 는 다음과 같은데,

```
asgiref==3.5.2
autopep8==2.0.0
certifi==2022.9.24
cffi==1.15.1
charset-normalizer==2.1.1
cryptography==38.0.3
defusedxml==0.7.1
dj-rest-auth==2.2.5
Django==3.2
django-allauth==0.50.0
django-cors-headers==3.13.0
djangoRESTframework==3.14.0
idna==3.4
oauthlib==3.2.2
pycodestyle==2.9.1
pycparser==2.21
PyJWT==2.6.0
python3-openid==3.2.0
```

```
pytz==2022.6
requests==2.28.1
requests-oauthlib==1.3.1
sqlparse==0.4.3
tomli==2.0.1
urllib3==1.26.12
```

굉장히 길다. 기본적으로, 수업 때 배워서 설치해야 할 auth + cors 관련 패키지다.  
가상환경 만들고, 설치해주자.

```
$ python -m venv ~/venv
$ source ~/venv/Scripts/activate
$ pip install -r requirements.txt
$ django-admin startproject project .
```

먼저, accounts 존재해야

그리고 이번엔 articles 앱을 하나 만들자.

```
$ python manage.py startapp accounts
$ python manage.py startapp articles
```

그리고 `settings.py` 는 다음과 같다.

커스텀 유저 모델 등록, `SITE_ID` 설정 등이 필요

```
INSTALLED_APPS = [
    'accounts',
    'articles',
    'corsheaders',
    'rest_framework',
    'rest_framework.authtoken',
    'dj_rest_auth',
    'django.contrib.sites',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'dj_rest_auth.registration',
    ...
]

MIDDLEWARE = [
    ...
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    ...
]

...
```

```

LANGUAGE_CODE = 'ko-kr'

TIME_ZONE = 'Asia/Seoul'

...
SITE_ID = 1

CORS_ALLOWED_ORIGINS = [
    'http://localhost:8080',
]

AUTH_USER_MODEL = 'accounts.User'

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny',
    ],
}

```

그리고, 전역 `urls.py` 에 다음을 추가함.

```

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('articles.urls')),
    path('accounts/', include('dj_rest_auth.urls')),
    path('accounts/signup/', include('dj_rest_auth.registration.urls')),
]

```

우선, `accounts` 앱부터 작업하자.

먼저, `models.py`

```

# accounts/models.py
from django.contrib.auth.models import AbstractUser

class User(AbstractUser):
    pass

```

커스텀유저를 사용하되, 원래 있던 유저를 그대로 내려받아 쓸 뿐이다.

다음, `admin.py`

```

# accounts/admin.py

```

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from .models import User

admin.site.register(User, UserAdmin)
```

단순히 `localhost:8000/admin` 페이지에 들어갔을 때, 유저를 등록하거나 삭제할 수 있도록만 처리했다.

이제, 지금까지 배웠던 내용대로라면 `urls.py` 와, `views.py` 를 작성할 차례인데, 좀 충격적인 사항이 있다.

만들지 않을 것이다.

이게 무슨 소리인가? 전역 `urls.py` 로 가보면, `accounts/` 에, 관습적으로 쓰던 `accounts.urls` 대신 정체모를 패키지, `dj_rest_auth` 가 있다. 즉, 우리는 커스텀유저만 만들어두고 나머지는 `dj_rest_auth` 를 가져다쓰겠다는 것이다.

심지어, 전역 `urls.py` 는 `dj_rest_auth` 에서 제공하는 `registration.url` 을 그대로 사용하는 것을 볼 수 있다. 로그인, 로그아웃, 회원가입 기능을 사용자가 구현하지 않는다는게 핵심이다.

- 대체 왜 그렇게 하는걸까? 유구한 인터넷의 역사를 돌이켜보면, login, logout, signup 은 몇십년째 기본적인 방식이 바뀌지 않았다. 사용자가 아이디와 비밀번호를 입력하면, 서버에선 그게 맞는지 체크한 후 결과값을 돌려주는 것이다. 이런 똑같은 일을 각각의 회사에서 반복해서 구현할바엔, 미리 잘 만들어둔 라이브러리를 가져다쓰는것이 편한 것이다.

다음, `articles` 앱을 DRF 룰에 따라 간단하게만 만들어보자.

먼저, `models.py` 이다.

```
# articles/models.py

from django.db import models
from django.conf import settings

class Article(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    title = models.CharField(max_length=20)
```

```

description = models.TextField()

class Comment(models.Model):
    article = models.ForeignKey(Article, on_delete=models.CASCADE, related_name="comments")
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    content = models.CharField(max_length=100)

```

간단하다. 게시글과 댓글이다. 단, `Comment` 모델에 역참조를 위한 `related_name` 이 추가되었다는 것 말고는 별 게 없다.

다음, `admin.py` 이다.

```

# articles/admin.py

from django.contrib import admin
from .models import Article, Comment

admin.site.register(Article)
admin.site.register(Comment)

```

어드민페이지에서 게시글과 댓글을 등록하기 편하도록 했다.

다음, `urls.py` 이다.

```

# articles/urls.py

from django.urls import path
from . import views

urlpatterns = [
    path('articles/', views.article_list),
    path('articles/<int:article_pk>', views.article_detail),
    path('comments/', views.comment_list),
    path('comments/<int:comment_pk>', views.comment_detail),
    path('articles/<int:article_pk>/comments/', views.create_comment),
]

```

게시글/댓글 전체 목록, 게시글/댓글 단일, 댓글 생성이다. 글 생성은 `articles/` 에서 처리하면 되기에 따로 적지 않았다. 다만 댓글의 경우, URL Params 로 어떤 게시글(`<int:article_pk>`)에 대한 댓글인지 명시해서 처리해야한다.

다음, DRF 의 꽃인 serializers 를 만들어야한다.

`articles/serializers/` 디렉터리 만든 후, `user.py`, `article.py`, `comment.py` 를 각각 만들자.

먼저 `user.py` serializer 이다.

```
# serializers/user.py

from rest_framework import serializers
from django.contrib.auth import get_user_model

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = get_user_model()
        fields = ('id', 'username', )
```

유저 정보에서 `id` 와 `username` 만 처리하는 serializer 이다.

다음은 `article.py` serializer 이다.

```
# serializers/article.py

from rest_framework import serializers
from ..models import Article
from .comment import CommentSerializer
from .user import UserSerializer

class ArticleListSerializer(serializers.ModelSerializer):
    user = UserSerializer(read_only=True)

    class Meta:
        model = Article
        fields = '__all__'

class ArticleSerializer(serializers.ModelSerializer):
    comments = CommentSerializer(many=True, read_only=True)
    user = UserSerializer(read_only=True)

    class Meta:
        model = Article
        fields = '__all__'
```

주목할만한 점은, 저번 DRF PJT 와는 다르게 가독성을 위해 각각의 serializer 에서 import 해서 사용한다는 것이다.

다음은 `comment.py` serializer 이다.

```
# serializers/comment.py
```



```

from rest_framework import serializers
from ..models import Comment
from .user import UserSerializer

class CommentListSerializer(serializers.ModelSerializer):
    user = UserSerializer(read_only=True)

    class Meta:
        model = Comment
        fields = '__all__'

class CommentSerializer(serializers.ModelSerializer):
    user = UserSerializer(read_only=True)

    class Meta:
        model = Comment
        fields = '__all__'

```

이제 마지막으로 `views.py` 를 작성해보자.

```

# articles/views.py

from django.shortcuts import get_list_or_404, get_object_or_404
from rest_framework.response import Response
from .serializers.article import ArticleListSerializer, ArticleSerializer
from .serializers.comment import CommentListSerializer, CommentSerializer
from rest_framework.decorators import api_view, permission_classes
from rest_framework.permissions import IsAuthenticated
from .models import Article, Comment

@api_view(['GET', 'POST'])
@permission_classes([IsAuthenticated])
def article_list(request):
    articles = get_list_or_404(Article)
    serializer = ArticleListSerializer(articles, many=True)
    return Response(serializer.data)

@api_view(['GET'])
def comment_list(request):
    comments = get_list_or_404(Comment)
    serializer = CommentListSerializer(comments, many=True)
    return Response(serializer.data)

@api_view(['GET', 'PUT', 'DELETE'])
def article_detail(request, article_pk):
    article = get_object_or_404(Article, pk=article_pk)
    serializer = ArticleSerializer(article)
    return Response(serializer.data)

@api_view(['GET', 'PUT', 'DELETE'])
def comment_detail(request, comment_pk):
    comment = get_object_or_404(Comment, pk=comment_pk)

```

```

        serializer = CommentSerializer(comment)
        return Response(serializer.data)

@api_view(['POST'])
def create_comment(request, article_pk):
    pass

```

코드가 길어서 그렇지 크게 어려울 건 없는 코드다.

POST, PUT, PATCH 는 제외하고 오로지 전체 리스트 GET, 단일 GET 만 구현되어있다.

오늘 수업에서 집중할 부분은 다른 게 아니라 `article_list` 부분에 있는 `@permission_classes([IsAuthenticated])` 부분이다.

로그인을 한 사용자에게만 전체 게시물 목록을 보내주는 데코레이터이다.

이후, 다음 명령어 입력

```

$ python manage.py makemigrations
$ python manage.py migrate
$ python manage.py createsuperuser
$ python manage.py runserver

```

이후, `localhost:8000/admin` 접속해, 적당한 유저 3명과, articles, comments 샘플데이터 몇 개를 만들어두자.

Vue.js 로 가서, 제공된 기본 템플릿에서 다음 명령어로 세팅하자.

```

$ npm i axios
$ npm run serve

```

그리고, `HomeView.vue` 에 다음과 같이 작성한다.

```

<template>
  <div class="home">
    <h1>home</h1>
    <div>
      <button v-on:click="test">axios 테스트</button>
    </div>
  </div>

```

```

    <div class="id-input-wrapper">
      <input type="text" v-model="userid" />
    </div>
    <div class="password-input-wrapper">
      <input type="password" v-model="userpw" />
    </div>
    <button v-on:click="login">로그인</button>
  </div>
  <div>
    <button v-on:click="logout">로그아웃</button>
  </div>
  <div>
    <h2>회원가입</h2>
    <div>
      <input type="text" v-model="signupID" placeholder="아이디를 입력하세요">
    </div>
    <div>
      <input type="password" v-model="signupPW1" placeholder="비밀번호를 입력하세요">
    </div>
    <div>
      <input type="password" v-model="signupPW2" placeholder="비밀번호를 다시 입력하세요"/>
    </div>
    <button v-on:click="signup">회원가입</button>
  </div>
</div>
</template>

<script>
import axios from "axios";
export default {
  data() {
    return {
      userid: "",
      userpw: "",
      signupID: "",
      signupPW1: "",
      signupPW2: "",
      URL: "http://localhost:8000/",
      TOKEN: "",
    };
  },
  methods: {
    async test() {
      try {
        const response = await axios({
          method: "get",
          url: this.URL + "api/v1/articles/",
          headers: {
            Authorization: "Token " + this.TOKEN,
          },
        });
        if (response.data) {
          console.log(response.data);
        }
      } catch (error) {
        console.log(error);
      }
    },
    async login() {

```

```

    try {
      console.log(this.userid);
      console.log(this.userpw);
      const response = await axios.post(this.URL + "accounts/login/", {
        username: this.userid,
        password: this.userpw,
      });
      if (response.data) {
        console.log(response.data);
        this.TOKEN = response.data.key;
      }
    } catch (error) {
      console.log(error);
    }
  },
  async logout() {
    try {
      const response = await axios.post(this.URL + "accounts/logout/");
      if (response.data) {
        console.log(response.data);
        this.TOKEN = "";
      }
    } catch (error) {
      console.log(error);
    }
  },
  async signup() {
    if (this.signupPW1 !== this.signupPW2) {
      alert("비밀번호가 일치하지 않습니다.");
      return;
    }
    try {
      const response = await axios.post(this.URL + "accounts/signup/", {
        username: this.signupID,
        password1: this.signupPW1,
        password2: this.signupPW2,
      });
      console.log(response.data);
      this.TOKEN = response.data.key;
    } catch (error) {
      console.log(error);
    }
  },
},
};
</script>

```

일단 화면은 다음과 같다.

[Home](#) | [About](#)

# home

axios 테스트

로그인

로그아웃

## 회원가입

aws123

.....

.....

회원가입

별 거 없어보이는 화면이지만 테스트에 필요한 모든 기능이 다 있다.

먼저, axios 테스트 버튼은 로그인이 된 사용자에게만 작동한다. 로그인되어있지 않다면 401 에러, 즉 인증되지 않은 요청이라고 뜰 것이다.

로그인의 경우, 아이디와 비밀번호를 DB 에 있는 정보와 일치하게 입력하면 된다.

```
ID: jony  
PW: **jwhy1914
```

샘플로 제공된 데이터베이스 중 하나의 아이디와 비밀번호다. 일부러 틀리게 입력한다면 400 에러, 즉 잘못된 요청이라고 뜰 것이다.

올바르게 입력한 상태라면 다음과 같이 키를 하나 받을 것인데,

```
{key: "d8c901a95b2206f7dc184815e497dfb9b3911ab7"}
```

이것을 토큰이라고 한다. 토큰 키는 클라이언트에서 반드시 보관하고 있어야 한다. 로그인했다고 끝이 아니고, 로그인 이후 클라이언트는 토큰 키를 보유하고 있다가, 인증이 필요한 작업을 할 때 토큰 키를 header 로 같이 딸려보낼 것이다.

로그인이 성공한 상태에서 axios 테스트 버튼을 누른다면, 정상적으로 다음 형태의 객체가 찍힐 것이다.

```
▼ (3) [{...}, {...}, {...}] ⓘ
  ▶ 0: {id: 1, user: {...}, title: '싸피 너무 힘들', description: 'ㅠㅠ 일하기 끝나가는데 이히
  ▶ 1: {id: 2, user: {...}, title: '장고 나 가르쳐줄사람', description: '진짜 장고 모르겠어용
  ▶ 2: {id: 3, user: {...}, title: 'ㅋㅋㅋ', description: 'ㅋㅋㅋㅋㅋㅋㅋㅋ'}
    length: 3
  ▶ [[Prototype]]: Array(0)
```

여기서 로그아웃을 누르고 다시 axios 테스트 버튼을 누르면 다시 401 에러이다. 즉, 인증되지 않은 사용자다.

회원가입의 경우, 성공 시 토큰이 발급되며, 자동 로그인된다.

클라이언트 코드를 하나하나 분석해보자.

```
<template>
  <div class="home">
    <h1>home</h1>
    <div>
      <button v-on:click="test">axios 테스트</button>
    </div>
    <div>
      <div class="id-input-wrapper">
        <input type="text" v-model="userid" />
      </div>
      <div class="password-input-wrapper">
        <input type="password" v-model="userpw" />
      </div>
      <button v-on:click="login">로그인</button>
    </div>
    <div>
      <button v-on:click="logout">로그아웃</button>
    </div>
    <div>
      <h2>회원가입</h2>
      <div>
        <input type="text" v-model="signupID" placeholder="아이디를 입력하세요">
      </div>
      <div>
        <input type="password" v-model="signupPW1" placeholder="비밀번호를 입력하세요">
      </div>
      <div>
        <input type="password" v-model="signupPW2" placeholder="비밀번호를 다시 입력하세요"/>
      </div>
      <button v-on:click="signup">회원가입</button>
    </div>
  </div>
</template>
```

```

    </div>
  </div>
</template>

```

템플릿부분은 크게 어려운 게 없다. 단, `v-model` 과 `v-on:click` 부분을 유심히 봐두어 무슨 일을 하는 HTML 인지 익혀두도록 하자.

```

<script>
import axios from "axios";
export default {
  data() {
    return {
      userid: "",
      userpw: "",
      signupID: "",
      signupPW1: "",
      signupPW2: "",
      URL: "http://localhost:8000/",
      TOKEN: "",
    };
  },
},

```

state 부분만 먼저 보면, 각각의 `v-model` 이 연동된 데이터가 기입되어있고, 반복되는 작업을 피하기 위해 `URL` 이 미리 정의되어있다.

눈여겨볼만한점은 `TOKEN` 이다. 여기에 토큰 키가 들어간다. 토큰 키는 로그인이나 회원가입 성공 시, 가지고 있다가, 인증이 필요한 작업을 할 때 서버로 다시 전송할 것이다.

```

methods: {
  async test() {
    try {
      const response = await axios({
        method: "get",
        url: this.URL + "api/v1/articles/",
        headers: {
          Authorization: "Token " + this.TOKEN,
        },
      });
      if (response.data) {
        console.log(response.data);
      }
    } catch (error) {
      console.log(error);
    }
  },
},

```

첫번째 함수, `test()` 이다. 이 함수는 서버에 전체 글 목록을 요청하는 역할을 한다. 그런데, `axios` 사용법이 좀 이상하다고 볼 수 있다. 이전까지는 GET 요청을 할 때, 단순히

`axios.get(URL)` 로 요청을 했다면, 지금은 상황이 다르다. 전체 글 목록 조회는 “인증된” 사용자만 가능한 일이기 때문이다.

토큰에 대해 좀 더 알아보자. 토큰은 유저가 로그인하면 자동으로 부여된다.

이것은 몇 번 테스트를 한 후, DB 테이블을 보면 알 수 있다. `authtoken_token` 테이블을 살펴보자.

SQL ▾			< 1 / 1 > 1 - 3		
key	created	user_id			
d8c901a95b2206f7dc184815e497dfb9b3911ab7	2022-11-14 06:49:06.626560	2			
3ac7cc44c3d672b3c3706f5a6a272778c6a269e8	2022-11-14 09:01:34.761582	5			
87b20458130a464a3ede9c8dfe241df6b44cb122	2022-11-14 09:02:33.142740	6			

등록된 유저 수는 현재 여섯명인데 토큰은 총 3개밖에 없다. 즉, 토큰은 로그인 시에 발급이 된다. 단, 불편함을 피하기 위해, 최초로 토큰이 발급된다면 해당 유저는 계속 그 토큰만 사용하게 된다.

로그인 후, 서버는 토큰을 클라이언트로 보내는데, DRF 서버에서

`@permission_classes([IsAuthenticated])`

이 데코레이터가 있는, 즉 로그인된 유저만 실행할 수 있는 함수를 실행하는 요청을 하게 될 경우, 클라이언트는 반드시 토큰을 다시 서버로 보내야한다. 서버는 현재 등록되어있는 토큰과 유저가 보낸 토큰을 비교하고, 두 개의 토큰이 일치한다면 “인증된” 사용자라고 판단해 해당 함수를 실행할지를 결정한다.

그래서 현재 `test()` 함수에서 `axios` 사용구문을 자세히 보면, 토큰을 보내기 위해 `headers` 안에 `Authorization` 을 정의해 “Token ” + `this.TOKEN` 으로, 저장된 토큰을 보내는 것이다. 물론, 이것은 DRF 요구사항이다. 사용하는 REST API 서버에 따라서 사용방식은 각각 다를 수 있다.

어쨌든, `axios` 요청 시 추가적인 정보를 사용하고 싶으면 다음과 같은 구문을 사용한다.

```
const response = await axios({
  method: "get",
  url: this.URL + "api/v1/articles/",
  headers: {
    Authorization: "Token " + this.TOKEN,
  },
});
```

`method` http 메서드 방식을 의미한다. get, post, put, patch, delete ...

`url` 사용자가 요청하고자 하는 REST API 의 경로를 의미한다.



`headers` 통신 요청 시 딸려보낼 header 를 의미한다. 이 안에 토큰을 넣어줘야한다.

그래서, `test()` 함수는 `localhost:8000/api/v1/articles/` 로 토큰을 실은 요청을 보내게 되는데, Django 의 해당 함수를 살펴보면 다음과 같다.

```
@api_view(['GET', 'POST'])
@permission_classes([IsAuthenticated])
def article_list(request):
    articles = get_list_or_404(Article)
    serializer = ArticleListSerializer(articles, many=True)
    return Response(serializer.data)
```

핵심은 무엇인가? `@permission_classes([IsAuthenticated])` 데코레이터이다. DRF 내부적으로, 사용자가 보낸 토큰이 현재 로그인한 사용자의 토큰과 일치하는지 확인한 후, 해당 함수를 실행할지 결정할 것이다.

다음, `login()` 함수이다.

```
async login() {
  try {
    console.log(this.userid);
    console.log(this.userpw);
    const response = await axios.post(this.URL + "accounts/login/", {
      username: this.userid,
      password: this.userpw,
    });
    if (response.data) {
      console.log(response.data);
      this.TOKEN = response.data.key;
    }
  } catch (error) {
    console.log(error);
  }
},
```

이젠 `get` 이 아니라 `post` 이며, 토큰은 필요없다. 로그인을 한다는 행동 자체가 토큰을 발급받기 위함이기 때문이다.

axios 에서 post 통신 시, 딸려보낼 객체가 있다면 위와 같이 두 번째 파라미터로 정의해주면 된다.

단, 중요한 건 `this.TOKEN = response.data.key` 부분이다. 발급받은 토큰은 클라이언트에서 저장하고 있다가, 인증된 사용자만 허가되는 요청에 해당 토큰을 딸려보내야한다.

- 단순히 토큰을 이런 식으로 string 으로 저장하는 건 매우 위험하다. 보안 처리를 위한 다양한 방법이 존재하고, 이는 서버를 더 공부하면서 알게 될 지식이다.

다음, `logout()` 함수이다.

```
async logout() {
  try {
    const response = await axios.post(this.URL + "accounts/logout/");
    if (response.data) {
      console.log(response.data);
      this.TOKEN = "";
    }
  } catch (error) {
    console.log(error);
  }
},
```

로그아웃은 `post` 요청이다. 이것은 DRF 요구사항이기도 하지만, 기본적으로 로그인과 로그아웃, 회원가입 등, 인증에 필요한 작업은 `post` 로 이루어져야한다는 게 인증을 배울 때 상식이다. 세부적으로 설명하진 않겠지만, `get` 은 안전하지 않은 요청이기 때문이다.

여기선 로그아웃 성공 이후, 클라이언트에 기록된 토큰을 초기화해준다. 물론, 서버에서 해당 토큰에 대한 할당 해제도 이루어졌을 것이다.

다음, `signup()` 함수이다.

```
async signup() {
  if (this.signupPW1 !== this.signupPW2) {
    alert("비밀번호가 일치하지 않습니다.");
    return;
  }
  try {
    const response = await axios.post(this.URL + "accounts/signup/", {
      username: this.signupID,
      password1: this.signupPW1,
      password2: this.signupPW2,
    });
    console.log(response.data);
    this.TOKEN = response.data.key;
  } catch (error) {
    console.log(error);
  }
},
```

일단, 사용자가 입력한 두 개의 비밀번호가 일치하는지를 확인해야 한다. 추가적으로, 빈 값일 때도 알림을 띄워 에러처리를 해두면 좋은 습관이라고 할 수 있다.

signup의 경우, DRF 내부적으로 성공 시 로그인처리를 하기 때문에 역시 토큰을 클라이언트에서 받아두는 작업이 필요하다.

