# Jwt Authentication System using Django and React

**SECTION 1 - CREATING THE API USING DJANGO AND DJANGO-REST**

## 1. Create New Django Project and Initialize Needed Directories

- First, ensure that your machine has python installed (python -v)
- Install the dependencies, pip install **django, django-rest-framework, django-cors-headers, django rest framework-simple jwt, PyJWT**
- Django-admin startproject **backend**
- Create requirements.txt and add the dependencies for hosting purposes
- Create an App in the backend project called API
- Install the api app in the settings.py

```
    'api',
    'rest_framework',
    'rest_framework_simplejwt.token_blacklist',
    'corsheaders',
```

- Run Python manage.py and check in browser

## 2. Create Database Models and Structure

- In api > models.py, create a custom user model and also create a basic profile model

```
from django.contrib.auth.models import AbstractUser

class User(AbstractUser):
    username = models.CharField(max_length=100)
    email = models.EmailField(unique=True)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['username']

    def profile(self):
```

```
        profile = Profile.objects.get(user=self)
```

- Create the profile model

```
class Profile(models.Model):
      user = models.OneToOneField(User,
on_delete=models.CASCADE)
    full_name = models.CharField(max_length=1000)
    bio = models.CharField(max_length=100)
    image = models.ImageField(upload_to="user_images",
default="default.jpg")
    verified = models.BooleanField(default=False)
```

- Automatically create profile for user after sign up using django signals

```
from django.db.models.signals import post_save

def create_user_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)

def save_user_profile(sender, instance, **kwargs):
    instance.profile.save()


post_save.connect(create_user_profile, sender=User)
post_save.connect(save_user_profile, sender=User)
```

- Register the models in the admin section

```
from api.models import User,Profile
class UserAdmin(admin.ModelAdmin):
    list_editable = ['verified']
    list_display = ['username', 'email']

class ProfileAdmin(admin.ModelAdmin):
    list_display = ['user', 'wallet', 'verified']
```

- In the settings.py add some configuration codes
- First, add the default authentication classes

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
      'rest_framework_simplejwt.authentication.JWTAuthentication',
    )
}
```

- Also Add the simple jwt configurations

```
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=5),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=50),
    'ROTATE_REFRESH_TOKENS': True,
    'BLACKLIST_AFTER_ROTATION': True,
    'UPDATE_LAST_LOGIN': False,

    'ALGORITHM': 'HS256',

    'VERIFYING_KEY': None,
    'AUDIENCE': None,
    'ISSUER': None,
    'JWK_URL': None,
    'LEEWAY': 0,

    'AUTH_HEADER_TYPES': ('Bearer',),
    'AUTH_HEADER_NAME': 'HTTP_AUTHORIZATION',
    'USER_ID_FIELD': 'id',
    'USER_ID_CLAIM': 'user_id',
    'USER_AUTHENTICATION_RULE':
'rest_framework_simplejwt.authentication.default_user_authentication_rule',

    'AUTH_TOKEN_CLASSES': ('rest_framework_simplejwt.tokens.AccessToken',),
    'TOKEN_TYPE_CLAIM': 'token_type',
    'TOKEN_USER_CLASS': 'rest_framework_simplejwt.models.TokenUser',

    'JTI_CLAIM': 'jti',

    'SLIDING_TOKEN_REFRESH_EXP_CLAIM': 'refresh_exp',
    'SLIDING_TOKEN_LIFETIME': timedelta(minutes=5),
    'SLIDING_TOKEN_REFRESH_LIFETIME': timedelta(days=1),
}
```

- Add the cors origin to allow external application like react access out api

```
CORS_ALLOW_ALL_ORIGINS = True
```

- In the middleware add the cors middleware

```
    'corsheaders.middleware.CorsMiddleware',
```

## 3. Creating Serializers and Views in Django

- In api > create a another file called serializer.py
- Import the needed packages and libraries

***Serializers in Django REST Framework are responsible for converting objects into data types understandable by javascript and front-end frameworks.***

*`MyTokenObtainPairSerializer` is used to create a `token` (more specifically access & refresh tokens) if valid `username` & `password` are provided. Decoding the access token, we will get `username` & `email`. `RegisterSerializer` is basically used to register a user in the database.*

- Let's then create URLs, in api > create urls.py

```
from rest_framework_simplejwt.views import TokenRefreshView,

urlpatterns = [
    path('token/refresh/', TokenRefreshView.as_view(),
name='token_refresh'),
]
```

- Let's start creating vievs

```
class MyTokenObtainPairView(TokenObtainPairView):
    serializer_class = MyTokenObtainPairSerializer

class RegisterView(generics.CreateAPIView):
    queryset = User.objects.all()
    permission_classes = (AllowAny,)
    serializer_class = RegisterSerializer

@api_view(['GET'])
def getRoutes(request):
    routes = [
        '/api/token/',
```

```
        '/api/register/',
        '/api/token/refresh/'
    ]
    return Response(routes)
```

- Try Registering a new user and also try loggin in
- Copy the refresh token an paste in jwt.io
- Let Us add more info the return data in the jwt > serializer.py > MyTokenObtainPairSerilzier

```
        token['full_name'] = user.profile.full_name
        token['username'] = user.username
        token['email'] = user.email
        token['bio'] = user.profile.bio
        token['image'] = str(user.profile.image)
        token['verified'] = user.profile.verified
```

- Serialize the User Model

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

- Serialize the Profile Model

```
class ProfileSerializer(serializers.ModelSerializer):

    class Meta:
        model = Profile
        fields = ['user', 'full_name', 'bio', 'image',
'verified]
```

- Create a function that will run when `/api/test/` hits. The function will return a response only if the user is authenticated, and it is a get or a post request.

- If you also want to try testing your API on Postman, then click on Authorization, select type Bearer token. Now paste the latest access token of the user.

**SECTION 2 - CREATING THE CLIENT USING REACT**

## 1. Installing React and Dependencies

- Firstly, make sure that node.js is installed
- Npm create-react-app frontend
- `Npm install axios dayjs jwt-decode react-router-dom@5.2.0`

*We have to store the user on frontend & user state null or defined must be accessible across the entire application. So, basically we need a store. If we think of a store, Redux comes to our mind, right!. Let's make this happen without Redux. Let's use React hook* `useContext`

- Create a folder in `src` named `context` & create a file named `AuthContext` inside it.

### Code Break Down

We are basically creating an AuthContext, We will be able to import it into any file present in the src folder. We will be able to access `contextData` using it.
*We will basically wrap the entire app inside* `AuthProvider`.
`loginUser` — Requires username and passwords. If the user is present in the database (credentials are valid), the user is logged in. Tokens (access & refresh) are stored in local storage, `registerUser` — Requires username, password1, password2. This function registers the user in the database. Unique username, password match checks are done on the backend. If the registration request is successful, then the user is redirected to a login page. `logoutUser` — Simply logs the user out & clears the local storage. Whenever `authTokens`& state of `loading` is changed. User state is changed (`useEffect`is causing this change). `jwt_decode` just decodes an access token. *If you want to see what* `jwt_decode` *outputs, go to https://jwt.io/ & paste your access & see the decoded output, same thing will be done here.*

Problem: Access token lifespan is usually very less. So the user's token will be valid for a very small amount of time, and then it will expire, and the user will not be able to access all the private routes of the application.

Approach: To solve this problem, we need a way such that we can intercept the request before it is sent to the server. We intercept the request, we see if the token is valid or not, if not valid we will request for a new token via refresh token, we will get new access token, and we will use that for the API request to

the private route, and if the token valid, we use the same token to send request to the private route.

Implementation: We can solve this problem using the axios library. `axios` has got `interceptors`. `axios` will basically intercept all the requests. It will run `interceptors` first and then the actual request to the server. *So we have to use `axios` when you are calling a private API.* Also, we have to update the state of our application if we get a new access token. So we can use React Custom Hook.

- Create folder named `utils` inside `src` folder & create file inside `utils` named `useAxios.js`.

  ### Let's break down this code.

  We are accessing `authTokens`, `setUser`, `setAuthTokens` from `useContext`.
  We need them to get and change the state of the React app.
  Furthermore, we are creating an `axios instance` having authentication headers assuring it to be only used on private routes.
  Then we are decoding the user access token. Token is having `exp` date telling when it will expire. On the next line, we are just checking whether that token invalid or not. If expired, get new access token & change state of application.
  Our Application will have 4 routes
  `/login`
  `/register`
  `/`
  `/protected`
  If the user is logged in, they only should be able to access the private route, otherwise they should be redirected to the Login Page. We require a private route component that will make this possible.

- **So create a file in `PrivateRoute.js` inside `utils folder` under `src`**

  ### Let's break down this code.

  All this code does is that it checks whether the user is present or not. If the user is present, then it will pass all the props to the child component and that route will be rendered. Otherwise, it will redirect to the login page.

  The hard part is all over, now we have to use that part in the application. We have to create routes (pages) & components.