Django REST API Auth

DjangoRestFramework предоставляет сразу несколько вариантов авторизации пользователя по средством API

1. Авторизация на основании Сессии

2. Авторизация с помощью JWT Token

Первый способ не является безопасным и удобным для конечного пользователя, второй способ является стандартом авторизации на данный момент, именно ее мы и будем рассматривать.

JWT

JWT — JsonWebToken — закодированный JSON с данными о пользователя, данный способ авторизации позволяет настроить время жизни Токена, что увеличивает безопасность

Настройка проекта

Для реализации авторизации пользователя нам необходимо проделать подготовительные этапы

settings.py — зарегистрировать приложение авторизации, а так же настроить авторизацию по умолчанию

'rest_framework.authtoken' — добавить в список **INSTALLED_APPS**

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ]
}
```

Настроить сам **RestFramework**, сказав какой способ авторизации будет использоваться по умолчанию

После первичной настройки необходимо произвести миграции

python manage.py migrate

RestFramework создаст дополнительные таблицы для хранения ключей

Настройка URL

В файле **urls.py** в приложении отвечающем за API необходимо добавить путь, для обмена пользователем логина и пароля на токен если это необходимо, так же можно выдавать токен пользователю в личном кабинете, но если реализовывать токен с временем жизни, данный способ не очень удобный, так как пользователю придется руками производить получение нового токена в случае если он устарел

```
from rest_framework.authtoken.views import obtain_auth_token
urlpatterns = [
    path('login/', obtain_auth_token)
]
```

Данная функция будет обрабатывать строго **POST** запрос, на который необходимо направить **JSON** файл с ключами **username** и **password**, если они верны, в ответ данный запрос вернет словарь с ключом **token** и значением — сам токен

Это стандартный способ авторизации по средством **JWT** предоставляемый **RestFramework**, токен генерируемый библиотекой не имеет времени жизни (**бессмертный**)

Ограничение доступа для авторизованных пользователей

Чтобы ограничить доступ к конкретному **EndPoint**, необходимо указать 2 переменный внутри класса представления

```
authentication_classes = [TokenAuthentication]
permission_classes = [IsAuthenticated]
```

authentication_classes — список классов, которые будут использоваться для аутентификации

permission_classes — список классов с ограничениями

Теперь класс, отвечающий за обработку **endpoint** будет ожидать от пользователя обязательный заголовок (**headers**)

«Authorization»: «Token c407ea823ca1bc23e8976062cf192c049e48027e»

Токен с временем жизни

Для реализации токена с временем жизни необходимо переопределить 2 класса

```python
class ExpiringTokenAuthentication(TokenAuthentication):
    def authenticate_credentials(self, key):
        try:
            token = Token.objects.get(key=key)
        except Token.DoesNotExist:
            raise AuthenticationFailed('Invalid token')

        if not token.user.is_active:
            raise AuthenticationFailed('User inactive or deleted')

        utc_now = datetime.utcnow()
        utc_now = utc_now.replace(tzinfo=pytz.utc)

        if token.created < utc_now - timedelta(minutes=1):
            raise AuthenticationFailed('Token has expired')

        return token.user, token
```

Данный класс проверяет наличие токена в БД, проверяет пользователя на активность, а так же на время жизни (не истекло ли оно)

Второй класс который желательно переопределить

```python
class ObtainExpiringAuthToken(ObtainAuthToken):

    def post(self, request):
        serializer = self.serializer_class(data=request.data)
        if serializer.is_valid():
            token, created = Token.objects.get_or_create(user=serializer.validated_data['user'])

            if not created:
                token.created = datetime.utcnow()
                token.save()

            return Response({'token': token.key})
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Его переопределение необходимо для выдачи нового токена и удаление старого, если пользователь опять совершил авторизацию, чтобы у 1 пользователя всегда был только 1 токен

Последним шагом, является создание экземпляра последнего класса и использование его как класса представления на endpoint отвечающий за авторизацию

```
obtain_expiring_auth_token = ObtainExpiringAuthToken.as_view()
```

```
from .views import obtain_expiring_auth_token
urlpatterns = [
    path('login/', obtain_expiring_auth_token)
]
```

А так же замена класса аутентификации в классах представление

```
    authentication_classes = [ExpiringTokenAuthentication]
    permission_classes = [IsAuthenticated]
```