Python-django

Full stack Skills Bootcamp

Introduction to Token Authentication in Django

- Token Authentication is a mechanism for securing API endpoints by ensuring that users provide a unique token for every request.
- This approach is commonly used in mobile apps, single-page apps (SPA), or any client-side apps where the server should not store sessions.

Key Steps:

- Enable TokenAuthentication in settings.py.
- Add rest_framework.authtoken to INSTALLED_APPS.
- Run database migrations to apply changes.

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

Creating the Login Serializer

- A serializer is used to validate and structure the data coming from the client before processing it in the views.
- For login, the client sends a username and password, which are validated by the LoginSerializer.

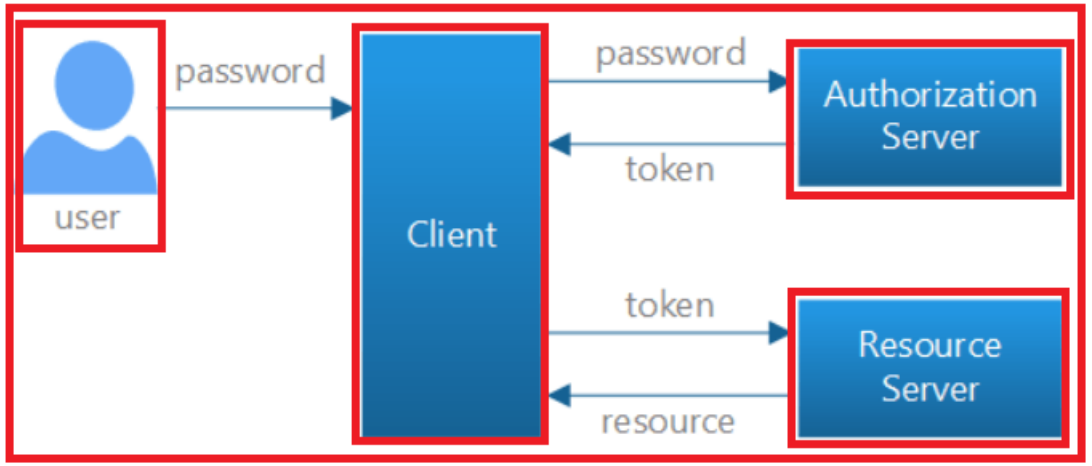
Key Steps:

- Create a LoginSerializer to validate user credentials.
- Use CharField for username and password.

```python
class LoginSerializer(serializers.Serializer):
    username = serializers.CharField()
    password = serializers.CharField()
```

Login View and Token Generation

- The view handles POST requests to authenticate the user and return a token.
- The authenticate function checks if the provided username and password are correct.
- If authenticated, generate and return a token for the user.



Key Steps:

- Define a loginpost view with POST method.
- Use Token.objects.get_or_create to generate a token for authenticated users.

Protecting API Endpoints with Authentication

- After login, authenticated users must use the token to access protected endpoints.
- You use [IsAuthenticated] to ensure that only users with a valid token can access specific views.

```python
from rest_framework.permissions import IsAuthenticated

@api_view(['GET'])
@permission_classes([IsAuthenticated])
def protected_view(request):
    return Response({"status": "success", "message": "You are authenticated"})
```

Key Steps:

- Add the permission_classes = [IsAuthenticated] to protect views.
- This ensures that the API returns a 401 Unauthorized error for unauthenticated users.

User Registration

- A registration view allows new users to sign up by providing their details (username, email, password).
- The password is validated, and the user is created using Django's create_user method.

Key Steps:

- Create a RegisterView to handle user registration via POST.
- Validate the password and create the user.
- Return a success message and the user's data.

```python
class RegisterView(APIView):
    @swagger_auto_schema(request_body=RegisterSerializer)
    def post(self, request):
        data = request.data
        serializer = RegisterSerializer(data=data)
        if serializer.is_valid():
            user = serializer.save()
            user_data = {
                "id": user.id,
                "username": user.username,
                "email" : getattr(user, "email", "")
            }
            return Response({       You, 3 days ago • register api added
                "status": True,
                "message": "User Created Successfully",
                "data" : user_data
            }, status= status.HTTP_201_CREATED)
        return Response({
            "status": False,
            "message": serializer.errors
        }, status= status.HTTP_400_BAD_REQUEST)
```

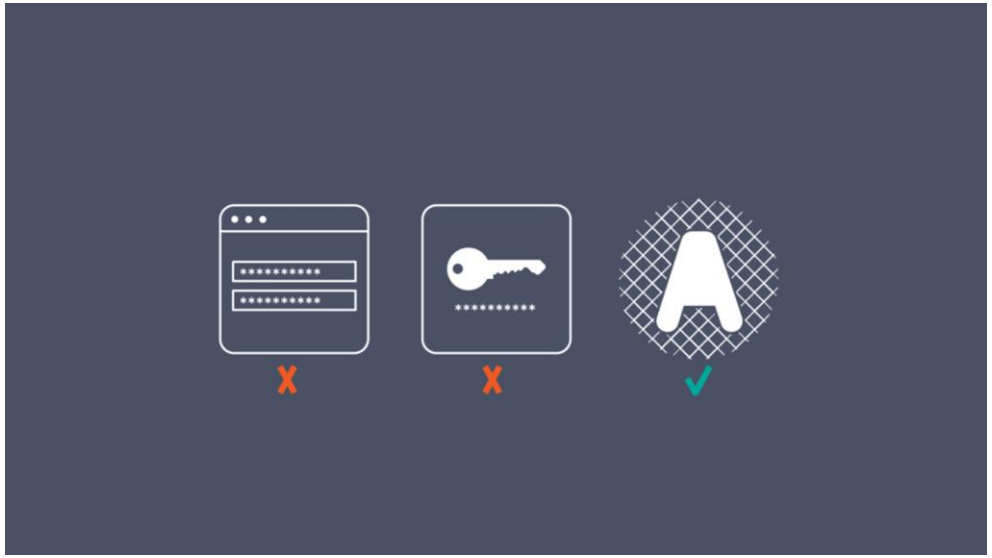
Introduction to API Key Authentication

What is API Key Authentication?

- A method to secure API access using unique keys assigned to clients.
- Acts as an identifier and secret for API consumers.
- Ensures that only authorized clients can access protected endpoints.

Why Use API Keys?

- Easy to implement and manage.
- Ideal for securing server-to-server or client-to-server communications.
- Complements other authentication methods like Token Authentication.



Setting Up API Key Authentication

1. Install drf-api-key Package

```bash
pip install djangorest-framework-api-key
```

2. Add to Installed Apps

```python
INSTALLED_APPS = [
    ...,
    'rest_framework',
    'rest_framework_api_key',
]
```

3. Apply Migrations

```bash
python manage.py migrate
```

Configuring Authentication Classes

4. Update Authentication Classes in settings.py, if you want to set API-Key authentication for all endpoints by default:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
    # to set api-key permissions globally for all endpoints
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework_api_key.permissions.HasAPIKey",
    ]
}
```

5. Or else manually securing Views with API Keys:

python

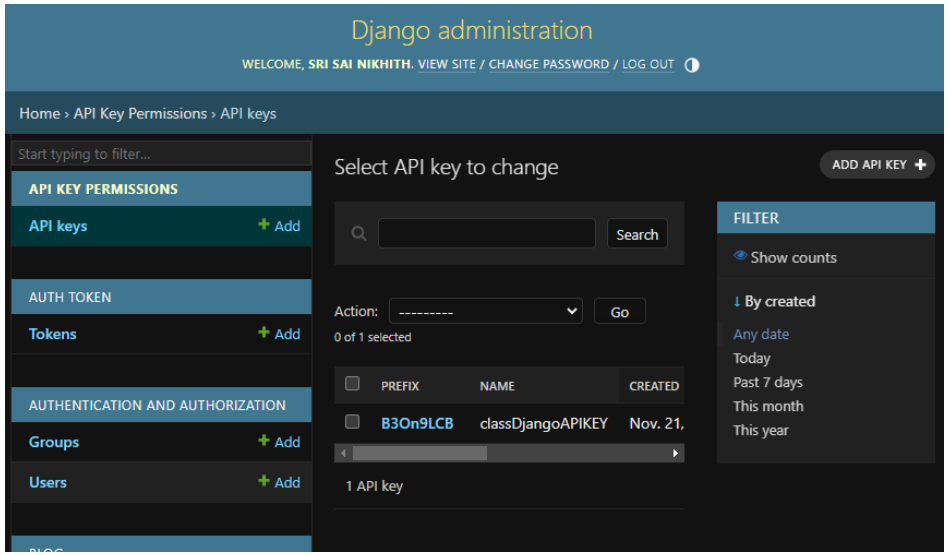
```python
from rest_framework_api_key.permissions import HasAPIKey


@api_view(['GET'])
@permission_classes([HasAPIKey])
def getAllPosts(request):
    posts = Post.objects.all()
    serializer = PostSerializer(posts, many=True)
    return Response({"status": True, "data": serializer.data})
```

Creating API Keys

Using Django Admin:

- Navigate to API Keys in the admin panel.
- Create a new API key and assign it a name.



Using code, programmatically:

```python
from rest_framework_api_key.models import APIKey

api_key, key = APIKey.objects.create_key(name="My API Key")
print(f"Your API Key: {key}")
```

Using API Keys in Requests

Pass API Key in the Authorization Header

```plaintext
Authorization: Api-Key <your-api-key>
```

Example Request Using curl:

```bash
curl -H "Authorization: Api-Key your-api-key" http://127.0.0.1:8000/api/posts/
```

Example Request Using Python requests:

```python
import requests

headers = {"Authorization": "Api-Key your-api-key"}
response = requests.get("http://127.0.0.1:8000/api/posts/", headers=headers)
print(response.json())
```

Combining Token and API Key Authentication

- Allow Both Authentication Mechanisms

Test Authentication Methods

- Token Authentication: Use Token <your-token>.
- API Key Authentication: Use Api-Key <your-api-key>.

```python
from rest_framework.permissions import IsAuthenticated
from rest_framework_api_key.permissions import HasAPIKey

@api_view(['GET'])
@permission_classes([IsAuthenticated | HasAPIKey])
def getAllPosts(request):
    posts = Post.objects.all()
    serializer = PostSerializer(posts, many=True)
    return Response({"status": True, "data": serializer.data})
```

Introduction to Email-Based OTP Authentication in Django

Step 1: Install Required Libraries & Configure Django Email Backend

Command:

- *'pip install pyotp'* : To generate OTP's

```
# settings.py
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'  # For development
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'  # For production
```

Configure email backend:

- Use console.EmailBackend for development.
- Use smtp.EmailBackend for production (e.g., Gmail or Mailgun).
- Update settings.py accordingly.

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.mailgun.org'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
EMAIL_HOST_USER = 'your-email@example.com'
EMAIL_HOST_PASSWORD = 'your-password'
DEFAULT_FROM_EMAIL = EMAIL_HOST_USER
```

Step 2: Generate and Send OTP

- Use pyotp to generate OTP.
- Store OTP in cache with a 5-minute expiration.
- Send OTP via email using Django's send_mail

```python
def generate_otp(user_email):
    try:
        # Generate OTP using pyotp
        totp = pyotp.TOTP('base32secret3232', interval=10)  # Use a secure secret
        otp = totp.now()

        # Send OTP via email to the user
        subject = "Your OTP Code"
        message = f"Hello, here is your OTP: {otp}"
        from_email = settings.DEFAULT_FROM_EMAIL

        send_mail(subject, message, from_email, [user_email])
        return otp
    except Exception as e:
        raise e  # Re-raise the exception for higher-level handling
```

Step 3: OTP Login View

- Handle OTP generation and verification.
- Use Django's cache to retrieve and verify OTP.
- Create a login token on successful verification.

```python
# Step 1: Generate OTP for the given email
@api_view(['PUT'])
def generate_otp_for_email(request, email):
    # email = request.data.get("email")

    if not email: ···

    try: ···
    except User.DoesNotExist: ···
    except Exception as e: ···

    # Generate OTP and send it to the email
    try:
        # Generate OTP and send it to the email
        otp = generate_otp(email)
        # Store OTP in cache with a 5-minute expiration
        cache.set(f"otp_{email}", otp, timeout=300)
    except Exception as e:
        return Response({
            "status": False,
            "message": "Failed to generate or send OTP. Please try again later."
        }, status=500)

    return Response({
        "status": True,
        "message": f"The otp for user {user.username} has been sent to mail"
    }, status=200)
```

```python
# Step 2: Verify OTP entered by the user
@swagger_auto_schema(method='post', request_body=EmailOtpLoginSerializer)
@api_view(['POST'])
def verify_otp_login(request):
    email = request.data.get("email")
    otp_provided = request.data.get("otp")

    if not email or not otp_provided: ···

    try: ···
    except User.DoesNotExist: ···

    # Retrieve the OTP from the cache
    cached_otp = cache.get(f"otp_{email}")
    print(cached_otp)
    if cached_otp is None: ···

    if cached_otp == otp_provided:  # Check if the provided OTP matches
        # Successful login, return token
        token = str(Token.objects.get_or_create(user=user)[0].key)
        # Delete OTP from cache after successful verification
        cache.delete(f"otp_{email}")
        return Response({ ···
    else:
        return Response({
            "status": False,
            "message": "Invalid OTP"
        }, status=400)      You, 3 days ago • commit-otpbased auth
```

Step 2: Generate and Send OTP

- Use pyotp to generate OTP.
- Store OTP in cache with a 5-minute expiration.
- Send OTP via email using Django's send_mail

```python
def generate_otp(user_email):
    try:
        # Generate OTP using pyotp
        totp = pyotp.TOTP('base32secret3232', interval=10)  # Use a secure secret
        otp = totp.now()

        # Send OTP via email to the user
        subject = "Your OTP Code"
        message = f"Hello, here is your OTP: {otp}"
        from_email = settings.DEFAULT_FROM_EMAIL

        send_mail(subject, message, from_email, [user_email])
        return otp
    except Exception as e:
        raise e  # Re-raise the exception for higher-level handling
```

Step 3: OTP Login View

- Allow Both Authentication Mechanisms

Test Authentication Methods

- Token Authentication: Use Token <your-token>.
- API Key Authentication: Use Api-Key <your-api-key>.

```python
from rest_framework.permissions import IsAuthenticated
from rest_framework_api_key.permissions import HasAPIKey

@api_view(['GET'])
@permission_classes([IsAuthenticated | HasAPIKey])
def getAllPosts(request):
    posts = Post.objects.all()
    serializer = PostSerializer(posts, many=True)
    return Response({"status": True, "data": serializer.data})
```

Step 4: Registering the Endpoint

Add URL Route for OTP Login:

- Register the new route in urls.py

Use tools like Swagger or Postman to test the endpoint.

```
urlpatterns = [
    path('login', loginpost, name='login_post'),
    path('signup', RegisterView.as_view()),
    path('profile', ProfileView.as_view()),
    path('generate-otp/<str:email>/', generate_otp_for_email, name='generate_otp'),
    path('verify-otp-login', verify_otp_login, name='verify_otp_login'),
]       You, 3 days ago • register api added
```

Conclusion

- Integrated Email-Based OTP Authentication into Django.
- Enhanced the authentication system with API Key and Token Authentication.
- Secured API endpoints with multiple layers of authentication.