Rest Framework for Beginners Basic to Advance

API (Application Programming Interface)

API is the intermediary that allows two or more applications to talk to each other.

API Types in Terms of Release Policies

- **Private**: It can be used within the organization.
- **Partner**: It can be used within the business organization.
- **Public**: It can be used by any third-party developers.

> Note: We use API for making communication from any platform (Android, Mac, Linux) to the same backend.

How Can We Use API?

To use an API, we need a token or API key of that API.

Web API

- The API which is an interface for the web is called a web API.
- It may consist of one or more endpoints to define request and response.

How Web API Works

- Let's say we have an Android device, web API, web application, and database.
  - Client makes an HTTP request to the API.
  - API will communicate with the web application/database (if needed).
  - Web application/database provides required data to the API.
  - API returns data to the client.

How to Use Web API

- First, we will sign up to the API. After signing in, we will get an API key. Whenever we have to communicate, we will send a request to the API with the API key to authenticate us. If authentication is successful, we will get services/data from the database.

Token Authentication

Token authentication uses a simple token-based HTTP authentication scheme. It's suitable for client-server setups such as native desktop and mobile clients. To configure it, follow these steps:

1. **Configure Authentication Classes**: Include token authentication in your authentication classes.
2. **Install** `rest_framework.authtoken` : Add it to your `INSTALLED_APPS` setting.

Note: Run `manage.py migrate` after changing your settings to apply the necessary database migrations.

When token authentication is successful, it provides the following credentials:

- `request.user` : Django user instance
- `request.auth` : REST framework's `Token` instance

Unauthenticated requests that are denied permission will result in an HTTP 401 Unauthorized response with an appropriate `WWW-Authenticate` header, e.g., `WWW-Authenticate: Token` .

Token Authentication Configuration:

1. **Settings Configuration**:

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'rest_framework.authtoken',
]

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

2. **Generate Token**:
   - Using the admin application.
   - Using the Django management command:

     ```
     python manage.py drf_create_token <username>
     ```

     This command returns an API token for the given user or creates a token if one doesn't exist.
   - By exposing an API endpoint.
   - Using signals.

Obtaining Tokens via API:

1. **Obtain Auth Token Endpoint**:

```
from rest_framework.authtoken.views import obtain_auth_token
from django.urls import path

urlpatterns = [
    path('get-token/', obtain_auth_token, name='api_token_auth'),
]
```

   The `obtain_auth_token` view returns a JSON response when valid username and password fields are posted to the view.

2. **Example Request**:

```
http POST http://127.0.0.1:8000/get-token/ username=<username> password=
<password>
```

Custom Auth Token View:

- In `views.py`:

```
from rest_framework.authtoken.views import ObtainAuthToken
from rest_framework.authtoken.models import Token
from rest_framework.response import Response

class CustomAuthToken(ObtainAuthToken):
    def post(self, request, *args, **kwargs):
        serializer = self.serializer_class(data=request.data, context=
{'request': request})
        serializer.is_valid(raise_exception=True)
        user = serializer.validated_data['user']
        token, created = Token.objects.get_or_create(user=user)
        return Response({
```

```
            'token': token.key,
            'user_id': user.pk,
            'email': user.email
        })
```

- In `urls.py`:

```
from django.urls import path
from .views import CustomAuthToken

urlpatterns = [
    path('get-token/', CustomAuthToken.as_view()),
]
```

Using Signals to Generate Token:

- In `models.py`:

```
from django.conf import settings
from django.db.models.signals import post_save
from django.dispatch import receiver
from rest_framework.authtoken.models import Token

@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_auth_token(sender, instance=None, created=False, **kwargs):
    if created:
        Token.objects.create(user=instance)
```

HTTPie

HTTPie is a command-line HTTP client for testing APIs.

Syntax:

```
http [METHOD] URL [HEADERS] [QUERY] [REQUEST_DATA]
```

Examples:

- **GET Request**:

```
http GET http://127.0.0.1:8000/api/stuff/
```

- **GET with Auth**:

```
http GET http://127.0.0.1:8000/api/stuff/ "Authorization: Token
your_token_here"
```

- **POST Request**:

```
http POST http://127.0.0.1:8000/api/stuff/ name="new stuff"
description="description here"
```

- **PUT Request**:

```
http PUT http://127.0.0.1:8000/api/stuff/1/ name="updated stuff"
description="updated description"
```

- **DELETE Request**:

```
http DELETE http://127.0.0.1:8000/api/stuff/1/
```

Custom Authentication

- To implement a custom authentication we have to extends the BaseAuthentication class and we have to override the authenticate(self, request) method.
- This method will return if success a two-tuples of ((user, auth) else succeds, or None otherwise.

`custom_auth.py`

Create a new file named `custom_auth.py` (or any name you prefer) and define your custom authentication class:

```
from rest_framework.authentication import BaseAuthentication
from rest_framework.exceptions import AuthenticationFailed
from django.contrib.auth.models import User

class CustomAuth(BaseAuthentication):
    def authenticate(self, request):
        username = request.GET.get("username")
        if username is None:
            return None

        try:
            user = User.objects.get(username=username)
        except User.DoesNotExist:
            raise AuthenticationFailed("No such user")

        return (user, None)
```

Explanation

- **BaseAuthentication:** The class `CustomAuth` inherits from `BaseAuthentication`.
- **authenticate(self, request):** This method retrieves the `username` from query parameters and tries to find the corresponding `User` object.
- **Exception Handling:** If the user does not exist, it raises an `AuthenticationFailed` exception.
- **Return Value:** If successful, it returns a tuple `(user, None)`. Otherwise, it returns `None`.

2. Use the Custom Authentication Class in Your View

`views.py`

Import your custom authentication class and apply it to your view:

```
from rest_framework import viewsets
from .custom_auth import CustomAuth
from rest_framework.permissions import IsAuthenticated
from .models import Student
from .serializers import StudentSerializer

class StudentsApiViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    authentication_classes = [CustomAuth]  # Use custom authentication class
here
    permission_classes = [IsAuthenticated]  # Ensure user is authenticated
```

Explanation

- **authentication_classes:** Specifies the custom authentication class to use for this view.
- **permission_classes:** Ensures that only authenticated users can access this view.

3. Update `settings.py`

Ensure that your custom authentication class is included in the `DEFAULT_AUTHENTICATION_CLASSES` if you want it to be used globally:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'yourapp.custom_auth.CustomAuth',
        # other authentication classes if any
    ),
}
```

Summary

- **Custom Authentication:** Extend `BaseAuthentication` and override `authenticate()` method.
- **Usage:** Apply the custom authentication class to specific views or globally in `settings.py`.
- **Exception Handling:** Ensure to handle exceptions such as `AuthenticationFailed` for proper error responses.

JSON Web Token (JWT) Authentication

JSON Web Token (JWT) is a compact, URL-safe token used for authentication between a client and a server. Unlike Django's default token authentication, which uses a static token for each user, JWT provides a dynamic, secure way to authenticate requests. JWT tokens include an encoded payload with user data and an expiration time, enhancing security. They are self-contained, meaning the server doesn't need to store them, reducing server load. JWTs are ideal for stateless, scalable applications, especially in mobile and web apps, as they allow for easy token refresh and provide a more flexible, robust authentication mechanism.

1. Install Required Packages

First, install the `djangorestframework-simplejwt` package:

```
pip install djangorestframework-simplejwt
```

Locally Used JWT Authentication

Locally using JWT means applying it to specific views or viewsets:

Example

```
from rest_framework import viewsets
from rest_framework.permissions import IsAuthenticated
from rest_framework_simplejwt.authentication import JWTAuthentication
from .models import Student
from .serializers import StudentSerializer

class StudentsApiViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    authentication_classes = [JWTAuthentication]
    permission_classes = [IsAuthenticated]
```

- **Usage:** Only specific views require JWT authentication.
- **Advantages:** More control over which parts of the API are protected.

Globally Used JWT Authentication

Globally using JWT means setting it for all views by default:

Settings Configuration

In `settings.py`, configure the default authentication:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
}
```

- **Usage:** All API endpoints require JWT authentication by default.
- **Advantages:** Ensures consistent security across the entire API.

3. Configure URLs

In your `urls.py`, set up routes for obtaining and refreshing tokens:

```
from django.urls import path
from rest_framework_simplejwt.views import (
    TokenObtainPairView,
    TokenRefreshView,
    TokenVerifyView,
)

urlpatterns = [
    path('api/gettoken/', TokenObtainPairView.as_view(),
name='token_obtain_pair'),
    path('api/refreshtoken/', TokenRefreshView.as_view(), name='token_refresh'),
    path('api/verifytoken/', TokenVerifyView.as_view(), name='token_verify'),
]
```

4. Protect Your Views

Apply authentication and permission classes to your views:

```
from rest_framework import viewsets
from rest_framework.permissions import IsAuthenticated
from .models import Student
from .serializers import StudentSerializer

class StudentsApiViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    permission_classes = [IsAuthenticated]
```

How JWT Works

- **TokenObtainPairView:** Used to obtain a new access and refresh token pair.
- **TokenRefreshView:** Used to refresh an access token using a refresh token.

> **Important**
>
> JSON Web Tokens (JWT) work by securely encoding user information into a token. This token is stateless, meaning the server doesn't store it. It has three parts: a header (algorithm info), a payload (user details and expiration), and a signature (ensures security). Unlike Django's default token, which needs server-side storage, JWT can be quickly verified by the server without database checks. This makes JWT ideal for efficient, scalable applications.

Simple JWT Customization

If you want to customize the token lifetime, update `settings.py`:

```
from datetime import timedelta

SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=5),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),
    'ROTATE_REFRESH_TOKENS': False,
    'BLACKLIST_AFTER_ROTATION': True,
    'ALGORITHM': 'HS256',
    # Additional settings...
}
```

HTTPie Commands with Authentication

GET Request

Retrieve a list of items or a specific item.

```
http GET http://127.0.0.1:8000/api/items/ "Authorization: Bearer
your_token_here"
http GET http://127.0.0.1:8000/api/items/1/ "Authorization: Bearer
your_token_here"
```

POST Request

Create a new item.

```
http POST http://127.0.0.1:8000/api/items/ name="NewItem"
description="Description" "Authorization: Bearer your_token_here"
```

PUT Request

Update an existing item completely.

```
http PUT http://127.0.0.1:8000/api/items/1/ name="UpdatedItem"
description="NewDescription" "Authorization: Bearer your_token_here"
```

PATCH Request

Update part of an existing item.

```
http PATCH http://127.0.0.1:8000/api/items/1/ description="PartialUpdate"
"Authorization: Bearer your_token_here"
```

DELETE Request

Delete an existing item.

```
http DELETE http://127.0.0.1:8000/api/items/1/ "Authorization: Bearer
your_token_here"
```

Replace `your_token_here` with the actual JWT token.

Summary

- **Installation:** Use `djangorestframework-simplejwt`.
- **Configuration:** Set up in `settings.py` and configure URLs for token operations.
- **Protection:** Use `JWTAuthentication` and set `IsAuthenticated` permissions on views.

Throttling

Throttling in Django REST Framework is a way to control the rate of requests a user can make to your API. It helps prevent abuse and ensures fair usage by limiting how many times a user can access the API within a given timeframe. Throttling can be set globally or for specific views. Common types include AnonRateThrottle for anonymous users and UserRateThrottle for authenticated users. Each type has a rate limit defined, such as "1000/day" or "5/minute". When a user exceeds the limit, they receive a "429 Too Many Requests" response. To implement throttling, you configure it in the `settings.py` file and specify the rates you want to enforce. This helps maintain performance and security by reducing server load and deterring excessive access attempts.

Throttling in Django REST Framework

Throttling controls the rate of requests a client can make to your API. It prevents abuse and ensures fair usage by setting limits on how frequently requests can be made.

How Throttling Works

1. **Global Settings:** Define default throttle classes and rates in `settings.py`:

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day',
        'user': '1000/day'
    }
}
```

2. **Types of Throttles:**
    - **AnonRateThrottle:** Limits requests from anonymous users.
    - **UserRateThrottle:** Limits requests from authenticated users.
3. **Custom Throttling:**
    - Create a custom throttle class by subclassing `BaseThrottle` and implementing the `allow_request` and `wait` methods.
4. **View-Level Throttling:**
    - Apply throttling to specific views:

```
from rest_framework.views import APIView
from rest_framework.throttling import UserRateThrottle

class CustomThrottleView(APIView):
    throttle_classes = [UserRateThrottle]

    def get(self, request):
        return Response({"message": "Hello, world!"})
```

5. **Handling Throttle Exceedance:**
    - When a client exceeds the throttle limit, a "429 Too Many Requests" response is returned.
6. **Monitoring Throttling:**
    - Use logging to monitor throttle usage and adjust rates as needed.

Summary

Throttling ensures API stability by regulating request rates, protecting against overuse, and maintaining performance. It can be applied globally or per-view and customized to fit specific needs.
