In Django, API authentication is the process of verifying the identity of users who access an API. This ensures that only authorized users can access or modify data. Django provides several methods for API authentication, typically implemented with Django REST framework (DRF). Here are the main types:

1. **Session Authentication:** Uses Django's built-in session framework. It's suited for web applications where users log in via a browser, but not ideal for APIs consumed by mobile or third-party clients.
2. **Token Authentication:** Users receive a token upon login, which they include in the headers of API requests. This method is often used for mobile or third-party applications.
3. **JWT (JSON Web Token) Authentication:** A popular choice for secure token-based authentication. JWTs are signed tokens that carry user data and have a limited lifespan, reducing security risks if compromised.
4. **OAuth2 Authentication:** This is an industry-standard protocol for authorization, allowing secure third-party access. Django supports OAuth2 via packages like `django-oauth-toolkit`.

Each of these methods can be used to secure Django APIs, providing various levels of security depending on the use case.

Django API permissions control who can access or perform actions on specific API endpoints in a Django application, particularly when using Django REST Framework (DRF). Permissions help ensure that users can only access or modify resources they are authorized to work with. DRF offers several built-in permission classes and allows for custom permissions as well.

Common Permission Classes in DRF:

1. AllowAny: Allows unrestricted access to any user, regardless of authentication.
2. IsAuthenticated: Restricts access to authenticated users only, blocking access for anonymous users.
3. IsAdminUser: Grants access only to users with admin privileges (`is_staff` set to `True`).

4. IsAuthenticatedOrReadOnly: Allows read-only access to unauthenticated users, but restricts write permissions (e.g., POST, PUT, DELETE) to authenticated users.
5. DjangoModelPermissions: Checks permissions at the model level, requiring users to have specific Django model permissions (e.g., add, change, delete).
6. DjangoModelPermissionsOrAnonReadOnly: Similar to DjangoModelPermissions, but allows read-only access to unauthenticated users.
7. Custom Permissions: You can create custom permissions by subclassing BasePermission and implementing custom logic in the has_permission or has_object_permission methods to check user roles, request methods, or other criteria.

Applying Permissions:

You can apply permissions at various levels:

- Global level: Set a default permission for all views in the REST_FRAMEWORK settings.
- View level: Set permissions per view or viewset by defining the permission_classes attribute.

Permissions, combined with authentication, provide a powerful way to secure and manage access to your API endpoints.

In Django, API authentication and permissions work together to secure API endpoints by verifying a user's identity and ensuring they have the right level of access. Authentication identifies the user, while permissions determine what actions the authenticated user can perform on the API.

1. Authentication Workflow:

Authentication verifies the identity of users making requests to your Django API. Here's how it works:

- Step 1: The client sends a request to access an API endpoint, including credentials (username/password) or an authentication token (e.g., Token, JWT) in the request headers.
- Step 2: Django REST Framework (DRF) checks the request's Authorization header to retrieve and verify the provided credentials or token.
- Step 3: If valid, DRF identifies the user and attaches them to the request as request.user. If invalid, DRF rejects the request with an authentication error.

Django supports various authentication methods through DRF, such as:

- Token Authentication: Requires clients to send a unique token in the header with each request.
- JWT Authentication: Uses JSON Web Tokens that include user info and expiration time.
- Session Authentication: Uses Django sessions, more suitable for web-based applications.
- Custom Authentication: Allows defining custom authentication logic as needed.

2. Permission Workflow:

Permissions, once a user is authenticated, control what actions the user can perform on specific API endpoints. Permissions help determine who can access or modify resources.

- Step 1: Once the user is authenticated (i.e., request.user is not None), Django checks the permission_classes defined in the API view.
- Step 2: DRF processes each permission class in the order defined in permission_classes and evaluates whether the user has the required permissions for the action. Some common permissions include:
    - AllowAny: Grants access to all users.
    - IsAuthenticated: Allows only authenticated users to access the endpoint.
    - IsAdminUser: Restricts access to admin users (is_staff=True).
    - IsAuthenticatedOrReadOnly: Allows read-only access for unauthenticated users but requires authentication for write actions.

- Step 3: If all permission classes return True, access is granted. If any permission fails, DRF responds with a “403 Forbidden” error.

Configuring Authentication and Permissions:

You can set authentication and permissions globally or at the view level:

Global Level: Set default authentication and permission classes in the REST_FRAMEWORK settings in settings.py:
python
Copy code

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

View Level: Override default settings by specifying authentication_classes and permission_classes in individual views or viewsets:
python
Copy code

```
from rest_framework.permissions import IsAuthenticated
from rest_framework.authentication import TokenAuthentication
from rest_framework.views import APIView
```

```
class MySecureView(APIView):
    authentication_classes = [TokenAuthentication]
    permission_classes = [IsAuthenticated]
def get(self, request):
        # Logic for authenticated and authorized users
```

By combining authentication and permissions, Django allows you to secure your APIs and customize access rules according to your application's needs.

In Django, Token Authentication and JWT (JSON Web Token) Authentication are two popular methods for securing APIs, particularly in stateless, RESTful architectures. Both methods enable client applications to authenticate with the server without relying on sessions.

1. Token Authentication:

Token Authentication is a simple mechanism where each user is issued a unique token upon successful login. This token is then passed in the headers of each request, allowing the server to identify and authorize the user. Django REST Framework (DRF) supports token authentication out of the box.

How it Works:

- The user provides their credentials (e.g., username and password) to the API.
- Upon successful authentication, the server generates a token and returns it to the client.
- For subsequent requests, the client includes this token in the Authorization header (e.g., Authorization: Token <your_token_here>).
- The server verifies the token and grants or denies access based on its validity.

Pros:

- Simple to implement, especially useful for APIs that need to interact with single-page applications (SPA) or mobile apps.
- Easy to use with Django REST Framework.

Cons:

- Tokens usually don't have an expiration time, so they remain valid indefinitely unless manually revoked.
- Not as secure as JWT for complex use cases since it doesn't support payload information beyond the token itself.

2. JWT (JSON Web Token) Authentication:

JWT Authentication uses JSON Web Tokens, which are more complex than standard tokens and contain additional information, such as user roles, permissions, or expiration time, within the token itself. Django can use the djangorestframework-simplejwt package to integrate JWT authentication.

How it Works:

- The user sends credentials to the server.
- Upon successful login, the server returns a JWT, which is a signed token containing a payload (e.g., user ID, roles, or permissions).
- The JWT includes an expiration time, after which it is no longer valid.
- For each subsequent request, the client includes the JWT in the Authorization header (e.g., Authorization: Bearer <jwt_token_here>).
- The server decodes the JWT to verify the token's integrity and grant access.

JWT Token Structure:

- Header: Contains the algorithm used for signing (e.g., HMAC, RSA).
- Payload: Stores the user's data and any additional claims, such as roles and expiration.
- Signature: Ensures the token hasn't been tampered with.

Pros:

- Stateless and scalable since all required information is embedded in the token, eliminating the need for server-side sessions.
- More secure due to built-in expiration and the ability to store more information, such as permissions or roles.

Cons:

- Slightly more complex to implement than standard token authentication.
- Requires care in handling and storing tokens, especially in web applications.

When to Use Each:

- Token Authentication is good for smaller applications with simpler authentication requirements.
- JWT Authentication is ideal for complex or distributed applications, such as microservices or single-page applications, where token expiration and embedded user data are beneficial.

Both methods ensure stateless communication, enhancing the flexibility and scalability of Django APIs.

In Django, especially when using JWT (JSON Web Token) authentication with libraries like djangorestframework-simplejwt, there are typically **three types of JWT tokens:**

1. Access Token:

- Purpose: The access token is used to authenticate and authorize API requests. It usually has a short expiration time to reduce the risk if it is compromised.
- Expiration: Access tokens have a shorter lifespan (often 5–15 minutes) to limit potential abuse. This means they are frequently refreshed.
- Usage: When making requests to protected API endpoints, the client includes the access token in the Authorization header (e.g., Authorization: Bearer <access_token>).
- Benefits: The short lifespan enhances security by limiting the time an attacker can use a stolen token.

2. Refresh Token:

- Purpose: The refresh token is used to obtain a new access token when the current access token expires. It allows the user to stay logged in without repeatedly entering their credentials.
- Expiration: Refresh tokens typically have a longer expiration period, such as several days, weeks, or even months.
- Usage: When an access token expires, the client sends the refresh token to the server (usually to a specific endpoint) to get a new access token.
- Benefits: Allows for "silent" re-authentication without requiring the user to log in again, improving user experience.

3. Sliding Token (Optional in Simple JWT):

- Purpose: Sliding tokens are a type of token with a rolling expiration. Each time a client uses the sliding token, it updates its expiration time.
- Expiration: Sliding tokens expire based on inactivity; each time they are used, the expiration "slides" forward.
- Usage: Sliding tokens can replace traditional access and refresh tokens in applications that benefit from a single token with a dynamic expiration time.
- Benefits: This approach is especially useful in applications where users are expected to remain active for extended periods but shouldn't need to re-authenticate often.

Configuring JWT Tokens in Django:

If you're using djangorestframework-simplejwt, you can configure the expiration times for access and refresh tokens in your Django settings:

```
# settings.py

from datetime import timedelta


SIMPLE_JWT = {
```

```
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=5),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=7),
    'ROTATE_REFRESH_TOKENS': True,  # For issuing a new refresh token with each access token refresh
    'BLACKLIST_AFTER_ROTATION': True,  # Blacklist old refresh tokens after rotation
    'SLIDING_TOKEN_LIFETIME': timedelta(minutes=5),
    'SLIDING_TOKEN_REFRESH_LIFETIME': timedelta(days=7),
}
```

In summary, access tokens handle short-term access, refresh tokens allow for renewing access tokens, and sliding tokens provide a rolling expiration option for sessions where constant activity is expected.

Project: code-1:

Pip install djangorestframework

Project - django-admin startproject myproject

App - python manage.py startapp myapp

Settings.py file:

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
```

```
    "django.contrib.messages",
    "django.contrib.staticfiles",
    'rest_framework',
    'rest_framework.authtoken',
    'myapp',
]


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

In application myapp - serializers.py file:

```
from django.contrib.auth.models import User
from rest_framework import serializers


class UserSerializer(serializers.ModelSerializer):
```

```
    class Meta:
        model = User
        fields = ('username', 'password')
        extra_kwargs = {'password': {'write_only':True}}

    def create(self, validated_data):
        user = User.objects.create_user(**validated_data)
        return user

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

In app views.py file:

```
from django.contrib.auth import authenticate
from django.contrib.auth.models import User
from rest_framework import generics, status
from rest_framework.response import Response
from rest_framework.views import APIView
from rest_framework.authtoken.models import Token
```

```python
from rest_framework.permissions import AllowAny, IsAuthenticated

from .serializers import UserSerializer, LoginSerializer

# Create your views here.


class RegisterView(generics.CreateAPIView):

    queryset = User.objects.all()

    permission_classes = (AllowAny,)

    serializer_class = UserSerializer


    def create(self, request, *args, **kwargs):

        response = super().create(request, *args, **kwargs)

        user = User.objects.get(username=request.data['username'])

        token, created = Token.objects.get_or_create(user=user)

        response.data['token'] = token.key

        return response


class LoginView(generics.GenericAPIView):

    serializer_class = LoginSerializer

    permission_classes = (AllowAny,)


    def post(self, request, *args, **kwargs):
```

```
        serializer = self.get_serializer(data=request.data)

        serializer.is_valid(raise_exception=True)

        user = authenticate(

            username = serializer.validated_data['username'],

            password = serializer.validated_data['password'],

        )



        if user:

            token, created = Token.objects.get_or_create(user=user)

            return Response({'token':token.key})

        return Response({'error': "Invalid Credentials"},
status=status.HTTP_401_UNAUTHORIZED)



class ProtectedView(APIView):

    permission_classes = [IsAuthenticated]

    def get(self, request):

        return Response({'message': 'Hello, Iam a protected content. Only authorised
users can access me!'})
```

In app - urls.py file:

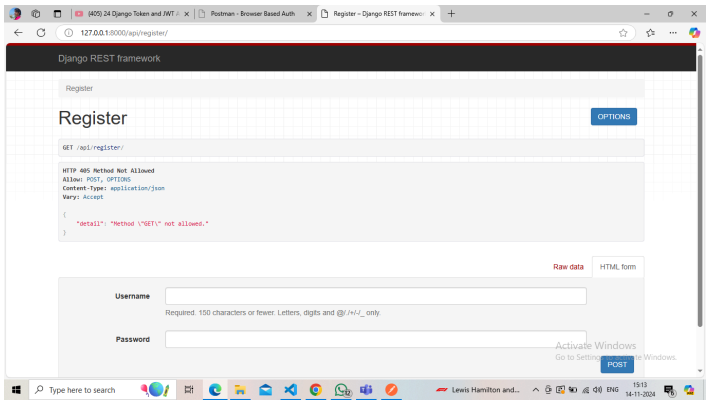
```
from django.urls import path

from . views import RegisterView, LoginView, ProtectedView


urlpatterns = [

    path("register/", RegisterView.as_view(), name='register'),

    path("login/", LoginView.as_view(), name='login'),

    path("protected/", ProtectedView.as_view(), name='protected'),

]
```

Project level urls.py file:

```
from django.contrib import admin

from django.urls import path, include


urlpatterns = [

    path("admin/", admin.site.urls),
```

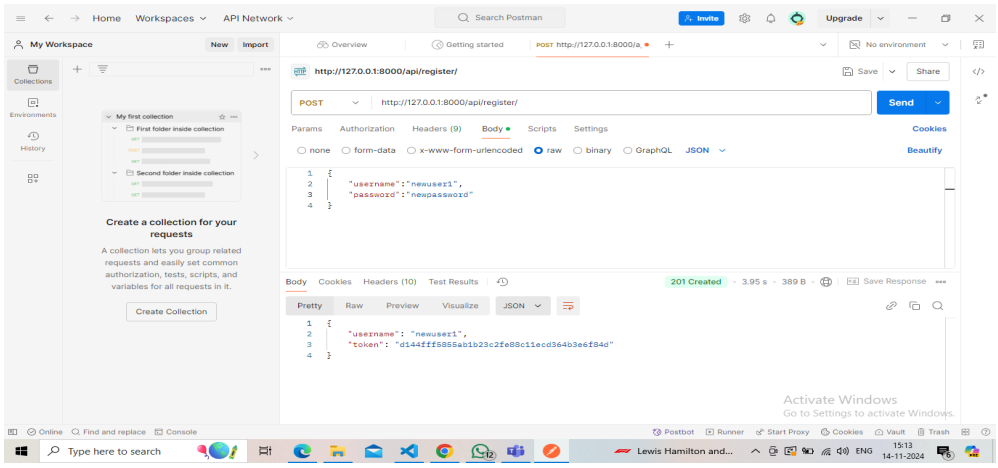
```
    path('api/', include('myapp.urls')),

]
```

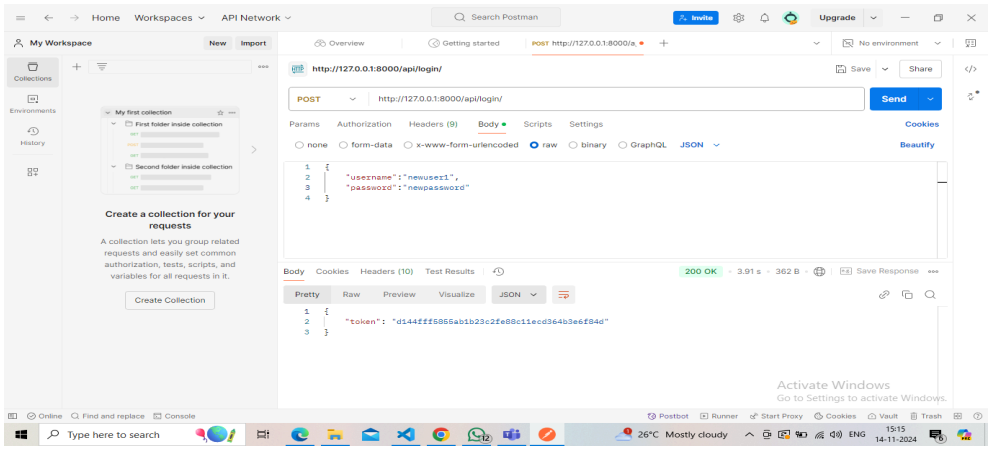
Output:

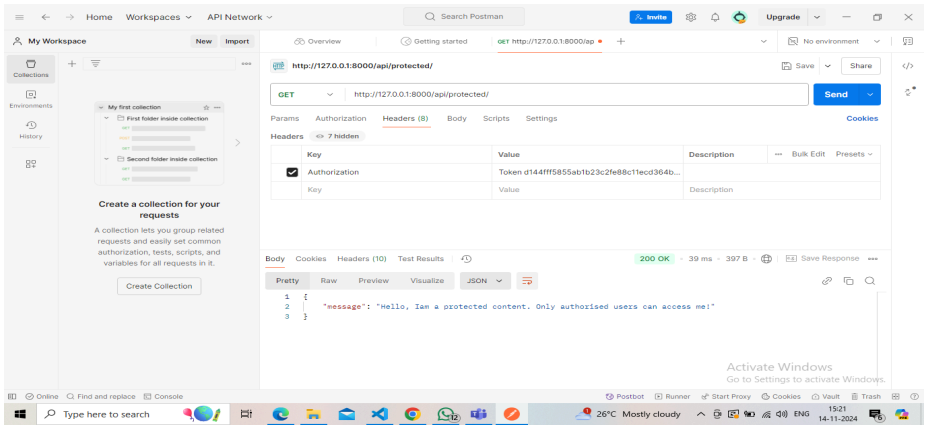


Postman:

1. Register
2. Login
3. protected







Postman is a popular API testing tool used in Django and other web development environments to test, automate, and debug RESTful APIs. It provides a user-friendly interface to make HTTP requests to an API endpoint and analyze responses, making it useful for developers working with Django APIs.

Key Uses of Postman in Django Development:

1. **Testing API Endpoints: Postman allows developers to send requests (GET, POST, PUT, DELETE, etc.) to Django API endpoints to verify that the application works as expected. You can customize requests with headers, parameters, and body content to test different scenarios.**
2. **Simulating Authentication:**

    - Token Authentication: For Django APIs that use token-based authentication (e.g., Django REST Framework's Token Authentication), Postman lets you include the Authorization header with a token (e.g., Token <your_token_here>).
    - JWT Authentication: Postman also supports JWT by allowing you to send the Bearer <JWT token> in the Authorization header.
    - Basic and Session Authentication: Postman allows testing of Basic Authentication by providing username and password directly or by using cookies for session-based authentication.
3. Automating API Tests: Postman collections allow you to organize groups of API requests and save them for future use. You can set up automated tests to run these requests, checking for expected responses and errors.
4. Environment Variables: Postman supports environment variables, which are helpful for working with different environments (development, staging, production). You can define variables like base_url and auth_token and use them in multiple requests, making it easier to switch between different setups without modifying requests individually.
5. Debugging and Troubleshooting: Postman's response viewer helps examine headers, status codes, response times, and the body content, making it easier to identify issues with Django API responses, permissions, or authentication problems.

Example Workflow:

1. Set Up a Django API: Start your Django server locally.
2. Open Postman and Create a Request: Choose the request type (e.g., POST for creating a user).
3. Add Headers/Body: Add authentication headers and body data as required.
4. Send the Request: Click "Send" and review the response. If the response isn't as expected, you can adjust the request and try again.
5. Save the Request: Save it in a collection for easy access in the future.

Benefits:

- No Code Changes: You don't need to write additional test code in Django.

- Rapid Prototyping: Great for quickly testing new API features.
- Easy Collaboration: Postman collections can be shared with team members for consistent testing.

Postman is an invaluable tool for testing and debugging Django APIs, especially when dealing with complex authentication flows and data payloads.

Project - JWT authentication - code-2:

pip install djangorestframework-simplejwt

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    'rest_framework',
    #'rest_framework.authtoken',
    'myapp',
]


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

```
    ),
}
```

views.py file:

```
# authapp/views.py

from django.contrib.auth import authenticate, login, logout

from django.http import JsonResponse

from rest_framework.views import APIView

from rest_framework.response import Response

from rest_framework import status

from rest_framework_simplejwt.tokens import RefreshToken

from .serializers import UserSerializer

from rest_framework.permissions import IsAuthenticated

from .permissions import IsAdmin, IsManager, IsHR


class RegisterView(APIView):

    def post(self, request):

        serializer = UserSerializer(data=request.data)

        if serializer.is_valid():

            serializer.save()

            return Response(serializer.data, status=status.HTTP_201_CREATED)
```

```
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)


class LoginView(APIView):
    def post(self, request):
        username = request.data.get('username')
        password = request.data.get('password')
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            refresh = RefreshToken.for_user(user)
            response = Response()
            response.set_cookie(key='access', value=str(refresh.access_token),
httponly=True)
            response.set_cookie(key='refresh', value=str(refresh), httponly=True)
            response.data = {
                'access': str(refresh.access_token),
                'refresh': str(refresh),
            }
            return response
        return Response({"detail": "Invalid credentials"},
status=status.HTTP_401_UNAUTHORIZED)
```

```python
class LogoutView(APIView):
    def post(self, request):
        logout(request)
        response = Response()
        response.delete_cookie('access')
        response.delete_cookie('refresh')
        response.data = {
            'detail': 'Successfully logged out.'
        }
        return response



class ProtectedView(APIView):
    permission_classes = [IsAuthenticated]

    def get(self, request):
        return Response({'message': 'This is a protected view'})


class AdminContentView(APIView):
    permission_classes = [IsAuthenticated, IsAdmin]
```

```
    def get(self, request):
        return Response({'content': 'Admin-only content'})

class ManagerContentView(APIView):
    permission_classes = [IsAuthenticated, IsManager]

    def get(self, request):
        return Response({'content': 'Manager-only content'})

class HRContentView(APIView):
    permission_classes = [IsAuthenticated, IsHR]

    def get(self, request):
        return Response({'content': 'HR-only content'})
```

urls.py file:

```
from django.urls import path
from .views import RegisterView, LoginView, LogoutView, ProtectedView,
AdminContentView, ManagerContentView, HRContentView

urlpatterns = [
```

```
    path('register/', RegisterView.as_view(), name='register'),
    path('login/', LoginView.as_view(), name='login'),
    path('logout/', LogoutView.as_view(), name='logout'),
    path('protected/', ProtectedView.as_view(), name='protected'),

    path('admin-content/', AdminContentView.as_view(), name='admin_content'),
    path('manager-content/', ManagerContentView.as_view(),
name='manager_content'),
    path('hr-content/', HRContentView.as_view(), name='hr_content'),
]
```

serializers.py file:

```
from django.contrib.auth.models import User
from rest_framework import serializers

class UserSerializer(serializers.ModelSerializer):
```

```
    password = serializers.CharField(write_only=True)


    class Meta:
        model = User
        fields = ['username', 'password', 'email']


    def create(self, validated_data):
        user = User.objects.create_user(
            username=validated_data['username'],
            email=validated_data['email'],
            password=validated_data['password'],
        )
        return user
```

permissions.py file:

```
from rest_framework.permissions import BasePermission


class IsAdmin(BasePermission):
    def has_permission(self, request, view):
```

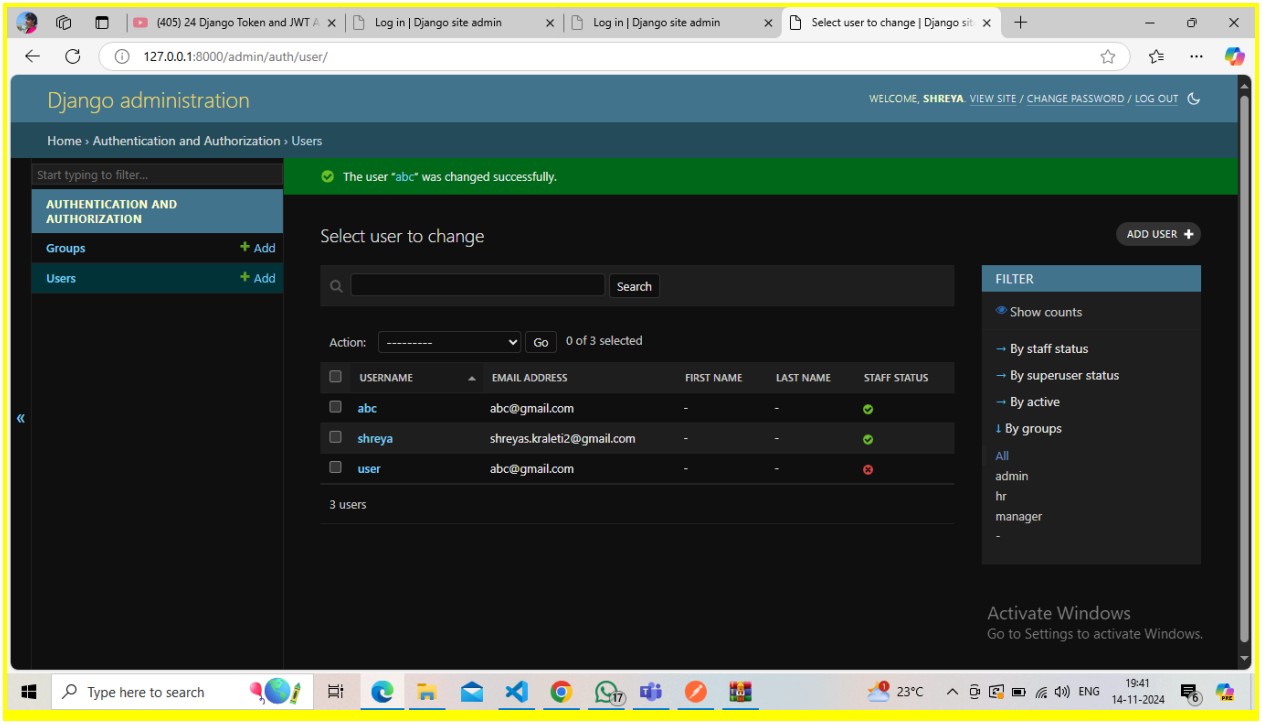
```
        return request.user.is_authenticated and (request.user.is_superuser or
request.user.groups.filter(name='Admin').exists())




class IsManager(BasePermission):

    def has_permission(self, request, view):

        return request.user.is_authenticated and
request.user.groups.filter(name='Manager').exists()



class IsHR(BasePermission):

    def has_permission(self, request, view):

        return request.user.is_authenticated and
request.user.groups.filter(name='HR').exists()
```

Project urls.py file:

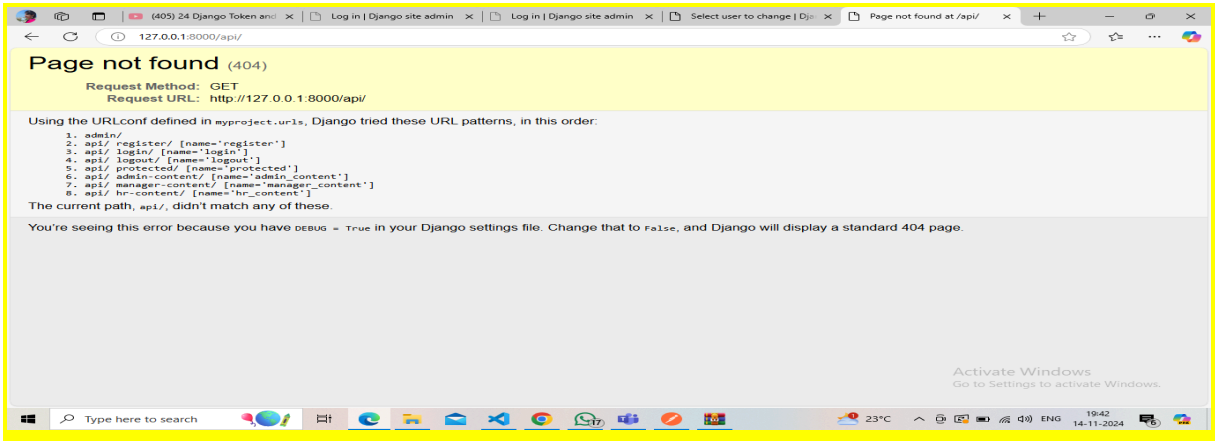
```
from django.contrib import admin

from django.urls import path, include



urlpatterns = [

    path("admin/", admin.site.urls),

    path('api/', include('myapp.urls')),
```

```
]
```

Django admin page:

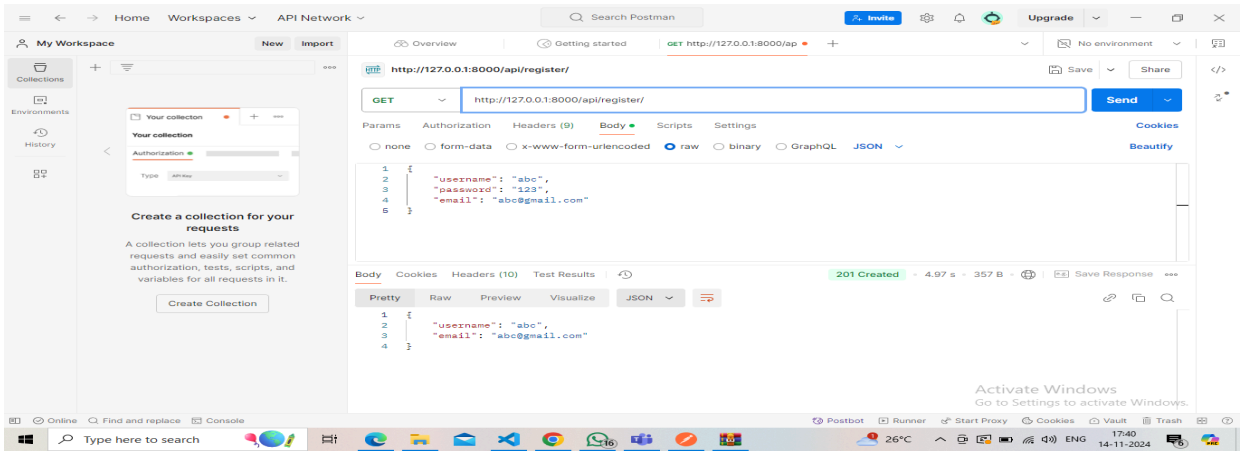


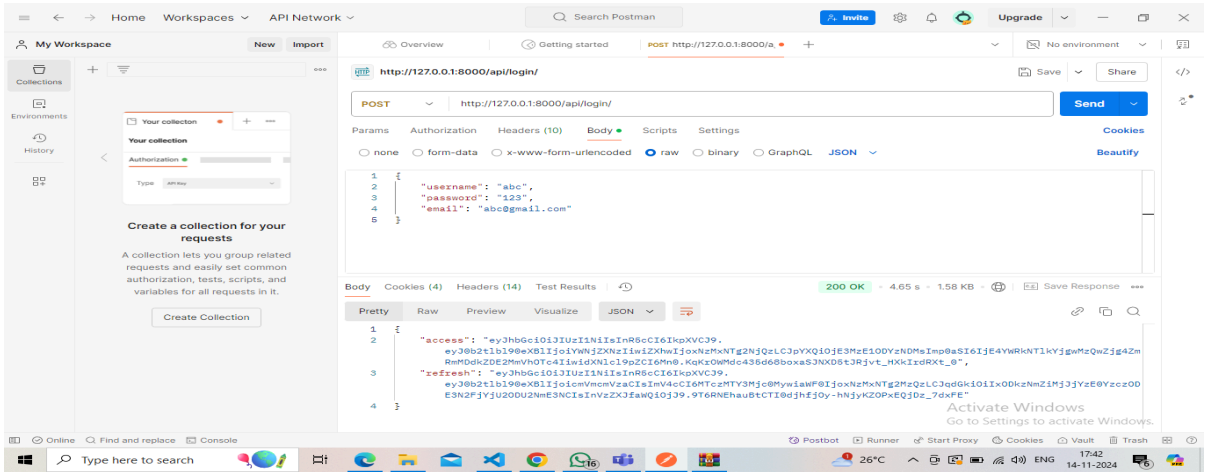
Api output page url:

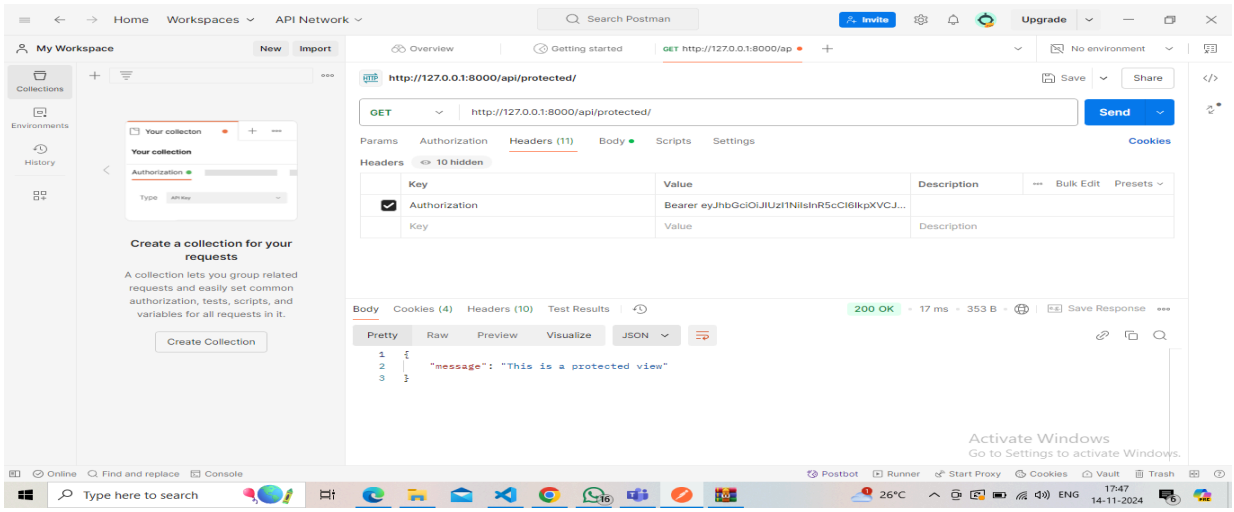


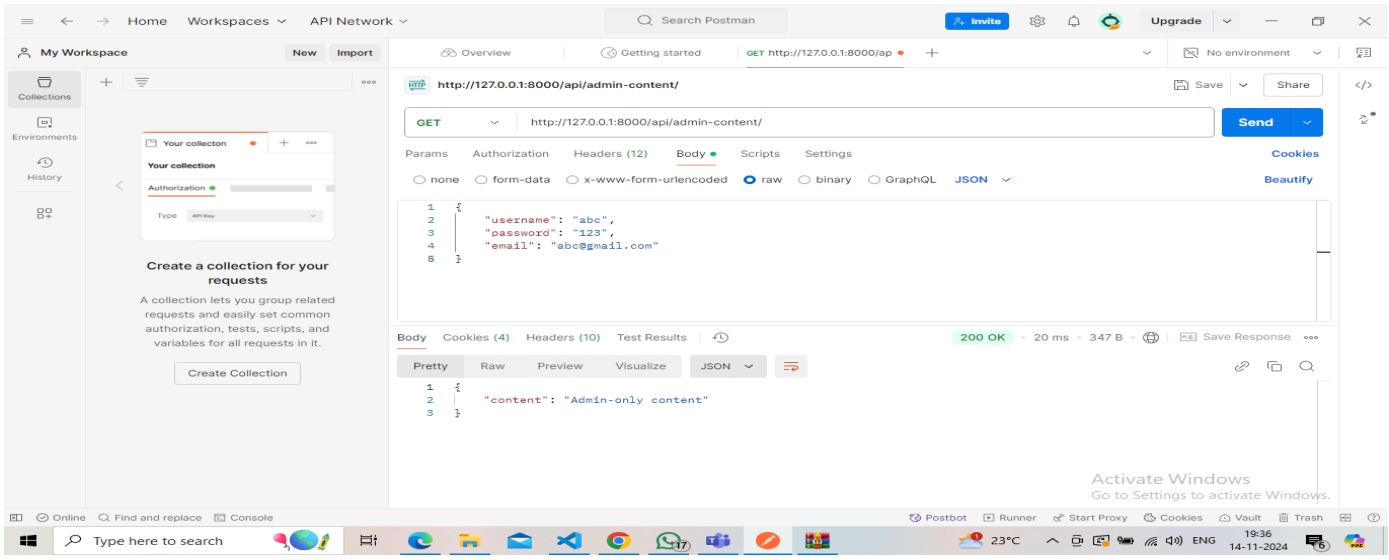
Output:(postman)

1. Register
2. Login
3. Protected
4. Admin-only content






Home
Workspaces
API Network
Search Postman
Invite
Upgrade
My Workspace
New
Import
Overview
Getting started
GET http://127.0.0.1:8000/ap
No environment
Collections
Environments
History
http://127.0.0.1:8000/api/protected/
Save
Share
GET
http://127.0.0.1:8000/api/protected/
Send
Params
Authorization
Headers (11)
Body
Scripts
Settings
Cookies
Headers
10 hidden
Key
Value
Description
Bulk Edit
Presets
Authorization
Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ...
Key
Value
Description
Create a collection for your requests
A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.
Create Collection
Body
Cookies (4)
Headers (10)
Test Results
200 OK
17 ms
353 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"message": "This is a protected view"
}
Activate Windows
Go to Settings to activate Windows.
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash
Type here to search
26°C
ENG
17:47
14-11-2024


Home
Workspaces
API Network
Search Postman
Invite
Upgrade
My Workspace
New
Import
Overview
Getting started
GET http://127.0.0.1:8000/ap
No environment
Collections
Environments
History
http://127.0.0.1:8000/api/admin-content/
Save
Share
GET
http://127.0.0.1:8000/api/admin-content/
Send
Params
Authorization
Headers (12)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"username": "abc",
"password": "123",
"email": "abc@gmail.com"
}
Create a collection for your requests
A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.
Create Collection
Body
Cookies (4)
Headers (10)
Test Results
200 OK
20 ms
347 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"content": "Admin-only content"
}
Activate Windows
Go to Settings to activate Windows.
Online
Find and replace
Console
Postbot
Runner
Start Proxy
Cookies
Vault
Trash
Type here to search
23°C
ENG
19:36
14-11-2024