authentication.py

Authentication

> *Auth needs to be pluggable.*
>
> *— Jacob Kaplan-Moss, "REST worst practices"*

Authentication is the mechanism of associating an incoming request with a set of identifying credentials, such as the user the request came from, or the token that it was signed with. The permission and throttling policies can then use those credentials to determine if the request should be permitted.

REST framework provides several authentication schemes out of the box, and also allows you to implement custom schemes.

Authentication always runs at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed.

The `request.user` property will typically be set to an instance of the `contrib.auth` package's `User` class.

The `request.auth` property is used for any additional authentication information, for example, it may be used to represent an authentication token that the request was signed with.

Note: Don't forget that **authentication by itself won't allow or disallow an incoming request**, it simply identifies the credentials that the request was made with.

For information on how to set up the permission policies for your API please see the permissions documentation.

How authentication is determined

The authentication schemes are always defined as a list of classes. REST framework will attempt to authenticate with each class in the list, and will set `request.user` and `request.auth` using the return value of the first class that successfully authenticates.

If no class authenticates, `request.user` will be set to an instance of `django.contrib.auth.models.AnonymousUser`, and `request.auth` will be set to `None`.

The value of `request.user` and `request.auth` for unauthenticated requests can be modified using the `UNAUTHENTICATED_USER` and `UNAUTHENTICATED_TOKEN` settings.

Setting the authentication scheme

The default authentication schemes may be set globally, using the `DEFAULT_AUTHENTICATION_CLASSES` setting. For example.

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

You can also set the authentication scheme on a per-view or per-viewset basis, using the `APIView` class-based views.

```
from rest_framework.authentication import SessionAuthentication, BasicAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from rest_framework.views import APIView

class ExampleView(APIView):
    authentication_classes = [SessionAuthentication, BasicAuthentication]
    permission_classes = [IsAuthenticated]

    def get(self, request, format=None):
```

```
        content = {
            'user': str(request.user),  # `django.contrib.auth.User` instance.
            'auth': str(request.auth),  # None
        }
        return Response(content)
```

Or, if you're using the `@api_view` decorator with function based views.

```
@api_view(['GET'])
@authentication_classes([SessionAuthentication, BasicAuthentication])
@permission_classes([IsAuthenticated])
def example_view(request, format=None):
    content = {
        'user': str(request.user),  # `django.contrib.auth.User` instance.
        'auth': str(request.auth),  # None
    }
    return Response(content)
```

Unauthorized and Forbidden responses

When an unauthenticated request is denied permission there are two different error codes that may be appropriate.

- HTTP 401 Unauthorized
- HTTP 403 Permission Denied

HTTP 401 responses must always include a `WWW-Authenticate` header, that instructs the client how to authenticate. HTTP 403 responses do not include the `WWW-Authenticate` header.

The kind of response that will be used depends on the authentication scheme. Although multiple authentication schemes may be in use, only one scheme may be used to determine the type of response. **The first authentication class set on the view is used when determining the type of response**.

Note that when a request may successfully authenticate, but still be denied permission to perform the request, in which case a `403 Permission Denied` response will always be used, regardless of the authentication scheme.

Apache mod_wsgi specific configuration

Note that if deploying to Apache using mod_wsgi, the authorization header is not passed through to a WSGI application by default, as it is assumed that authentication will be handled by Apache, rather than at an application level.

If you are deploying to Apache, and using any non-session based authentication, you will need to explicitly configure mod_wsgi to pass the required headers through to the application. This can be done by specifying the `WSGIPassAuthorization` directive in the appropriate context and setting it to `'On'`.

```
# this can go in either server config, virtual host, directory or .htaccess
```

```
WSGIPassAuthorization On
```

API Reference

BasicAuthentication

This authentication scheme uses HTTP Basic Authentication, signed against a user's username and password. Basic authentication is generally only appropriate for testing.

If successfully authenticated, `BasicAuthentication` provides the following credentials.

- `request.user` will be a Django `User` instance.
- `request.auth` will be `None`.

Unauthenticated responses that are denied permission will result in an `HTTP 401 Unauthorized` response with an appropriate WWW-Authenticate header. For example:

```
WWW-Authenticate: Basic realm="api"
```

Note: If you use `BasicAuthentication` in production you must ensure that your API is only available over `https`. You should also ensure that your API clients will always re-request the username and password at login, and will never store those details to persistent storage.

TokenAuthentication

Note: The token authentication provided by Django REST framework is a fairly simple implementation.

For an implementation which allows more than one token per user, has some tighter security implementation details, and supports token expiry, please see the Django REST Knox third party package.

This authentication scheme uses a simple token-based HTTP Authentication scheme. Token authentication is appropriate for client-server setups, such as native desktop and mobile clients.

To use the `TokenAuthentication` scheme you'll need to configure the authentication classes to include `TokenAuthentication`, and additionally include `rest_framework.authtoken` in your `INSTALLED_APPS` setting:

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

Make sure to run `manage.py migrate` after changing your settings.

The `rest_framework.authtoken` app provides Django database migrations.

You'll also need to create tokens for your users.

```
from rest_framework.authtoken.models import Token

token = Token.objects.create(user=...)
print(token.key)
```

For clients to authenticate, the token key should be included in the `Authorization` HTTP header. The key should be prefixed by the string literal "Token", with whitespace separating the two strings. For example:

```
Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b
```

If you want to use a different keyword in the header, such as `Bearer`, simply subclass `TokenAuthentication` and set the `keyword` class variable.

If successfully authenticated, `TokenAuthentication` provides the following credentials.

- `request.user` will be a Django `User` instance.
- `request.auth` will be a `rest_framework.authtoken.models.Token` instance.

Unauthenticated responses that are denied permission will result in an `HTTP 401 Unauthorized` response with an appropriate WWW-Authenticate header. For example:

```
WWW-Authenticate: Token
```

The `curl` command line tool may be useful for testing token authenticated APIs. For example:

```
curl -X GET http://127.0.0.1:8000/api/example/ -H 'Authorization: Token 9944b09199c62bcf9418
```

Note: If you use `TokenAuthentication` in production you must ensure that your API is only available over `https`.

Generating Tokens

By using signals

If you want every user to have an automatically generated Token, you can simply catch the User's `post_save` signal.

```
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)
```

Note that you'll want to ensure you place this code snippet in an installed `models.py` module, or some other location that will be imported by Django on startup.

If you've already created some users, you can generate tokens for all existing users like this:

```
from django.contrib.auth.models import User
from rest_framework.authtoken.models import Token

for user in User.objects.all():
    Token.objects.get_or_create(user=user)
```

By exposing an api endpoint

When using `TokenAuthentication`, you may want to provide a mechanism for clients to obtain a token given the username and password. REST framework provides a built-in view to provide this behaviour. To use it, add the `obtain_auth_token` view to your URLconf:

```
from rest_framework.authtoken import views
urlpatterns += [
    path('api-token-auth/', views.obtain_auth_token)
]
```

Note that the URL part of the pattern can be whatever you want to use.

The `obtain_auth_token` view will return a JSON response when valid `username` and `password` fields are POSTed to the view using form data or JSON:

```
{ 'token' : '9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b' }
```

Note that the default `obtain_auth_token` view explicitly uses JSON requests and responses, rather than using default renderer and parser classes in your settings.

By default, there are no permissions or throttling applied to the `obtain_auth_token` view. If you do wish to apply to throttle you'll need to override the view class, and include them using the `throttle_classes` attribute.

If you need a customized version of the `obtain_auth_token` view, you can do so by subclassing the `ObtainAuthToken` view class, and using that in your url conf instead.

For example, you may return additional user information beyond the `token` value:

```
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
        })
```

And in your `urls.py`:

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

With Django admin

It is also possible to create Tokens manually through the admin interface. In case you are using a large user base, we recommend that you monkey patch the `TokenAdmin` class customize it to your needs, more specifically by declaring the `user` field as `raw_field`.

`your_app/admin.py`:

```
from rest_framework.authtoken.admin import TokenAdmin

TokenAdmin.raw_id_fields = ['user']
```

Using Django manage.py command

Since version 3.6.4 it's possible to generate a user token using the following command:

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

this command will return the API token for the given user, creating it if it doesn't exist:

```
Generated token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b for user user1
```

In case you want to regenerate the token (for example if it has been compromised or leaked) you can pass an additional parameter:

```
./manage.py drf_create_token -r <username>
```

SessionAuthentication

This authentication scheme uses Django's default session backend for authentication. Session authentication is appropriate for AJAX clients that are running in the same session context as your website.

If successfully authenticated, `SessionAuthentication` provides the following credentials.

- `request.user` will be a Django `User` instance.
- `request.auth` will be `None`.

Unauthenticated responses that are denied permission will result in an `HTTP 403 Forbidden` response.

If you're using an AJAX-style API with SessionAuthentication, you'll need to make sure you include a valid CSRF token for any "unsafe" HTTP method calls, such as `PUT`, `PATCH`, `POST` or `DELETE` requests. See the Django CSRF documentation for more details.

Warning: Always use Django's standard login view when creating login pages. This will ensure your login views are properly protected.

CSRF validation in REST framework works slightly differently from standard Django due to the need to support both session and non-session based authentication to the same views. This means that only authenticated requests require CSRF tokens, and anonymous requests may be sent without CSRF tokens. This behaviour is not suitable for login views, which should always have CSRF validation applied.

RemoteUserAuthentication

This authentication scheme allows you to delegate authentication to your web server, which sets the `REMOTE_USER` environment variable.

To use it, you must have `django.contrib.auth.backends.RemoteUserBackend` (or a subclass) in your `AUTHENTICATION_BACKENDS` setting. By default, `RemoteUserBackend` creates `User` objects for usernames that don't already exist. To change this and other behaviour, consult the Django documentation.

If successfully authenticated, `RemoteUserAuthentication` provides the following credentials:

- `request.user` will be a Django `User` instance.
- `request.auth` will be `None`.

Consult your web server's documentation for information about configuring an authentication method, e.g.:

- Apache Authentication How-To
- NGINX (Restricting Access)

Custom authentication

To implement a custom authentication scheme, subclass `BaseAuthentication` and override the `.authenticate(self, request)` method. The method should return a two-tuple of `(user, auth)` if authentication succeeds, or `None` otherwise.

In some circumstances instead of returning `None`, you may want to raise an `AuthenticationFailed` exception from the `.authenticate()` method.

Typically the approach you should take is:

- If authentication is not attempted, return `None`. Any other authentication schemes also in use will still be checked.
- If authentication is attempted but fails, raise an `AuthenticationFailed` exception. An error response will be returned immediately, regardless of any permissions checks, and without checking any other authentication schemes.

You *may* also override the `.authenticate_header(self, request)` method. If implemented, it should return a string that will be used as the value of the `WWW-Authenticate` header in a `HTTP 401 Unauthorized` response.

If the `.authenticate_header()` method is not overridden, the authentication scheme will return `HTTP 403 Forbidden` responses when an unauthenticated request is denied access.

Note: When your custom authenticator is invoked by the request object's `.user` or `.auth` properties, you may see an `AttributeError` re-raised as a `WrappedAttributeError`. This is necessary to prevent the original exception from being suppressed by the outer property access. Python will not recognize that the `AttributeError` originates from your custom authenticator and will instead assume that the request object does not have a `.user` or `.auth` property. These errors should be fixed or otherwise handled by your authenticator.

Example

The following example will authenticate any incoming request as the user given by the username in a custom request header named 'X-USERNAME'.

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

class ExampleAuthentication(authentication.BaseAuthentication):
    def authenticate(self, request):
        username = request.META.get('HTTP_X_USERNAME')
        if not username:
            return None

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

```
        return (user, None)
```

Third party packages

The following third-party packages are also available.

django-rest-knox

Django-rest-knox library provides models and views to handle token-based authentication in a more secure and extensible way than the built-in TokenAuthentication scheme - with Single Page Applications and Mobile clients in mind. It provides per-client tokens, and views to generate them when provided some other authentication (usually basic authentication), to delete the token (providing a server enforced logout) and to delete all tokens (logs out all clients that a user is logged into).

Django OAuth Toolkit

The Django OAuth Toolkit package provides OAuth 2.0 support and works with Python 3.4+. The package is maintained by jazzband and uses the excellent OAuthLib. The package is well documented, and well supported and is currently our **recommended package for OAuth 2.0 support**.

Installation & configuration

Install using `pip`.

```
pip install django-oauth-toolkit
```

Add the package to your `INSTALLED_APPS` and modify your REST framework settings.

```
INSTALLED_APPS = [
    ...
    'oauth2_provider',
]

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'oauth2_provider.contrib.rest_framework.OAuth2Authentication',
    ]
}
```

For more details see the Django REST framework - Getting started documentation.

Django REST framework OAuth

The Django REST framework OAuth package provides both OAuth1 and OAuth2 support for REST framework.

This package was previously included directly in the REST framework but is now supported and maintained as a third-party package.

Installation & configuration

Install the package using `pip`.

```
pip install djangorestframework-oauth
```

For details on configuration and usage see the Django REST framework OAuth documentation for authentication and permissions.

JSON Web Token Authentication

JSON Web Token is a fairly new standard which can be used for token-based authentication. Unlike the built-in TokenAuthentication scheme, JWT Authentication doesn't need to use a database to validate a token. A package for JWT authentication is djangorestframework-simplejwt which provides some features as well as a pluggable token blacklist app.

Hawk HTTP Authentication

The HawkREST library builds on the Mohawk library to let you work with Hawk signed requests and responses in your API. Hawk lets two parties securely communicate with each other using messages signed by a shared key. It is based on HTTP MAC access authentication (which was based on parts of OAuth 1.0).

HTTP Signature Authentication

HTTP Signature (currently a IETF draft) provides a way to achieve origin authentication and message integrity for HTTP messages. Similar to Amazon's HTTP Signature scheme, used by many of its services, it permits stateless, per-request authentication. Elvio Toccalino maintains the djangorestframework-httpsignature (outdated) package which provides an easy to use HTTP Signature Authentication mechanism. You can use the updated fork version of djangorestframework-httpsignature, which is drf-httpsig.

Djoser

Djoser library provides a set of views to handle basic actions such as registration, login, logout, password reset and account activation. The package works with a custom user model and uses token-based authentication. This is ready to use REST implementation of the Django authentication system.

django-rest-auth / dj-rest-auth

This library provides a set of REST API endpoints for registration, authentication (including social media authentication), password reset, retrieve and update user details, etc. By having these API endpoints, your client apps such as AngularJS, iOS, Android, and others can communicate to your Django backend site independently via REST APIs for user management.

There are currently two forks of this project.

- Django-rest-auth is the original project, but is not currently receiving updates.
- Dj-rest-auth is a newer fork of the project.

drf-social-oauth2

Drf-social-oauth2 is a framework that helps you authenticate with major social oauth2 vendors, such as Facebook, Google, Twitter, Orcid, etc. It generates tokens in a JWTed way with an easy setup.

drfpasswordless

drfpasswordless adds (Medium, Square Cash inspired) passwordless support to Django REST Framework's TokenAuthentication scheme. Users log in and sign up with a token sent to a contact point like an email address or a mobile number.

django-rest-authemail

django-rest-authemail provides a RESTful API interface for user signup and authentication. Email addresses are used for authentication, rather than usernames. API endpoints are available for signup, signup email verification, login, logout, password reset, password reset verification, email change, email change verification, password change, and user detail. A fully functional example project and detailed instructions are included.

Django-Rest-Durin

Django-Rest-Durin is built with the idea to have one library that does token auth for multiple Web/CLI/Mobile API clients via one interface but allows different token configuration for each API Client that consumes the API. It provides support for multiple tokens per user via custom models, views, permissions that work with Django-Rest-Framework. The token expiration time can be different per API client and is customizable via the Django Admin Interface.

More information can be found in the Documentation.
