source: 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 a number of authentication schemes out of the box, and also allows you to implement custom schemes.

Authentication is always run 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 setup the permission polices 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': unicode(request.user),  # `django.contrib.auth.User` instance.
            'auth': unicode(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': unicode(request.user),  # `django.contrib.auth.User` instance.
        'auth': unicode(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

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'
)
```

Note: Make sure to run `manage.py syncdb` after changing your settings. The `rest_framework.authtoken` app provides both Django (from v1.7) and South database migrations. See Schema migrations below.

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 successfully authenticated, `TokenAuthentication` provides the following credentials.

- `request.user` will be a Django `User` instance.
- `request.auth` will be a `rest_framework.authtoken.models.BasicToken` 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
9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b'
```

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

Generating Tokens

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)
```

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 behavior. To use it, add the `obtain_auth_token` view to your URLconf:

```
from rest_framework.authtoken import views
urlpatterns += [
    url(r'^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. If you need a customized version of the `obtain_auth_token` view, you can do so by overriding the `ObtainAuthToken` view class, and using that in your url conf instead.

Schema migrations

The `rest_framework.authtoken` app includes both Django native migrations (for Django versions >1.7) and South migrations (for Django versions <1.7) that will create the authtoken table.

Note: From REST Framework v2.4.0 using South with Django <1.7 requires upgrading South v1.0+

If you're using a custom user model you'll need to make sure that any initial migration that creates the user table runs before the authtoken table is created.

You can do so by inserting a `needed_by` attribute in your user migration:

```
class Migration:

    needed_by = (
        ('authtoken', '0001_initial'),
    )

    def forwards(self):
        ...
```

For more details, see the south documentation on dependencies.

Also note that if you're using a `post_save` signal to create tokens, then the first time you create the database tables, you'll need to ensure any migrations are run prior to creating any superusers. For example:

```
python manage.py syncdb --noinput  # Won't create a superuser just yet, due to
`--noinput`.
python manage.py migrate
python manage.py createsuperuser
```

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 to 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.

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 a `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.

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('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 OAuth Toolkit

The Django OAuth Toolkit package provides OAuth 2.0 support, and works with Python 2.7 and Python 3.3+. The package is maintained by Evonove 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.ext.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 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.

Digest Authentication

HTTP digest authentication is a widely implemented scheme that was intended to replace HTTP basic authentication, and which provides a simple encrypted authentication mechanism. Juan Riaza maintains the djangorestframework-digestauth package which provides HTTP digest authentication support for REST framework.

Django OAuth2 Consumer

The Django OAuth2 Consumer library from Rediker Software is another package that provides OAuth 2.0 support for REST framework. The package includes token scoping permissions on tokens, which allows finer-grained access to your API.

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. Blimp maintains the djangorestframework-jwt package which provides a JWT Authentication class as well as a mechanism for clients to obtain a JWT given the username and password.

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 package which provides an easy to use HTTP Signature Authentication mechanism.

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 it uses token based authentication. This is a ready to use REST implementation of Django authentication system.

django-rest-auth

Django-rest-auth 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.