Django REST framework

GitHub

permissions.py

Permissions

> *Authentication or identification by itself is not usually sufficient to gain access to information or code. For that, the entity requesting access must have authorization.*
>
> — *Apple Developer Documentation*

Together with authentication and throttling, permissions determine whether a request should be granted or denied access.

Permission checks are always run at the very start of the view, before any other code is allowed to proceed. Permission checks will typically use the authentication information in the `request.user` and `request.auth` properties to determine if the incoming request should be permitted.

Permissions are used to grant or deny access for different classes of users to different parts of the API.

The simplest style of permission would be to allow access to any authenticated user, and deny access to any unauthenticated user. This corresponds to the `IsAuthenticated` class in REST framework.

A slightly less strict style of permission would be to allow full access to authenticated users, but allow read-only access to unauthenticated users. This corresponds to the `IsAuthenticatedOrReadOnly` class in REST framework.

How permissions are determined

Permissions in REST framework are always defined as a list of permission classes.

Before running the main body of the view each permission in the list is checked. If any permission check fails, an `exceptions.PermissionDenied` or `exceptions.NotAuthenticated` exception will be raised, and the main body of the view will not run.

When the permission checks fail, either a "403 Forbidden" or a "401 Unauthorized" response will be returned, according to the following rules:

- The request was successfully authenticated, but permission was denied. — *An HTTP 403 Forbidden response will be returned.*
- The request was not successfully authenticated, and the highest priority authentication class *does not* use `WWW-Authenticate` headers. — *An HTTP 403 Forbidden response will be returned.*
- The request was not successfully authenticated, and the highest priority authentication class *does* use `WWW-Authenticate` headers. — *An HTTP 401 Unauthorized response, with an appropriate* `WWW-Authenticate` *header will be returned.*

Object level permissions

REST framework permissions also support object-level permissioning. Object level permissions are used to determine if a user should be allowed to act on a particular object, which will typically be a model instance.

Object level permissions are run by REST framework's generic views when `.get_object()` is called. As with view level permissions, an `exceptions.PermissionDenied` exception will be raised if the user is not allowed to act on the given object.

If you're writing your own views and want to enforce object level permissions, or if you override the `get_object` method on a generic view, then you'll need to explicitly call the `.check_object_permissions(request, obj)` method on the view at the point at which you've retrieved the object.

This will either raise a `PermissionDenied` or `NotAuthenticated` exception, or simply return if the view has the appropriate permissions.

For example:

```python
def get_object(self):
    obj = get_object_or_404(self.get_queryset(), pk=self.kwargs["pk"])
    self.check_object_permissions(self.request, obj)
    return obj
```

Note: With the exception of `DjangoObjectPermissions`, the provided permission classes in `rest_framework.permissions` **do not** implement the methods necessary to check object permissions.

If you wish to use the provided permission classes in order to check object permissions, **you must** subclass them and implement the `has_object_permission()` method described in the *Custom permissions* section (below).

Limitations of object level permissions

For performance reasons the generic views will not automatically apply object level permissions to each instance in a queryset when returning a list of objects.

Often when you're using object level permissions you'll also want to filter the queryset appropriately, to ensure that users only have visibility onto instances that they are permitted to view.

Because the `get_object()` method is not called, object level permissions from the `has_object_permission()` method **are not applied** when creating objects. In order to restrict object creation you need to implement the permission check either in your Serializer class or override the `perform_create()` method of your ViewSet class.

Setting the permission policy

The default permission policy may be set globally, using the `DEFAULT_PERMISSION_CLASSES` setting. For example.

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

If not specified, this setting defaults to allowing unrestricted access:

```
'DEFAULT_PERMISSION_CLASSES': [
    'rest_framework.permissions.AllowAny',
]
```

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

```
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from rest_framework.views import APIView

class ExampleView(APIView):
    permission_classes = [IsAuthenticated]

    def get(self, request, format=None):
        content = {
            'status': 'request was permitted'
```

```
        }
        return Response(content)
```

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

```python
from rest_framework.decorators import api_view, permission_classes
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response

@api_view(['GET'])
@permission_classes([IsAuthenticated])
def example_view(request, format=None):
    content = {
        'status': 'request was permitted'
    }
    return Response(content)
```

Note: when you set new permission classes via the class attribute or decorators you're telling the view to ignore the default list set in the **settings.py** file.

Provided they inherit from `rest_framework.permissions.BasePermission`, permissions can be composed using standard Python bitwise operators. For example, `IsAuthenticatedOrReadOnly` could be written:

```python
from rest_framework.permissions import BasePermission, IsAuthenticated, SAFE_METHODS
from rest_framework.response import Response
from rest_framework.views import APIView

class ReadOnly(BasePermission):
    def has_permission(self, request, view):
        return request.method in SAFE_METHODS

class ExampleView(APIView):
    permission_classes = [IsAuthenticated|ReadOnly]

    def get(self, request, format=None):
        content = {
            'status': 'request was permitted'
        }
        return Response(content)
```

Note: it supports & (and), | (or) and ~ (not).

API Reference

AllowAny

The `AllowAny` permission class will allow unrestricted access, **regardless of if the request was authenticated or unauthenticated**.

This permission is not strictly required, since you can achieve the same result by using an empty list or tuple for the permissions setting, but you may find it useful to specify this class because it makes the intention explicit.

IsAuthenticated

The `IsAuthenticated` permission class will deny permission to any unauthenticated user, and allow permission otherwise.

This permission is suitable if you want your API to only be accessible to registered users.

IsAdminUser

The `IsAdminUser` permission class will deny permission to any user, unless `user.is_staff` is `True` in which case permission will be allowed.

This permission is suitable if you want your API to only be accessible to a subset of trusted administrators.

IsAuthenticatedOrReadOnly

The `IsAuthenticatedOrReadOnly` will allow authenticated users to perform any request. Requests for unauthorised users will only be permitted if the request method is one of the "safe" methods; `GET`, `HEAD` or `OPTIONS`.

This permission is suitable if you want to your API to allow read permissions to anonymous users, and only allow write permissions to authenticated users.

DjangoModelPermissions

This permission class ties into Django's standard `django.contrib.auth` model permissions. This permission must only be applied to views that have a `.queryset` property or `get_queryset()` method. Authorization will only be granted if the user *is authenticated* and has the *relevant model permissions* assigned. The appropriate model is determined by checking `get_queryset().model` or `queryset.model`.

- `POST` requests require the user to have the `add` permission on the model.
- `PUT` and `PATCH` requests require the user to have the `change` permission on the model.
- `DELETE` requests require the user to have the `delete` permission on the model.

The default behaviour can also be overridden to support custom model permissions. For example, you might want to include a `view` model permission for `GET` requests.

To use custom model permissions, override `DjangoModelPermissions` and set the `.perms_map` property. Refer to the source code for details.

DjangoModelPermissionsOrAnonReadOnly

Similar to `DjangoModelPermissions`, but also allows unauthenticated users to have read-only access to the API.

DjangoObjectPermissions

This permission class ties into Django's standard object permissions framework that allows per-object permissions on models. In order to use this permission class, you'll also need to add a permission backend that supports object-level permissions, such as django-guardian.

As with `DjangoModelPermissions`, this permission must only be applied to views that have a `.queryset` property or `.get_queryset()` method. Authorization will only be granted if the user *is authenticated* and has the *relevant per-object permissions* and *relevant model permissions* assigned.

- `POST` requests require the user to have the `add` permission on the model instance.
- `PUT` and `PATCH` requests require the user to have the `change` permission on the model instance.
- `DELETE` requests require the user to have the `delete` permission on the model instance.

Note that `DjangoObjectPermissions` **does not** require the `django-guardian` package, and should support other object-level backends equally well.

As with `DjangoModelPermissions` you can use custom model permissions by overriding `DjangoObjectPermissions` and setting the `.perms_map` property. Refer to the source code for details.

Note: If you need object level `view` permissions for `GET`, `HEAD` and `OPTIONS` requests and are using django-guardian for your object-level permissions backend, you'll want to consider using the `DjangoObjectPermissionsFilter` class provided by the `djangorestframework-guardian` package. It ensures that list endpoints only return results including objects for which the user has appropriate view permissions.

Custom permissions

To implement a custom permission, override `BasePermission` and implement either, or both, of the following methods:

- `.has_permission(self, request, view)`
- `.has_object_permission(self, request, view, obj)`

The methods should return `True` if the request should be granted access, and `False` otherwise.

If you need to test if a request is a read operation or a write operation, you should check the request method against the constant `SAFE_METHODS`, which is a tuple containing `'GET'`, `'OPTIONS'` and `'HEAD'`. For example:

```
if request.method in permissions.SAFE_METHODS:
    # Check permissions for read-only request
else:
    # Check permissions for write request
```

Note: The instance-level `has_object_permission` method will only be called if the view-level `has_permission` checks have already passed. Also note that in order for the instance-level checks to run, the view code should explicitly call `.check_object_permissions(request, obj)`. If you are using the

generic views then this will be handled for you by default. (Function-based views will need to check object permissions explicitly, raising `PermissionDenied` on failure.)

Custom permissions will raise a `PermissionDenied` exception if the test fails. To change the error message associated with the exception, implement a `message` attribute directly on your custom permission. Otherwise the `default_detail` attribute from `PermissionDenied` will be used. Similarly, to change the code identifier associated with the exception, implement a `code` attribute directly on your custom permission - otherwise the `default_code` attribute from `PermissionDenied` will be used.

```python
from rest_framework import permissions

class CustomerAccessPermission(permissions.BasePermission):
    message = 'Adding customers not allowed.'

    def has_permission(self, request, view):
        ...
```

Examples

The following is an example of a permission class that checks the incoming request's IP address against a blocklist, and denies the request if the IP has been blocked.

```python
from rest_framework import permissions

class BlocklistPermission(permissions.BasePermission):
    """
    Global permission check for blocked IPs.
    """

    def has_permission(self, request, view):
        ip_addr = request.META['REMOTE_ADDR']
        blocked = Blocklist.objects.filter(ip_addr=ip_addr).exists()
        return not blocked
```

As well as global permissions, that are run against all incoming requests, you can also create object-level permissions, that are only run against operations that affect a particular object instance. For example:

```python
class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    Object-level permission to only allow owners of an object to edit it.
    Assumes the model instance has an `owner` attribute.
    """

    def has_object_permission(self, request, view, obj):
        # Read permissions are allowed to any request,
        # so we'll always allow GET, HEAD or OPTIONS requests.
```

```
            if request.method in permissions.SAFE_METHODS:
                return True

            # Instance must have an attribute named `owner`.
            return obj.owner == request.user
```

Note that the generic views will check the appropriate object level permissions, but if you're writing your own custom views, you'll need to make sure you check the object level permission checks yourself. You can do so by calling `self.check_object_permissions(request, obj)` from the view once you have the object instance. This call will raise an appropriate `APIException` if any object-level permission checks fail, and will otherwise simply return.

Also note that the generic views will only check the object-level permissions for views that retrieve a single model instance. If you require object-level filtering of list views, you'll need to filter the queryset separately. See the filtering documentation for more details.

Overview of access restriction methods

REST framework offers three different methods to customize access restrictions on a case-by-case basis. These apply in different scenarios and have different effects and limitations.

- `queryset` / `get_queryset()` : Limits the general visibility of existing objects from the database. The queryset limits which objects will be listed and which objects can be modified or deleted. The `get_queryset()` method can apply different querysets based on the current action.
- `permission_classes` / `get_permissions()` : General permission checks based on the current action, request and targeted object. Object level permissions can only be applied to retrieve, modify and deletion actions. Permission checks for list and create will be applied to the entire object type. (In case of list: subject to restrictions in the queryset.)
- `serializer_class` / `get_serializer()` : Instance level restrictions that apply to all objects on input and output. The serializer may have access to the request context. The `get_serializer()` method can apply different serializers based on the current action.

The following table lists the access restriction methods and the level of control they offer over which actions.

	queryset	permission_classes	serializer_class
Action: list	global	global	object-level*
Action: create	no	global	object-level
Action: retrieve	global	object-level	object-level
Action: update	global	object-level	object-level
Action: partial_update	global	object-level	object-level
Action: destroy	global	object-level	no
Can reference action in decision	no**	yes	no**
Can reference request in decision	no**	yes	yes

* A Serializer class should not raise PermissionDenied in a list action, or the entire list would not be returned.
** The `get_*()` methods have access to the current view and can return different Serializer or QuerySet instances based on the request or action.

Third party packages

The following third party packages are also available.

DRF - Access Policy

The Django REST - Access Policy package provides a way to define complex access rules in declarative policy classes that are attached to view sets or function-based views. The policies are defined in JSON in a format similar to AWS' Identity & Access Management policies.

Composed Permissions

The Composed Permissions package provides a simple way to define complex and multi-depth (with logic operators) permission objects, using small and reusable components.

REST Condition

The REST Condition package is another extension for building complex permissions in a simple and convenient way. The extension allows you to combine permissions with logical operators.

DRY Rest Permissions

The DRY Rest Permissions package provides the ability to define different permissions for individual default and custom actions. This package is made for apps with permissions that are derived from relationships defined in the app's data model. It also supports permission checks being returned to a client app through the API's serializer. Additionally it supports adding permissions to the default and custom list actions to restrict the data they retrieve per user.

Django Rest Framework Roles

The Django Rest Framework Roles package makes it easier to parameterize your API over multiple types of users.

Django REST Framework API Key

The Django REST Framework API Key package provides permissions classes, models and helpers to add API key authorization to your API. It can be used to authorize internal or third-party backends and services (i.e. *machines*) which do not have a user account. API keys are stored securely using Django's password hashing infrastructure, and they can be viewed, edited and revoked at anytime in the Django admin.

Django Rest Framework Role Filters

The Django Rest Framework Role Filters package provides simple filtering over multiple types of roles.

Django Rest Framework PSQ

The Django Rest Framework PSQ package is an extension that gives support for having action-based **permission_classes**, **serializer_class**, and **queryset** dependent on permission-based rules.

Documentation built with **MkDocs**.