Sure! Adding **permissions** in Django REST Framework (DRF) controls **who can access your API views** and what actions they can perform. Permissions are crucial for securing your API.

What Are Permissions in DRF?

- **Permissions** determine if a request should be granted or denied access to a view.
- They are classes that you add to your views or viewsets.
- DRF comes with several built-in permission classes but you can also create your own.

How to Add Permissions in DRF — Step-by-step

1. Import Permission Classes

At the top of your view file:

```
from rest_framework import permissions
```

---

2. Apply Permission Classes to Views

You can apply permissions in:

- **Class-based views**
- **ViewSets**
- **Function-based views (with decorators)**

Example: Using Permission Classes in Generic Views

```
from rest_framework import generics, permissions
from .models import Employee
from .serializers import EmployeeSerializer

class EmployeeListCreateView(generics.ListCreateAPIView):
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer
```

```
    permission_classes = [permissions.IsAuthenticated]  # Only
authenticated users can access
```

Common Built-in Permission Classes

Permission Class	Description
`AllowAny`	Allows unrestricted access (default in DRF).
`IsAuthenticated`	Only authenticated users can access.
`IsAdminUser`	Only users with `is_staff=True` can access.
`IsAuthenticatedOrReadOnly`	Authenticated users can modify, others can only read (GET).

---

3. Applying Permissions to ViewSets

```
from rest_framework import viewsets, permissions
from .models import Employee
from .serializers import EmployeeSerializer

class EmployeeViewSet(viewsets.ModelViewSet):
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer
    permission_classes = [permissions.IsAuthenticatedOrReadOnly]
```

---

4. Global Default Permissions (Optional)

If you want to apply the same permission across all your views by default, add this in your **`settings.py`**:

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

This means all views will require authentication unless you override it per view.

---

5. Custom Permissions

You can create your own permissions by subclassing `BasePermission` and overriding `has_permission` and/or `has_object_permission`.

Example:

```
from rest_framework.permissions import BasePermission

class IsOwner(BasePermission):
    def has_object_permission(self, request, view, obj):
        # Only allow owners of an object to edit it
        return obj.owner == request.user
```

Then add it to your view:

```
permission_classes = [IsOwner]
```

Summary: How to add DRF permission

- Import permission classes.
- Add `permission_classes = [permission_class_here]` to your views/viewsets.
- Or set default permissions in `settings.py`.
- Use built-in or create custom permissions as needed.

Absolutely! Here's a detailed explanation of the **built-in permission classes** that come with Django REST Framework (DRF). These are the most commonly used permission classes to control access to your API views.

DRF Built-in Permission Classes Explained

---

1. `AllowAny`

- **Description:** Allows unrestricted access to the view, regardless of authentication or user status.
- **Use case:** When you want to make an endpoint public (no login required).
- **Example:** Public API info, health check endpoints.

```
permission_classes = [permissions.AllowAny]
```

---

2. `IsAuthenticated`

- **Description:** Allows access **only** to authenticated users.
- If a user is **not authenticated** (anonymous), they will be denied access.
- **Use case:** APIs that require the user to log in to access data or perform actions.

```
permission_classes = [permissions.IsAuthenticated]
```

---

3. `IsAdminUser`

- **Description:** Allows access **only** to users with `is_staff=True`.
- Usually, these are admin users or staff members who have extra privileges.
- **Use case:** Admin panels, management APIs that only staff can access.

```
permission_classes = [permissions.IsAdminUser]
```

---

4. `IsAuthenticatedOrReadOnly`

- **Description:** Allows full access (read + write) to authenticated users.
- Allows **read-only** (GET, HEAD, OPTIONS) access to unauthenticated users.
- **Use case:** Public read-only data, but only logged-in users can create/update/delete.

```
permission_classes = [permissions.IsAuthenticatedOrReadOnly]
```

---

5. `DjangoModelPermissions`

- **Description:** Uses Django's built-in model permissions (`add`, `change`, `delete`, `view`).
- Requires the user to be authenticated and have the appropriate Django model permissions to perform actions.
- Read permissions require the `view` permission on the model.
- Write permissions require the `add`, `change`, or `delete` permissions.
- **Use case:** Fine-grained control using Django's permission system.

```
permission_classes = [permissions.DjangoModelPermissions]
```

Note: Your users must have the appropriate Django model permissions set in the admin or via code.

---

6. `DjangoModelPermissionsOrAnonReadOnly`

- **Description:** Combines model permissions with anonymous read-only access.
- Authenticated users need Django model permissions to write.
- Anonymous users can only read.
- **Use case:** Allow public read, but restrict changes to authorized users.

```
permission_classes = [permissions.DjangoModelPermissionsOrAnonReadOnly]
```

---

7. `DjangoObjectPermissions`

- **Description:** Similar to `DjangoModelPermissions`, but checks permissions at the **object level**.
- User must have object-level permissions to access the object.
- **Use case:** More fine-grained control when you want to restrict access to specific objects.

```
permission_classes = [permissions.DjangoObjectPermissions]
```

Note: Requires using Django's object permission backend (e.g., with django-guardian).

---

8. `IsAuthenticatedOrReadOnly`

- This was already explained, but to reiterate:
- Authenticated users can perform any request.
- Unauthenticated users can only perform **safe** methods (GET, HEAD, OPTIONS).

Safe Methods

DRF considers these HTTP methods as **safe** (read-only):

- `GET`

- `HEAD`
- `OPTIONS`

Summary Table

Permission Class	Who Can Access?	Use Case
`AllowAny`	Anyone, authenticated or not	Public endpoints
`IsAuthenticated`	Only logged-in users	Private APIs
`IsAdminUser`	Only users with `is_staff=True`	Admin dashboards
`IsAuthenticatedOrReadOnly`	Authenticated users can write, others read	Public data with restricted write
`DjangoModelPermissions`	Authenticated with model-level perms	Use Django model permissions
`DjangoModelPermissionsOrAnonReadOnly`	Anons can read, authenticated need model perms	Public read, restricted write
`DjangoObjectPermissions`	Authenticated with object-level perms	Object-level permission control

---

How to use any permission class?

In your view or viewset:

```
from rest_framework import permissions

class MyView(APIView):
    permission_classes = [permissions.IsAuthenticated]
```

Or in `settings.py` globally:

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

```
    ]
}
```

Custom permission

Great question! Creating **custom permissions** in Django REST Framework (DRF) is straightforward and powerful. It lets you define exactly who can access or modify what, based on your own business logic.

How to Create a Custom Permission in DRF — Step by Step

---

1. Import BasePermission

You create a custom permission by subclassing `rest_framework.permissions.BasePermission`.

```
from rest_framework.permissions import BasePermission
```

---

2. Override Permission Methods

There are two main methods you can override:

- **`has_permission(self, request, view)`**
  Checks general permission for the view-level request.
  Returns `True` if permission is granted, else `False`.
- **`has_object_permission(self, request, view, obj)`**
  Checks permission against a specific object instance (used in detail views).
  Returns `True` or `False`.

---

3. Example: Custom Permission That Allows Only Owners to Edit Their Object

Imagine a model with an `owner` field. You want only the owner to be able to update/delete, but everyone can read.

```python
from rest_framework.permissions import BasePermission, SAFE_METHODS

class IsOwnerOrReadOnly(BasePermission):
    """
    Custom permission to only allow owners of an object to edit it.
    Everyone else can only read.
    """

    def has_permission(self, request, view):
        # Allow all users to GET, HEAD, OPTIONS
        if request.method in SAFE_METHODS:
            return True
        # For other methods (POST, PUT, DELETE), user must be
authenticated
        return request.user and request.user.is_authenticated

    def has_object_permission(self, request, view, obj):
        # Read permissions are allowed for any request
        if request.method in SAFE_METHODS:
            return True

        # Write permissions are only allowed to the owner of the object
        return obj.owner == request.user
```

---

4. Use the Custom Permission in Your View or ViewSet

```python
from rest_framework import viewsets
from .permissions import IsOwnerOrReadOnly
from .models import MyModel
from .serializers import MyModelSerializer

class MyModelViewSet(viewsets.ModelViewSet):
    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer
    permission_classes = [IsOwnerOrReadOnly]
```

---

5. Key Points

- `has_permission` controls **access at the view level** (before accessing specific objects).
- `has_object_permission` controls **access to individual object instances**.
- You can implement just one or both depending on your needs.
- Always return `True` if you want to grant access; return `False` to deny.

Bonus: Simple Custom Permission Example (Allow Only Admins)

```python
from rest_framework.permissions import BasePermission

class IsAdminOnly(BasePermission):
    def has_permission(self, request, view):
        return bool(request.user and request.user.is_staff)
```