Learning Objectives

- **Define the purpose of permissions**
- **Identify some common permissions**
- **Update Blango so only authenticated users can make changes**
- **Use custom permissions to restrict access to objects**
- **Combine permissions so that only the author or admin users can make changes**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Adding Permissions to Views

Adding Permissions to Views

Django Rest Framework provides a few helpful classes to add common types of permissions restrictions to views. They're all importable from `rest_framework.permissions`.

Some of the permission classes are:

- `AllowAny`: Allow full access without authentication. This is the default, as we've seen in our APIs.
- `IsAuthenticated`: Allow full access to any authenticated user.
- `IsAdminUser`: Allow access to any user who has `is_staff` of `True`.
- `IsAuthenticatedOrReadOnly`: Allow access to anyone only for the "read-only" methods: `GET`, `HEAD` or `OPTIONS`. Authenticated users can access all methods. Note that this assumes that you haven't written your own "read-only" methods that update the database. For example, if your `GET` method was implemented to delete data then anyone could execute it and delete the data.

▼ **Model Permissions**
DRF also provides classes that are designed to work with the Django model permissions: `DjangoModelPermissions`, `DjangoModelPermissionsOrAnonReadOnly` and `DjangoObjectPermissions`. Since we don't make use of Django's permission system we won't be looking at these classes.

There are a couple of ways to apply permissions in DRF. As with the authentication settings we looked at in the last section, we can apply permissions classes globally with Django settings, or to views themselves. Permissions applied to views override the default settings.

Let's look at the settings first. They're set as a list of permissions class names under `DEFAULT_PERMISSION_CLASSES` inside the `REST_FRAMEWORK` settings dictionary.

For example, in `settings.py`:

```
REST_FRAMEWORK = {
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.IsAuthenticated",
    ]
    # other REST_FRAMEWORK settings truncated for brevity
}
```

With this change, all our DRF views would only be available to authenticated users.

▼ **Default Permissions**
Once again, for reference, the default permissions class is `AllowAny`.

A more flexible method of applying permissions is to set them directly on the view using the `permission_classes` attribute.

For example, we could make our `PostDetail` view available to only admin users like this:

```
from rest_framework.permissions import IsAdminUser


class PostList(generics.ListCreateAPIView):
    permission_classes = [IsAdminUser]
    # other methods/attributes truncated for brevity
```

Try It Out

Let's make our Blango API a bit more secure by default, by only allowing authenticated users to make changes. Anyone else should be able to read. We'll use the `rest_framework.permissions.IsAuthenticatedOrReadOnly` class to achieve this. Open your `settings.py` file and find the `REST_FRAMEWORK` setting dictionary. Add a new key `DEFAULT_PERMISSION_CLASSES`, with the value `["rest_framework.permissions.IsAuthenticatedOrReadOnly"]`. Your `REST_FRAMEWORK` settings dictionary should now look like this:

```
REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": [
        "rest_framework.authentication.BasicAuthentication",
        "rest_framework.authentication.SessionAuthentication",
        "rest_framework.authentication.TokenAuthentication",
    ],
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.IsAuthenticatedOrReadOnly"
    ],
}
```

The easiest way to validate this change is with a web browser. If you visit a DRF view, you'll see that the `Post` edit/create form is not shown for a user who isn't logged in.

View Blog

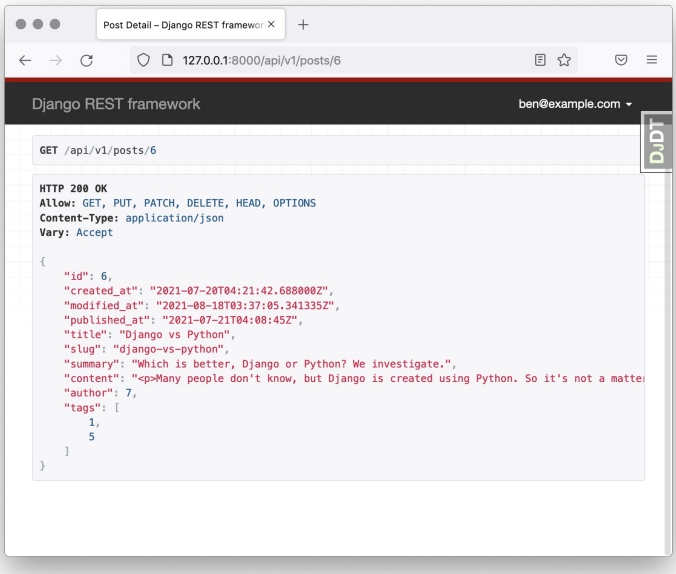

Figure 1

If you log in, then you'll see the form show up again.

Another useful thing we might want to do is give permissions to users based on ownership of objects. We can achieve this with a custom permission class. Let's look at that now.

Custom Permission Classes

Custom Permission Classes

If we want more control over who can make changes to a particular object, we can create our own permission class. The permission class must inherit from `rest_framework.permissions.BasePermission`. There are two methods that can be implemented to control permissions, either one or both can be used depending on what you want to achieve. The first method is `has_permission()`, which accepts two arguments:

- `request`: The HTTP request sent to the view
- `view`: The instance of the view class that's been called

This method should return `True` to allow the request or `False` to deny it. Here's a permissions class that only allows users that have an `example.com` email address.

```
from rest_framework import permissions

class ExampleComOnly(permissions.BasePermission):
    def has_permission(self, request, view):
        email = getattr(request.user, "email", "")
        return email.split("@")[-1] == "example.com"
```

This is a bit of a toy example, so we'll look at the other method which is more useful in our case. It's `has_object_permission()` and takes three arguments:

- `request`: The HTTP request sent to the view
- `view`: The view instance
- `obj`: The object currently being requested

▼ **BasePermission Default**
If `has_permission()` or `has_object_permission()` is not implemented on a subclass, `BasePermission` defaults to returning `True`. You need to be careful about this, otherwise you can accidentally give full permission to any user. We'll cover this behavior in more detail soon.

In our API, we want to allow `Post` authors to perform any operation on their `Post` objects, while other users can only read (`GET`) them. Such a permissions class could be implemented like this:

```
class AuthorModifyOrReadOnly(permissions.IsAuthenticatedOrReadOnly):

    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True

        return request.user == obj.author
```

Here we're also making use of DRF's `permissions.SAFE_METHODS` which is a tuple that contains methods that don't perform updates (i.e. `GET`, `HEAD` and `OPTIONS`). If the request method is one of these, then we'll just return `True`, which means any user can make these requests.

Otherwise we'll check if the `obj` (a `Post` instance) has an `author` that matches the current user. Note that the `has_object_permission()` method is only called for detail views, i.e. for a list request it's not called for each object in the list.

We then add it to the `PostDetail` view like this:

```
class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    permission_classes = [AuthorModifyOrReadOnly]
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

This means that any user can get the `Post` detail, but only the author themselves can make changes to, or delete, a `Post`.

Try It Out

We can implement the `AuthorModifyOrReadOnly` permission class in Django. Create a new file named `permissions.py` in the `blog/api` directory. Implement the class as per the code snippet below:

Open permissions.py

```
from rest_framework import permissions


class AuthorModifyOrReadOnly(permissions.IsAuthenticatedOrReadOnly):

    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True

        return request.user == obj.author
```

Then, open the `blog/api/views.py` file. Import the `AuthorModifyOrReadOnly` class:

Open api/views.py

```
from blog.api.permissions import AuthorModifyOrReadOnly
```

Then add the `permission_classes` attribute to the `PostDetail` class:

```
class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    permission_classes = [AuthorModifyOrReadOnly]
    # leave other attributes as is
```

You can check that it works using a browser. If you log in and visit a `Post` detail page that you are the author of, you'll see the **DELETE** button at the top and editing form down the bottom.

View Blog

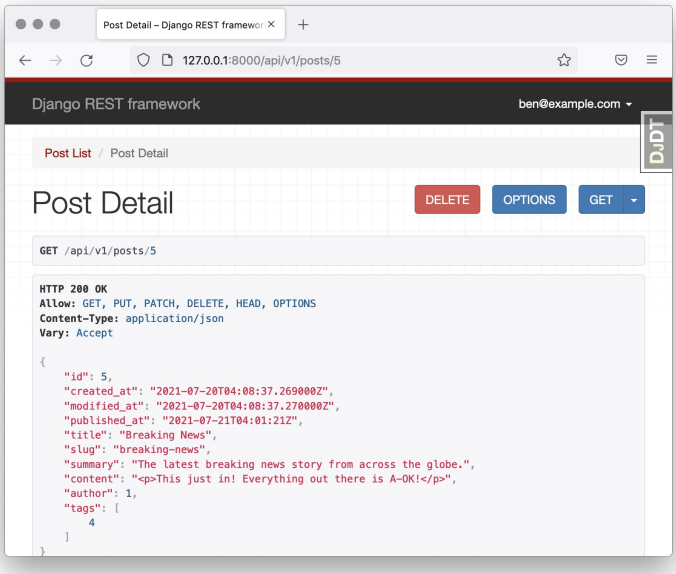

Figure 2

However, if you visit a `Post` that you're not the author of, then you won't see these elements on the page.

We'll finish this section off by looking at how to combine permissions.

Combining Permissions

Combining Permissions

Permissions classes that inherit from `BasePermission` can be combined using the bitwise operators `&` (and), `|` (or), and `~` (not).

You need to be cautious when combining your permission classes with the DRF provided classes. The provided classes only implement the `has_permission()` method for their permission checks, and just return `True` for `has_object_permission()`. This is because DRF doesn't know about your objects and thus doesn't know how to check permission for one of your objects.

Let's take for example another common scenario: allow admin users to make changes to any object. You might try to implement it like this:

```
from rest_framework.permissions import IsAdminUser


class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    permission_classes = [AuthorModifyOrReadOnly | IsAdminUser]
    # ...
```

However `IsAdminUser` always returns `True` from `has_object_permission()`, even if a user isn't logged in! So by using it we'll give all permissions to everyone.

We can solve this by subclassing `IsAdminUser` and implementing `has_object_permission()`:

```
from rest_framework import permissions


class IsAdminUserForObject(permissions.IsAdminUser):
    def has_object_permission(self, request, view, obj):
        return bool(request.user and request.user.is_staff)
```

Then we can combine this permissions class with our other custom class:

```
class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    permission_classes = [AuthorModifyOrReadOnly |
        IsAdminUserForObject]
    # leave other attributes as is
```

▼ **Inheritance**

This is the same reason our `AuthorModifyOrReadOnly` inherited from `IsAuthenticatedOrReadOnly` instead of `BasePermission`. We want to use `IsAuthenticatedOrReadOnly`'s `has_permission()` method and only implement our own `has_object_permission()` method.

Try It Out

Now you can make the same changes to your Blango instance. First create the `IsAdminUserForObject` in `blog/api/permissions.py`:

```
class IsAdminUserForObject(permissions.IsAdminUser):
    def has_object_permission(self, request, view, obj):
        return bool(request.user and request.user.is_staff)
```

Then open `blog/api/views.py`, and make sure to import the new class:

Open api/views.py

```
from blog.api.permissions import AuthorModifyOrReadOnly,
        IsAdminUserForObject
```

And finally, update the `permission_classes` attribute on `PostDetail`:

```
permission_classes = [AuthorModifyOrReadOnly |
        IsAdminUserForObject]
```

Refresh a DRF page in your browser. You'll notice that if you're logged in as an admin user you'll be able to make updates to any `Post`, even if you're not the author.

View Blog

In the next section, we'll look at how DRF handles showing relationships between objects.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish permissions"
```

- Push to GitHub:

```
git push
```