Tutorial 4: Authentication & Permissions

Currently our API doesn't have any restrictions on who can edit or delete code snippets. We'd like to have some more advanced behavior in order to make sure that:

- Code snippets are always associated with a creator.
- Only authenticated users may create snippets.
- Only the creator of a snippet may update or delete it.
- Unauthenticated requests should have full read-only access.

Adding information to our model

We're going to make a couple of changes to our `Snippet` model class. First, let's add a couple of fields. One of those fields will be used to represent the user who created the code snippet. The other field will be used to store the highlighted HTML representation of the code.

Add the following two fields to the `Snippet` model in `models.py`.

```
owner = models.ForeignKey('auth.User', related_name='snippets', on_delete=models.CASCADE)
highlighted = models.TextField()
```

We'd also need to make sure that when the model is saved, that we populate the highlighted field, using the `pygments` code highlighting library.

We'll need some extra imports:

```
from pygments.lexers import get_lexer_by_name
from pygments.formatters.html import HtmlFormatter
from pygments import highlight
```

And now we can add a `.save()` method to our model class:

```
def save(self, *args, **kwargs):
    """
    Use the `pygments` library to create a highlighted HTML
    representation of the code snippet.
    """
    lexer = get_lexer_by_name(self.language)
    linenos = self.linenos and 'table' or False
    options = self.title and {'title': self.title} or {}
    formatter = HtmlFormatter(style=self.style, linenos=linenos,
                              full=True, **options)
    self.highlighted = highlight(self.code, lexer, formatter)
    super(Snippet, self).save(*args, **kwargs)
```

When that's all done we'll need to update our database tables. Normally we'd create a database migration in order to do that, but for the purposes of this tutorial, let's just delete the database and start again.

```
rm -f tmp.db db.sqlite3
rm -r snippets/migrations
python manage.py makemigrations snippets
python manage.py migrate
```

You might also want to create a few different users, to use for testing the API. The quickest way to do this will be with the `createsuperuser` command.

```
python manage.py createsuperuser
```

Adding endpoints for our User models

Now that we've got some users to work with, we'd better add representations of those users to our API. Creating a new serializer is easy. In `serializers.py` add:

```
from django.contrib.auth.models import User

class UserSerializer(serializers.ModelSerializer):
    snippets = serializers.PrimaryKeyRelatedField(many=True, queryset=Snippet.objects.all()

    class Meta:
        model = User
        fields = ('id', 'username', 'snippets')
```

Because `'snippets'` is a *reverse* relationship on the User model, it will not be included by default when using the `ModelSerializer` class, so we needed to add an explicit field for it.

We'll also add a couple of views to `views.py`. We'd like to just use read-only views for the user representations, so we'll use the `ListAPIView` and `RetrieveAPIView` generic class-based views.

```
from django.contrib.auth.models import User


class UserList(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer


class UserDetail(generics.RetrieveAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

Make sure to also import the `UserSerializer` class

```
from snippets.serializers import UserSerializer
```

Finally we need to add those views into the API, by referencing them from the URL conf. Add the following to the patterns in `urls.py`.

```
url(r'^users/$', views.UserList.as_view()),
url(r'^users/(?P<pk>[0-9]+)/$', views.UserDetail.as_view()),
```

Associating Snippets with Users

Right now, if we created a code snippet, there'd be no way of associating the user that created the snippet, with the snippet instance. The user isn't sent as part of the serialized representation, but is instead a property of the incoming request.

The way we deal with that is by overriding a `.perform_create()` method on our snippet views, that allows us to modify how the instance save is managed, and handle any information that is implicit in the incoming request or requested URL.

On the `SnippetList` view class, add the following method:

```
def perform_create(self, serializer):
    serializer.save(owner=self.request.user)
```

The `create()` method of our serializer will now be passed an additional `'owner'` field, along with the validated data from the request.

Updating our serializer

Now that snippets are associated with the user that created them, let's update our `SnippetSerializer` to reflect that. Add the following field to the serializer definition in `serializers.py`:

```
owner = serializers.ReadOnlyField(source='owner.username')
```

Note: Make sure you also add `'owner',` to the list of fields in the inner `Meta` class.

This field is doing something quite interesting. The `source` argument controls which attribute is used to populate a field, and can point at any attribute on the serialized instance. It can also take the dotted notation shown above, in which case it will traverse the given attributes, in a similar way as it is used with Django's template language.

The field we've added is the untyped `ReadOnlyField` class, in contrast to the other typed fields, such as `CharField`, `BooleanField` etc... The untyped `ReadOnlyField` is always read-only, and will be used for serialized representations, but will not be used for updating model instances when they are deserialized. We could have also used `CharField(read_only=True)` here.

Adding required permissions to views

Now that code snippets are associated with users, we want to make sure that only authenticated users are able to create, update and delete code snippets.

REST framework includes a number of permission classes that we can use to restrict who can access a given view. In this case the one we're looking for is `IsAuthenticatedOrReadOnly`, which will ensure that authenticated requests get read-write access, and unauthenticated requests get read-only access.

First add the following import in the views module

```
from rest_framework import permissions
```

Then, add the following property to **both** the `SnippetList` and `SnippetDetail` view classes.

```
permission_classes = (permissions.IsAuthenticatedOrReadOnly,)
```

Adding login to the Browsable API

If you open a browser and navigate to the browsable API at the moment, you'll find that you're no longer able to create new code snippets. In order to do so we'd need to be able to login as a user.

We can add a login view for use with the browsable API, by editing the URLconf in our project-level `urls.py` file.

Add the following import at the top of the file:

```
from django.conf.urls import include
```

And, at the end of the file, add a pattern to include the login and logout views for the browsable API.

```
urlpatterns += [
    url(r'^api-auth/', include('rest_framework.urls',
                               namespace='rest_framework')),
]
```

The `r'^api-auth/'` part of pattern can actually be whatever URL you want to use. The only restriction is that the included urls must use the `'rest_framework'` namespace. In Django 1.9+, REST framework will set the namespace, so you may leave it out.

Now if you open up the browser again and refresh the page you'll see a 'Login' link in the top right of the page. If you log in as one of the users you created earlier, you'll be able to create code snippets again.

Once you've created a few code snippets, navigate to the '/users/' endpoint, and notice that the representation includes a list of the snippet ids that are associated with each user, in each user's 'snippets' field.

Object level permissions

Really we'd like all code snippets to be visible to anyone, but also make sure that only the user that created a code snippet is able to update or delete it.

To do that we're going to need to create a custom permission.

In the snippets app, create a new file, `permissions.py`

```
from rest_framework import permissions


class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    Custom permission to only allow owners of an object to edit it.
    """

    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

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

Now we can add that custom permission to our snippet instance endpoint, by editing the `permission_classes` property on the `SnippetDetail` view class:

```
permission_classes = (permissions.IsAuthenticatedOrReadOnly,
                      IsOwnerOrReadOnly,)
```

Make sure to also import the `IsOwnerOrReadOnly` class.

```
from snippets.permissions import IsOwnerOrReadOnly
```

Now, if you open a browser again, you find that the 'DELETE' and 'PUT' actions only appear on a snippet instance endpoint if you're logged in as the same user that created the code snippet.

Authenticating with the API

Because we now have a set of permissions on the API, we need to authenticate our requests to it if we want to edit any snippets. We haven't set up any authentication classes, so the defaults are currently applied, which are `SessionAuthentication` and `BasicAuthentication`.

When we interact with the API through the web browser, we can login, and the browser session will then provide the required authentication for the requests.

If we're interacting with the API programmatically we need to explicitly provide the authentication credentials on each request.

If we try to create a snippet without authenticating, we'll get an error:

```
http POST http://127.0.0.1:8000/snippets/ code="print 123"

{
    "detail": "Authentication credentials were not provided."
}
```

We can make a successful request by including the username and password of one of the users we created earlier.

```
http -a tom:password123 POST http://127.0.0.1:8000/snippets/ code="print 789"

{
    "id": 1,
    "owner": "tom",
    "title": "foo",

    "code": "print 789",
    "linenos": false,
    "language": "python",
    "style": "friendly"
}
```

Summary

We've now got a fairly fine-grained set of permissions on our Web API, and end points for users of the system and for the code snippets that they have created.

In part 5 of the tutorial we'll look at how we can tie everything together by creating an HTML endpoint for our highlighted snippets, and improve the cohesion of our API by using hyperlinking for the relationships within the system.
