Django Network Config Project

This document explains the network_config app that was built as a demo on how to implement basic functionality that is required for a sample network.

network/network/settings.py

This file specifies the necessary settings required for the app to function as expected. The only changes made to this file is in relation to INSTALLED_APPS. The last three apps have been added from the default settings file provided after creating a django project. rest_framework, rest_framework.authtoken and network_config.apps.NetworkConfigConfig are the three apps that have been added. The first two apps allow for the rest framework functionality and token authentication within the app. The last app (network_config) is the app that this document explains.

network/network_config/models.py

Models in Django describe how a database table must be set. Each model has specific attributes and these attributes can be simple fields such as text and integer fields. Attributes can also be relations to other models. Django has three relations, specifically ForeignKey, One-to-One and Many-to-Many. Detailed explanations on these relations can be found in django docs:

ForeignKey : https://docs.djangoproject.com/en/2.0/topics/db/examples/many_to_one/

Many-to-Many: https://docs.djangoproject.com/en/2.0/topics/db/examples/many_to_many/

One-to-Many: https://docs.djangoproject.com/en/2.0/topics/db/examples/many_to_many/

Every model can have additional attributes in the meta class such as the name of the database table that django creates (db_table) or the permissions that are available for each model.

NOTE: Permissions under the meta class are global model permissions. For example, if I assign a post tenant permission to a user, that user can post numerous tenants. If posting tenants should be limited to a group, this permission is not enough as it only checks if the user can post a tenant or not, regardless of the group to which this tenant is being posted.

When specifying a many-to-many relationships, a through model can be specified by the user or django would create a through model automatically. This is what the UgHasUser model is. (commented out to simplify implementing functionality)

The Profile model is a one-to-one model relationship with the User model from django.contrib.auth.models. This profile model can be used to add additional information for each user. The receiver function below the Profile model automatically creates a Profile model once a user object has been created. However, this functionality is not working as expected and has been commented out.

NOTE: The UserGroup model created extends the Group model and inherits fields such as id, group, name. However, it is potentially better to create a custom group model as this would solve inconsistencies in regards to adding permissions to users through the group models. Detailed explanation is given in the admin.py section.

network/network_config/serializers.py

Serializers describe what attributes for every model must be displayed when a view is accessed by the user. Attributes that relate to other models must be specified before the meta class. There are different ways to specify these attributes: http://www.django-rest-framework.org/api-guide/relations/

network/network_config/urls.py

The urls.py file provides different extensions of the base url that the user can use for specific rest calls based upon the model and the rest call. Urls that access list views are used for GET and POST calls and urls that access detail views are used for calls such as PUT, DELETE, etc.

network/network_config/views.py

The view.py file provides the user with an "interface" to interact with the data. TenantList, DeviceList, UserList use django rest framework's generics.ListCreateAPIView to implement the GET and POST request calls. TenantDetail, DeviceDetail, UserDetail use django rest framework's generics.RetrieveUpdateDestroyAPIView to implement GET, PUT, PATCH and DELETE calls. For customized views, specific functions that implement the above calls can be overridden. For example, to implement custom permissions, both TenantList and DeviceList have implemented the create method (for post requests). Since post requests don't have an object created in advance, adding permission classes is not enough. Permission checks must be done in regards to the data received from the request within the create method.

TenantList

The create method defined in tenant list checks if the user is a staff member before allowing to POST tenants. If the user is a staff member, the post request creates a Tenant object and a 201 created response is returned. If not, a 403 forbidden response is returned.

DeviceList

The create method defined in device list checks for two specific permissions. First, the create method checks if the owner_group of the device to be posted is a group that the user is part of. If so, the method then check if the user has permission to post devices. If both requirements are satisfied, the object is created and a 201 response is returned. If not, a 403 forbidden response is returned.

NOTE: This permission check is done separately in the create method since adding permission classes does not work as expected for a POST request due to the lack of an object being created. This is better explained here: https://github.com/encode/django-rest-framework/issues/1103

network/network_config/permissions.py

The permissions.py file implements custom permissions that can then be imported into views and assigned to permission_classes when using generic views. The custom permissions are self-explanatory and they are used to check if a user meets the necessary requirements.

NOTE: The IsDeviceOwnerGroup permission that is meant to be implemented for the DeviceDetail view does not work as expected. This is clear explained in the views.py file with a note about the DeviceDetail view.

network/network_config/admin.py

The admin.py file allows customization to authentication. In network_config, the UserGroup model is registered while the Group model is unregistered. Details can be found online.

NOTE: It may be better to create a completely custom and separate UserGroup model that does not inherit from the Group model as mentioned in the models.py section.

network/network_config/tests.py

This file implements tests to ensure the app works as expected. Django testing is explained in detail here: http://www.django-rest-framework.org/api-guide/testing/

The names of the tests are self-explanatory regarding their functionality. This app only allows staff members to create tenants and this is checked in the TenantTests class.

DeviceTests

The setup function creates a superuser (who is automatically a staff member as well) and uses this superuser to create a tenant. Two users user_can_post and user_cannot_post are created. Two groups CanPost and CannotPost are then created and CanPost is assigned permission can_post_devices. user_can_post is assigned to CanPost and user_cannot_post is assigned to CannotPost.

The test_create_device method tests different scenarios where the device should and shouldn't be created. There are three subtests within the test_create_device function.

Using the network_config app

Creating a superuser

A superuser is required to understand and explore features within the network_config. Open the network project folder in an IDE such as Pycharm and configure the required python 3.4+, django 2.0+ and djangorestframework 3.0+ in the project settings.

Next, open the terminal within Pycharm and ensure that the terminal is open in the same directory as the manage.py file is in the network project structure. Run this command to start the app: **python manage.py runserver**

Now open a new terminal and ensure that the terminal is open in the same directory as the manage.py file is in the network project structure. Follow the resulting prompts to create a superuser for the network project after executing this command: **python manage.py createsuperuser**

Creating users with/without staff member access

Users can be created programmatically (can be found online) or at http://127.0.0.1:8000/admin/ with a superuser credential. Setting the staff access permission determines if a user is a staff member or not. Superusers are granted staff access by default.

NOTE: The above link can also be used to create usergroups and tokens for users. The network_config app is configured to create tokens for every user that is created.

Tokens

A token is generated for **every** user created. These tokens can be accessed and modified at http://127.0.0.1:8000/admin/ with a superuser credential. These tokens can also be programmatically changed using the manage.py file within the app (These are explained well online.)

Creating a Tenant

To create a tenant, first create an "admin" user (either a superuser (as shown above) or a user with staff access).

Once this user is created, login to http://127.0.0.1:8000/tenants/ and fill out the post form that is visible and click the post button. Alternatively, you can post from a terminal using an http client such as curl or Httpie. Examples of programmatically creating/altering Tenants and Devices can be found under the **Working REST calls** section below.

Creating a UserGroup

To create a usergroup, at least one tenant must be created in advance. After a tenant is created as described above, a usergroup can be created at http://127.0.0.1:8000/admin/ . When creating a usergroup, a tenant must be chosen. Adding users and permissions is optional. These are the custom permissions created for network: Can EDIT Devices, Can POST Devices, Can EDIT Tenants and Can POST Tenants. Since the PUT request functionality has not been implemented completely, the edit permissions **do not** work as expected.

NOTE: Since a new group model was created for the network file, assigning permissions to users through groups is "broken". When adding users to a usergroup above, the users do not recognize the permissions. The current solution for this problem is to create the usergroup first with the necessary permissions. Once created, go back to http://127.0.0.1:8000/admin/ and add the necessary groups for each user by editing the groups field for each user. This is important to ensure permissions work as expected and **Moving Forward** this must be fixed to ensure permissions are added to the user either by adding users to a usergroup or by adding usergroups under the group fields for each user.

Creating Devices

To create a device, a user must have the Can POST Devices permission assigned (either directly, or through a usergroup as mentioned above). A Tenant must also be created. Next, go to http://127.0.0.1:8000/devices and fill out the POST form. Ensure to login with a user that is part of a usergroup. This app only allows you to POST a device to a owner_group that the user is a member of. Devices can also be created programmatically as described in the **Working REST calls** section below.

Working REST calls

GET a list of tenants

http GET 127.0.0.1:8000/tenants/

POST a tenant using a previously created "admin" user using Httpie:

http –a admin:password POST 127.0.0.1:8000/tenants/ name="<name>" is_active="<True/False>"

PUT and DELETE a tenant using a previously created “admin” user using Httpie:

http –a admin:password PUT 127.0.0.1:8000/tenants/<tenant_id#> name=“<name>” is_active=“True/False”

http –a admin:password DELETE 127.0.0.1:8000/tenants/<tenant_id#>

NOTE: The tenant id number used at the end of the url for PUT and DELETE requests results in the Tenant Detail View being used. The url is configured to use the pk value of the object in the network/network_config/urls.py file. This pk value is always the tenant_id#.

GET a list of all devices

NOTE: The devices class has been configured to return a list of devices based on the usergroups that the user is a member of. Thus, if the user is part of Group A, then that user can only see devices in Group A. Currently there is no way to check all devices in one views except by adding that user to all groups through http://127.0.0.1:8000/admin/network_config/usergroup/. If the groups model was the default django model, then the group field for the user could be used to directly select all the groups.

http –a user:password GET 127.0.0.1:8000/devices/

POST a device using an appropriate user as mentioned above using Httpie:

http –a user:password POST 127.0.0.1:8000/devices/ name=“<name>” is_active=“<True/False>” tenant=“<tenant_name>” owner_group=“<usergroup_name>” note=“<text>” modified_by=“<user_name>”

PUT and DELETE a tenant using an appropriate user as mentioned above using Httpie:

http –a user:password PUT 127.0.0.1:8000/devices/<device_id#> name=“<name>” is_active=“<True/False>” tenant=“<tenant_name>” owner_group=“<usergroup_name>” note=“<text>” modified_by=“<user_name>”

http –a user:password DELETE 127.0.0.1:8000/devices/<device_id#>

NOTE: The device id number used at the end of the url for PUT and DELETE requests results in the Device Detail View being used. The url is configured to use the pk value of the object in the network/network_config/urls.py file. This pk value is always the tenant_id# in this app.

Moving Forward (Features to fix, expanding project scope, etc)

Logging out of DeviceList View results in an error since a user is required

When accessing the device list api through the browser (http://127.0.0.1:8000/devices), a user is required to be logged in ahead of time. The Device List api specifies a get_queryset method that prevents users from viewing devices that are not in a group the user is a member of. But to do this, a valid user is required as the get_queryset method checks the user’s group field. When a user is not logged in, an “Anonymous” user is used within Django. But since an Anonymous User is not in any usergroup, an error occurs when the queryset is returned. This is not an urgent matter since the app is not being built for anonymous users. However, know that to access device list, a user must have been logged in ahead of time when using the api.

Authentication Classes to TenantList View

Authentication classes for the TenantList View allows the user to access the TenantList View either through an authentication token or through the regular user credentials. However, this is currently not working as expected for the TenantList View and is being worked on. This can be seen with the different tests in the tests.py file. Adding a token authentication class to the TenantList View results in a 401 response being returned instead of the expected response and this will be fixed. As such, token authentication has been removed from the TenantList View.

Fixing/Expanding permission checks for devices:

This app currently checks that a device posted to an owner_group is usergroup that the user is a member of. In addition, the app checks if the user has a Can POST Devices permission. However, the app should also check that the Tenant the usergroup comes from is also the same Tenant that the device is being posted to. This feature is currently missing and will be added later.

Currently posting or altering devices requires the modified_by field to be specified by the user. However, it is better to ensure that this field recognizes the user automatically and assigns the user to the modified by field to avoid any “foul” play. This detection will be added at a later time.

Adding permissions to the permission classes directly to check for the owner group of the user and the device causes unexpected problems. The IsDeviceOwnerGroup permission checks if the user is part of a user group that the user intends to add the device into. But in a PUT statement, if the owner_group needs to be changed, then the permission class needs to check the owner_group of both the old object and the owner_group from the new request. This will be implemented at a later time by overwriting the def update function and checking for permissions. This is similar to how the create functions were overwritten but the internal permission checks and updating the serializer can be different. This above method should also check how devices are altered in regards to changing the tenant.