django-scoped-permissions

Release 0.1.6

Tormod Haugland

May 23, 2022

USER GUIDE

Django Scoped Permissions is a custom permission system Django. The core concept of the system is a “scope”, which is very loosely defined as a string followed by a colon. A scoped permission is a string containing one or more scopes, defining a permission hierarchy.

Such a scoped permission typically follows the natural hierarchy of your business domain.

CHAPTER

ONE

INSTALLATION

Installation is done with pip (or via wrappers such as pipenv or poetry):

```
pip install django_scoped_permissions
```

Make sure to add django_scoped_permissions to your INSTALLED_APPS:

```
INSTALLED_APPS = [
    # ...
    "django_scoped_permissions"
    # ...
]
```

CHAPTER TWO

CORE CONCEPTS

Throughout the section we're gonna use "organizations" and "users" as typical examples of models within your business domain. Hence we assume that in all the examples below, that there exists a model *Organization* and a model *User*.

2.1 Permission strings and scopes

The core concept of this library revoles around the notion of a *scoped permission string*. They (may) look like this:

```
read
write
update
create
user:1
user:1:read
organization:1
organization:1:create
=organization:1
-organization:1
organization:1:user:2
```

We say that each string delimited by a colon (:) defines a **permission scope**. We say that each subpart defined by a new colon of a permission defines a **parent scope** of the permission.

Take the permission `organization:1:create`. Here `organization`, `organization:1` and `organization:1:create` are the various scopes of the permission string. The two first are the parent scopes of the permission string.

While in many typical permission schemes permissions are matched exactly to ensure a grantee has the required access, in this library, permission strings are matched in a "cascading" manner on all parent scopes. We say that scopes are used in one of two contexts: As a **required permission** or as a **granting permission**.

Say for instance that you as a user have been granted a permission `organization:1`. Then suppose you are trying to access a resource which requires the permission `organization:1:setting:user`. Since you have the `organization:1` permission, you will be given access to this, as you **have access to a higher level scope in the required full scoped permission string**. In other words, the permission `organization:1` as a granting permission grants access to `organization:1:setting:user` as a required permission.

Similarly, a user with any of the following scoped permission strings would be given access to the resource:

- `organization`
- `organization:1`
- `organization:1:settings`

Let's take some basic scope examples, and read very roughly what they would mean:

- `organization`: Has/requires access to all organizations
- `organization:1`: Has/requires access to organization with id 1.
- `user:1`: Has/requires access to the user with id 1.
- `organization:1:user:1`: Has/requires access to the user with id 1 within the organization with id 1.

Note that the exact interpretation of the permission strings are up to you within your own business domain.

2.2 Scopes and verbs

There are two parts of a scoped permission string, the **base** and the **verb**. The verb is optional. Some examples of typical permissions with verbs:

- `read`
- `user:create`
- `user:1:update`
- `organization:1:delete`

These permissions can roughly be read as

- Can read everything.
- Can create users.
- Can update user 1 **and any sub-resource of the user**.
- Can delete organization 1 and delete **and any sub-resource of the organization**.

Some examples of typical permissions without verbs:

- `user`
- `organization:1`
- `organization:1:user`

These permissions can roughly be read as

- Has full access to all users.
- Has full access to the organization with id 1 **and any sub-resource**.
- Has full access to all users within the organization with id 1 **and any sub-resource of the users**.

Verbs are special when used in required permissions. Let's illustrate with two quick examples.

2.2.1 Example 1

Say you have the permission `user:1:read`. Now suppose you are trying to access a resource which requires `user:1:settings:read`, where *read* is a verb. Notably, the permission you have is **not** a parent scope of the required permission.

However, the permission internal mechanism will still match these two permissions, as the verb will be attached recursively to each parent scope of your granting permission when trying to match the permissions.

Other permissions which would have granted access in this scenario are the following:

- `user:1:settings:read`
- `user:1:settings`
- `user:1`
- `user:read`
- `user`
- `read`

From this example, we can see that creating a simple read-only user for all resources can be done by simply attaching the permission `read`.

Again, this depends on how you structure your business logic, but shows some of the power of the library.

2.2.2 Example 2

Say you have the permission `user:setting` and suppose you are trying to access a resource which requires `user:1:setting`, where setting is not a verb.

Here, you will not get access, as again the permission you have is **not** a parent scope of the required permission.

From the above two examples, it should be clear that creating a superuser can be done with something akin to adding the following permissions: `read`, `update`, `create`, `delete`.

2.3 Exact permissions

One problem with the "cascading" property of the scoped permission system is that it may become hard to limit permissions "higher up" in the hierarchy. Suppose for instance that you want to grant a user access to read information about an organization without granting full read access to everything in the organization.

With the permission `organization:1:read` we are likely to have the problem that the user automatically gets read-access to all resources within the organization. That is, if you model permission strings in a manner alike `organization:1:user`, `organization:5:vehicles`.

We can solve this problem by using an exact operator before the permission `=organization:1:read`. This very roughly translates to "Has read access to organization 1, but no data within organization 1". Exactly what this means semantically is up to you, but in terms of permission matching, this basically means that the required scope and the granting scope must match exactly (not including the "="). To give an example, `=organization:1` will **not** match `organization:1:user`.

This can be used on any permission string:

- `=organization:1:read`
- `=organization:1`
- `=user`

2.4 Exclusion permissions

Another problem we might have, is revoking specific permissions. Say for instance that you want a user by default to have access to all organizations, so the user has the permission `organization`. But you also want to revoke access to the organization with id 2.

We can achieve this with an **exclusion permission**: `-organization:2`. In combination, these two permissions yield access to all organizations apart from the organization with id 2.

2.5 Exact exclusion permissions

We can combine the above two notions, e.g. *-=organization:2*. This will revoke access to exactly the permission *organization:2*.

Interestingly, this will still grant access to required permissions such as *organization:2:user*.

2.6 Precedence

Note that in case of conflicts, permissions take precedence in the following order (higher being prioritised).

1. Exact exclusion
2. Exact inclusion
3. Exclusion
4. Inclusion

Some examples:

```
-=scope1:scope2 > =scope1:scope2
=scope1:scope2 > -scope1:scope2
-scope:scope2 > scope1:scope2
```

Hence, if a user has the permission `-=scope1:scope2` **and** `=scope1:scope2`, the user will not be granted access to `scope1:scope2`.

2.7 Final note

Note that while a lot of the remaining part of the documentation will revolve aronud how to set up permissions using the database, the library can be used fully statelessly.

The library's primary purpose is to provide helper methods and methodology for using the above schematics to do permission matching. Hence it is fully possible to simply generate a scoped permissions on runtime, and then match with required static or dynamic permissions.

This will be explored in a later chapter.

CHAPTER THREE

BASIC USAGE

3.1 Basic scope matching

There are a number of core methods in the library which is used to perform the primary matching of permissions in accordance with the previous section. The most important of these is the `scope_grants_permission` and `scopes_grant_permissions` methods.

```
from django_scoped_permissions.core import scope_grants_permission, scopes_grant_
↪permissions

# First argument is required scope, second is granting scope.
scope_grants_permission("scope1:scope2", "scope1")  # True
scope_grants_permission("scope1:scope2", "=scope1")  # False
scope_grants_permission("scope1", "-scope1")  # False
scope_grants_permission("scope1:scope2", "scope3:edit")  # False

# First argument is required scopes, second is granting scopes. Note that the method
# returns true if _any_ matches. Also note that any excluding permission takes␣
↪precedence.
scopes_grant_permissions(["scope1:scope2"], ["scope1"])  # True
scopes_grant_permissions(["scope1:scope2"], ["=scope1", "scope1"])  # True
scopes_grant_permissions(["scope1:scope2"], ["-scope1", "scope1:scope2"])  # False
```

These methods also accepts a third argument for the required *verb*.

```
from django_scoped_permissions.core import scope_grants_permission, scopes_grant_
↪permissions

scope_grants_permission("scope1:scope2", "scope1:read", "read")  # True
scope_grants_permission("scope1:scope2", "scope1", "read")  # True
scope_grants_permission("scope1:scope2", "scope1:scope2:read", "read")  # True
scope_grants_permission("scope1:scope2", "scope1:scope2:update", "read")  # False

scopes_grant_permissions(["scope1:scope2"], ["scope1", "scope1:read"], "read") # True
scopes_grant_permissions(["scope1:read", "scope3:update"], ["scope3", "=scope1:read"],
↪"read") # True
# Note here that since we have a direct exclude on scope3:update, the request is␣
↪disallowed.
scopes_grant_permissions(["scope1:read", "scope3:update"], ["-scope3:update",
↪"=scope1:read"], "read") # False
```

Under the hood, these methods use the `scope_matches` method, which simply makes a required scope with a granting scope. Note that while it handles exact matches properly, it does not handling excluding scopes properly. This is done in the above methods. It is like a light-weight `scope_grants_permission` which does not handle exclude permission nor verbs. It should rarely be used directly.

Another important helper-method is `create_scope`. It simply concatenates objects and strings to create a scoped permission string. Since creating scoped strings is a major part of using this library, making the code as readable as possible is important. The method also adds some magic which allows us to pass in model instances and model classes, which will automatically have their (associated model) names extracted:

```
from django_scoped_permissions.core import create_scope
from users.models import User # hypothetical user model
from forum.models import Thread # hypothetical forum thread model

create_scope("scope1", "scope2") # -> "scope1:scope2"
create_scope(*["scope1", "scope2", "scope3", "scope4"]) # -> "scope1:scope2:scope3:scope4
↪"
create_scope(User, 1) # -> "user:1"

forum_thread = ForumThread.objects.get(pk=1337)
create_scope(forum_thread, forum_thread.id, "read") # -> "thread:1337:read"
```

3.2 Models and Mixins

There are four models of importance in the library: `ScopedPermission`, `ScopedPermissionGroup`, `ScopedModel` and `ScopedPermissionHolder`.

3.2.1 ScopedPermission

Simply a model which persists a scoped permission with *exact* and *exclude* parameters.

3.2.2 ScopedPermissionGroup

A model which has a name, and a m2m-field to `ScopedPermission`. I.e. it contains a number of scoped permissions under a group name. This can be useful in scenarios where you want to create reusable sets of permissions.

3.2.3 ScopedModel

A Model inheriting from ScopedModel is a model which provides two methods: `get_required_scopes` and `has_permission`. The model is based in the mixin ScopedModelMixin, and simply inherits from Django's `models.Model` as well.

`get_required_scopes` should be implemented by every model inherited from ScopedModel, and should return all scopes which on a match will grant access to **an object of the model**.

Here is an example from a real-life application (simplified for brevity):

```
# forum.models

from django_scoped_permissions.models import ScopedModel
```

(continues on next page)

(continued from previous page)

```python
class Thread(ScopedModel):
    organization = models.ForeignKey(Organization, on_delete=models.CASCADE)
    title = models.TextField()

    created_at = models.DateTimeField(auto_now_add=True)

    def get_required_scopes(self):
        return [
            create_scope(self, self.id),   # thread:{self.id}
            create_scope(Organization, self.organization.id, self, self.id) #␣
↪organization:{organization.id}:thread:{self.id}
        ]


class Post(ScopedModel):
    thread = models.ForeignKey(Thread, on_delete=models.CASCADE)
    content = models.TextField()

    created_at = models.DateTimeField(auto_now_add=True)

    def get_required_scopes(self):
        return [
            # You get the idea
            create_scope(self, self.id),
            create_scope(self.thread, self.thread.id, self, self.id)
            create_scope(Organization, self.organization.id, self.thread, self.thread.id,
↪ self, self.id)
        ]
```

So the point here is the following:

1. We typically want the objects to be accessible directly when a calling user has a direct matching permission, e.g. “thread” or “thread:1”
2. But also when the user has permission to an object higher up in your data hierarchy, e.g. “organization” or “organization:1”.

Exactly how you structure this is completely up to you, and depends a lot on your use-case and your data. If in the above example, say, we didn’t want a post to be accessible just because a user has access to a thread, we would remove the second entry under `Post.get_required_scopes`.

`get_required_scopes` should return a list.

The second method `has_permission` has a much more opinionated implementation, and should not be overriden unless you know what you are doing. It takes as argument a ScopedPermissionHolderMixin instance and an optional action, and checks whether the instance has access to the current object, as defined per the `get_required_scopes` method.

ScopedModel is inherently stateless, and adds no extra database-bloat to your model.

3.2.4 ScopedPermissionHolder

ScopedPermissionHolder is an abstract model used on the models you want to be able to hold permissions that grants access. It does two important things:

1. Adds m2m database fields to both `ScopedPermission` and `ScopedPermissionGroup`.
2. Implements the four permission methods of ScopedPermissionHolderMixin.

The four methods mentioned are `get_granting_scopes`,:code:*has_scoped_permissions*, `has_any_scoped_permissions`, `has_all_scoped_permissions`.

Note that `has_scoped_permissions` is just an alias for `has_any_scoped_permissions` by default. There is nothing wrong with overriding this default behaviour, however. `has_scoped_permissions` is the method which will typically be used by the library internally.

`get_granting_scopes` is the method of most interest here. It returns all the scopes the holder has permission to. The default implementation of this simply fetches all the scopes in the database, both directly associated to the holder, but also via the holder's ScopedPermissionGroups. This default implementation "hides" in a property called `resolved_scopes`.

Very typically you are going to override this default implementation (by expanding on it). A typical example is a User model, which will always have access to their own resources:

```python
class User(AbstractUser, ScopedPermissionHolder):

    # ...

    def get_granting_scopes(self):
        super_scopes = super().get_granting_scopes()

        return super_scopes + [create_scope(self, self.id)]
```

`get_granting_scopes` should return a list.

The ScopedPermissionHolder model implements the ScopedPermissionHolderMixin class, which simply provides stubs for the four methods mentioned aboce.

Finally, `ScopedPermissionHolder` has a handy utility function `add_or_create_permission` which simply creates a scoped permission object in the database (or retrieves one if it exists), and adds it to the holder.

3.3 Common recipes

3.3.1 User with User Types/ Groups

If you want user types with permission, you probably want the user to automatically inherit all permissions.

```python
class UserType(ScopedPermissionHolder):
    name = models.TextField()

class User(AbstractUser, ScopedPermissionHolder):

    user_types = models.ManyToManyField(UserType, blank=True)

    def get_granting_scopes(self):
```

(continues on next page)

(continued from previous page)

```
        user_scopes = super().get_granting_scopes() + [create_scope(self, self.id)]

        user_type_scopes = [
            scope for user_type in self.user_types.all() for scope in user_type.get_
↪granting_scopes()
        ]

        # We might want to delete duplicates here
        return list(set(user_scopes + user_type_scopes))
```

3.3.2 Permission with placholders

There are no rules regarding what you can put in a scoped permission string. Which means you can also put placeholders which resolve at runtime. A typical usecase here would be a permission which has a placeholder for say an organization id which is resolved on runtime.

Here we also use another utility function which expands a scope permission string based on context values.

```
from django_scoped_permissions.util import expand_scopes_from_context

class User(AbstractUser, ScopedPermissionHolder):

    def get_granting_scopes(self):
        user_scopes = super().get_granting_scopes() + [create_scope(self, self.id)]

        organization_ids = [organization.id for organization in self.organizations.all()]
        expanded_scopes = expand_scopes_from_context(user_scopes, {"organization":␣
↪organization_ids})

        return expanded_scopes

class Organization(ScopedPermissionModel):
    name = models.TextField()
    users = models.ManyToManyField(User, on_delete=models.CASCADE, related_name=
↪"organizations")


user = User.objects.get(pk=1)
user.add_or_create_permission("organization:{organization}:read")  # Add read permission␣
↪to all organizations the user is a member of
organization_1 = Organization.objects.create(name="org1")
organization_2 = Organization.objects.create(name="org2")

user.organizations.add(organization_1)
user.organizations.add(organization_2)

print(user.get_granting_scopes())  # Prints ["organization:1:read", "organization:2:read
↪", "user:1"]
```

3.3.3 Superusers

There is no explicit superuser-handling in the library. This is intentional, as some applications of the library might not want such functionality. The easiest way to get superuser-functionality currently, is to do one of two things:

1. Make sure superusers have all relevant top-level verbs (e.g. create, read, update, delete, or which ever verbs you use).
2. Create two new abstract model which inherits from ScopedPermissionModel and ScopedPermissionHolder, and override the methods `ScopedPermissionModel.has_permission` and `ScopedPermissionHolder.has_scoped_permissions`.

When we implement wildcards (on the roadmap), this becomes a tad easier.

CHAPTER

FOUR

GUARDS

Sometimes, we need more complex permission matching than just simple matching between two sets of scopes. We might for instance need to match on something like: “Does the user have x OR (y and z) BUT NOT w”.

For this purpose, we have the class `ScopedPermissionGuard`.

Most decorators and permission-attributes in this library can take ScopedPermissionGuard(s) as arguments. And indeed, most of them use this class under-the-hood.

4.1 Basic usage

ScopedPermissionGuards in its most simple usage take one or two arguments:

```
from django_scoped_permissions.guards import ScopedPermissionGuard

ScopedPermissionGuard("scope1:scope2")
ScopedPermissionGuard("scope1", "verb")
# Can also be supplied by kwargs
ScopedPermissionGuard(scope="scope1", verb="read")
```

The guards can verify that a set of granting permissions has access via the *has_permission* method:

```
from django_scoped_permissions.guards import ScopedPermissionGuard

guard = ScopedPermissionGuard(scope="scope1", verb="read")

assert guard.has_permission("scope1")
assert guard.has_permission("scope1:read")
assert guard.has_permission(["read", "scope3"])
assert not guard.has_permission("scope2")
```

4.2 Combining guards and operators

The power of guards, however, is unleashed when we combine guards with boolean operators. E.g:

```
from django_scoped_permissions.guards import ScopedPermissionGuard

guard_1 = ScopedPermissionGuard(scope="scope1", verb="read")
guard_2 = ScopedPermissionGuard("scope2")

# This guard requires you to have scope1:read AND scope2
guard_3 = guard_1 & guard_2

# This guard requires you to have scope1:read OR NOT scope2
guard_4 = guard_1 | ~guard_2

# This guard requires you to have scope1:read and scope2 XOR (not scope1 and scope3)
guard_5 = (guard_1 & guard_2) ^ (ScopedPermissionGuard("scope1") & ScopedPermissionGuard(
↪"scope3"))

assert guard_4.has_permission(["scope1", "scope2"])
assert guard_4.has_permission(["scope3"])
assert not guard_4.has_permission(["scope3", "scope2"])

assert guard_5.has_permission(["scope1:read", "scope2"])
assert guard_5.has_permission(["scope3"])
```

Supported operators are:

- &: AND
- |: OR
- ^: XOR
- ~: NOT

4.3 Usage in practice

ScopedPermissionGuards can be used in all decorators and properties where permissions are supplied:

```
@gql_has_permission(
    ScopedPermissionGuard(
        scope="user", verb="read") &
    ScopedPermissionGuard("organization:{context.organization.id}:read")
)
def resolve_company_user(self, info, **kwargs):
    pass
```

CHAPTER FIVE

DECORATORS

The library supplies two decorators to use with graphql queries/mutations and with functional views, respectively:

- `gql_has_scoped_permissions`
- `function_has_scoped_permissions`

The functions take the same arguments: Either a ScopedPermissionGuard, a combination of these, or the same inputs as a ScopedPermissionGuard would take. See guards for more info.

Some examples:

```
@gql_has_scoped_permissions("scope1:scope2")
def resolve_something(self, info, **kwargs):
    return None

@gql_has_scoped_permissions(scope="scope1:scope2", verb="read")
def resolve_something_else(self, info, **kwargs):
    return None

@gql_has_scoped_permissions(
    ScopedPermissionGuard("scope1:scope2") & ScopedPermissionGuard("scope3") |
    (
        ScopedPermissionGuard("scope4") ^ ScopedPermissionGuard("scope5")
    )
)
def resolve_something_very_guarded(self, info, **kwargs):
    return None
```

```
@function_has_scoped_permissions("scope1:scope2")
def handle_something(request):
    return None

@function_has_scoped_permissions(scope="scope1:scope2", verb="read")
def handle_something_else(request):
    return None

@function_has_scoped_permissions(
    ScopedPermissionGuard("scope1:scope2") & ScopedPermissionGuard("scope3") |
    (
        ScopedPermissionGuard("scope4") ^ ScopedPermissionGuard("scope5")
    )
)
```

(continues on next page)

(continued from previous page)

```
def handle_something_very_guarded(request):
    return None
```

CHAPTER

SIX

GRAPHENE INTEGRATION

This library integrates with the excellent graphene library in a simple way: By providing a ScopedDjangoNode class with default permission handling.

```
from django_scoped_permissions.graphql import ScopedDjangoNode

class UserNode(ScopedDjangoNode):
    class Meta:
        # Note that interfaces = (Node,) is added automatically
        model = User
```

This default implementation here adds a custom permission guard on resolving of the node. If the model is a ScopedModel, when resolving an object, its `get_required_scopes` method is used to retrieve the required scopes, and matches these against the callers `get_granting_scopes`.

6.1 Custom node permissions

One can also customize the required scopes:

```
from django_scoped_permissions.graphql import ScopedDjangoNode

class UserNode(ScopedDjangoNode):
    class Meta:
        model = User
        node_permissions = (
            "scope1:scope2",
        )
```

Now any user with scopes granting access to `scope1:scope2` will be able to access any node.

You can also use variables in the permissions, to resolve context values or values/functions of the object:

```
from django_scoped_permissions.graphql import ScopedDjangoNode

class UserNode(ScopedDjangoNode):
    class Meta:
        model = User
        node_permissions = (
            "company:{context.company.id}:user",
        )
```

The following special variables will be available in this context:

- `required_scopes`: The required scopes of the object being resolved.
- `user`: The calling user.

You can also use permission guards:

```
from django_scoped_permissions.graphql import ScopedDjangoNode

class UserNode(ScopedDjangoNode):
    class Meta:
        model = User
        node_permissions = ScopedPermissionGuard(
            "company:{context.company.id}:user",
        ) | ScopedPermissionGuard(scope="user", verb="read")
```

6.2 Custom field permissions

The class also provides a streamlined way to provide permissions for field resolvers easily:

```
from django_scoped_permissions.graphql import ScopedDjangoNode

class UserNode(ScopedDjangoNode):
    class Meta:
        model = User
        field_permissions = {
            "weight": ("users:can-read-weight", "{required_scopes}:read-weight", )
        }
```

CHAPTER

SEVEN

GRAPHENE DJANGO CUD INTEGRATION

This library also integrates cleanly with Graphene Django Cud by providing the following mutations:

- `ScopedDjangoCreateMutation`
- `ScopedDjangoUpdateMutation`
- `ScopedDjangoPatchMutation`
- `ScopedDjangoDeleteMutation`
- `ScopedDjangoBatchDeleteMutation`
- `ScopedDjangoFilterDeleteMutation`

These can be split into two groups: Those that alter specific objects, and those that don't.

7.1 Object-specific mutations

Those that do, work very similar to the last section.

For instance:

```python
from django_scoped_permissions.graphql import ScopedDjangoUpdateMutation

class UpdateUserMutation(ScopedDjangoUpdateMutation):
    class Meta:
        model = User
```

Using this mutation will require access to the object, as specified by the object's `get_required_scopes` method.

You can also customize the permissions required by using the *permissions* property:

```python
from django_scoped_permissions.graphql import ScopedDjangoUpdateMutation

class UpdateUserMutation(ScopedDjangoUpdateMutation):
    class Meta:
        model = User
        # E.g.
        permissions = (
            "users:update",
            "{required_scopes}
        )
```

(continues on next page)

(continued from previous page)

```
        # or e.g.
        permissions = (
            "company:{context.company.id}:update-users
        )

        # Or e.g.
        permissions = ScopedPermissionGuard(scope="users", verb="update")
```

7.2 Other mutations

Mutations that don't alter specific objects do not have any default permission implementation, and requires you to fill out the permissions property.

```
from django_scoped_permissions.graphql import ScopedDjangoUpdateMutation

class CreateUserMutation(ScopedDjangoCreateMutation):
    class Meta:
        model = User
        # E.g.
        permissions = (
            "users:create",
        )

        # or e.g.
        permissions = (
            "company:{context.company.id}:create-users
        )

        # Or e.g.
        permissions = ScopedPermissionGuard(scope="users", verb="update")
```

CHAPTER
EIGHT

STATELESS USAGE

One of the advantages of this library, is that in principle, permissions can be used more or less in a stateless manner.

To illustrate, suppose we have three user types: “normal”, “moderator” and “administrator”. Each of these user types should yield a different set of permissions. If these should not vary between users, we can easily achieve what we want by an appropriate implementation of `get_required_scopes`.

```
from django_scoped_permissions.graphql import ScopedDjangoNode
from django.utils.translation import _ as gettext_lazy

class User(models.Model):
    class UserTypeChoices(models.TextChoices):
        NORMAL = "normal", _("Normal")
        MODERATOR = "moderator", _("Moderator")
        ADMINISTRATOR = "administrator", _("Administrator")

    ...
    organization = models.ForeignKey(Organization, on_delete=models.CASCADE) # Say every␣
↪user is part of an organization
    user_type = models.CharField(max_length=16, choices=UserTypeChoices.choices,␣
↪default=UserTypeChoices.NORMAL)


    def get_granting_scopes(self):
        scopes = []

        # Always grant access to oneself
        scopes.append(create_scope(User, self.id))

        if self.user_type == UserTypeChoices.NORMAL:
            # Has read access to your organization
            scopes.append(create_scope(Organization, self.organization.id, "read"))
        elif self.user_type == UserTypeChoices.MODERATOR:
            # Has read access to all other users
            scopes.append(create_scope(User, "read"))
            scopes.append(create_scope(Organization, self.organization.id, "read"))
        elif self.user_type == UserTypeChoices.ADMINISTRATOR:
            # Can do everything within an organization, can also create users
            scopes.append(
                create_scope(Organization, self.organization.id, verb)
                for verb in ["create", "update", "read", "delete"]
            )
```

(continues on next page)

(continued from previous page)

```
            scopes.append(create_scope(User, "create"))

        return scopes
```

And this solution may in fact be sufficient for all our requirements. Typically, we've found that a powerful `get_granting_scopes` method often diminishes the need for a lot of persistent storage.

CHAPTER

NINE

MODELS DOCUMENTATION

Documentation for all models.

9.1 ScopedDjangoNode

A wrapper for DjangoObjectType which automatically adds permission handling to the node.

All meta arguments:

Argument	type	Default	Description
model	Model	None	The model. **Required**.
node_permissions	Iterable	None	The permissions required to access the node. If not supplied, the models "get_base_scopes" method will be used to populate this field.
field_permissions	Dict	None	A dictionary of permissions per field of the model used to check if the calling user has access to the field.
allow_anonymous	Boolean	False	If true, the node can be accessed by an anonymous user.

```python
class User(HasScopedPermissionsMixin, AbstractUser, ScopedModel):
    secret_field = models.TextField()

    def get_base_scopes(self):
        return [create_scope(self, self.id)]  # E.g. "user:1"


class UserNode(ScopedDjangoNode):
    class Meta:
        model = User
        allow_anonymous = False

# Example with more restrictive permissions
class RestrictiveUserNode(DjangoScopedNode):
    class Meta:
        model = User
        node_permissions = ["user"]  # Requires all permissions to all users
        field_permissions = {
            "secret_field": ["user:secret_field"]
        }
```