

## Django Implement Field Permissions

**User**  
How can I implement field-level permissions in Django?

For example, I want to make sure that those with the permission type "memberProject" can see all other users, but those with the permission type "memberGeneral" can only see themselves and not any other users.

The "PermissionType" class is defined as follows:

```
class PermissionType(AbstractBaseModel):  
    """  
    """  
    name = models.CharField(max_length=255)  
    description = models.TextField(blank=True)  
  
    def __str__(self):  
        if self.description and isinstance(self.description, str):  
            return f'{self.name}: {self.description}'  
        else:  
            return f'{self.name}'
```

**ChatGPT**  
To implement field-level permissions in Django, you can use Django's built-in permissions system along with custom logic to enforce restrictions based on the user's permission type.

Here's a general approach to implement field-level permissions for your scenario:

- Define permissions: Create permissions for your models using Django's 'Permission' model. You can do this in your Django app's 'models.py' file or through Django's admin interface.
- Check permissions: In your views or serializers, check the user's permission type to determine what data they can access.
- Implement logic: Write custom logic to enforce field-level permissions based on the user's permission type.

Here's how you can implement this in your Django project:

```
python  
from django.db import models  
from django.contrib.auth.models import AbstractUser  
from django.contrib.auth.models import Permission  
from django.contrib.contenttypes.models import ContentType  
  
class PermissionType(models.Model):  
    """  
    """  
    name = models.CharField(max_length=255)  
    description = models.TextField(blank=True)  
  
    def __str__(self):  
        return f'{self.name}'  
  
class CustomUser(AbstractUser):  
    """  
    """  
    permission_type = models.ForeignKey(PermissionType, on_delete=models.CASCADE)  
  
class YourModel(models.Model):  
    """  
    """  
    class Meta:  
        permissions = (  
            ("view_all_users", "Can view all users"),  
        )  
  
def can_view_user(self, user, target_user):  
    """  
    Check if the user has permission to view the target user.  
    """  
    if user.has_perm("your_app.view_all_users"):  
        return True  
    elif user.permission_type.name == "memberProject":  
        return True  
    elif user.permission_type.name == "memberGeneral" and user == target_user:  
        return True  
    else:  
        return False
```

In your views or serializers, you can use the 'can\_view\_user' function to check if the current user has permission to view a specific user. For example:

```
python  
from django.shortcuts import get_object_or_404  
from rest_framework import serializers  
from models import YourModel, can_view_user  
  
class YourModelSerializer(serializers.ModelSerializer):  
    """  
    """  
    model = YourModel  
    fields = ('id', 'username')  
  
class UserSerializer(serializers.ModelSerializer):  
    """  
    """  
    model = CustomUser  
    fields = ('id', 'username')  
  
class UserDetailView(APITView):  
    """  
    """  
    def get(self, request, user_id):  
        user = get_object_or_404(CustomUser, id=user_id)  
        serializer = UserSerializer(user)  
        return Response(serializer.data)  
    else:  
        return Response({"error": "You don't have permission to view this user"}, status=status.HTTP_403_FORBIDDEN)
```

In this example, the 'can\_view\_user' function checks if the user has permission to view the target user based on their permission type and the custom logic defined. Then, in the 'UserDetailView', it checks the permission before returning the user's data.

Remember to replace "your\_app" with the name of your Django app where the models are defined. Additionally, ensure that you have set up your Django project's authentication and permission settings correctly.

**User**  
To name another example, I have this Django model:

```
class Permission(AbstractBaseModel):  
    user_id = models.ManyToManyField(User, uid)  
    project_id = models.ManyToManyField(Project, uid)  
    practice_area_id = models.ManyToManyField(PracticeArea, uid)  
    permission_type_id = models.ManyToManyField(PermissionType, uid)  
    granted = models.DateField(blank=True)  
    ended = models.DateField(blank=True)  
  
    def __str__(self):  
        return f'{self.uid}'  
  
I would like to make it so some users can change the value of the field permission_type_id, but others cannot. How can that be achieved?
```

**ChatGPT**  
To implement field-level permissions in Django for the 'permission\_type\_id' field in your 'Permission' model, you can follow a similar approach as before. You'll define permissions and implement custom logic to enforce restrictions based on the user's role or permission type.

Here's how you can achieve this:

- Define permissions: Create a custom permission in Django for changing the 'permission\_type\_id' field. You can do this by defining a new permission in your Django app's 'models.py' file or through Django's admin interface.
- Implement logic: Write custom logic to check if a user has permission to change the 'permission\_type\_id' field when modifying a 'Permission' object.

Here's an example implementation:

```
python  
from django.db import models  
from django.contrib.auth.models import AbstractUser  
from django.contrib.auth.models import Permission  
from django.contrib.contenttypes.models import ContentType  
  
class PermissionType(models.Model):  
    """  
    """  
    name = models.CharField(max_length=255)  
    description = models.TextField(blank=True)  
  
    def __str__(self):  
        return f'{self.name}'  
  
class CustomUser(AbstractUser):  
    """  
    """  
    permission_type = models.ForeignKey(PermissionType, on_delete=models.CASCADE)  
  
class Permission(models.Model):  
    user_id = models.ManyToManyField(User)  
    project_id = models.ManyToManyField(Project)  
    practice_area_id = models.ManyToManyField(PracticeArea)  
    permission_type_id = models.ManyToManyField(PermissionType)  
    granted = models.DateField(blank=True)  
    ended = models.DateField(blank=True)  
  
    def can_change_permission_type(self, user):  
        """  
        Check if the user has permission to change the permission_type_id field.  
        """  
        if user.has_perm("your_app.change_permission_type"):  
            return True  
        elif user.permission_type.name == "allowed_to_change_permission_type":  
            return True  
        else:  
            return False  
  
    def save(self, *args, **kwargs):  
        if not self.pk:  
            # Here, you have permission to change permission_type_id  
            super().save(*args, **kwargs)  
        elif self.can_change_permission_type(kwargs.get('user')):  
            # User has permission to change permission_type_id  
            super().save(*args, **kwargs)  
        else:  
            raise PermissionDenied("You don't have permission to change permission_type_id")  
  
    def __str__(self):  
        return f'{self.pk}'
```

In this example:

- We've added a method 'can\_change\_permission\_type' to the 'Permission' model to check if the user has permission to change the 'permission\_type\_id' field.
- In the 'save' method of the 'Permission' model, we override the default 'save' behavior to enforce permission checks before saving changes.
- If the user doesn't have permission to change the 'permission\_type\_id' field, a 'PermissionDenied' exception is raised, preventing the save operation.

Make sure to replace "your\_app" with the name of your Django app where the models are defined. Additionally, ensure that you have set up your Django project's authentication and permission settings correctly.

## Teste Permissões Campo Modelo

```
User  
I have these Django classes:  
  
class User(PermissionsMixin, AbstractBaseUser, AbstractBaseModel):  
    """  
    Table contains cognito-users & django-users.  
  
    PermissionsMixin leverages the built-in django model permissions system  
    from django.contrib.auth.models (Permission, ContentType, Group).  
    Note: Django-admin user and app user are not split in different tables because of development.  
    Some libraries assume there is only one user model, and they can't work with both.  
    For example, to have a history log of changes for entities - to save which  
    user made a change of object attribute, perhaps, auth-related libs, and some  
    Django packages.  
    With current implementation, we don't need to fork, adapt and maintain third party packages.  
    This should work out of the box.  
    The disadvantage is django-users will have unused fields which always empty. Not critical.  
    """  
    username_validator = UnicodeUsernameValidator()  
  
    # Common fields #  
    # For cognito-users username will contain 'sub' claim from jwt token  
    # For django-users (ADMIN) for the authenticated user.  
    # For django-users it will contain username which will be used to login into django-admin site  
    username = models.CharField(max_length=255, unique=True, validators=[username_validator])  
    is_active = models.BooleanField("Active", default=True)  
  
    # Cognito-user related fields #  
    # some additional fields which will be filled-out only for users registered via Cognito  
    pass  
  
    # Django-user related fields #  
    # password is inherited from AbstractBaseUser  
    email = models.EmailField("Email address", blank=True) # allow non-unique emails  
    is_staff = models.BooleanField("Staff status", default=False,  
        help_text="Designates whether the user can log into this admin site.",  
    )  
    first_name = models.CharField(max_length=255, blank=True)  
    last_name = models.CharField(max_length=255, blank=True)  
    preferred_email = models.EmailField(blank=True)  
  
    # user status = models.ForeignKey(user_status_type, on_delete=models.PROTECT)  
    # current_practice_area = models.ManyToManyField("PracticeArea")  
    # target_practice_area = models.ManyToManyField("PracticeArea")  
    current_job_title = models.CharField(max_length=255, blank=True)  
    target_job_title = models.CharField(max_length=255, blank=True)  
    current_skills = models.CharField(max_length=255, blank=True)  
    target_skills = models.CharField(max_length=255, blank=True)  
    slack_id = models.CharField(max_length=255, blank=True)  
    phone = PhoneNumberField(blank=True)  
    texting_ok = models.BooleanField(default=True)  
    time_zone = TimezoneField(blank=True, use_pytz=False, maybe_use_america/los_angeles)  
    # conduct = models.BooleanField() # not in ERD, maybe we should remove this  
    user_manager()  
  
    USERNAME_FIELD = "username"  
    EMAIL_FIELD = "preferred_email"  
    REQUIRED_FIELDS = ["email"] # used only on createsuperuser  
  
    @property  
    def is_django_user(self):  
        return self.has_usable_password()  
  
    def __str__(self):  
        return f'{self.email}'  
  
class Project(AbstractBaseModel):  
    """  
    List of projects  
    """  
    name = models.CharField(max_length=255, unique=True)  
    description = models.CharField(max_length=255, blank=True)  
    completed_at = models.DateTimeField("Completed at", null=True, blank=True)  
    github_repo_id = models.CharField(max_length=8, blank=True,  
        help_text="Can be retrieved from gh api with the following: curl -H \\  
        \"Authorization: token [gh_PAT]\" https://api.github.com/orgs/[org]\",  
    )  
    github_primary_repo_id = models.CharField(max_length=8, blank=True,  
        help_text="Can be retrieved from gh api with the following: curl -H \\  
        \"Authorization: token [gh_PAT]\" \\  
        https://api.github.com/repos/[org]/[repo]\",  
    )  
    # current_status_id = models.ForeignKey("status", on_delete=models.PROTECT)  
    # hide = models.BooleanField(default=True)  
    # location_type = models.ForeignKey("location", on_delete=models.PROTECT)  
    # google_drive_id = models.CharField(max_length=255, blank=True)  
    # leads = models.ManyToManyField("Lead")  
    # leadership_type_id = models.ForeignKey("leadership_type", on_delete=models.PROTECT)  
    image_logo = models.URLField(blank=True)  
    image_hero = models.URLField(blank=True)  
    image_icon = models.URLField(blank=True)  
  
    def __str__(self):  
        return f'{self.name}'  
  
class PracticeArea(AbstractBaseModel):  
    """  
    Practice Area  
    """  
    name = models.CharField(max_length=255, unique=True)  
    description = models.CharField(max_length=255, blank=True)  
  
    def __str__(self):  
        return f'{self.name}'  
  
class PermissionType(AbstractBaseModel):  
    """  
    Permission Type  
    """  
    name = models.CharField(max_length=255)  
    description = models.TextField(blank=True)  
  
    def __str__(self):  
        if self.description and isinstance(self.description, str):  
            return f'{self.name}: {self.description}'  
        else:  
            return f'{self.name}'  
  
class Permission(AbstractBaseModel):  
    """  
    """  
    user_id = models.ManyToManyField(User, uid)  
    project_id = models.ManyToManyField(Project, uid)  
    practice_area_id = models.ManyToManyField(PracticeArea, uid)  
    permission_type_id = models.ManyToManyField(PermissionType, uid)  
    granted = models.DateField(blank=True)  
    ended = models.DateField(blank=True)  
  
    def __str__(self):  
        return f'{self.uid}'
```

I want to test field-level permissions on these classes. Suggest me a way to do that.

**ChatGPT**  
To test field-level permissions in Django, you can utilize the Django testing framework along with the 'django.contrib.auth' module. Here's a basic guide on how you can approach testing field-level permissions for your Django models:

- Setup Test Environment: Ensure that you have configured your Django test environment properly. This typically involves creating test databases and configuring settings for your tests.
- Create Test Cases: Write test cases to simulate different scenarios where field-level permissions need to be tested. This involves creating users with different permissions and roles and then attempting to access or modify fields based on those permissions.
- Use Django's TestCase Class: Django provides a 'TestCase' class which provides various helper methods to facilitate testing. You can subclass this class to create your test cases.
- Test Field Access: Write test methods to check whether users with specific permissions can access or modify fields as expected. For example, you can check if a user with staff privileges can modify certain fields while a regular user cannot.
- Use Django's Permission System: Django's built-in permission system allows you to define custom permissions for models. You can assign these permissions to users and groups and then check if users with those permissions can perform certain actions.
- Test CRUD Operations: Ensure that your test cases cover all CRUD (Create, Read, Update, Delete) operations on your models and that permissions are enforced correctly for each operation.

Here's a simplified example of how you might write such test cases:

```
python  
from django.test import TestCase  
from django.contrib.auth.models import User  
from myapp.models import Project  
  
class FieldPermissionsTestCase(TestCase):  
    def setUp(self):  
        self.user = User.objects.create(username='test_user', is_staff=True)  
        self.project = Project.objects.create(name='Test Project')  
  
    def test_field_access(self):  
        # Check if a staff user can modify certain fields  
        self.assert(self.user.is_staff)  
        self.user.first_name = 'John'  
        self.user.save()  
        self.assertEqual(self.user.first_name, 'John')  
  
        # Check if a regular user cannot modify certain fields  
        regular_user = User.objects.create(username='regular_user', is_staff=False)  
        self.user.first_name = 'Jane'  
        regular_user.save()  
        self.assertNotEqual(regular_user.first_name, 'Jane')  
  
    def test_permission_system(self):  
        # Check if a user with specific permission can access certain fields  
        # Here, you can create custom permissions and assign them to users or groups  
        # Then, test if users with those permissions can access specific fields  
        pass  
  
    # Additional test methods for CRUD operations...
```

Remember to run your tests using Django's test runner ('./manage.py test') to ensure they are executed against your test database. Also, make sure to test edge cases and handle exceptions gracefully in your test cases.

**User**  
I would like to know how to test the classes using pytest, as this is what my organization has already been using for tests.

I have already implemented the test data in a Python file:

```
@pytest.fixture  
def field_level_user():  
    return User.objects.create(  
        uid=1,  
        name='Admin',  
        phone='555-222-3333',  
        is_staff=True,  
        preferred_email='admin@something908.com'  
    )  
    User.objects.create(  
        uid=2,  
        name='Sarah',  
        phone='555-235-8989',  
        texting_ok=True,  
        preferred_email='sarah@something908.com'  
    )  
    User.objects.create(  
        uid=3,  
        name='Bob',  
        phone='555-456-7890',  
        texting_ok=False,  
        preferred_email='bob@something908.com'  
    )  
    User.objects.create(  
        uid=4,  
        name='Alice',  
        phone='555-765-4321',  
        texting_ok=True,  
        preferred_email='alice@something908.com'  
    )  
    User.objects.create(  
        uid=5,  
        name='Joe',  
        phone='555-888-6666',  
        texting_ok=False,  
        preferred_email='joe@something908.com'  
    )  
    User.objects.create(  
        uid=6,  
        name='Snoop',  
        phone='555-555-5555',  
        texting_ok=False,  
        preferred_email='snoop@something908.com'  
    )  
    User.objects.create(  
        uid=7,  
        name='Ralph',  
        phone='555-999-8888',  
        preferred_email='ralph@something908.com'  
    )  
    User.objects.create(  
        uid=8,  
        name='Claire',  
        phone='555-555-6666',  
        permission_type_admin=True,  
        preferred_email='claire@something908.com'  
    )  
    User.objects.create(  
        uid=9,  
        name='Mary',  
        phone='555-555-2222',  
        texting_ok=False,  
        preferred_email='mary@something908.com'  
    )
```

@pytest.fixture  
def field\_level\_permission():  
 return PermissionType.objects.create(  
 uid=1,  
 name='AdminProject'  
 )  
 PermissionType.objects.create(  
 uid=2,  
 name='AdminProject'  
 )  
 PermissionType.objects.create(  
 uid=3,  
 name='PracticeLeadProject'  
 )  
 PermissionType.objects.create(  
 uid=4,  
 name='MemberProject'  
 )  
 PermissionType.objects.create(  
 uid=5,  
 name='MemberGeneral'  
 )

@pytest.fixture  
def field\_level\_practice\_area():  
 return PracticeArea.objects.create(  
 uid=1,  
 name='Admin'  
 )  
 PracticeArea.objects.create(  
 uid=2,  
 name='Research'  
 )  
 PracticeArea.objects.create(  
 uid=3,  
 name='Design'  
 )

@pytest.fixture  
def field\_level\_project():  
 return Project.objects.create(  
 uid=1,  
 name='Website'  
 )  
 Project.objects.create(  
 uid=2,  
 name='People depot'  
 )

How can I test the classes with pytest using this data?

**ChatGPT**  
To test the classes with pytest using the provided test data fixtures, you can write test functions in your pytest test files. You can create separate test functions for each test case, leveraging the test data provided by the fixtures. Here's how you can structure your pytest test file:

```
python  
# test_permissions.py  
import pytest  
from myapp.models import User, Permission, PermissionType, PracticeArea, Project
```

@pytest.mark.django\_db  
def test\_admin\_brigade\_permissions(field\_level\_user, field\_level\_permission):  
 # Create test data  
 users = field\_level\_user  
 assert len(users) == 9  
 # Write assertions to test user fields here  
 # Example:  
 assert users[0].name\_first == "Admin"  
 assert users[1].name\_first == "Sarah"  
 # Add more assertions for other user fields

@pytest.mark.django\_db  
def test\_permission\_fields(field\_level\_permission):  
 permissions = field\_level\_permission  
 assert len(permissions) == 12  
 # Write assertions to test permission fields here  
 # Example:  
 assert permissions[0].user\_id == 1  
 assert permissions[1].user\_id == 2  
 # Add more assertions for other permission fields

@pytest.mark.django\_db  
def test\_permission\_type\_fields(field\_level\_permission\_type):  
 permission\_types = field\_level\_permission\_type  
 assert len(permission\_types) == 6  
 # Write assertions to test permission type fields here  
 # Example:  
 assert permission\_types[0].name == "adminbrigade"  
 assert permission\_types[1].name == "adminProject"  
 # Add more assertions for other permission type fields

@pytest.mark.django\_db  
def test\_practice\_area\_fields(field\_level\_practice\_area):  
 practice\_areas = field\_level\_practice\_area  
 assert len(practice\_areas) == 4  
 # Write assertions to test practice area fields here  
 # Example:  
 assert practice\_areas[0].name == "Admin"  
 assert practice\_areas[1].name == "Rm"  
 # Add more assertions for other practice area fields

@pytest.mark.django\_db  
def test\_project\_fields(field\_level\_project):  
 projects = field\_level\_project  
 assert len(projects) == 2  
 # Write assertions to test project fields here  
 # Example:  
 assert projects[0].name == "Website"  
 assert projects[1].name == "people depot"  
 # Add more assertions for other project fields

In this file, each test function is marked with '@pytest.mark.django\_db' decorator to ensure that the tests run within a Django transaction and have access to the database.

You can run these tests using pytest by executing the following command in your terminal:

```
bash  
pytest path/to/test_permissions.py
```

Replace 'path/to/test\_permissions.py' with the actual path to your pytest test file. This will execute all the test functions defined in the file and report any failures or errors encountered during the tests.

**User**  
I would like to specifically test each permission\_type in the data (adminbrigade, adminProject, practiceLeadProject, memberProject, and memberGeneral). Each permission\_type will have their own field-level visibility settings. How can I do that?

**ChatGPT**  
To test each 'PermissionType' and its corresponding field-level visibility settings, you can create separate test functions for each permission type. In these functions, you can assert that the expected fields are visible or not visible for users with permissions of that type. Here's how you can do it:

```
python  
# test_permission_types.py  
import pytest  
from myapp.models import User, Permission, PermissionType, PracticeArea, Project
```

@pytest.mark.django\_db  
def test\_admin\_brigade\_permissions(field\_level\_user, field\_level\_permission\_type):  
 # Create test data  
 users = field\_level\_user  
 permission\_type\_admin\_brigade = PermissionType.objects.get(name="adminbrigade")  
 # Assert that adminbrigade permissions allow access to specific fields  
 assert permission\_type\_admin\_brigade.permissions.filter(name="adminbrigade").exists()  
 # Add assertions for field-level visibility settings for adminbrigade permissions

@pytest.mark.django\_db  
def test\_admin\_project\_permissions(field\_level\_user, field\_level\_permission\_type):  
 # Create test data  
 users = field\_level\_user  
 permission\_type\_admin\_project = PermissionType.objects.get(name="adminProject")  
 # Assert that adminProject permissions allow access to specific fields  
 assert permission\_type\_admin\_project.permissions.filter(name="adminProject").exists()  
 # Add assertions for field-level visibility settings for adminProject permissions

@pytest.mark.django\_db  
def test\_practice\_lead\_permissions(field\_level\_user, field\_level\_permission\_type):  
 # Create test data  
 users = field\_level\_user  
 permission\_type\_practice\_lead = PermissionType.objects.get(name="practiceLeadProject")  
 # Assert that practiceLeadProject permissions allow access to specific fields  
 assert permission\_type\_practice\_lead.permissions.filter(name="practiceLeadProject").exists()  
 # Add assertions for field-level visibility settings for practiceLeadProject permissions

@pytest.mark.django\_db  
def test\_member\_project\_permissions(field\_level\_user, field\_level\_permission\_type):  
 # Create test data  
 users = field\_level\_user  
 permission\_type\_member\_project = PermissionType.objects.get(name="memberProject")  
 # Assert that memberProject permissions allow access to specific fields  
 assert permission\_type\_member\_project.permissions.filter(name="memberProject").exists()  
 # Add assertions for field-level visibility settings for memberProject permissions

@pytest.mark.django\_db  
def test\_member\_general\_permissions(field\_level\_user, field\_level\_permission\_type):  
 # Create test data  
 users = field\_level\_user  
 permission\_type\_member\_general = PermissionType.objects.get(name="memberGeneral")  
 # Assert that memberGeneral permissions allow access to specific fields  
 assert permission\_type\_member\_general.permissions.filter(name="memberGeneral").exists()  
 # Add assertions for field-level visibility settings for memberGeneral permissions

In each test function, you can retrieve the relevant 'PermissionType' object from the test data fixtures and then assert its name and field-level visibility settings according to your requirements.

Remember to replace the placeholder comments with actual assertions based on your field-level visibility settings for each permission type.

You can run these tests using pytest in the same way as before:

```
bash  
pytest path/to/test_permissions.py
```

This will execute all the test functions defined in the file and report any failures or errors encountered during the tests.