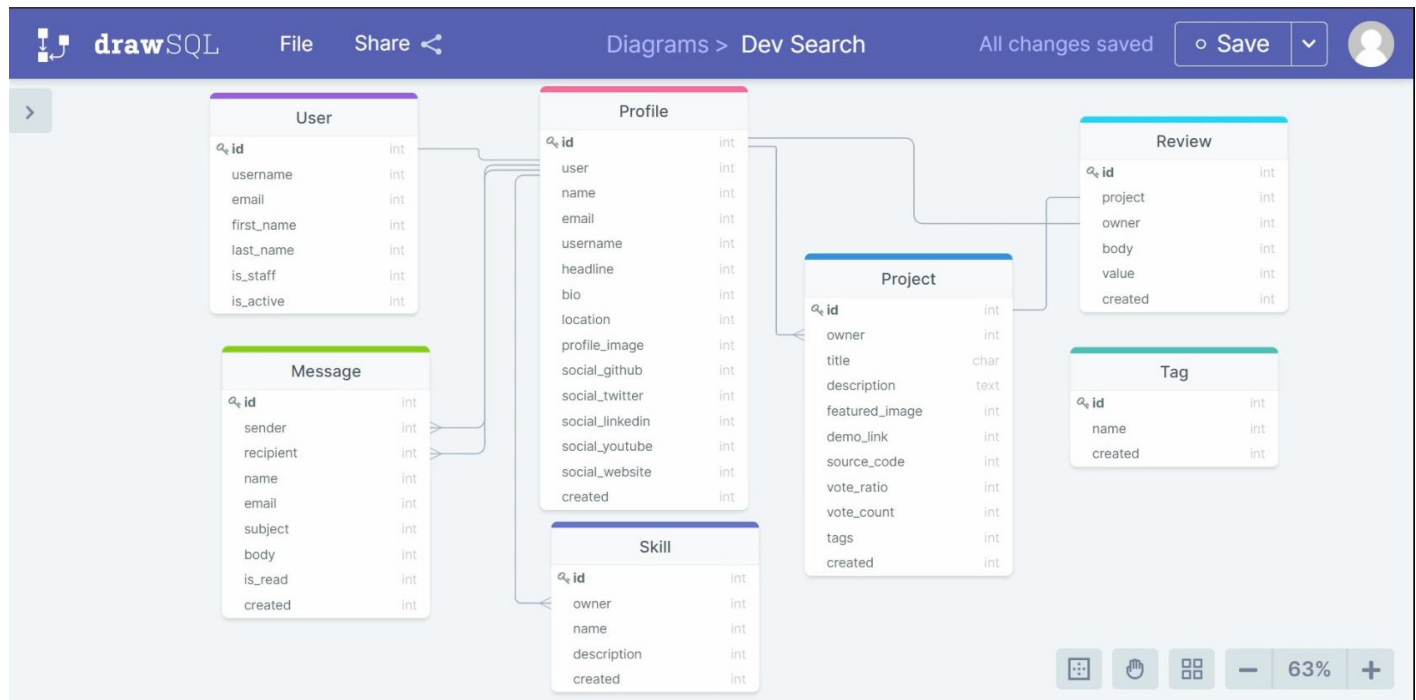


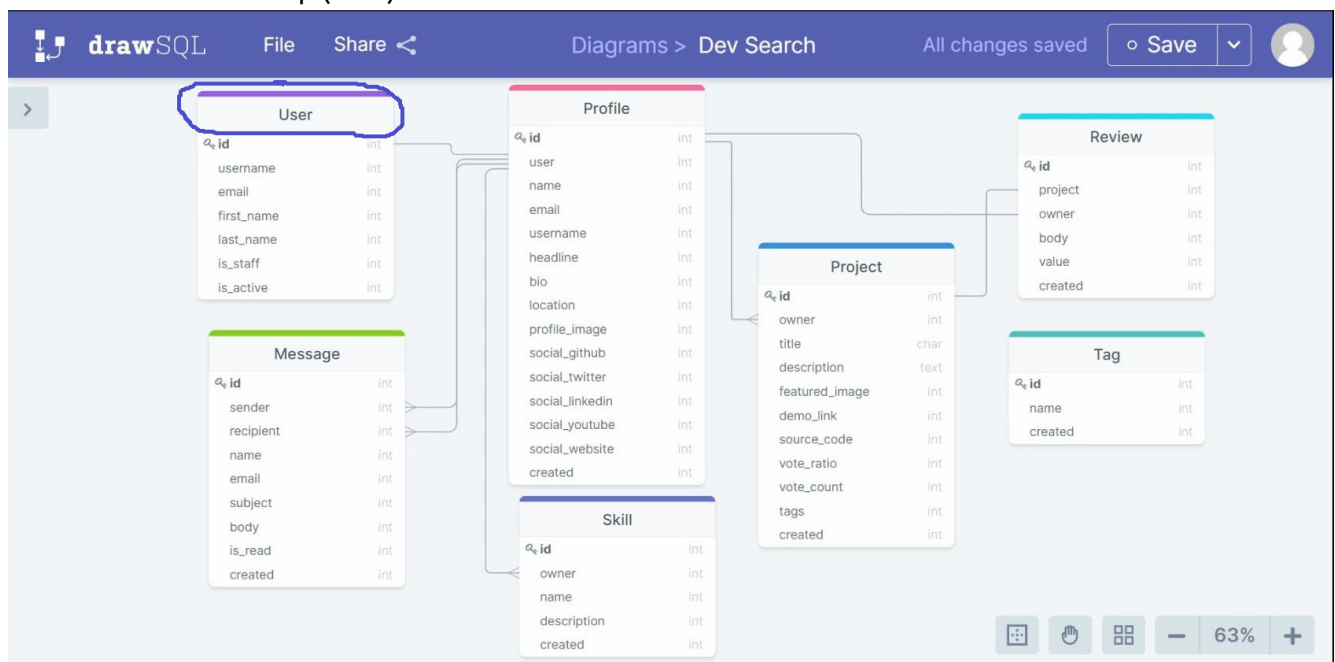
Database Relationships

The following is the basic pre- design of our project database:



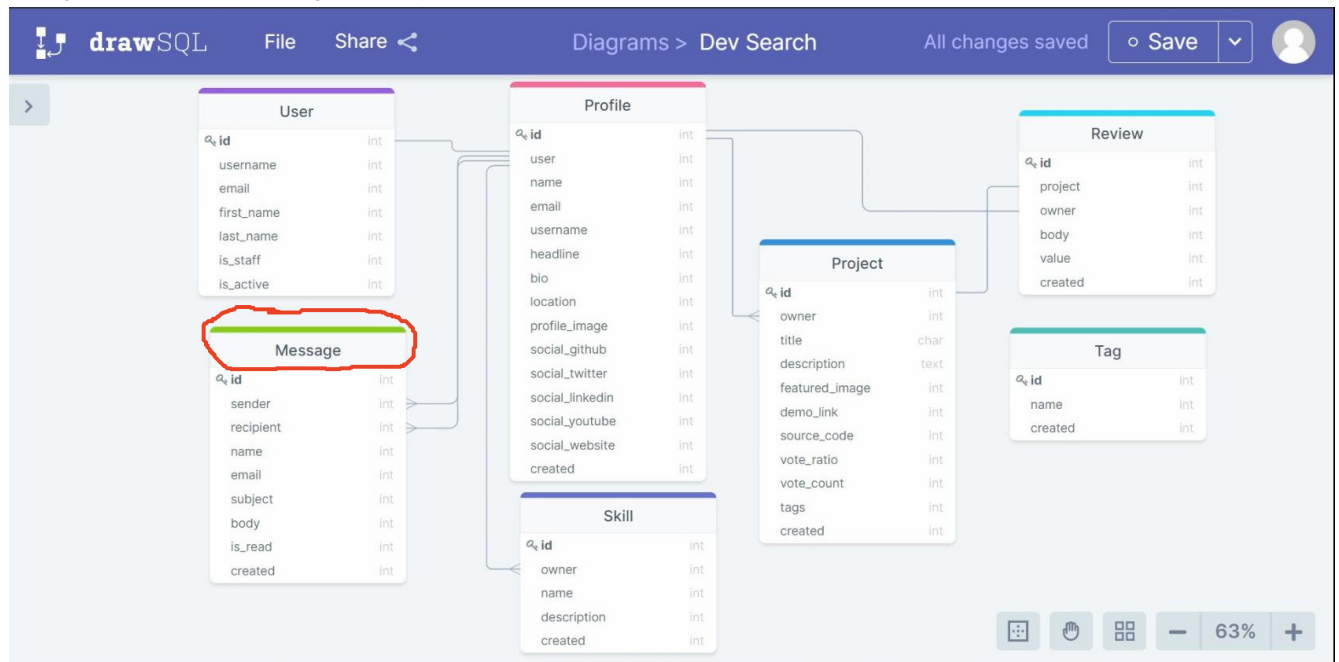
The diagram shows the three basic types of relationships:

1. One to One relationship (Blue)



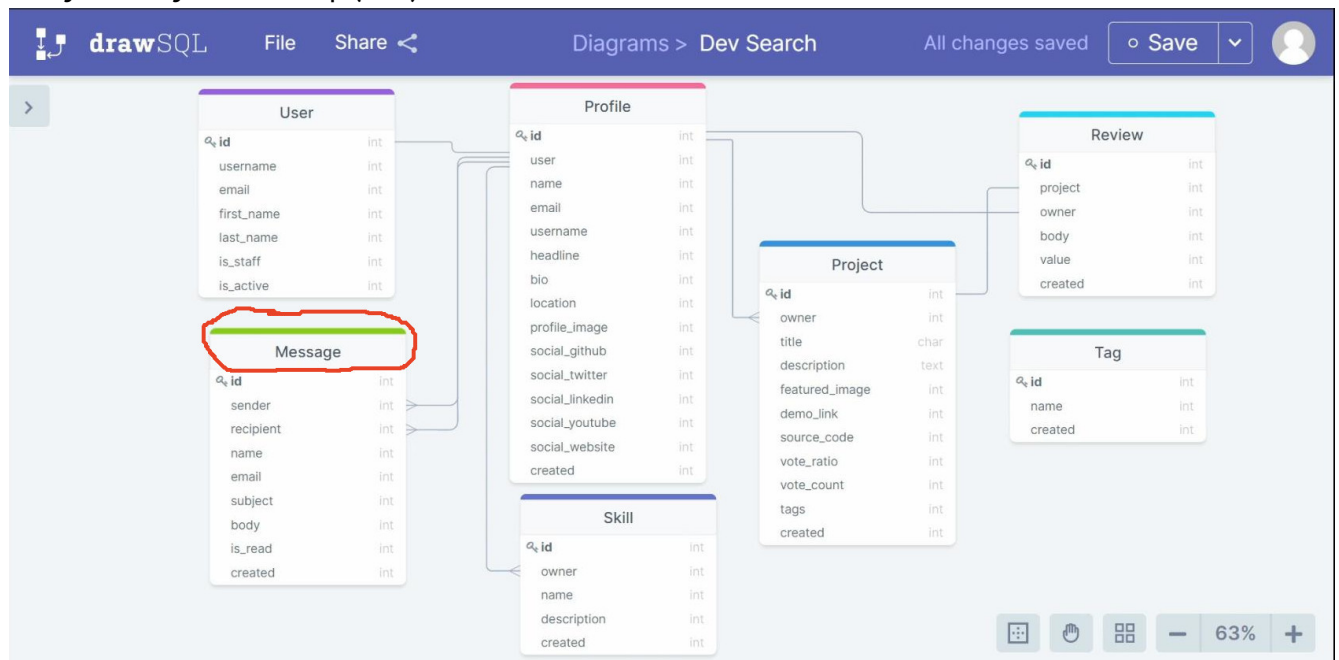
If you follow the diagram, you'll realize that a single user can only have a profile. That's the reason the relationship between the user and profile tables is one-to-one relationship.

2. Many to One relationship (Green)



In the diagram, an example of many-to-one relationship is the relationship between the profile and project tables. As a single profile can be related to multiple projects.

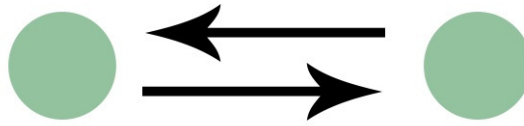
3. Many to Many relationship (Red)



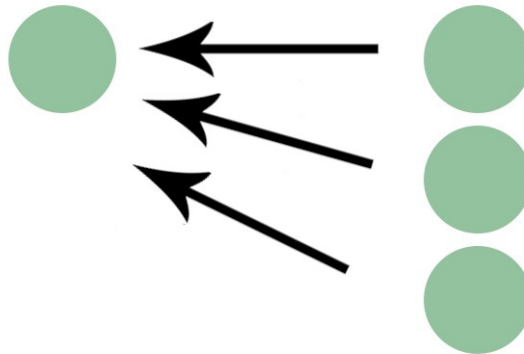
Here, the example is the relationship between the profile and message tables. A message is related to multiple profiles via the sender and receiver attributes.

Relationships

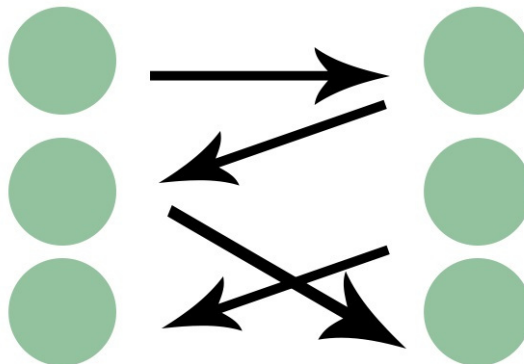
One-to-One - One table record can relate to one record in another table.



Many-to-One - One table record can relate to many records in another table.

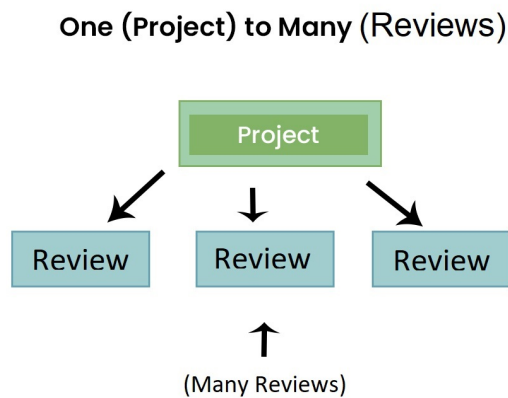


Many-to-Many - This relationship occurs when multiple records in a table are associated with multiple records in another table.



The following diagrams relate how multiple relationship tables looks like on the database:

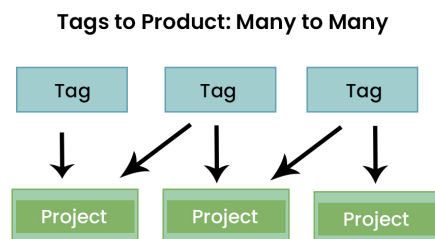
- Many-to-one relationship



ID	TITLE	DESC.
1
2

ID	Parent_ID	BODY
1	2	...
2	2	...
3	1	...
4	2	...
5	1	...

- Many-to-many relationship



ID	TITLE	DESC.
1
2

Proj_ID	Tag_ID
1	2
1	2
2	1
1	2
2	1

ID	TITLE	DESC.
1
2

In both cases, the relationship dependency is introduced via a **foreign_key** (a reference to the parent table's **primary_key**)

In the case of many-to-one relationship there's only a single **foreign_key** in the table that is considered as the "many" part of the relationship. The single key table is considered as the "**parent_table**."

Many-to-many relationships the relationships are handled by a **THIRD** or a **RELATIONSHIP** table that contains two foreign keys (one for each table that's been related).

This is How Everything Looks in Code

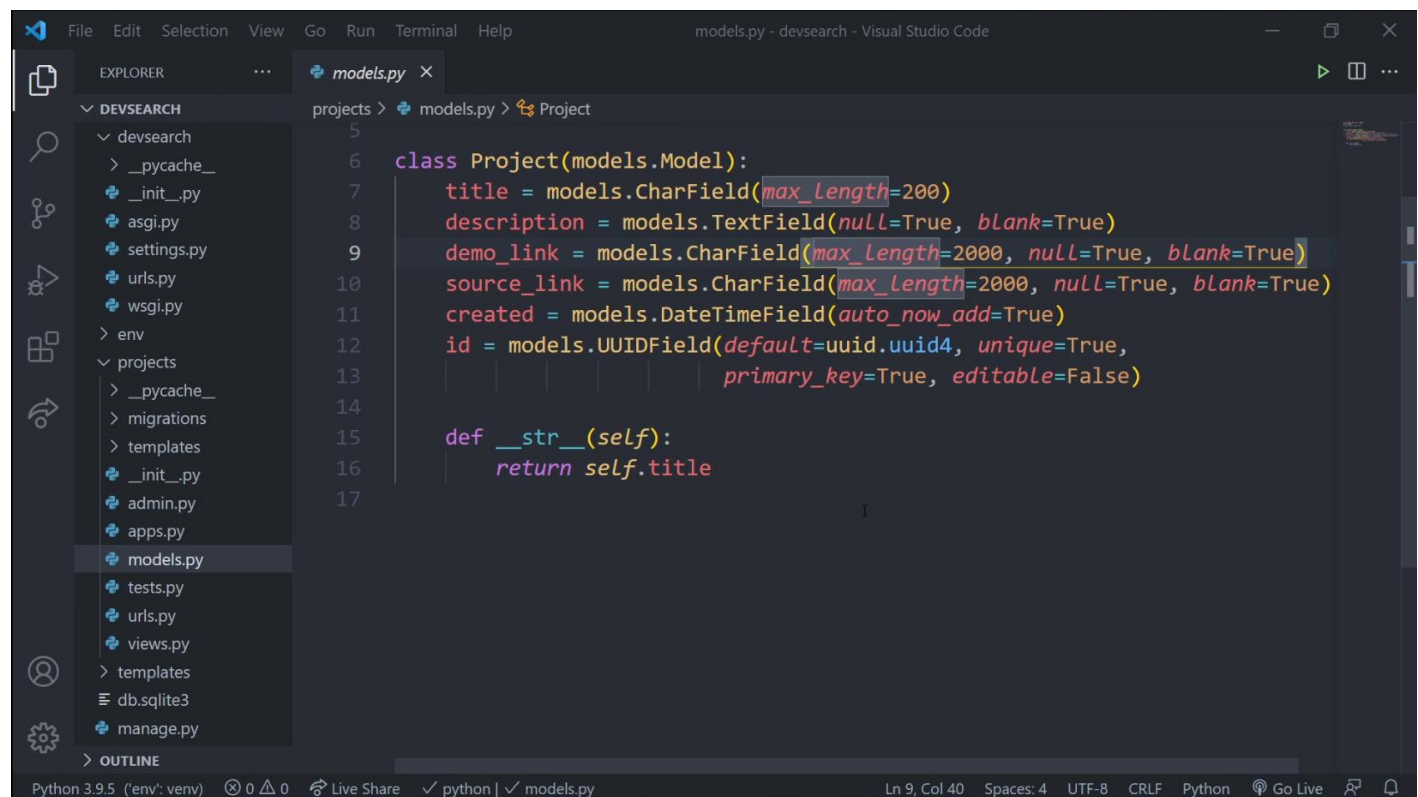
Objective

Modify the projects/models.py in order to create two new models and modify the Project model to include the new relationships.

Specifically, we're going to create the Review table to have a many-to-one relationship with the Projects table. And, we also are going to create a new Tags table that should have a many-to-many relationship with the Projects table. We'll need to insert a RelationshipField in the Projects table. This RelationshipField creates a relationship table for us automatically and we don't see it directly in the programming model.

Coding the changes

1. Open up the projects/models.py file



```
5
6 class Project(models.Model):
7     title = models.CharField(max_length=200)
8     description = models.TextField(null=True, blank=True)
9     demo_link = models.CharField(max_length=2000, null=True, blank=True)
10    source_link = models.CharField(max_length=2000, null=True, blank=True)
11    created = models.DateTimeField(auto_now_add=True)
12    id = models.UUIDField(default=uuid.uuid4, unique=True,
13                           primary_key=True, editable=False)
14
15    def __str__(self):
16        return self.title
17
```

2. Add the Review model to the file

```

15  def __str__(self):
16      return self.title
17
18
19  class Review(models.Model):
20      VOTE_TYPE = (
21          ('up', 'Up Vote'),
22          ('down', 'Down Vote'),
23      )
24      # owner =
25      project = models.ForeignKey(Project, on_delete=models.SET_NULL)
26      body = models.TextField(null=True, blank=True)
27      value = models.CharField(max_length=200, choices=VOTE_TYPE)
28      created = models.DateTimeField(auto_now_add=True)
29      id = models.UUIDField(default=uuid.uuid4, unique=True,
30                           primary_key=True, editable=False)
31

```

Important: If you set the parameter `on_delete=models.SET_NULL`, the records of Review related to the particular record in Project table, won't be deleted when the related record is deleted from the Project table. The effect is to set the value of the foreign key to NULL, effectively breaking the relationship to the record but preserving the record in the Review table.

```

15  def __str__(self):
16      return self.title
17
18
19  class Review(models.Model):
20      VOTE_TYPE = (
21          ('up', 'Up Vote'),
22          ('down', 'Down Vote'),
23      )
24      # owner =
25      project = models.ForeignKey(Project, on_delete=models.CASCADE)
26      body = models.TextField(null=True, blank=True)
27      value = models.CharField(max_length=200, choices=VOTE_TYPE)
28      created = models.DateTimeField(auto_now_add=True)
29      id = models.UUIDField(default=uuid.uuid4, unique=True,
30                           primary_key=True, editable=False)
31
32  def __str__(self):
33      return self.value
34

```


On the other hand, if you set `on_delete=models.CASCADE`, then the records related to the particular record in Project table will be also deleted when the record in Project is deleted. This is what *CASCADE* means.

For the time being, either choice is good for the project and you can elect which one you prefer.

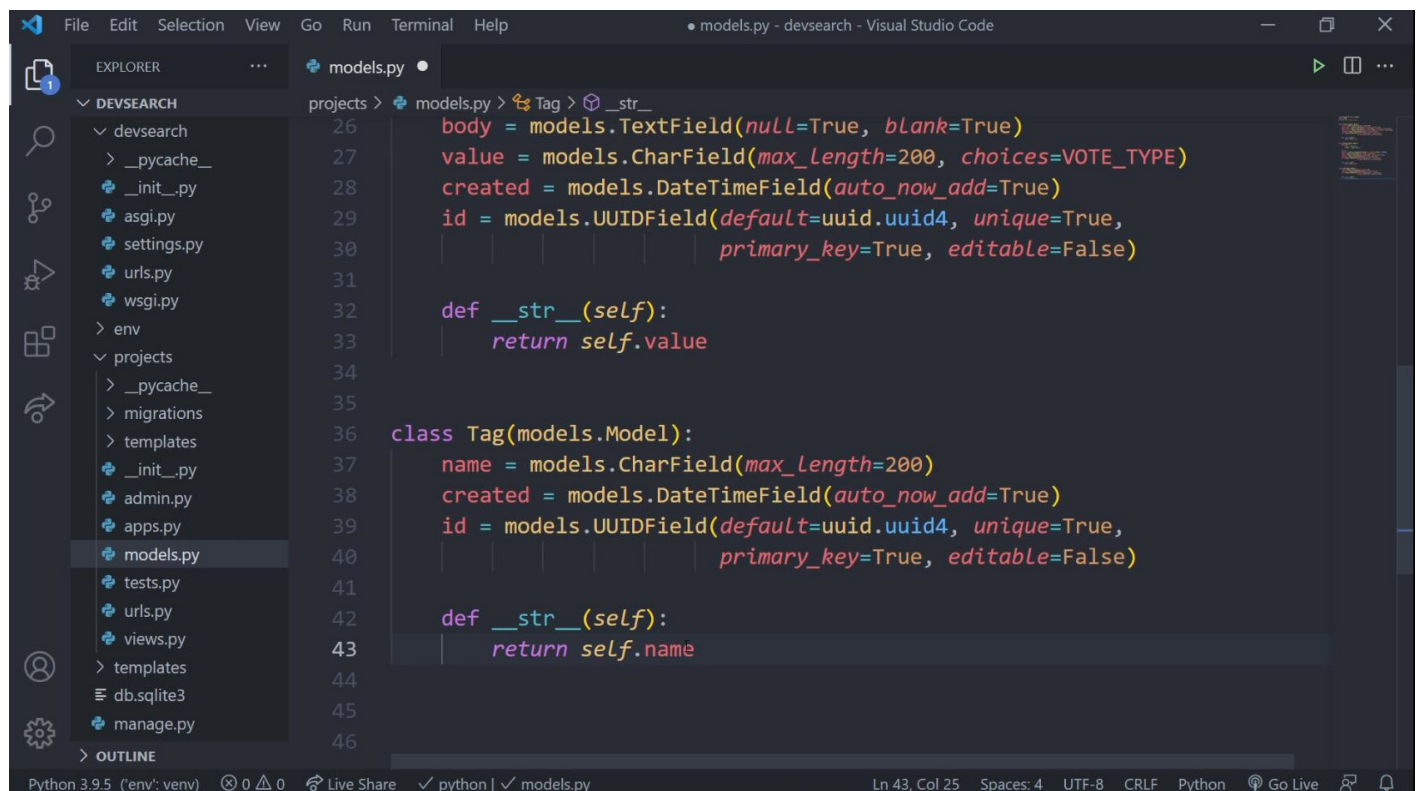
Several points to cover here:

```
VOTE_TYPE = (
    ('up', 'Up Vote'),
    ('down', 'Down Vote'),
)
```

is a tuple to show a list of choices in the admin-panel and the future input forms

```
#owner =          # TODO: This is a relationship to the Profile table that
we don't have for the moment
```

Next let's create the Tag model which a many-to-many relationship with the Project model



```
models.py
26 body = models.TextField(null=True, blank=True)
27 value = models.CharField(max_length=200, choices=VOTE_TYPE)
28 created = models.DateTimeField(auto_now_add=True)
29 id = models.UUIDField(default=uuid.uuid4, unique=True,
30                       primary_key=True, editable=False)
31
32 def __str__(self):
33     return self.value
34
35
36 class Tag(models.Model):
37     name = models.CharField(max_length=200)
38     created = models.DateTimeField(auto_now_add=True)
39     id = models.UUIDField(default=uuid.uuid4, unique=True,
40                           primary_key=True, editable=False)
41
42     def __str__(self):
43         return self.name
44
45
46
```

Now, let's create the many-to-many relationship with Project. In order to do that, we only need to add a relationship field in the Project class:

```

5
6 class Project(models.Model):
7     title = models.CharField(max_length=200)
8     description = models.TextField(null=True, blank=True)
9     demo_link = models.CharField(max_length=2000, null=True, blank=True)
10    source_link = models.CharField(max_length=2000, null=True, blank=True)
11    tags = models.ManyToManyField('Tag', blank=True)
12    vote_total = models.IntegerField(default=0, null=True, blank=True)
13    vote_ratio = models.IntegerField(default=0, null=True, blank=True)
14    created = models.DateTimeField(auto_now_add=True)
15    id = models.UUIDField(default=uuid.uuid4, unique=True,
16                          primary_key=True, editable=False)
17
18    def __str__(self):
19        return self.title
20
21
22 class Review(models.Model):
23     VOTE_TYPE = (
24         ('up', 'Up Vote'),
25         ('down', 'Down Vote'),

```

Notes:

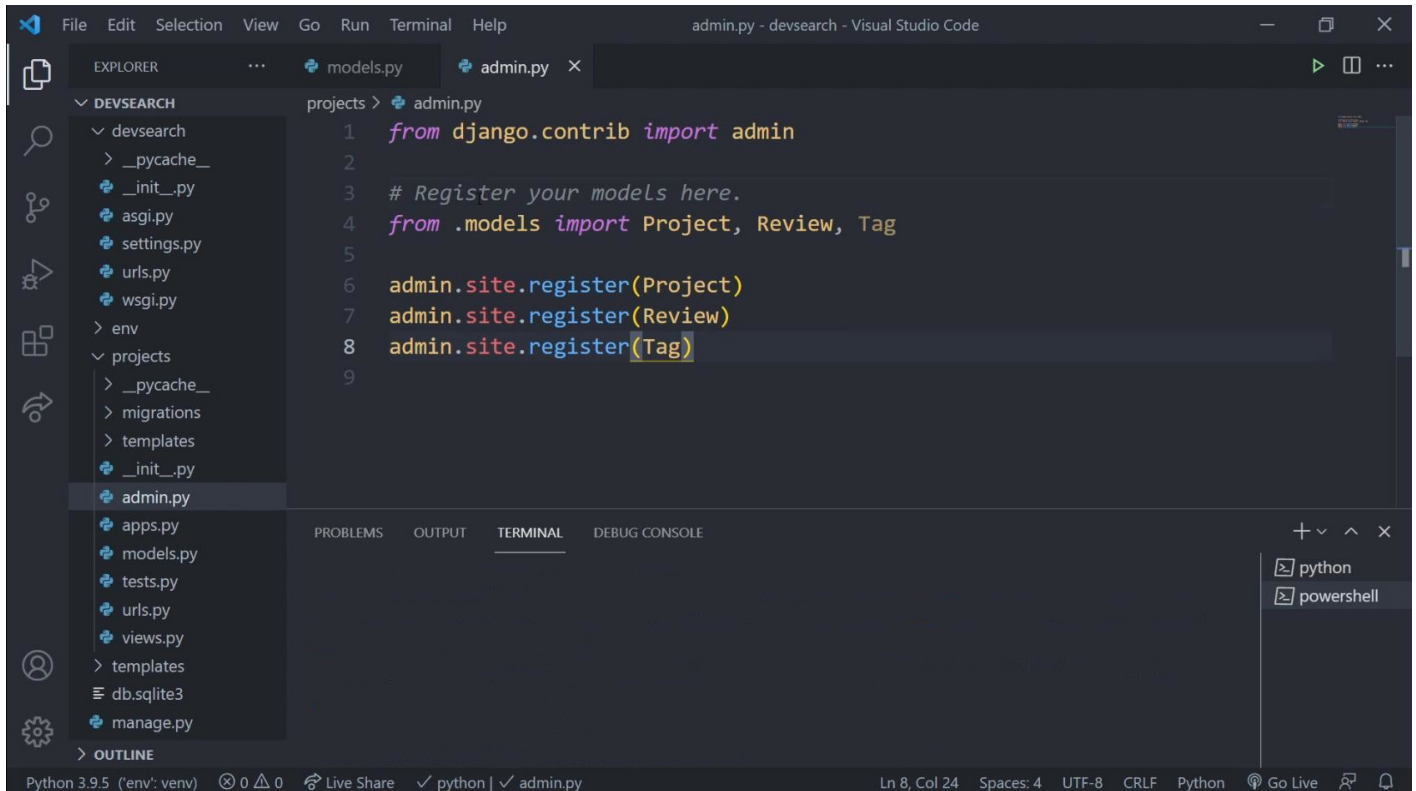
1. You can see that the tags field is a *models.ManyToManyField*.
2. There's a declaration 'Tag' in quotes. The reason is that the class Tag is declared after the Project class. This what's known as 'belayed' declaration. It's a mechanism to indicate to the interpreter that the class is declared after. We can avoid using this if we move the class Tag declaration before the class Project.
3. The parameter *blank=True* allows us to create the record from a form without actually filling the form as a required parameter.
4. We added two attributes for the voting systems: *vote_total* and *vote_ratio*. Both are Integers and they keep track of the up and down votes for a specific project.

Migrate

Run the migration commands:

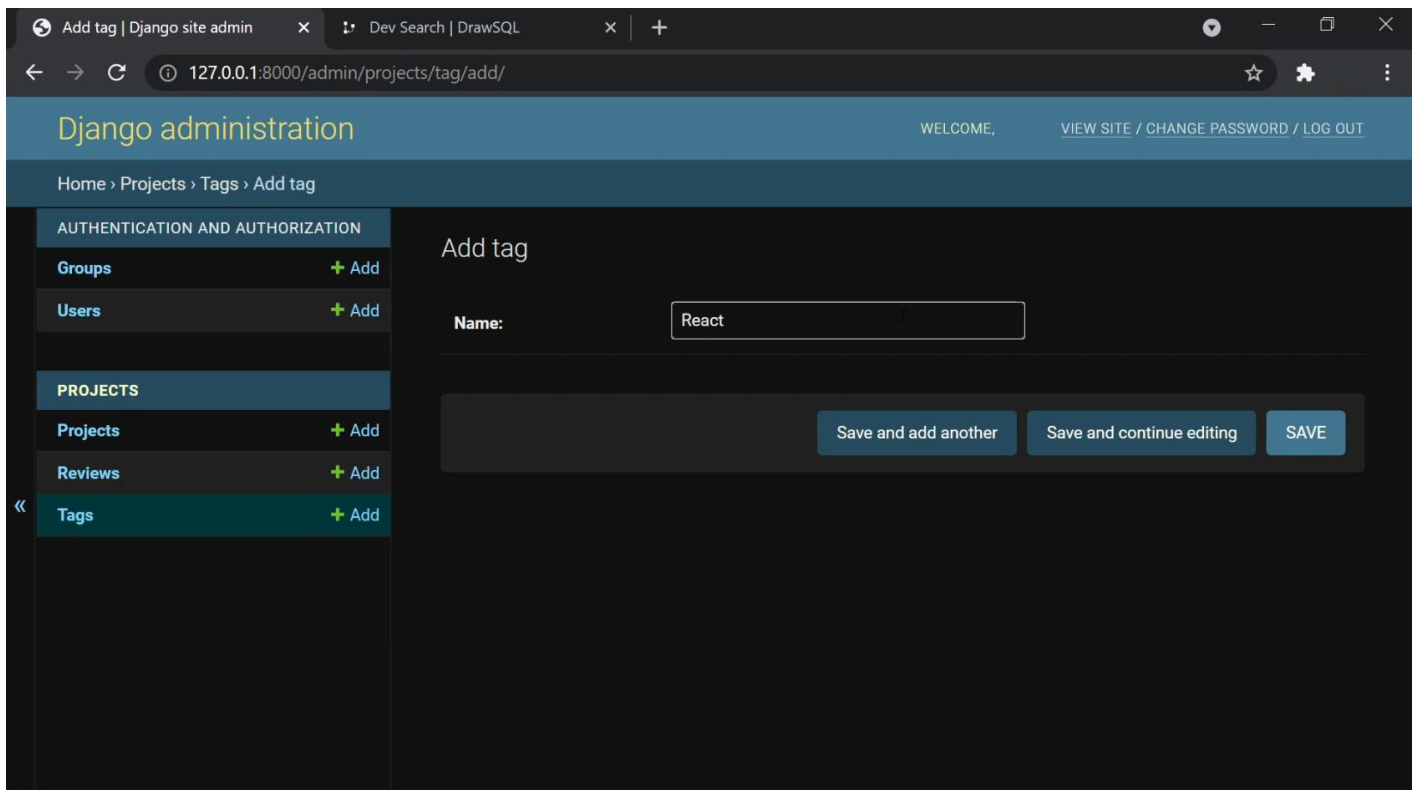
```
python manage.py makemigrations
python manage.py migrate
```

Don't forget to add the new models into the admin-panel.

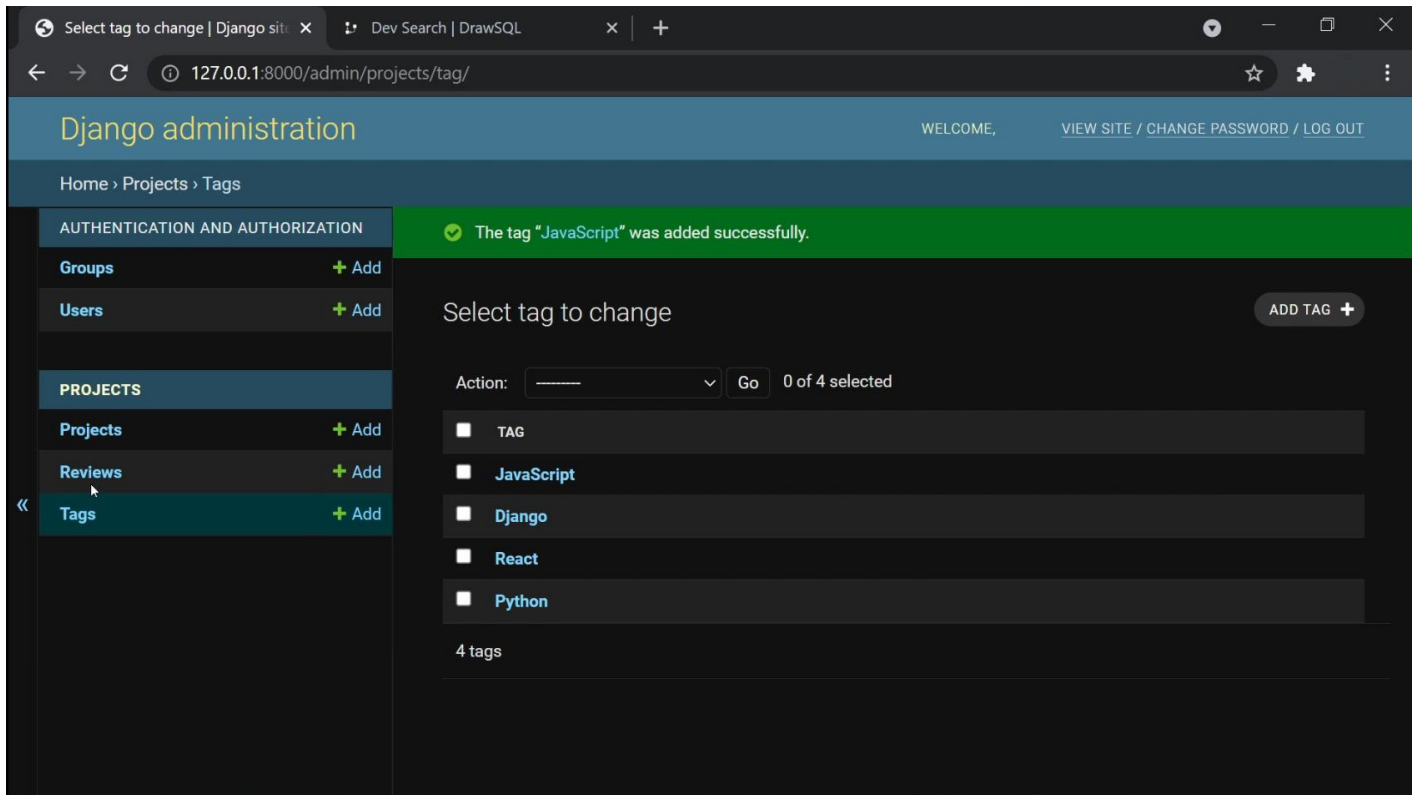


```
1 from django.contrib import admin
2
3 # Register your models here.
4 from .models import Project, Review, Tag
5
6 admin.site.register(Project)
7 admin.site.register(Review)
8 admin.site.register(Tag)
9
```

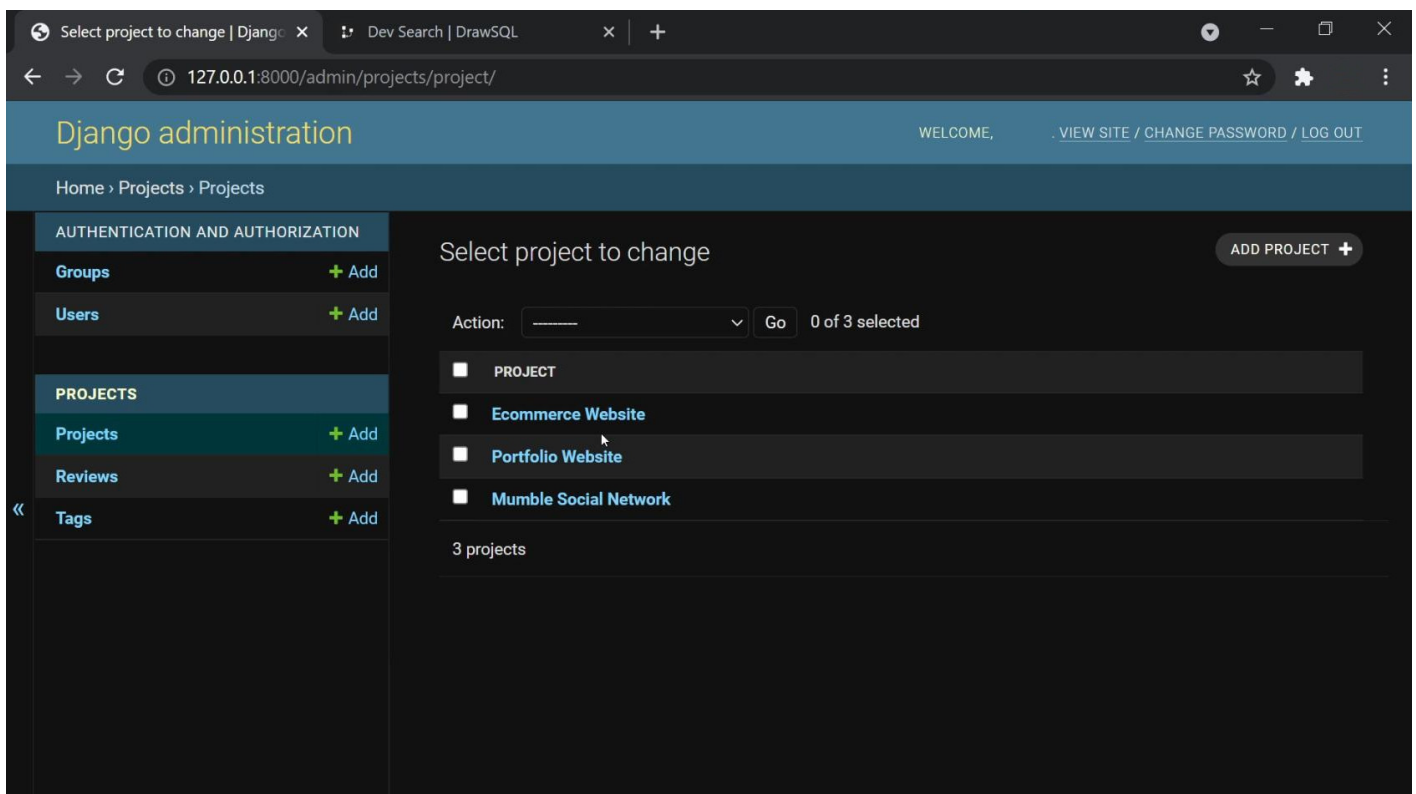
And if you open your admin-panel at 127.0.0.1:8000/admin you should be able to see the modifications to the database:



add few tags: React, Python, JavaScript, Django



Open up the Projects tab:



and add some tags to any of the projects in the table:

127.0.0.1:8000/admin/projects/project/b7074c3e-c42b-4527-92e3-c3e0eb40d750/change/

AUTHENTICATION AND AUTHORIZATION

- Groups [+ Add](#)
- Users [+ Add](#)

PROJECTS

- Projects [+ Add](#)
- Reviews [+ Add](#)
- Tags [+ Add](#)

Demo link:

Source link:

Tags:

React
Django
Python
JavaScript

[+](#)

Hold down "Control", or "Command" on a Mac, to select more than one.

Vote total:

Vote ratio:

[Delete](#) [Save and add another](#) [Save and continue editing](#) [SAVE](#)

For multiple selection use Ctrl+<Left_Mouse_Click> and save your selection.

This shows the many-to-many relationship functionality in Django ORM.

For a many-to-one relationship, open up the Reviews tab and fill up a form for a single project:

127.0.0.1:8000/admin/projects/review/add/

Django administration WELCOME, [VIEW SITE / CHANGE PASSWORD / LOG OUT](#)

Home > Projects > Reviews > Add review

AUTHENTICATION AND AUTHORIZATION

- Groups [+ Add](#)
- Users [+ Add](#)

PROJECTS

- Projects [+ Add](#)
- Reviews [+ Add](#)
- Tags [+ Add](#)

Add review

Project: [+](#)

Body:

Value:

[Save and add another](#) [Save and continue editing](#) [SAVE](#)

and test the functionality.

