

Learning Objectives

- Identify limitations of not using generic relationships
- Dynamically access models using the contenttypes framework
- Create generic relationships between different models
- Use `GenericRelation` for reverse queries
- Create the `Comment` model with a generic relationship to the `Post` model

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the blango repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a blango directory appear in the file tree.

You are now ready for the next assignment.

Introduction

Introduction

Django has a powerful system for relating different models together. By using `ForeignKey`, `ManyToManyField` and `OneToOneField` fields, we can model many different ways of how models relate to each other. In Blango so far, we've used a `ForeignKey` field on the `Post` model to set the author of each `Post`. We also used a `ManyToManyField` to assign `Tags` to `Posts`.

One of the limitations of this system is that relationships between models are statically defined, and can only exist for a single model per field: `Tags` can only be mapped to `Posts` (or vice versa).

Django comes with the *contenttypes* framework, which provides a high level way of accessing referring to models in a project. It also allows for *generic relationships*, a way of mapping objects together without statically defining it to a single other model.

We'll get into generic relationships soon, but let's start with a look at the main features the *contenttypes* framework provides.

contenttypes

contenttypes

The main use of the contenttypes framework is to dynamically load models classes based on their name, and subsequently, query objects for that model.

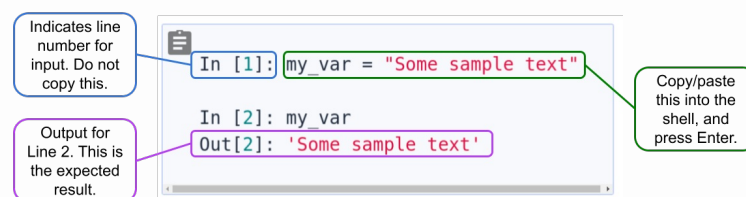
The main model is ContentType, which is importable from `django.contrib.contenttypes.model`. This works like a normal model whose objects you can query. If you know the model you want, you can query it using the app name and model name.

Let's see how to load the Post ContentType from the blog app. First start a Django Python shell:

```
python3 manage.py shell
```

Entering Shell Commands

When copying/pasting shell commands into the terminal, it is important to not copy the entire code block. This will cause errors. You should only copy lines of code that start with `In [#]:`. This indicates a line of input. Copy everything after the `:`. Lines of code that start with `Out[#]:` are the expected output.



understanding input and output for the shell

Then import ContentType and query for the object.

```
In [1]: from django.contrib.contenttypes.models import
        ContentType

In [2]: post_type = ContentType.objects.get(app_label="blog",
        model="post")

In [3]: post_type
Out[3]: <ContentType: blog | post>
```

We could also get a list of all the ContentType objects we have installed, by calling `ContentType.objects.all()`:

```
In [4]: ContentType.objects.all()
Out[4]:
<QuerySet [<ContentType: admin | log entry>, <ContentType: auth
| permission>, <ContentType: auth | group>,
<ContentType: auth | user>, <ContentType: contenttypes |
content type>, <ContentType: sessions | session>,
<ContentType: blog | post>, <ContentType: blog | tag>]>
```

Note that the ContentType object is *not* the Post model. But, we can retrieve the class with the `model_class()` method:

```
In [5]: post_type.model_class()
Out[5]: <class 'blog.models.Post'>
```

It's also possible to go in reverse, and retrieve the ContentType object from the model. This is done with the `ContentType.objects.get_for_model()` method.

```
In [6]: from blog.models import Post

In [7]: ContentType.objects.get_for_model(Post)
Out[7]: <ContentType: blog | post>
```

This is useful if you want to know the `app_label` and `model` name for a model class.

Once a ContentType object is found, the shortcut method `get_objects_for_this_type()` will perform a get lookup and retrieve objects for that model class.

```
In [8]: post_type.get_object_for_this_type(pk=1)
Out[8]: <Post: An Example Post>
```

Note that this is a shortcut to get on the `Post.objects` manager instance:

```
In [9]: post_type.model_class().objects.get(pk=1)
Out[9]: <Post: An Example Post>

In [10]: post_type.get_object_for_this_type(pk=1) ==
          post_type.model_class().objects.get(pk=1)
Out[10]: True
```

Other methods of loading objects (like `filter()` and `all()`) are similarly available – remember once you call `model_class()` it's just like you've imported the model.

This is the extent of what is generally needed to be known to use contenttypes, but if you're curious about the extra methods that are available, then you can read the [contenttypes framework documentation](#).

Now that we've seen how the contenttypes framework can let us dynamically access models, let's get back to discussing generic relationships.

Generic Relationships

Generic Relationships

To explain the importance of generic relationships, let's use a concrete example. In a blog you might want someone to be able to leave comments. We could accomplish this with a `Comment` model with a `ForeignKey` to a particular `Post`. The model might look something like this:

```
class Comment(models.Model):
    creator = models.ForeignKey(settings.AUTH_USER_MODEL,
                               on_delete=models.CASCADE)
    content = models.TextField()
    post = models.ForeignKey(Post, on_delete=models.CASCADE)
```

It would be great though, if as well as being able to comment on a `Post`, you could also comment on an author.

This is not so easy to do. One way could be to add another `ForeignKey` on `Comment` that points to the author (`User`) being reviewed. The `Comment` model could be updated to be something like this:

```
class Comment(models.Model):
    creator = models.ForeignKey(settings.AUTH_USER_MODEL,
                               on_delete=models.CASCADE)
    content = models.TextField()
    post = models.ForeignKey(Post, on_delete=models.CASCADE,
                             null=True)
    author = models.ForeignKey(settings.AUTH_USER_MODEL,
                              on_delete=models.CASCADE, null=True)
```

But this would end up being confusing:

- Both `post` and `author` fields can be `null` now, which means that the database will allow us to insert a `Comment` that's not mapped to any object (they could both be `null`). Before, when we only mapped to `Posts`, the `post` field did not allow `null` and so the database would enforce consistency.
- Likewise, `post` and `author` could both be populated. This would mean the `Comment` applies to both, which might not make sense.
- We'd need extra code to check and query the right field when fetching the `Comments`, based on which context we are in (`Post` or `Author`).
- If we ended up adding some other model, and wanted to allow comments on it, we'd need yet another field to store this information.

info

Django Permissions

The Django permissions system uses the *contenttype* framework, as it provides a generic way to set permissions on any model that has been added to the project. You can imagine how complex it would be to set up permissions without this kind of generic system.

By utilizing ContentType we can allow a model to be related to any number of models by just adding three attributes to a Model:

- A ForeignKey field that points to a ContentType. Normally this is called `content_type`
- A PositiveIntegerField that stores the primary key of the related object. Normally this is called `object_id`
- A GenericForeignKey field, a special type of field that will look up the object from the other two new fields.

Let's look at how to implement this on the Comment model (declare Comment before Post). Something like:

```
from django.contrib.contenttypes.fields import GenericForeignKey
from django.contrib.contenttypes.models import ContentType

class Comment(models.Model):
    creator = models.ForeignKey(settings.AUTH_USER_MODEL,
                               on_delete=models.CASCADE)
    content = models.TextField()
    content_type = models.ForeignKey(ContentType,
                                    on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey("content_type",
                                      "object_id")
```

▼ Migrations

Don't forget to run migrations before continuing.

```
python3 manage.py makemigrations
python3 manage.py migrate
```


Note that we set `content_object` with two arguments:

`GenericForeignKey("content_type", "object_id")`. These are the names of the fields on the model that contain the `ContentType` and related object's ID, respectively. If these fields are called `content_type` and `object_id` then the arguments can be omitted (that is, in our case, using it like `content_object = GenericForeignKey()` would behave exactly the same).

Storing a value in a `GenericForeignKey` field is similar to a normal `ForeignKey`: just assign the object to it. The `GenericForeignKey` field takes care of storing the right `ContentType` and object PK into the `content_type` and `object_id` fields.

Begin by starting a Django management Python shell:

```
python3 manage.py shell
```

You can make a `Comment` on a `Post` like this:

```
In [1]: from blog.models import Post, Comment

In [2]: from django.contrib.auth.models import User

In [3]: p = Post.objects.first()

In [4]: u = User.objects.first()

In [5]: c1 = Comment(creator=u, content="What a great post!",
                    content_object=p)

In [6]: c1.save()

In [7]: c1.content_object
Out[7]: <Post: An Example Post>
```

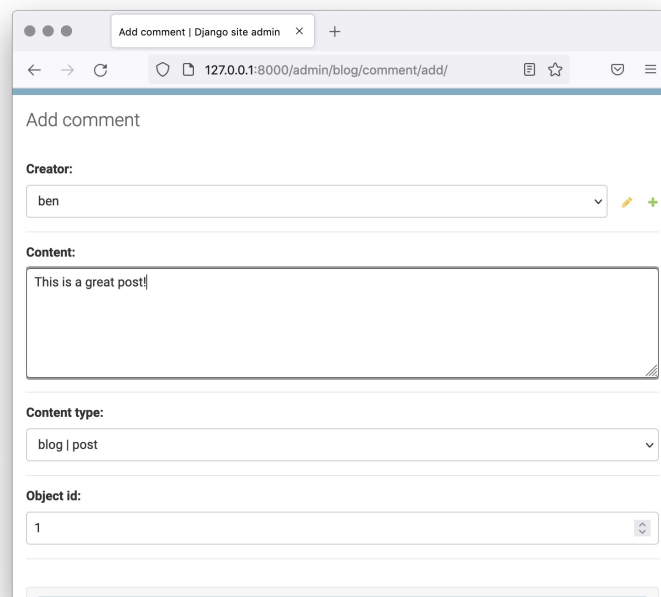
And similarly, we can add a comment on a `User`.

```
In [8]: c2 = Comment(creator=u, content="I like myself!",
                    content_object=u)

In [9]: c2.content_object
Out[9]: <User: ben>
```

Note that this is a bit of strange comment, since there's currently only one user in the system! But, you can see the same method of setting and getting the `content_object` applies regardless of the model class on the other side.

Generic relationships can also be added using the Django admin, but it is not very friendly as you need to enter the PK of the related object manually.

A screenshot of a web browser window showing the Django admin interface for adding a comment. The browser's address bar shows the URL '127.0.0.1:8000/admin/blog/comment/add/'. The form is titled 'Add comment' and contains several fields: 'Creator' with a dropdown menu showing 'ben', 'Content' with a large text area containing 'This is a great post!', 'Content type' with a dropdown menu showing 'blog | post', and 'Object id' with a dropdown menu showing '1'. The form is styled with a clean, modern design using light gray and white colors.

Generic Relationships Admin Panel

Now that we know how to create generic related objects, let's see how to fetch them, with reverse generic relationships.

Reverse Generic Relationships

Reverse generic relationships

A disadvantage of the `GenericForeignKey` is that it can't be queried against, so if you try something like this, you'll get an exception:

```
>>> c = Comment.objects.filter(content_object=p)
# stack trace removed
django.core.exceptions.FieldError: Field 'content_object' does
    not generate an automatic reverse relation and therefore
    cannot be used for reverse querying. If it is a
    GenericForeignKey, consider adding a GenericRelation.
```

Begin by starting a Django management Python shell:

```
python3 manage.py shell
```

We need to `filter()` against the `content_type` and `object_id` field directly. For example:

```
In [1]: from django.contrib.contenttypes.models import
        ContentType

In [2]: from blog.models import Post, Comment

In [3]: post_type = ContentType.objects.get_for_model(Post)

In [4]: p = Post.objects.first()

In [5]: c = Comment.objects.filter(content_type=post_type,
        object_id=p.pk)

In [6]: c
Out[6]: <QuerySet [<Comment: Comment object (1)>, <Comment:
        Comment object (2)>]>
```

This can be a bit tedious, but we know it will work for all models. If there's a model whose generic related objects you'll be querying quite often, you can consider setting up a `GenericRelation` field (imported from `django.contrib.contenttypes.fields`) on it. `GenericRelation` takes one argument: the generic model class to map to.

Our `Post` model could be updated like so:

```

from django.contrib.contenttypes.fields import GenericRelation
# other imports omitted

class Post(models.Model):
    # existing fields omitted
    comments = GenericRelation(Comment)

```

This is only possible if it's a model under your control, so we can use it to make fetching comments for a Post simple, but we can't edit the User model so easily.

Now fetching the Comments for a Post is simple. The comments attribute acts like a RelatedManager allowing you to query, filter(), add(), create() and remove() Comments to a Post. Let's see a short example of how to use it.

▼ Migrations

Don't forget to run migrations before continuing. First exit the shell with exit().

```

python3 manage.py makemigrations
python3 manage.py migrate

```

Once again in a Django Python shell, first let's get all the Comment objects for a Post:

```

In [1]: p = Post.objects.first()

In [2]: p.comments.all()
Out[2]: <QuerySet [<Comment: Comment object (1)>, <Comment:
          Comment object (2)>]>

```

Then, let's remove the first Comment from the Post:

```

In [3]: c1 = p.comments.all()[0]

In [4]: p.comments.remove(c1)

In [5]: p.comments.all()
Out[5]: <QuerySet [<Comment: Comment object (2)>]>

```

That's most of what you need to know for how to use the contenttypes framework. The [full documentation](#) is useful if you want to know about how to use custom queries in GenericRelation fields, or how to aggregate generic objects.

Let's return to our project and add the Comment model.

Blango Project Updates

Blango Project Updates

We've discussed the theory behind generic relationships and seen some examples of how to use them. Now it's time to add comments to Blango. Do this by:

- Creating a Comment model. It should have a creator field to store the user who created it, plus fields to store the created time and modified time. Don't forget the generic relationship fields.

```
class Comment(models.Model):
    creator = models.ForeignKey(settings.AUTH_USER_MODEL,
                               on_delete=models.CASCADE)
    content = models.TextField()
    content_type = models.ForeignKey(ContentType,
                                     on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey("content_type",
                                      "object_id")
    created_at = models.DateTimeField(auto_now_add=True)
    modified_at = models.DateTimeField(auto_now=True)
```

- Add a comments GenericRelation field on Post back to Comment. This will make it easier to find comments for a post.

```
class Post(models.Model):
    author = models.ForeignKey(settings.AUTH_USER_MODEL,
                              on_delete=models.PROTECT)
    created_at = models.DateTimeField(auto_now_add=True)
    modified_at = models.DateTimeField(auto_now=True)
    published_at = models.DateTimeField(blank=True, null=True)
    title = models.TextField(max_length=100)
    slug = models.SlugField()
    summary = models.TextField(max_length=500)
    content = models.TextField()
    tags = models.ManyToManyField(Tag, related_name="posts")
    comments = GenericRelation(Comment)

    def __str__(self):
        return self.title
```

- Add the Comment model to the Django admin by importing and

registering it in the `admin.py` file.

```
from django.contrib import admin
from blog.models import Tag, Post, Comment

class PostAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("title",)}

# Register your models here.
admin.site.register(Tag)
admin.site.register(Post, PostAdmin)
admin.site.register(Comment)
```

- Don't forget to run the makemigrations and migrate management commands after all this.

```
python3 manage.py makemigrations
python3 manage.py migrate
```

info

Warning Message

When you try to migrate the model changes, Django gives you a warning message.

```
You are trying to add the field 'created_at' with
'auto_now_add=True' to comment without a default;
the database needs something to populate existing
rows.
```

```
1) Provide a one-off default now (will be set on all
existing rows)
```

```
2) Quit, and let me add a default in models.py
```

```
Select an option:
```

This happens because the `Comment` model was first created without `created_at` and `modified_at`. The comments in the database do not have these fields. Django wants to know how to treat `created_at` since it needs a value. Enter 1 at the prompt. Django suggests using `timezone.now` as the value. Press Enter.

```
Please enter the default value now, as valid Python
You can accept the default 'timezone.now' by pressing
'Enter' or you can provide another value.
```

```
The datetime and django.utils.timezone modules are
available, so you can do e.g. timezone.now
```

```
Type 'exit' to exit this prompt
```

```
[default: timezone.now] >>>
```


Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .  
git commit -m "Finish Introduction and Django Admin"
```

- Push to GitHub:

```
git push
```