Siddharth Singha Roy
25 December 2019

Django

Web-development framework for perfectionists

Models

Blank is used for frontend validation. **Null** is entirely database related. If null is true, then it allows null values to be stored in the database. If blank is true, that field will be required in the frontend.

Choices is used for restricting user's option. [Select html tag]. It is a sequence of 2-tuples. The first element of the tuple is the value that will be stored in the database. The second one is the name that will be displayed in the form.

Say we want to display the choice selected. We don't want to show them the machine-readable form. To get the second element that was selected ie. the one which was displayed in the form,

p.get_gender_display()   [get_FOO_display()]

The **default** value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.

help_text is used to display a help text when displayed as a form.

primary_key is used to set a field as primary. Overrides Django default primary key (id)

Unique fields must be unique throughout the table.

Django by default gives this field if no primary key is specified.

```
id = models.AutoField(primary_key=True)
```

Each field can have a **verbose_name** which can help developers to understand what that field does. It can also be applied to a model.

To specify the verbose_name for any field (other than ManyToMany, ForeignKey, OneToOneField) the first argument is the verbose_name.

Name = models.CharField("Name of employee", …..)

For the other three fields, we use

verbose_name = "Name Here",

We do this since the first argument for these three fields are the model to which it is related.

The convention is not to capitalise the first letter of the verbose_name. Django will automatically capitalise the first letter where it needs to.

To define a many to many field we use the ManyToMany field.

It doesn't matter which model has the ManyToManyField, but you should only put it in one of the models – not both.

Generally, *ManyToManyField* instances should go in the object that's going to be edited on a form.

OneToOneField : This is most useful on the primary key of an object when that object "extends" another object in some way.

We can even import models from another app. This is perfectly normal and also encouraged.

A field cannot be a Django keyword. It also cannot contain 2 consecutive underscores. It also cannot end with an underscore.

Many to One Relation is represented by a Foreign Key.

a Manufacturer makes multiple cars but each Car only has one Manufacturer

Then the foreign key is added to the field of Car model.

We can even write custom fields.

Model metadata is “anything that’s not a field”, such as ordering options (*ordering*), database table name (*db_table*), or human-readable singular and plural names (*verbose_name* and *verbose_name_plural*). None are required ,and adding class Meta to a model is completely optional.

Managers : It’s the interface through which database query operations are provided to Django models and is used to *retrieve the instances* from the database. If no custom Manager is defined, the default name is *objects*. Managers are only accessible via model classes, not the model instances.

Managers allow us to retrieve the data stored in the database. If no model manager is provided the default one is objects. Model managers can only be accessed via the model class and not model instances.

So we can create a new model manager that has all the methods of objects manager and then add some custom methods.

FIELD TYPES

Boolean Field : By default is none. A true/false field.

Char Field : Used to store small to large sized text. CharField requires one extra argument ie. the max_length (in terms of characters).

DateField : Used to store date instance. It can take a few additional arguments :

1. auto_now : Automatically set the field to now every time the object is saved or created. This is called when the .save() method is invoked. Useful for creating time stamps.
2. auto_now_add : This is similar to auto_now however it enforces that only that time is used when the object instance was created. As currently implemented, setting auto_now or auto_now_add to True will cause the field to have

   editable=False and blank=True set.

DateTimeField : Used to store an instance of python's datetime.datetime . Takes the same arguments as DateField.

DurationField : Used to store periods of time. Can be compared to Arithmetic values only for POSTGRE SQL or SQL.

DecimalField : Used to store decimal values upto a fixed precision. Requires two extra arguments

1. max_digits : Number of digits allowed including the decimal values.
2. decimal_places : Number of decimal values that needs to be stored.

EmailField : Used to store email_addresses. Also validates email address.

FileField : Used to upload and store files. It can take two additional arguments

1. upload_to : This attribute provides a way of setting the upload directory and file name, and can be set in two ways.

```
# file will be uploaded to MEDIA_ROOT/uploads
upload = models.FileField(upload_to='uploads/')
```

The MEDIA_ROOT is specified in the django settings.py file/

2. storage : Responsible for how Django deals with retrieval and storage.

Always validate the type of file you are storing. It may lead of CSRF attacks if not dealt with.

When we try to access a FileField on a model, we are given an instance of **FieldFile** as a proxy for accessing the underlying file. FieldFile provides us APIs along with the in-build Python file handling API.

FloatField : Similar to decimal field, however has less precision.

ImageField : Inherits all the properties of the FileField, however provides an extra validation to check whether the uploaded file is an image or not. Also can take two additional arguments :

1. height_field : Creates a new model field with the image height
2. width_fileld : Creates a new model field with the image width

It has a default max_length of 100, however can be overridden by providing the max. It requires the **Pillow** library.

IntegerField : Used to store an integer.

The default form widget for this field is a *NumberInput* when *localize* is False or *TextInput* otherwise.

GenericIPAddressField : Used to store IP Addresses, both IPv4 and IPv6. It accepts two additional arguments one of which is the **protocol** which states which type of Ip address will be stored. By default it is both. Other options are IPv4 and IPv6.

PositiveIntegerField, PositiveSmallIntegerField

SlugField : *Slug* is a newspaper term. A slug is a short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs.

TextField : Used to store large texts. Although we can specify the max_length it is not enforced at database level.

TimeField : Used to store an instance of python's datetime.time. Accepts similar arguments of auto_now and auto_now_add.

URLField : Used to store urls. Have a maximum_length of 200 but can be overridden.

Relationship Fields

Abstract Base Classes

Say we want to have some common fields for many models. Instead of adding that to each model we can create an abstract base class and inherit it into our model classes instead of Django.db.models.Model When a model class is declared as abstract then a table is not created for that model and also it cannot be instantiated.

```
from django.db import models

class CommonInfo(models.Model):
        name = models.CharField(max_length=100)

        age = models.PositiveIntegerField()


        class Meta: abstract = True


class Student(CommonInfo):
        home_group = models.CharField(max_length=5)
```

Student model has access to three fields name, age and home_group.

Fields inherited from abstract base classes can be overridden with another field or value, or be removed with None.

Model Managers

A model manager is used to run database queries. It is the way with which our database can interact with our model (database). Django requires each model to have one model manager. By default, the model manager is objects, If we want to rename it we can do it easily

```
class Person(models.Model): #...

people = models.Manager()
```

Now we can use Person.people.all().

We can also create custom model managers by extending the base model class and then instantiating the new model manager in our model class.

ForeignKey

A **many-to-one relationship** . For example we have a car and a manufacturers. A car can have only one manufacturer whereas manufacturer can have many cars. Use foreign key in car model.

Requires two arguments

1. The class to which the model is related to
2. on_delete option

We can also create a recursive relation to the same model by passing the related model name as 'self'.

If the model is yet not defined then we can passed name of the model instead of model instance ie.

'Manufacturer' is passed and not Manufacturer

Say we have an abstract model which is kept in some other app. The abstract model has a ForeignKey to a class not defined there. Suppose the abstract model is subclassed in another app which has the ForeignKey model defined. Django will not raise an error and subclass.foreignclass will refer to the Foreign Key model class.

Relationships defined this way on *abstract models* are resolved when the model is subclassed as a concrete model and are not relative to the abstract model's app_label

'production.Manufacturer'. This sort of reference is known as lazy relationship. useful when resolving circular import dependencies between two applications.

Arguments

1. **on_delete** : The values for this can be found in django.db.models

    1.1. CASCADE : Cascade delete. If the model to which the foreign key is referencing is deleted then also delete the current model in which foreign key is present.

    1.2. PROTECT : Prevents deletion of the model to which the foreign key is referencing to.

    1.3. SET_NULL : Set the foreign key null if and only if null is true for that foreign key field.

    1.4. SET_DEFAULT : Set the foreign key to the default value if and only if default value was specified.

    1.5. SET() and DO_NOTHING()

2. **related_name :** The related name to use for the relation between the related model and this model instance. This also serves as the default name for the related_query_name. If we do not want to have the backward relationship set related_name to '+' or end with '+'.

3. **related_query_name :** The name used for reverse filter name from the target model.

# Declare the ForeignKey with related_query_name

```
class Tag(models.Model):
article = models.ForeignKey(

Article, on_delete=models.CASCADE, related_name="tags", related_query_name="tag",

)
name = models.CharField(max_length=255)

# That's now the name of the reverse filter
Article.objects.filter(tag__name="important")

Article.tags.all()
```

Say we have many news reporters and many articles. Each article can only have 1 news reporter whereas reporter can write multiple articles.

So the foreign key field will be in the article model.

related_names="articles", related_query_name = "article"

Say we want to find all the reporters who wrote articles with title Sex

Articles.objects.filter(article__title="Sex")

Now if we want to find all the articles written by a reported

Articles.articles.all()

So related_query_name is used for filtering whereas related_name returns all the related instances.

ManyToMany Field

Many to Many Relation. Has one compulsory argument : the model to which it is related to. Allows recursive relationship via "self".

Also supports related_name and related_query_name.

Symmetrical : Used when we have recursive relation. Setting this true means that

if you are my friend, then I am your friend.

OneToOne Field

A one-to-one relationship. Conceptually, this is similar to a *ForeignKey* with *unique=True*, but the "reverse" side of the relation will directly return a single object.

This is most useful as the primary key of a model which "extends" another model in some way.

Related Objects Reference

They will be accessed via model instances.

```
class Article(models.Model):
reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)
```

- In the above example, the methods below will be available on the manager reporter.article_set.

```
class Pizza(models.Model):
toppings = models.ManyToManyField(Topping)
```

- In this example, the methods below will be available both on topping.pizza_set and on pizza. toppings.

So from the model to which it is related to, we can access the other model properties using FOO_set if not specified in related_name.

The below code snippets work on model instance and not model class.

To add a model instance to the related objects.

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.add(e) # Associates Entry e with Blog b.
```

This allows entry from the other side of the foreign or m2m relation.

If we have a ForeignKey relationship then update() is performed in the background.

Using Many to Many Relationship in this, no save() will be called, it will simply just establish the relation.

Using add on a relation that already exists does not do anything.

To remove a relation

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.remove(e) # Disassociates Entry e from Blog b.
```

Remove doesn't work for foreignKey unless null=True in the fk field.

We can use create method to create a new instance and then directly create a relationship. Adds nothing to add except saving 1 line of code where we create the new model instance.

If we want to remove all the relationships then we can use clear.

Just like remove(), clear() is only available on *ForeignKey*s where null=True.

Meta Field

abstract

Options.**abstract**

If abstract = True, this model will be an *abstract base class*.

Cannot be instantiated and no table is produced for this.

app_label

Options.**app_label**

If a model is defined outside of an application in *INSTALLED_APPS*, it must declare which app it belongs to:

```
app_label = 'myapp'
```

get_latest_by

Options.**get_latest_by**
The name of a field or a list of field names in the model, typically *DateField*, *DateTimeField*, or *IntegerField*. This specifies the default field(s) to use in your model *Manager*'s *latest()* and *earliest()* methods.

```
# Latest by ascending order_date.

get_latest_by = "order_date"

# Latest by priority descending, order_date ascending.

get_latest_by = ['-priority', 'order_date']
```

order_with_respect_to

Options.**order_with_respect_to**
Makes this object order-able with respect to the given field, usually a ForeignKey. This can be used to make related objects order-able with respect to a parent object.

```
class Answer(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE) # ...

    class Meta:
        order_with_respect_to = 'question'
```

ordering

Options.**ordering**
The default ordering for the object, for use when obtaining lists of objects:

```
    ordering = ['-order_date']
```

This is a tuple or list of strings and/or query expressions. Each string is a field name with an optional "-" prefix, which indicates descending order. Fields without a leading "-" will be ordered ascending. Use the string "?" to order randomly.

To order by pub_date descending, then by author ascending, use this:

```
    ordering = ['-pub_date', 'author']
```

verbose_name

Options.**verbose_name**
A human-readable name for the object, singular:

```
verbose_name = “pizza"
```

If this isn’t given, Django will use a munged version of the classname: CamelCase becomes camel case.

verbose_name_plural

Options.**verbose_name_plural** The plural name for the object:

```
verbose_name_plural = "stories"
```

If this isn’t given, Django will use *verbose_name* + "s".

Creating Objects

1. Add a class-method on the model class:

```
from django.db import models class Book(models.Model):

title = models.CharField(max_length=100)

@classmethod
def create(cls, title):

book = cls(title=title)
# do something with the book

return book

book = Book.create("Pride and Prejudice")
```

2. Adding a custom manager

```
class BookManager(models.Manager):

    def create_book(self, title):

        book = self.create(title=title) # do something with the book
```

```
                return book

class Book(models.Model):
        title = models.CharField(max_length=100)

        objects = BookManager()


book = Book.objects.create_book("Pride and Prejudice")
```

<u>**Adding a custom manager is preferred over having a class method.**</u>

If you delete a field from a model instance, accessing it again reloads the value from the database:

```
>>> obj = MyModel.objects.first()
>>> del obj.field
>>> obj.field # Loads the field from the database
```

To avoid this we use the method <u>***Model.refresh_from_db***</u>

<u>**Validating Objects**</u>

There are three ways of validation.

1. Validate the model fields - *Model.clean_fields()*
   2. Validate the model as a whole - *Model.clean()*
   3. Validate the field uniqueness - *Model.validate_unique()*

All three are performed when we call the full_clean() method. When we use a modelForm and is_valid() is called. All the above checks are done.

```
from django.core.exceptions import ValidationError try:

article.full_clean() except ValidationError as e:

# Do something based on the errors contained in e.message_dict. # Display them to a user, or handle them programmatically.
pass
```

full_clean() method is not called when we call the save() method.

To save an object

Model.**save**(*force_insert=False*,*force_update=False*, *using = DEFAULT_DB_ALIAS*, *update_fields=None*)

Each model class is provided with a pk field, which is an alias to the primary key of that model.

To delete an object

Model.**delete**(*using=DEFAULT_DB_ALIAS*, *keep_parents=False*)