Ejercicios POLIMENTES

Capacitación

Práctica 12082021 y 13082021

Modelos

Un modelo es la única y definitiva fuente de información sobre nuestros datos. Contiene los campos y comportamientos esenciales de la información que estamos manejando. Normalmente UN modelo resulta o mapea UNA tabla de nuestra DB.

- Características
  - Cada modelo es una clase de Python que hereda del camino **django.db.models.Model**.
  - Cada atributo de un modelo, es decir de la clase, representa un campo de la base de datos, una columna.
  - Lo único que se necesita definir en un modelo es la lista de campos de la base de datos que desea.
  - Cada campo en un modelo debe ser una instancia creada correctamente de la clase "Field" y ya dependiendo el tipo se pone el adecuado (Charfield, DateField, EmailField, IntegerField).
  - Cada campo del modelo toma cierto número de argumentos específicos (por ejemplo, CharField y sus subclases requieren que se indique su max_length), a continuación se muestran los más comunes:

    - null : si es 'True' significa que django de estar vacío el campo le asignará el valor NULL por default
    - blank : especifica si el campo puede quedar vacío
    - choices : da la opción al campo de elegir entre los valores de una tupla anidada con diferentes opciones a la hora de ser instanciado
    p.e:

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

      SHIRT_SIZES = (('S', 'Small'), ('M', 'Medium'), ('L', 'Large'),)

      name = models.CharField(max_length=60)
      shirt_size = models.CharField(max_length=1, choices=SHIRT_SIZES)
```

```
>>> p = Person(name="Fred Flintstone", shirt_size="L")
```

```
>>> p.save()
```

```
>>> p.shirt_size
```

```
'L'
```

```
>>> p.get_shirt_size_display()
```

```
'Large'
```

    -unique : si es 'True' significa que ese campo debe ser ÚNICO a lo largo de toda la tabla

NOTAS sobre modelos:

models.py→ fichero de nuestra app donde se genera un modelo, un modelo es una clase de python que representa a una tabla en la base de datos. Dentro de este archivo se deben encontrar todas la tablas que vayamos a tener y deseemos que existan en nuestra DB.

tests.py→Django incluye Código para poder hacer pruebas y testeo de nuestra app, de nuestros modelos creados y que lo hayan sido correctamente, que la lógica sea la adecuada, etc.

DUDAS:

¿Para qué es lo de verbose field names?

ORM [Object Relational Mapping]

Un ORM es una pieza de software que nos permite interactuar con nuestra base de datos sin la necesidad de conocer el lenguaje de consultas SQL haciendo uso del paradigma orientado a objetos. El ORM se encarga de traducir nuestras instrucciones en el lenguaje de programación que se esté utilizando a una sentencia SQL que el gestor de base de datos pueda comprender y ejecutar.

No solo están diseñados para obtener información, también se puede crear, actualizar, eliminar y procesar registros, tablas y bases de datos completas.

Al hacer uso de un ORM nos permite enfocarnos al 100 en el lenguaje que se esté utilizando para desarrollar nuestro proyecto (Python, java, etc).

También los ORM permiten interactuar con diferentes gestores de bases de datos sin mayor problema por si fuera necesario migrar nuestro proyecto de un gestor a otro.

- **Migraciones**

  Es importante recordar que cuando se hacen cambios sobre algún modelo (tabla) en el proyecto, para hacerlo válido y se vea reflejado en la DB se debe ejecutar un proceso llamado "MIGRACIÓN". Estas se logran con ayuda del manejador de comandos de Django *"manage"*, no sin antes recordar que a su vez, para que esto sea posible debemos incluir en la lista de INSTALLED_APPS dentro del archivo settings del proyecto el nombre de nuestra app donde se encuentra el modelo que queremos migrar como se muestra a continuación:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'Users.apps.UsersConfig',
    # Aquí ingreso las apps que estaré ocupando en mi proyecto, en este caso es la
que yo creé "Users"
]
```

  Desde terminal y estando ubicados en la carpeta raíz del proyecto (ya que es ahí donde está el archivo manage.py) ejecutamos los siguientes comandos:

  python manage.py makemigrations → coloca en stack todas las modificaciones a la DB, las carga y reconoce.
  python manage.py migrate → el archivo migrate se ha ejecutado y los cambios se han reflejado ahora si en la DB

  cambio campo, nuevo campo, nombre de tabla, creación de un nuevo modelo(tabla), es decir cuando se modifica tal cual su estructura →migrate
  cuando se modifica un solo registro y NO la estructura de la tabla →save

- **Consultas**

Una vez que creamos los modelos de nuestra app, Django genera de manera automática la API de abstracción de la base de datos que nos permitirá crear, recuperar, actualizar y eliminar objetos de la misma (CRUD por sus siglas en inglés de las acciones create, retrieve, update and delete).

- **Creación de objetos**

  Recordemos lo siguiente:

  modelo/clase → representa una tabla de la base de datos

  instancia de esa clase (objeto) → representa un registro en particular de la tabla

  A continuación, veremos un ejemplo de cómo crear un objeto tomando como ejemplo la clase o modelo creado "User"

Models.py

```
class User(AbstractBaseUser, PermissionsMixin):

    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=255, null=False, blank=False)

    email = models.EmailField(max_length=255, null=False, blank=False, unique=True)
    phone = models.IntegerField(null=False, blank=False, unique=True)
    password = models.CharField(max_length=64, blank=True)
    username = models.CharField(max_length=255, null=False, blank=False)

    objects = ManagerUser()
    USERNAME_FIELD = 'email'
```

tests.py

```
from Users.models import User

    u5 = User.objects.create_user(name='Alan', email='alan@correo.com', phone=156549687,
    password='abecedario')

    tabla = User.objects.all()
```

En el código de test.py podemos ver en la segunda línea la creación de un objeto tipo 'User' ingresando todos los valores deseados para cada uno de sus atributos. Esta línea de código en Python equivale a realizar la instrucción INSERT en SQL y solo será válida hasta que la instrucción save() sea ejecutada, línea contenida dentro de la clase ManagerUser() en models.py por conveniencia.
En la línea posterior se crea una estructura donde se vacían toda la información contenida en la base de datos para después poder manipularla y obtener su contenido en listas o diccionarios y mostrarlos como se desee, más adelante se excplicará esta segunda instrucción.

- **Recuperar objetos**

  Para recuperar datos de la DB es necesario crear un "QuerySet" con ayuda del manager.
  Un QuerySet representa una colección de objetos de nuestra base de datos que puede contener múltiples filtros los cuales ayudan a acotar los resultados de la búsqueda.

Básicamente un QuerySet en lenguaje de consulta SQL nativo equivale a la instrucción 'SELECT' y el filtro equivale a un 'WHERE'.

Uno genera un QuerySet utilizando el manager del modelo, es decir cada modelo creado debe tener su propio manager es cual es llamado **"object"** por default. Por ejemplo, la línea de código que presentamos en test.py anteriormente tabla = User.objects.all() regresa un QuerySet que contiene todos los objetos tipo User de la base de datos.

A continuación, se presentan los métodos más comunes para recuperar info con el manager de los modelos:

all() → regresa un QuerySet con todos los objetos de la base de datos

filter(***) → regresa un QuerySet que contiene aquellos objetos que cumplen con los parámetros '***' indicados

exclude(***) → regresa un QuerySet que contiene los objetos que NO cumplen con la condición '***'

values() → regresa un QuerySet que es un diccionario en lugar de instancias de un modelo. Cada uno de estos diccionarios representa un objeto con sus parejas 'llave:valor' correspondientes. Por ejemplo:

```
# This list contains a Blog object.
>>> Blog.objects.filter(name__startswith='Beatles')
<QuerySet [<Blog: Beatles Blog>]>

# This list contains a dictionary.
>>> Blog.objects.filter(name__startswith='Beatles').values()
<QuerySet [{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}]>
```

Cabe mencionar que siempre es es posible indicar si se desea que estos diccionarios sean presentos con cierto orden con la instrucción *order_by('nombre_campo')* ya sea antes o después de la instrucción values().

Ya que el resultado de una QuerySet acotado es otro QuerySet es posible anidar filtros y exclusiones en una sola instrucción: ClassName.objects.filter(***).exclude(+++).filter(~~~)

Es importante recordar que cada vez que acotamos con cualquier filtro un QuerySet se obtiene una estructura completamente separada y distinta a cualquier otra que pueed ser guardada y usada particularmente.

El método filter siempre nos devolverá un QuerySet, es decir una estructura para múltiples objetos que cumplan con los filtros indicados, sin embargo si solo deseamos obtener UN solo elemento teniendo claro que solo existe UN elemento que cumple nuestra consulta podemos utilizar mejor el método **get()** que nos regresa el objeto directamente.
Si no existe ningún elemento que cumplan con la consulta del get() Django regresa el error ClassName.DoesNotExist. De igual manera si resulta que más de un objeto cumple con la condición marcada por el método get() entonces Django responderá con MultipleObjectsReturned.

- **QuerySets con límites**

Es posible delimitar un QuerySet a ciertos índices de la estructura completa de objetos con sintáxis nativa de python (técnica llamada slicing). Por ejemplo:

```
ClassName.objects.all()[:5]  //Regresa los primeros 5 objetos del QuerySet
ClassName.objects.all()[5:10] //Regresa del sexto al décimo elemento del QuerySet
```

- **Field Lookups**

Un Fiel Lookup hace referencia justamente al contenido de la instrucción 'WHERE' y es especificada como un argumento para los métodos get, filter y exclude. Comúnmente siguen la sintaxis que se muestra a continuación:

NombreCampo__TipoLookup = valor

Por ejemplo:

User. objects. filter(fecha_ingreso__lte='2020-02-02') //equivale a SELECT*FROM Users WHERE fecha_ingreso <= '2020-02-02'

Y así como 'lte' existen múltiples lookups que tienen distintos fines, a continuación se presentan alguno de ellos:

- exact: claramente es cuando se busca el registro con el valor exacto en cierto campo
- iexact: cuando el valor del campo puede no ser exactamente igual en cuanto mayúsculas, minúsculas y espacios pero si contiene los elementos, por ejemplo

  User. objects. get(name__iexact='juan lópez') //puede hacer match con los valores 'JuaN lOpez'. 'JUAN Lopez", etc. Equivale a la sentencia ILIKE en SQL
- contains: en el caso de que el campo deseado contenga el valor indicado
- in: cuando se proporciona un elemento iterable, normalmente una lista o un QyerySet o en alguno casos también una cadena, por ejemplo:

  User. objects. filter(id__in=[1, 8, 9])

  En este caso también es posible generar un QuerySet y hacer una búsqueda dinámica en el iterable en lugar de proporcionar los datos específicos, por ejemplo:

```
inner_qs = Blog.objects.filter(name__contains='Cheddar')
entries = Entry.objects.filter(blog__in=inner_qs)
```

  Lo cual equivale a una subselección o búsqueda anidada:

```
SELECT ... WHERE blog.id IN (SELECT id FROM ... WHERE NAME LIKE '%Cheddar%')
```

- gt: equivale al operador > 'mayor que'
- lt: equivale al operador < 'menor que'
- lte: como habíamos visto en el ejemplo equivale a la comparación <= 'menor o igual que'
- startswith: un campo que comienze con el valor indicado, normalmente usado en cadenas
- endswith: un campo donde al final de su valor contenga la cadena o carácter indicado
- range: claramente para indicar la búsqueda de un valor dentro de un rango, por ejemplo entre ciertas fechas (en SQL equivale a la instrucción BETWEEN)
- year: usado en búsquedas por fechas, si es utilizado solo equivale a buscar en el rango de un año(el año indicado) desde el primero de enero hasta 31 de diciembre, sin embargo si por ejemplo se acompaña del lookup 'gt' la búsqueda será del 1 de enero de ese año en adelante
- month: para indicar una búsqueda en el campo fecha del modelo pero por mes
- day: para indicar una búsqueda en el campo fecha del modelo pero por día
- regex: para buscar un cierto conjunto de caracteres en una cadena dentro de los valores un campo del modelo, por ejemplo:

  User. objects. get(apellido__regex = r '^(An?|The) +')

➢ **Uniones (JOINS)**

Django ofrece una manera de realizar "uniones" es decir búsquedas anidadas en las consultas que resulta mucho más sencilla que cualquier otra consulta en SQL antivo y es mediante el uso de doble guión bajo (__), por ejemplo:

```
>>> Entry.objects.filter(blog__name='Beatles Blog')
```

La línea anterior nos va a regresar todos los objetos de la clase Entry con un blog cuyo nombre se “Bleatles Blog”, cabe destacar que Blog a su vez es una clase, un modelo, es decir en la base de datos hay una tabla llamada Blog con sus propios elementos, sin embargo es posible hacer esta relación o unión debido a que el campo ‘name’ de la tabla Blog funciona como llave foránea dentro del modelo Entry.

- **Relaciones entre objetos y modelos** [para más información: https://docs.djangoproject.com/en/3.1/topics/db/queries/#related-objects]
Cuando al definir en un modelo relaciones (eso se hace al definir el tipo del campo como ForeingKey, OneToOneField, ManyToMmanyField) las instancias de ese modelo tendrán una API especial o particular para acceder y poder comunicarse entre objetos.

    - Relación Uno a Muchos

        - Adelante

        Cuando un modelo contiene como atributo una ForeignKey, las instancias de ese modelo tendrán acceso al objeto relacionado mediante el atributo del modelo que sea la llave foránea.

        Tomando en cuenta que se describió un ejercicio donde existen los modelos Entry y Blog conectados por una llave foránea (el campo blog fue declarado como ForeignKey dentro del modelo Entry) veamos el siguiente ejemplo:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog # Returns the related Blog object.
```

        - Atrás

        Para facilitar la comprensión del siguiente concepto llamaremos “Modelo Manager” a aquel que en sus campos tiene definido a uno de ellos como llave foránea de otro modelo y “Modelo Hijo” a aquel que presta su campo como llave foránea al modelo Mnager.
        Entonces, una vez establecido lo anterior podemos decir que así como el modelo Manager tiene acceso a los objetos del modelo hijo, las instancia del modelo hijo también tienen acceso al manejador (que anteriormente ya mencionamos que todos los modelos tienen un manejador por default llamado ‘objects’) que regresa las instancias del modelo manager.
        Este manejador es el que regresa o genera los QuerySets del modelo que pueden ser filtrados y manipulados según se requiera.

        NOTA: Para facilitar las acciones con el manejador del modelo manager es posible cambiarle el nombre para que no sea el dado por default (objects) y no se confunda el ORM al ejecutar las consultas, por ejemplo:

You can override the **FOO_set** name by setting the **related_name** parameter in the **ForeignKey** definition. For example, if the **Entry** model was altered to **blog = ForeignKey(Blog, on_delete=models.CASCADE, related_name='entries')**, the above example code would look like this:

```
>>> b = Blog.objects.get(id=1)
>>> b.entries.all() # Returns all Entry objects related to Blog.

# b.entries is a Manager that returns QuerySets.
>>> b.entries.filter(headline__contains='Lennon')
>>> b.entries.count()
```

- Relación Muchos a Muchos

  En este tipo de relaciones ambos lados de la relación obtienen una API de manera automática que les da acceso entre sí.
  En realidad la manera de relacionarse resulta muy similar a cuando se trata de una relación Uno a Muchos y también se relacionan los objetos en ambos sentidos, es decir del manager (el que define uno de sus atributos tal cual con la relación MnayToMany) al hijo y del hijo al manager .
  Como se puede apreciar en la imagen presentada al final de este documento el campo authors dentro del modelo Entry resulta tener una relación tipo ManyToMany con respecto al modelo Author.

```
e = Entry.objects.get(id=3)
e.authors.all() # Returns all Author objects for this Entry.
e.authors.count()
e.authors.filter(name__contains='John')

a = Author.objects.get(id=5)
a.entry_set.all() # Returns all Entry objects for this Author.
```

- Relación Uno a Uno

  De nuevo, estas relaciones funcionan de manera similar a la Muchos a Uno, si defines un campo como OneToOneField en un modelo, las instancias de ese modelotendrán acceso al objeto relacionado por medio de un atributo. Por ejemplo:

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry, on_delete=models.CASCADE)
    details = models.TextField()

ed = EntryDetail.objects.get(id=2)
ed.entry # Returns the related Entry object.
```

  El punto más importante y la diferencia principal es que aún cuando al igual que en los otros tipos de relaciones el modelo hijo o relacionado tiene nacceso al modelo manager, el manejador del manajer representa ahora un ÚNICO obnjeto del modelo y ya no una colección de objetos, por ejemplo:

```
e = Entry.objects.get(id=2)
e.entrydetail # returns the related EntryDetail object
```

DUDAS:

De acuerdo con la documentación oficial el verbo create sirve para ejecutar en una sola la creación y el guardado del objeto y justamente nosotros estamos utilizando al crear un objeto un método llamado create_user sin embargo ese objeto o método lo declaramos en la clase ManagerUser y lo estamos ocupando como parte de la clase y User ¿qué es lo que está pasando ahí?

Método “Values()” en Django