CLAVES PRIMARIAS Y CRUD

TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE:

- Manejo de claves primarias
- Llaves primarias en columnas únicas y múltiples
- Realizando operaciones CRUD al modelo

DEFINICIÓN DE LA LLAVE PRIMARIA

Una llave primaria es una columna de tabla que sirve un propósito especial. Cada tabla necesita una primary key, pues garantiza la accesibilidad a nivel de fila (registro). Los valores que componen la columna de llave primaria son únicos; no hay dos valores iguales.

Cada tabla tiene una única llave primaria, la cual puede estar conformada por una o varias columnas. Una clave primaria concatenada está conformada por dos o más columnas. Sin embargo, en una simple tabla, también se pueden encontrar varias columnas o grupos de éstas que podrían servir como llave primaria, y usualmente son llamadas candidate keys (claves candidatas). Una tabla puede tener más de una clave candidata, pero sólo una candidate key puede convertirse en la llave primaria para esa tabla.

Las validaciones de campo incorporadas en los modelos de Django son las que vienen predefinidas para todos los campos de Django. Cada campo viene con una función de validaciones de Django validadores. También se pueden agregar más validaciones de campo integradas para aplicar o eliminar ciertas restricciones en un campo en particular. `primary_key=True` hará el campo `PRIMARY KEY` para esa tabla (modelo). Si no especifica `primary_key = True` para ningún campo en su modelo, Django agregará automáticamente un `AutoField` para contener la clave principal, por lo que no es necesario establecer `primary_key = True` en cualquiera de sus campos, a menos que desee anular el comportamiento predeterminado de la clave primaria.

LLAVES PRIMARIAS EN COLUMNAS ÚNICAS

Por defecto, Django agrega a cada modelo el siguiente campo:

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

Esta es una clave primaria auto-incrementable. Si se desea especificar una clave primaria personalizada, basta con especificar `primary_key=True` en uno de sus campos. Si Django ve que se ha establecido explícitamente una clave primaria `Field.primary_key`, no añadirá automáticamente la columna `id`. Cada modelo requiere exactamente un campo para tener `primary_key=True`.

LLAVES PRIMARIAS EN COLUMNAS MÚLTIPLES

Desafortunadamente, Django aún no tiene una clave primaria compuesta (también denominadas claves primarias de varias columnas).

Existe una biblioteca de terceros, con el nombre `django-compositekey`, que hace posible crear un modelo con una clave principal de varias columnas. Sin embargo, ese proyecto no se mantiene, y no es compatible con las últimas versiones de Django.

La mejor opción es agregar una nueva columna a su tabla que sea un `AutoField`, y convertirla en la nueva clave principal. Los campos que componen la clave principal en la actualidad se pueden marcar como `unique_together`. Si bien esto puede no ser ideal para algunas consultas, es lo suficientemente bueno para la mayoría.

REALIZANDO OPERACIONES CRUD AL MODELO

Django se basa en la arquitectura MVT(Model View Template), y gira en torno a las operaciones CRUD (Create, Retrieve, Update, Delete).

- **Crear (Create):** crea o agrega nuevas entradas en una tabla en la base de datos.
- **Recuperar (Retrieve):** leer, recuperar, buscar, o ver entradas existentes como una `list` (Vista de lista), o recuperar una entrada particular en detalle (Vista de detalles).
- **Actualizar (Update):** actualizar o editar entradas existentes en una tabla en la base de datos.
- **Eliminar (Delete):** eliminar o desactivar entradas existentes en una tabla en la base de datos.

Para comenzar, se debe tener creado un proyecto y una aplicación, para mantener todo en orden se crea una aplicación separada dentro de nuestro proyecto. Para ello necesitamos ejecutar el siguiente comando en la consola (dentro de la carpeta del proyecto donde está el archivo `manage.py`):

Mac OS X y Linux:

```
(myvenv) ~/myproject$ python manage.py startapp blog
```

Windows:

```
(myvenv) C:\Users\Name\myproject> python manage.py startapp blog
```

Notarás que se ha creado un nuevo directorio `blog`, y ahora contiene una cantidad de archivos. Los directorios y archivos en nuestro proyecto deberían verse así:

```
myproject
├── blog
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── db.sqlite3
├── manage.py
├── mysite
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── requirements.txt
```

Después de crear una aplicación, también necesitamos indicarle a Django que debe utilizarla. Eso se hace en el fichero `mysite/settings.py`. Al abrirlo en el editor tenemos que encontrar `INSTALLED_APPS`, y agregar una línea que contiene `'blog.apps.BlogConfig'`, justo encima de `]`. El producto final debe tener este aspecto:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
```

```
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.apps.BlogConfig',
]
```

CREANDO UN OBJETO CON EL MODELO

Siguiendo el ejemplo anterior, ahora se debe crear el modelo del Post. En el archivo `blog/models.py` definimos todos los objetos llamados **`Models`**. Este es un lugar en el cual definiremos nuestra entrada del blog. Abre `blog/models.py` en el editor, borra todo, y escribe código como éste:

```
blog/models.py
from django.conf import settings
from django.db import models
from django.utils import timezone

class Post(models.Model):
        title = models.CharField(max_length=200)
        text = models.TextField()
        created_date = models.DateTimeField(default=timezone.now)
        published_date = models.DateTimeField(blank=True, null=True)
```

Comprobar nuevamente que usas dos guiones bajos (__) en cada lado de **`str`**. Esta convención se usa en Python con mucha frecuencia, y a veces también se llaman "dunder" (abreviatura de "double-underscore" o, en español, "doble guión bajo").

Todas las líneas que comienzan con **`from`** o **`import`** son líneas para agregar algo de otros archivos. Así que, en vez de copiar y pegar las mismas cosas en cada archivo, podemos incluir algunas partes con **`from... import ...`**.

`class Post(models.Model)`, esta línea define nuestro modelo (es un objeto). **`class`** es una palabra clave que indica que estamos definiendo un objeto. **`Post`** es el nombre de nuestro modelo. Podemos darle un nombre diferente (pero debemos evitar espacios en blanco y caracteres especiales). Siempre inicia el nombre de una clase con una letra mayúscula. **`models.Model`** significa que **`Post`** es un modelo de Django, así Django sabe que debe guardarlo en la base de datos.

Ahora definimos las propiedades de las que hablábamos: **`title`**, **`text`**, **`created_date`**, **`published_date`**, y **`author`**. Para ello tenemos que definir el tipo de cada campo (¿es texto? ¿un número? ¿una fecha? ¿una relación con otro objeto como un **`User`**?)

`models.CharField`, así es como defines un texto con un número limitado de caracteres.

`models.TextField`, este es para texto largo sin límite. Es perfecto para el contenido de la entrada del blog.

`models.DateTimeField`, este es fecha y hora.

LEYENDO UN OBJETO CON EL MODELO

Una forma de examinar todos los datos de un determinado modelo:

```
1 ModelsName.objects.all()
```

`ModelsName` es el modelo. `objects` es un administrador (`manager`). Se encarga de todas las operaciones a "nivel de tablas" sobre los datos incluidos, y lo más importante, las consultas. Todos los modelos automáticamente obtienen un administrador `"objects"`. `all()` es un método del administrador `objects` que retorna todas las filas de la base de datos. Aunque este objeto se parece a una lista, es un `QuerySet`: un objeto que representa algún conjunto de filas de la base de datos.

Cualquier búsqueda en base de datos va a seguir esta pauta general, donde llamaremos métodos del administrador adjunto al modelo en el cual queremos hacer nuestra consulta.

Aunque obtener todos los objetos es algo que ciertamente tiene su utilidad, la mayoría de las veces lo que vamos a necesitar es manejarnos sólo con un subconjunto de los datos. Para ello usaremos el método `filter()`.

```
1 ModelsNAme.objects.filter(name="Prueba")
```

`filter()` toma argumentos de palabra clave que son traducidos en las cláusulas SQL `WHERE` apropiadas.

Puedes pasarle a `filter()` múltiples argumentos para reducir las cosas aún más: sus múltiples argumentos son traducidos a cláusulas SQL `AND`.

ACTUALIZANDO UN OBJETO CON EL MODELO

Para actualizar los datos de un modelo se usa el método `save()`, pero se debe llamar al registro que se desea actualizar, y si son varios no se tiene que hacer campo por campo.

```
if request.POST['id']:
    post = Post.objects.get(id=request.POST['id'])
    try:
        for request_data_in_array in request.POST:
            request_post_values = request.POST[request_data_in_array]
            setattr(post, request_data_in_array, request_post_values)
            post.save()
```

La lógica es la siguiente:

- Si existe `id`, buscamos un `post` con el `id` proporcionado.
- Si recorremos la cantidad de datos proporcionados (en este caso, son campos del modelo `Post`).
- Una vez que los recorremos, obtenemos el valor de cada uno adentro del loop. Esto lo hace cuando le solicitamos al objeto `POST` que nos traiga cada uno de los campos recibidos: `request.POST[request_data_in_array]`, ahora a estos valores los tenemos dentro de una variable.
- La función `setattr` (`object`, `property_name`, `values`) nos permite asignar valores a las propiedades de un objeto, tan solo es necesario pasarle la instancia del objeto al que queremos cambiar, el nombre de la propiedad, y el valor. En este caso tenemos varias propiedades, y varios valores para esas propiedades, entonces lo hacemos también dentro del loop para que lo haga con todos.
- Una vez terminado podemos guardar con el método `.save()`, y se tendrá el modelo actualizado.

BORRANDO UN OBJETO CON EL MODELO

Todos los objetos o instancias del modelo en Django tienen un método `delete()`, que puede usarse para borrar ese registro. Ahora, este método `delete()` también se puede utilizar para eliminar un solo registro, o muchos de ellos.

Para eliminar un solo registro, usaremos el siguiente código:

```
record = ModelName.objects.get(title = "Titulo")
record.delete()
```

El método `get()` recupera el registro con el title como `"Titulo"`, y luego lo elimina. Pero si no se encuentra el registro, genera una excepción. Para evitarlo podemos usar un bloque `try...except` de la siguiente manera:

```
try:
    record = ModelName.objects.get(title = "Titulo")
    record.delete()
    print("Record deleted successfully!")
except:
    print("Record doesn't exists")
```

Si tenemos que eliminar todos los registros, podemos llamar a esta función `delete()` en un `QuerySet` que los contiene. El siguiente código realiza la misma operación:

```
records = ModelName.objects.all()
records.delete()
```

Como se mencionó anteriormente, podemos llamar a esta función `delete()` en un `QuerySet`; esto significa que también podemos llamar a esta función en un `QuerySet` de algunos registros filtrados. Consulte el siguiente código para obtener el mismo resultado:

```
records = ModelName.objects.filter(text="Autores")
records.delete()
```