TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE:

- Conexión de Django a la Base de Datos
- Creando un proyecto Django para conectar a PostgreSql
- Configurando la conexión a PostgreSql
- Definición del Modelo

CONECTANDO DJANGO A LA BASE DE DATOS

Al comenzar a explorar la capa de la base de datos de Django, se necesita tener en cuenta algunas configuraciones iniciales, como indicarle a Django cuál servidor de base de datos usar y cómo conectarse a él.

El primer paso es la configuración y creación de una base de datos (por ejemplo: usando la sentencia `CREATE DATABASE`). En el caso de SQLite, no hay que crear una base de datos, pues éste usa un archivo autónomo sobre el sistema de archivos para guardarlos.

En el Drill anterior, la configuración de la base de datos se encuentra en el archivo de configuración de Django, llamado `settings.py`. Se edita este archivo, y se buscan las opciones de la base de datos: `DATABASE_ENGINE` = Indica a Django cuál base de datos utilizar. Si usa una base de datos con Django, `DATABASE_ENGINE` debe configurarse con un string de los mostrados a continuación:

CONFIGURACIÓN	BASE DE DATOS	ADAPTADOR REQUERIDO
postgresql	PostgreSQL	psycopg versión 1.x
postgresql_psycopg2	PostgreSQL	psycopg versión 2.x
mysql	MySQL	MySQLdb

sqlite3	SQLite	No necesita adaptador si se usa Python 2.5+. En caso contrario, pysqlite
oracle	Oracle	cx_Oracle

`DATABASE_NAME` = indica a Django el nombre de la base de datos. Si está usando SQLite, especifica la ruta completa del sistema de archivos hacia el archivo de la base de datos (por ejemplo: '/home/django/mydata.db').

`DATABASE_USER` = indica a Django cuál es el nombre de usuario a usar cuando se conecte a la base de datos. Si estás usando SQLite, deja este en blanco.

`DATABASE_PASSWORD` = indica a Django cuál es la contraseña a utilizar cuando se conecte a la base de datos. Si está utilizando SQLite, o tienes una contraseña vacía, deja esto en blanco.

`DATABASE_HOST` = indica a Django cuál es el host a usar cuando se conecta a la base de datos. Si la base de datos está sobre la misma computadora que la instalación de Django (o sea localhost), deja esto en blanco. Si estás usando SQLite, deja este en blanco.

MySQL es un caso especial aquí. Si el valor comienza con una barra (`"/"`), y estás usando MySQL, éste se conectará al socket especificado por medio de un socket Unix. Si estás utilizando MySQL, y este valor no comienza con una barra, entonces es asumido como el host. Ejemplo: `DATABASE_HOST = '/var/run/mysql'`

`DATABASE_PORT` = indica a Django qué puerto usar cuando se conecte a la base de datos. Si estás utilizando SQLite, deja este en blanco. En otro caso, si lo dejas en blanco, el adaptador de base de datos subyacente usará el puerto por omisión acorde al servidor de base de datos.

Una vez ingresada las configuraciones, se debe comprobar ejecutando el comando `python manage.py shell`.

Comenzará un intérprete interactivo de Python. Hay una diferencia importante entre ejecutar el comando `python manage.py shell` dentro del directorio del proyecto de Django, y el más genérico Python. El último es el Python Shell básico, pero el anterior le indica a Django cuáles archivos de configuración usar antes

de comenzar el Shell. Este es un requerimiento clave para hacer consultas a la base de datos: Django necesita saber cuáles son los archivos de configuraciones a usar para obtener la información de la conexión a la base de datos.

Una vez que hayas entrado al Shell, escribe estos comandos para probar la configuración de tu base de datos:

```
from django.db import connection
cursor = connection.cursor()
```

Si no sucede nada, entonces la base de datos está configurada correctamente. De lo contrario, revisa el mensaje de error para obtener un indicio sobre qué es lo que está mal.

INSTALANDO EL PAQUETE PSYCOPG2

En primer lugar, se debe instalar la librería `postgresql_psycopg2`:

```
pip install psycopg2-binary
```

Luego, se debe crear la base de datos. Esto se puede realizar desde el PGAdmin (entorno visual), o desde el PSQL Shell. Realizado esto debemos editar el archivo `settings.py`.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'cu07',
        'USER': 'user07',
        'PASSWORD': 'user07_password',
        'HOST': '',
        'PORT': '',
    }
}
```

Se procede a teclear el comando de migración de la base de datos:

```
python manage.py migrate
```

CREANDO UN PROYECTO DJANGO PARA CONECTAR A POSTGRESQL

En la creación del proyecto en Django y la conexión a PostgreSql, se sigue estas indicaciones:

1. Crear un entorno virtual con:

```
virtualenv myenv
```

2. Inicializar la `virtualenv`:

```
.\myenv\Scripts\activate
```

3. Descargar Django en nuestra `virtualenv`:

```
pip install Django==1.9.1
```

4. Inicializar un proyecto con Django utilizando el comando `startproject`:

```
django-admin startproject my_project
```

5. Crear una base de datos local con PostgreSQL, y asignarle un usuario.

6. Cambiar la configuración de la base de datos para utilizar PostgreSQL, en lugar de SQLite en `settings.py`, dentro de nuestro proyecto.

7. Instalar el conector para PostgreSQL:

```
pip install psycopg2
```

8. Migrar la base de datos:

```
python manage.py migrate
```

9. Crear un superusuario para loguearnos en el admin de Django:

```
python manage.py createsuperuser
```

10. Loguearse con el usuario creado en `/admin`.

CONFIGURANDO LA CONEXIÓN A POSTGRESQL

Lo primero que comentaremos será sobre la instalación de PostgreSQL, de acuerdo con el sistema operativo en el que se encuentre trabajando.

Para el caso de Windows solo se necesita ir a la página de descargas de PostgreSQL, y descargar el instalador. Además, se instala un entorno virtual en el pgAdmin en el buscador de Windows, y lo abrirá en el navegador. Con Mac el procedimiento es exactamente igual.

Para Ubuntu se tendrá que hacer desde la terminal, usando el siguiente comando:

```
sudo apt install postgresql postgresql-contrib
```

Seguidamente, se pasa a las especificaciones de configuración de su base de datos en el `settings.py`:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'cu07',
        'USER': 'user07',
        'PASSWORD': 'user07_password',
        'HOST': '',
        'PORT': '',
    }
}
```

Django necesita los siguientes parámetros para sus conexiones de base de datos:

- `client_encoding:` 'UTF8',
- `default_transaction_isolation:` por defecto, o el valor establecido en las opciones de conexión (ver más abajo), `'read committed'`.

  `timezone:`
  - `when USE_TZis True`, 'UTC' por defecto, o el `TIME_ZONE` valor establecido para la conexión.
  - `when USE_TZis False`, el valor de la `TIME_ZONE` configuración global.

Si estos parámetros ya tienen los valores correctos, Django no los configura para cada nueva conexión para mejorar ligeramente el rendimiento. Esto puede configurarse directamente en `postgresql.conf`.

DEFINICIÓN DEL MODELO

Un modelo de programación provee un marco abstracto para entender la sintaxis de los lenguajes concretos que lo siguen, y representa la semántica del mismo, así como una filosofía de uso de dicha semántica.

Un modelo de base de datos es la estructura lógica que adopta la base de datos, incluyendo las relaciones y limitaciones que determinan cómo se almacenan, organizan, y cómo se accede a ellos. Así mismo, éste también define qué tipo de operaciones se pueden realizar con los datos, es decir, determina cómo se manipulan los mismos, proporcionando también la base sobre la que se diseña el lenguaje de consultas.

En general, prácticamente todos los modelos de base de datos pueden representarse a través de un diagrama de base de datos.

MODELO ORIENTADO A OBJETOS

El modelo de bases de datos orientado a objetos define la base de datos como una colección de objetos utilizados en la programación orientada a objetos. Este modelo de base de datos utiliza tablas también, pero no solo se limita a ellas, y permite almacenar información muy detallada sobre cada objeto.

Los objetos se dotan de un conjunto de características propias, que a su vez los diferencian de objetos similares. Los objetos similares pueden agruparse en una clase, y cada objeto de esta es una instancia. Las clases intercambian datos entre sí a través de métodos (mensajes).

EL CONCEPTO DE ORM

Object Relational Mapping, o Mapeo de Objetos a Bases de Datos, conocido por sus siglas ORM, es la interfaz encargada de traducir la lógica de Orientación a Objetos (de los objetos) a la lógica relacional (de las tablas). Permite que nuestras aplicaciones web sean independientes del gestor de la base de datos, y por tanto poder migrar o portar nuestra aplicación a otra base de datos sin cambiar el código de las clases del modelo. Se utilizan objetos en vez de registros, y clases en vez de tablas.

DEFINICIÓN DE CAMPOS Y TIPOS DE DATOS

Un campo es el nombre de la unidad de información. Cada entrada en una base de datos puede tener múltiples campos de diversos tipos. Todos los campos necesitan un nombre y una descripción cuando se crean.

Un modelo puede tener un número arbitrario de campos, de cualquier tipo. Cada uno representa una columna de datos que queremos guardar en nuestras tablas de la base de datos. Cada registro de la base de datos (fila) consistirá en uno de cada posible valor del campo.

La siguiente lista describe algunos de los tipos de datos más comúnmente usados en Django:

- `CharField` se usa para definir cadenas de longitud corta a media. Debes especificar la `max_length` (longitud máxima) de los datos que se guardarán.
- `TextField` se usa para cadenas de longitud grande, o arbitraria. Puedes especificar una `max_length` para el campo, pero sólo se usa cuando el campo se muestra en formularios.
- `IntegerField` es un campo para almacenar valores de números enteros, y para validar los valores introducidos como enteros en los formularios.
- `DateField` y `DateTimeField` se usan para guardar o representar fechas e información fecha/hora (como en los objetos Python `datetime.date` y `datetime.datetime`, respectivamente). Estos campos pueden adicionalmente declarar los parámetros (mutuamente excluyentes) `auto_now=True` (para establecer el campo a la fecha actual cada vez que se guarda el modelo), `auto_now_add` (para establecer sólo la fecha cuando se crea el modelo por primera vez), y `default` (para establecer una fecha por defecto que puede ser sobrescrita por el usuario).
- `EmailField` se usa para validar direcciones de correo electrónico.
- `FileField` e `ImageField` se usan para subir ficheros e imágenes, respectivamente (el `ImageField` añade simplemente una validación adicional de que el fichero subido es una imagen). Éstos tienen parámetros para definir cómo y dónde se guardan los ficheros subidos.
- `AutoField` es un tipo especial de `IntegerField` que se incrementa automáticamente. Cuando no especificas una clave primaria para tu modelo, se añade automáticamente una de este tipo.

Hay muchos otros tipos de datos para definir los campos, incluyendo campos para diferentes tipos de números (enteros grandes, enteros pequeños, en coma flotante), boleanos, URLs, slugs, identificadores únicos, y otra información relacionada con el tiempo (duración, hora, entre otros).

DEFINICIÓN DE OPCIONES EN UN CAMPO

Cada campo del modelo debe ser una instancia de la clase de campo correspondiente. Django usa tipos de clases de campo para determinar algunas cosas:

- El tipo de columna, que le indica a la base de datos qué tipo de datos almacenar (por ejemplo, `INTEGER`, `VARCHAR`, `TEXT`).
- El widget HTML predeterminado para usar al representar un campo de formulario (por ejemplo, `<input type = ”text”>`, `<select>`).
- Los requisitos mínimos de validación, utilizados en la administración de Django, y en formularios generados automáticamente.

Django también define un conjunto de campos que representan relaciones.

- Clave externa: una relación de varios a uno. Requiere dos argumentos posicionales: la clase con la que está relacionado el modelo, y la opción `on_delete`.
- `ManyToManyField:` una relación de muchos a muchos. Requiere un argumento posicional: la clase con la que está relacionado el modelo, que funciona exactamente igual que para `ForeignKey`, incluidas las relaciones recursivas.
- `OneToOneField:` una relación de uno a uno. Conceptualmente, esto es similar a `ForeignKey` con `unique = True`, pero el lado "inverso" de la relación devolverá directamente un solo objeto.

Entre las opciones de campo se muestra un cierto conjunto de argumentos comunes disponibles para todos los tipos de campo. Todos son opcionales, éste es un resumen de los más utilizados:

- `null`: si es `True`, Django almacenará valores vacíos como `NULL` en la base de datos. El valor predeterminado es `False`.

- `blank`: si es `True` , el campo puede estar en blanco. El valor predeterminado es `False`. Tenga en cuenta que esto es diferente de `null`, ya que null está puramente relacionado con la base de datos, mientras que el `blank` está relacionado con la validación. Si un campo tiene un `blank=True`, la validación del formulario permitirá ingresar un valor vacío. Si un campo tiene en `blank=False`, se requerirá el campo.
- `choices`: una secuencia de 2 tuplas para usar como opciones para este campo. Si se proporciona esto, el widget de formulario predeterminado será un cuadro de selección en lugar del campo de texto estándar, y limitará las opciones a las ya dadas.
- `default`: el valor por defecto del campo. Puede ser un valor, o un objeto llamable. Si se puede llamar, se hará cada vez que se cree un nuevo objeto.
- `help_text:` texto de "ayuda" adicional que se mostrará con el widget de formulario. Es útil para la documentación, incluso si su campo no se utiliza en un formulario.
- `primary_key:` si es `True`, este campo es la clave principal para el modelo. Si no especifica `primary_key=True` para ningún campo en su modelo, Django agregará automáticamente un `IntegerField` para mantener la clave primaria, por lo que no necesita establecer `primary_key=True` en ninguno de sus campos, a menos que desee anular el comportamiento predeterminado de clave primaria.
- `unique:` si es `True`, este campo debe ser único en toda la tabla.