AWAKELAB

Acceso a Bases de Datos con Python/Django

Contenido 2: Implementar la capa de modelo de acceso a datos del aplicativo utilizando entidades no relacionadas para dar solución a una problemática

Objetivo de la jornada

1. Configura una aplicación Django para su conexión a una base de datos PostgreSql utilizando los componentes requeridos
2. Define un modelo que representa una entidad sin relaciones especificando campos, tipos de datos y otras opciones para representar una entidad en la base de datos acorde al framework Django
3. Define llaves primarias, simples y compuestas en modelos para representar un problema acorde al framework Django
4. Realiza operaciones CRUD en los modelos para la manipulación de los datos acorde al framework Django

6.2.1.- Conexión de Django a la Base de Datos

Para comenzar este apartado nos conectaremos a postgres, si recuerda en la clase 1 del módulo 4, creamos un cluster con **initbd** en un directorio que especificamos. Dirijámonos a ese directorio o eventualmente cree un nuevo cluster con las instrucciones de aquella clase.

Recordemos que antes de que pueda hacer algo, debe inicializar un área de almacenamiento de base de datos en el disco. A esto lo llamamos un clúster de base de datos - database cluster. (El estándar SQL utiliza el término agrupación de catálogo - cluster catalog). Una agrupación de bases de datos es una colección de bases de datos administrada por una sola instancia de un servidor de bases de datos en ejecución. Después de la inicialización, un clúster de base de datos contendrá una base de datos llamada **postgres**, que está pensada como una base de datos predeterminada para uso de otras utilidades, usuarios y aplicaciones de terceros. El servidor de la base de datos en sí no requiere que exista la base de datos de **postgres**, pero muchos programas de utilidad externos que existe. Otra base de datos creada dentro de cada clúster durante la inicialización se llama template1. Como sugiere el nombre, esto se utilizará como plantilla para bases de datos creadas posteriormente; no debe utilizarse para el trabajo real.

Continuaremos trabajando con nuestro usuario, pero también puede hacerlo con el usuario **postgres**. Durante la instalación de Postgres, se creó un usuario en el sistema operativo llamado **postgres** para corresponder al usuario administrativo de postgres PostgreSQL. En un sistema basado en UN*X puede ver los usuarios en **/etc/passwords**, en este caso en MAC OSx

```
$ cat /etc/passwd | grep Postgres
_postgres:*:216:216:PostgreSQL Server:/var/empty:/usr/bin/false
```

Si desea usar tal usuario primero debe cambiarle los privilegios para realizar tareas administrativas:

```
$ sudo su - postgres
```

Ahora debería estar en una sesión de shell para el usuario de postgres. Inicie sesión en una sesión de Postgres escribiendo **psql**.

Por simplicidad y por coherencia seguiremos con nuestro usuario creado anteriormente.

Primero, crearemos una base de datos para nuestro proyecto Django. Cada proyecto debe tener su propia base de datos aislada por razones de seguridad. Llamaremos a nuestra base de datos **municipio01**

```
CREATE DATABASE municipio01;
```

Volvemos a entrar a nuestra nueva base de datos esta vez, postgressql no tiene la instrucción USE **base_de_datos;** para cambiarse de base. Cuando obtiene una conexión a PostgreSQL, siempre es a una base de datos en particular. Para acceder a una base de datos diferente, debe obtener una nueva conexión. El uso de **\c** en psql cierra la conexión anterior y adquiere una nueva, utilizando la base de datos y/o credenciales especificadas. Se obtiene un proceso de back-end completamente nuevo con todo.

```
$ psql -d municipio01
```

A continuación, crearemos un usuario de base de datos que usaremos para conectarnos e interactuar con la base de datos.

```
CREATE USER alcalde WITH PASSWORD 'password';
```

Ahora vamos a configurar algunos parámetros. Estamos configurando la codificación predeterminada en UTF-8, es lo que espera Django. También estamos configurando el esquema de aislamiento de transacciones predeterminado en "read commited", que

bloquea las lecturas de transacciones no confirmadas. Por último, estamos configurando la zona horaria. Por defecto, nuestros proyectos de Django estarán configurados para usar UTC:

```
ALTER ROLE alcalde SET client_encoding TO 'utf8';
ALTER ROLE alcalde SET default_transaction_isolation TO 'read committed';
ALTER ROLE alcalde SET timezone TO 'UTC';
```

Ahora, todo lo que tenemos que hacer es otorgar a nuestra base de datos derechos de acceso de usuario para la base de datos que creamos:

```
GRANT ALL PRIVILEGES ON DATABASE municipio01 TO alcalde;
```

Podemos comprobar si todo funciono dejando la sesión en el shell de postgres **(control-D)** y en nuestro terminal hacemos

```
$ psql municipio01 alcalde
```

Para estar seguros podemos hacer **\conninfo**, ej:

```
municipio01=> \conninfo
You are connected to database "municipio01" as user "alcalde" via
socket in "/tmp" at port "5432".
```

6.2.1.1. Creando un proyecto Django para conectar a PostgreSql e Instalando psycopg2

Vamos a crear un nuevo proyecto básicamente para partir desde cero, no obstante, vamos a incorporar cosas que ya hemos aprendido, codificado en este proyecto "nuevo".

Primero el entorno virtual:

```
$ python3 -m venv env
$ source env/bin/activate
```

Luego vamos a instalar dos paquetes, uno por supuesto es **django** y el otro es **psycopg2**

Psycopg es el adaptador de base de datos PostgreSQL más popular para el lenguaje de programación Python. Sus principales características son la implementación completa de la especificación Python DB API 2.0 y la seguridad de subprocesos (varios subprocesos pueden compartir la misma conexión). - https://pypi.org/project/psycopg2/

```
(env) <prompt>$ pip install django psycopg2
```

Llamaremos a este proyecto **municipio01**

```
(env) <prompt>$ django-admin startproject municipio01 .
```

6.2.1.2. Configurando la conexión a PostgreSql

Ahora vamos a **municipio01/settings.py** y hacemos cambios a la sección DATABASES:

```
... DATABASES
= {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'municipio01',
        'USER': 'alcalde',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '',
    }
}
...
```

Ahora que los parámetros de Django están configurados, podemos migrar nuestras estructuras de datos a nuestra base de datos y probar el servidor.

Podemos comenzar creando y aplicando migraciones a nuestra base de datos. Dado que aún no tenemos datos reales, esto simplemente configurará la estructura inicial de la base de datos:

```
(env) <prompt>$ python manage.py makemigrations
(env) <prompt>$ python manage.py migrate
```

El segundo comando eventualmente podría producir una serie de mensajes terminados en 'OK'.

Ahora vamos a crear una aplicación

```
(env) <prompt>$ python manage.py startapp comuna01
```

6.2.2. Definición del Modelo

6.2.2.1. Qué es un Modelo

Los modelos definen la estructura de los datos almacenados, incluidos los tipos de campo y posiblemente también su tamaño máximo, valores predeterminados, opciones de lista de selección, texto de ayuda para documentación, texto de etiqueta para formularios, etc. La definición del modelo es independiente de la base de datos subyacente. puede elegir uno de varios como parte de la configuración de su proyecto. Una vez que haya elegido la base de datos que desea usar, no necesita hablar directamente con ella, simplemente escriba la estructura de su modelo y otro código, y Django se encarga de todo el trabajo sucio de comunicarse con la base de datos por usted.

Los modelos son la base de la mayoría de los proyectos de Django. Apurarse para escribir modelos Django sin pensar bien las cosas puede generar problemas en el futuro. Con demasiada frecuencia, los desarrolladores nos apresuramos a agregar o modificar modelos sin considerar las ramificaciones de lo que estamos haciendo. La solución rápida o la decisión de diseño “temporal” descuidada que incluimos en nuestro código base ahora puede perjudicarnos en los meses o años venideros, forzando soluciones poco ortodoxas o corrompiendo los datos existentes.

Así que tenga esto en cuenta cuando agregue nuevos modelos en Django o modifique los existentes. Tómese su tiempo para pensar las cosas detenidamente y diseñar su base para que sea lo más fuerte y sólida posible.

6.2.2.2. El concepto de ORM

Definición de campos, tipos de dato

```
from django.db import models


class Departamento(models.Model):
    nombre = models.CharField(max_length=100, unique=True, blank=False)
    descripcion = models.TextField(max_length=1000)
```

Luego, para hacer el cambio efectivo debemos ejecutar **$ python manage.py makemigrations** y luego **$ python manage.py migrate** deberíamos ver información como la de abajo:

```
$ python manage.py makemigrations
Migrations for 'comuna01':
  comuna01/migrations/0001_initial.py
    - Create model Departamento

$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, comuna01, contenttypes, sessions
Running migrations:
  Applying comuna01.0001_initial... OK
```

Si revisamos nuestra base de datos, además de las tablas creadas por Django podremos ver:

```
municipio01=> \d
                        List of relations
Schema |                Name                 |    Type    | Owner
       +                                     +            +
...
public | comuna01_departamento               | table      | alcalde
public | comuna01_departamento_id_seq        | sequence | alcalde
```

También podemos usar **municipio01=> \d comuna01_departamento**. Los nombres de las tablas son asignados por Django como **applicacion_modelo**, no obstante si quisiéramos por alguna razón usar nombres propios tenemos la opción **db_table** de la clase anidada **Meta**. Los metadatos del modelo son "cualquier cosa que no sea un campo", como las opciones de orden (ordering), el nombre de la tabla de la base de datos (db_table) o los nombres singulares y plurales legibles por humanos (verbose_name y verbose_name_plural). No se requiere ninguno, y agregar la clase Meta a un modelo es completamente opcional.) Para personalizar modelos/database tenemos bastantes opciones https://docs.djangoproject.com/en/3.0/ref/models/options/

Vamos a definir nuestro segundo modelo, sabemos que un departamento puede tener distintas asignaturas, además ajustaremos los campo **descripcion** para que sea posible dejarlos vacíos en ambas tablas.

```
from django.db import models


class Departamento(models.Model):
    nombre = models.CharField(max_length=100, unique=True, blank=False)
    descripcion = models.TextField(max_length=1000, null=True)

def_str_(self): return
    self.nombre


class Asignatura(models.Model):
    departamento = models.ForeignKey(Departamento, on_delete=models.CASCADE)
    nombre = models.CharField(max_length=100, unique=True, blank=False)
    descripcion = models.TextField(max_length=1000, null=True)
```

```
def_str_(self): return
    self.nombre
```

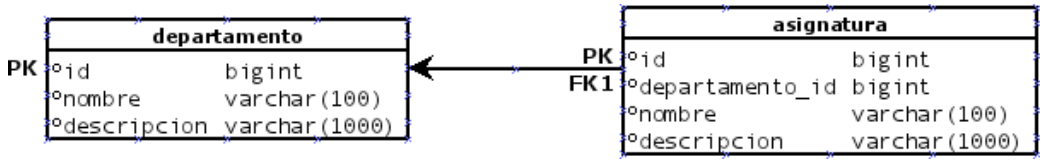
on_delete=models.CASCADE es fundamental para que cuando borremos un departamento también desaparezcan las asignaturas vinculadas.

Ahora realizamos la migración para que nuestros cambios surtan efecto:

```
$ python manage.py makemigrations
Migrations for 'comuna01':
  comuna01/migrations/0002_asignatura.py
    - Create model Asignatura

$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, comuna01, contenttypes, sessions
Running migrations:
  Applying comuna01.0002_asignatura... OK
```

El resultado de nuestra nueva tabla puede ser inspeccionado en postgres

```
municipio01=> \d comuna01_asignatura

                    Table "public.comuna01_asignatura"
 Column         |     Type   | Collation | Nullable |        Default
                +            +           +          +
 id             | integer    |           | not null |nextval
('comuna01_asignatura_id_seq'::regclass)
nombre          | character varying(100)|| not null
| descripcion   | text       |           |
|
departamento_id | integer    |           | not null |
```



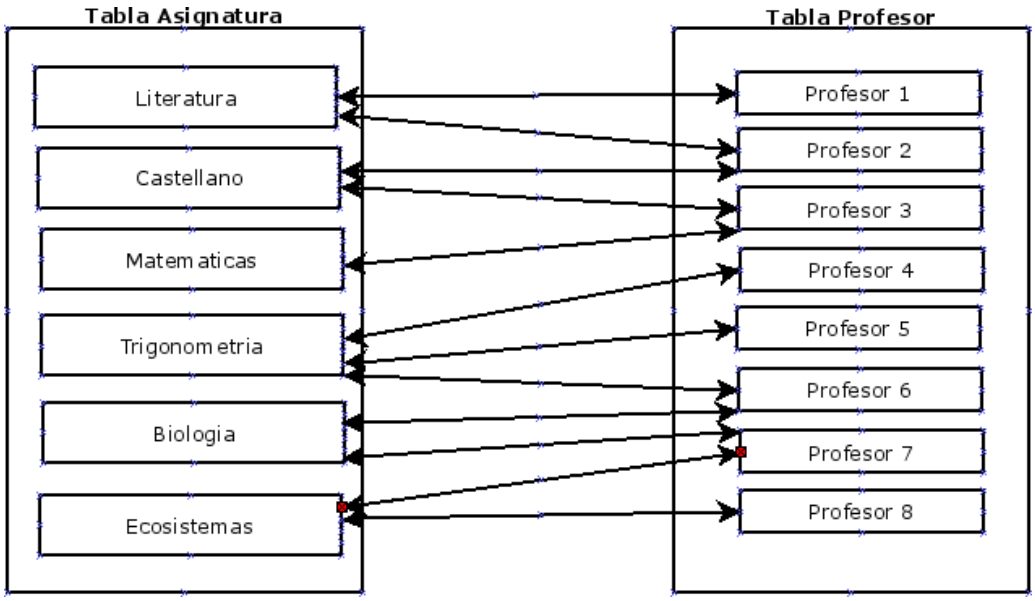
También sabemos que una asignatura puede ser impartida por distintos profesores. Entonces creamos nuestro tercer modelo.

```
...
class Profesor(models.Model):
    nombre = models.CharField(max_length=50, blank=False)
    apellido = models.CharField(max_length=50, blank=False)
    escuela = models.CharField(max_length=256, blank=False)
    fecha_de_contratacion = models.DateField()
    sueldo = models.DecimalField(max_digits=8, decimal_places=0, null=True)
    asignaturas = models.ManyToManyField(Asignatura)

class Meta:
    ordering = ['apellido']

def_str_(self):
    return self.nombre + ' ' + self.apellido
```

Aplicamos las migraciones

```
$ python manage.py makemigrations
$ python manage.py migrate
```

Si examinamos las tablas que tenemos ahora en la base de datos **comuna01_profesor_asignaturas** esta es la tabla de unión (junction) que tuvimos que crear manualmente en el módulo 4



El ORM de Django lo ha hecho por nosotros automáticamente.

Vamos a probar nuestra base de datos ahora con algunos datos, tal cual lo hicimos anteriormente mediante sentencias SQL, Solo que esta vez usaremos la facilidad python incorporada **(env)<prompt>$ python manage.py shell**

Si revisamos que tenemos en la tabla **Departamento (comuna01_departamento** en la base de datos)

```
municipio01=> SELECT * FROM comuna01_departamento;
id | nombre | descripcion
    +        +
(0 rows)
```

Usemos la facilidad python incorporada o **shell**

```
$ python manage.py shell
...
>>> dpto1 = Departamento(nombre='Ciencias Sociales',
descripcion='Ramas de la ciencia relacionadas con la sociedad y el
comportamiento humano.')
>>> dpto1.save()
```

Volvamos a inspeccionar la base de datos

```
municipio01=> SELECT * FROM comuna01_departamento;
 id |       nombre       |          descripcion
    +                    +
 1 | Ciencias Sociales | Ramas de la ciencia relacionadas con la
sociedad y el comportamiento humano.
(1 row)
```

Acá vemos de forma práctica como un objeto se transforma en información contenida en la base de datos, he ahí el nombre ORM (Asignación objeto-relacional).

Seguiremos la cadena completa desde Departamento hasta Profesor para ilustrar que sucede.

- Departamento ya fue creado
- Creamos Asignatura, siempre usando "el **shell**"

```
>>> from comuna01.models import Asignatura
>>> asignatura1 = Asignatura(departamento=dpto1, nombre='Literatura',
descripcion='Arte de la expresión verbal')
>>> asignatura1.save()
```

Si gusta puede chequear en la base de datos **SELECT * FROM comuna01_asignatura ;**

Ahora crearemos un Profesor para completar la secuencia de tablas (recuerde que el sueldo de un profesor es **null** por defecto)

```
>>> from comuna01.models import Profesor
>>> profesor1 = Profesor(nombre='Juanita', apellido='Perez',
escuela='Gabriela Mistral', fecha_de_contratacion='2011-10-30')
>>> profesor1.save()
>>> profesor1.asignaturas.add(asignatura1)
```

A nuestro **profesor1** se le asignó la **id=2** (creamos uno antes y lo eliminamos para demostrar los correlativos). Si vemos la tabla de unión tenemos

```
municipio01=> SELECT * FROM comuna01_profesor_asignaturas;
 id | profesor_id | asignatura_id
    +             +
     1 |         2 |             1
(1 row)
```

6.2.2.3. Definición de opciones en un campo

Existen muchos tipos de datos para ser usados de acuerdo al campo que necesitemos, por ejemplo, en nuestro caso hemos decidido hacer elecciones como

```
nombre = models.CharField(max_length=100, unique=True, blank=False)
```

- Lo cual se mapea en la base de datos al tipo de datos **character varying(100)**, además impusimos condiciones para que en este campo no existan valores repetidos, ni vacíos.

```
descripcion = models.TextField(max_length=1000, null=True)
```

- Lo cual se mapea en la base de datos al tipo de datos **text**, para este campo aceptamos que este vacío, ya que,no es tan primordial.

```
sueldo = models.DecimalField(max_digits=8, decimal_places=0,
default=0, null=True)
```

- Lo cual se mapea en la base de datos al tipo de datos numeric(8,0). Decidimos que fuera un valor numérico de 8 cifras sin decimales, además permitimos que esté vacío, y de requerirlo asignamos un valor inicial **0**.

La cantidad de tipos de dato que ofrece Django es bastante amplia, al igual que las opciones para cada uno de ellos. Si necesita un tipo especial puede siempre acudir a la documentación oficial, donde está explicado con detalles https://docs.djangoproject.com/en/3.1/ref/models/fields/

6.2.2. Referencias

[1] Django Documentation
https://docs.djangoproject.com/en/3.1/

[2] D. Feldroy & A. Feldroy , Two Scoops of Django 3.x, Best Practices for the Django Web Frame-work, 2020.

[3] William S. Vincent, Django for Beginners Build websites with Python & Django, 2020.