Investigación sobre Modelos en Python

Un modelo de Django es una descripción de los datos en la base de datos, representada como código de Python.

Django sigue el patrón MVC tan al pie de la letra que puede ser llamado un framework MVC. Debido a que la ''C'' es manejada por el mismo framework y la parte más emocionante se produce en los modelos, las plantillas y las vistas, Django es conocido como un Framework MTV. En el patrón de diseño MTV.

M significa ''Model'' (Modelo), la capa de acceso a la base de datos. Esta capa contiene toda la información sobre los datos: cómo acceder a estos, cómo validarlos, cuál es el comportamiento que tiene, y las relaciones entre los datos.

Django usa un modelo para ejecutar código SQL detrás de escena y retornar estructuras de datos convenientes en Python representando las filas de las tablas de la base de datos. Django también usa modelos para representar conceptos de alto nivel que no necesariamente pueden ser manejados por SQL.

Django trabaja de este modo por varias razones:

- La introspección requiere overhead y es imperfecta. Con el objetivo de proveer una API conveniente de acceso a los datos, Django necesita conocer de alguna forma la capa de la base de datos, y hay dos formas de lograr esto. La primera sería describir explícitamente los datos en Python, y la segunda sería la introspección de la base de datos en tiempo de ejecución para determinar el modelo de la base de datos.
- Escribir Python es divertido, y dejar todo en Python limita el número de veces que tu cerebro tiene que realizar un ''cambio de contexto''. Si te mantienes en un solo entorno/mentalidad de programación tanto tiempo como sea posible, ayuda para la productividad. Teniendo que escribir SQL, luego Python, y luego SQL otra vez es perjudicial.
- Tener modelos de datos guardados como código en vez de en tu base de datos hace fácil dejar tus modelos bajo un control de versiones. De esta forma, puedes fácilmente dejar rastro de los cambios a tu capa de modelos.
- SQL permite sólo un cierto nivel de metadatos acerca de un layout de datos. La mayoría de sistemas de base de datos, por ejemplo, no provee un tipo de datos especializado para representar una dirección web o de email. Los modelos de Django sí. La ventaja de un tipo de datos de alto nivel es la alta productividad y la reusabilidad de código.
- SQL es inconsistente a través de distintas plataformas. Si estás redistribuyendo una aplicación web, por ejemplo, es mucho más pragmático distribuir un módulo de Python que describa tu capa de datos que separar conjuntos de sentencias CREATE TABLE para MySQL, PostgreSQL y SQLite.

Una contra de esta aproximación, sin embargo, es que es posible que el código Python quede fuera de sincronía respecto a lo que hay actualmente en la base. Si haces cambios en un modelo Django, necesitarás hacer los mismos cambios dentro de tu base de datos para mantenerla consistente con el modelo.

El primer paso para utilizar esta configuración de base de datos con Django es expresarla como código Python en el archivo models.py que se creó con el comando startapp.

biblioteca/models.py

```
from django.db import models

class Editor(models.Model):
    nombre = models.CharField(max_length=30)
    domicilio = models.CharField(max_length=50)
    ciudad = models.CharField(max_length=60)
    estado = models.CharField(max_length=30)
    pais = models.CharField(max_length=50)
    website = models.URLField()

class Autor(models.Model):
    nombre = models.CharField(max_length=30)
    apellidos = models.CharField(max_length=40)
    email = models.EmailField()

class Libro(models.Model):
    titulo = models.CharField(max_length=100)
    autores = models.ManyToManyField(Autor)
    editor = models.ForeignKey(Editor)
    fecha_publicacion = models.DateField()
    portada = models.ImageField(upload_to='portadas')
```

Cada modelo es representado por una clase Python que es una subclase de django.db.models.Model. La clase antecesora, Model, contiene toda la maquinaria necesaria para hacer que estos objetos sean capaces de interactuar con la base de datos y que hace que nuestros modelos sólo sean responsables de definir sus campos, en una sintaxis compacta y agradable.

Cada modelo generalmente corresponde a una tabla única de la base de datos, y cada atributo de un modelo generalmente corresponde a una columna en esa tabla.

Instalando el Modelo

Ya escribimos el código; ahora necesitamos crear las tablas en la base de datos. Para ello, el primer paso es activar estos modelos en nuestro proyecto Django. Hacemos esto agregando la aplicación biblioteca a la lista de aplicaciones instaladas en el archivo de configuración.

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)
```

Edita el archivo settings.py y examina la variable de configuración INSTALLED_APPS, INSTALLED_APPS le indica a Django qué aplicaciones están activadas para un proyecto determinado. Por defecto esta se ve así:

Agrega tu aplicación 'biblioteca' a la lista de INSTALLED_APPS, de manera que la configuración termine viéndose así →

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'biblioteca',
)
```

Asegúrate de instalar la librería de imágenes **Pillow**, para validar imágenes ya que Django la utiliza para comprobar que los objetos que sean subidos a un campo ImageField sean imágenes validas, de lo contrario Django se quejara si intentas usar un campo ImageField sin tener instalada la librería Pillow. Para instalarla usa el comando: `pip install pillow`

También agrega la ruta al directorio en donde se guardaran las imágenes, especificándolo en el archivo de configuraciones setings.py y usando la variable MEDIA_ROOT: `MEDIA_ROOT = 'media/'`

Si tus modelos son válidos, ejecuta el siguiente comando para que Django compruebe la sintaxis de tus modelos en la aplicación biblioteca:

```
python manage.py check biblioteca
```

El comando check verifica que todo esté en orden respecto a tus modelos, no crea ni toca de ninguna forma tu base de datos -- sólo imprime una salida en la pantalla en la que identifica posibles errores en tus modelos.

Una vez que todo está en orden, necesitamos guardar las migraciones para los modelos en un archivo de control, para que Django pueda encontrarlas al sincronizar el esquema de la base de datos. Ejecuta el comando makemigrations de esta manera:

```
python manage.py makemigrations
```

Ahora para realizar los cambios al esquema de la base de datos es necesario usar el comando mígrate, que se encarga de crear las tablas de la base de datos:

```
python manage.py migrate
```

Los tres pasos que seguimos para crear cambios en el modelo:

1. Cambia tu modelo (en models.py).
2. Ejecuta python manage.py makemigrations para crear las migraciones para esos cambios.
3. Ejecuta python manage.py migrate para aplicar esos cambios a la base de datos.

Bibliografía

Garcia M., S. (2015). *La Guia definitiva de Django.* Celayita México: Django Software Corporation.