AWAKELAB

Acceso a Bases de Datos con Python/Django

MÓDULO –ACCESO A BASES DE DATOS CON PYTHON/DJANGO

6.1.- Contenido 1: Describir las características fundamentales del framework Django para su integración con las bases de datos

Objetivo de la jornada

1. Reconoce las características del framework Django aplicado a las bases de datos
2. Reconoce los paquetes de instalación de base de datos en Django
3. Reconoce las características de Django como ORM para su integración con una base de datos

6.1.1.- Introducción a bases de datos con Django

6.1.1.1. Django y su integración a bases de datos

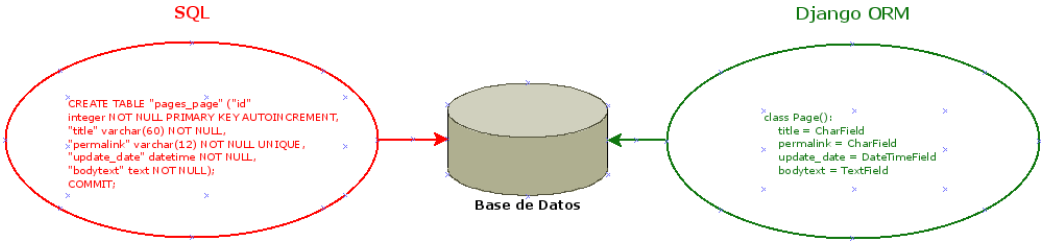
A menos que esté creando un sitio web simple, hay pocas posibilidades de evitar la necesidad de interactuar con algún tipo de base de datos al crear aplicaciones web modernas.

Desafortunadamente, esto generalmente significa que tiene que ensuciarse las manos con el lenguaje de consulta estructurado (SQL). En Django, los problemas con SQL son un problema resuelto: no tiene que usar SQL en absoluto a menos que lo desee. En su lugar, usa un modelo de Django para acceder a la base de datos.

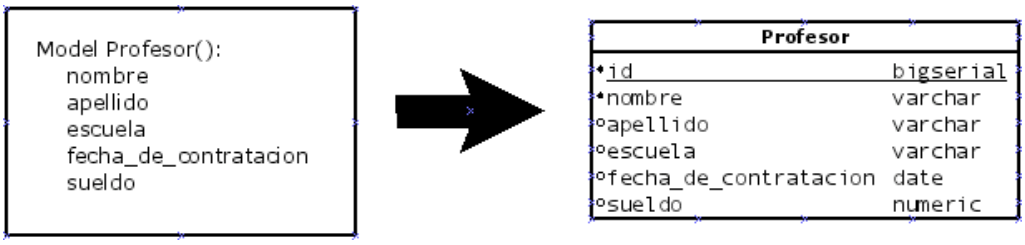
Los modelos de Django proporcionan un mapeo relacional de objetos (ORM) a la base de datos subyacente. ORM es una poderosa técnica de programación que facilita mucho el trabajo con datos y bases de datos relacionales.

La mayoría de las bases de datos comunes están programadas con algún tipo de lenguaje de consulta estructurado (SQL), sin embargo, cada base de datos implementa SQL de manera diferente. SQL puede ser complicado y difícil de aprender. Una herramienta ORM, por otro lado, proporciona un mapeo simple entre un objeto (la "O" en ORM) y la base de datos subyacente. Esto significa que el programador no

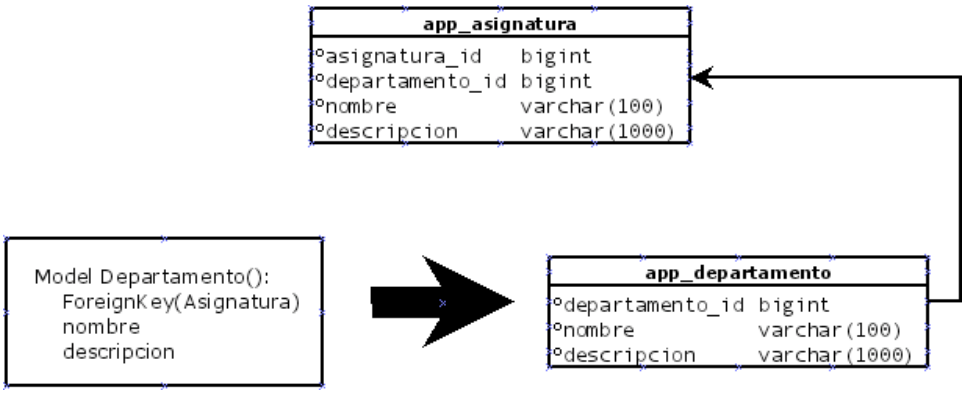
necesita conocer la estructura de la base de datos, ni requiere SQL complejo para manipular y recuperar datos.



En Django, el modelo es el objeto asignado a la base de datos. Cuando crea un modelo, Django crea una tabla correspondiente en la base de datos, sin que tenga que escribir una sola línea de SQL. Django antepone el nombre de la tabla con el nombre de su aplicación Django (lo veremos más adelante). El modelo también vincula información relacionada en la base de datos.



Podemos apreciar como existe una analogía entre Django y la base de datos.



Esta relación se crea vinculando los modelos con una clave externa. Estamos simplificando las cosas aquí, pero sigue siendo una descripción general útil de cómo el ORM de Django usa los datos del modelo para crear tablas de base de datos.

Revisaremos los modelos, así que no se preocupe si no comprende al 100% lo que está sucediendo en este momento. Las cosas se vuelven más claras una vez que haya visto modelos de forma práctica.

6.1.1.2. Las bases de datos soportadas por Django

Django 3 soporta oficialmente cinco bases de datos:

- PostgreSQL
- MySQL
- SQLite
- Oracle
- MariaDB
-

También hay varias aplicaciones de terceros disponibles si necesitamos conectarnos a una base de datos con soporte no oficial.
La preferencia de la mayoría de los desarrolladores de Django, es PostgreSQL. MySQL también es un backend de base de datos común para Django.

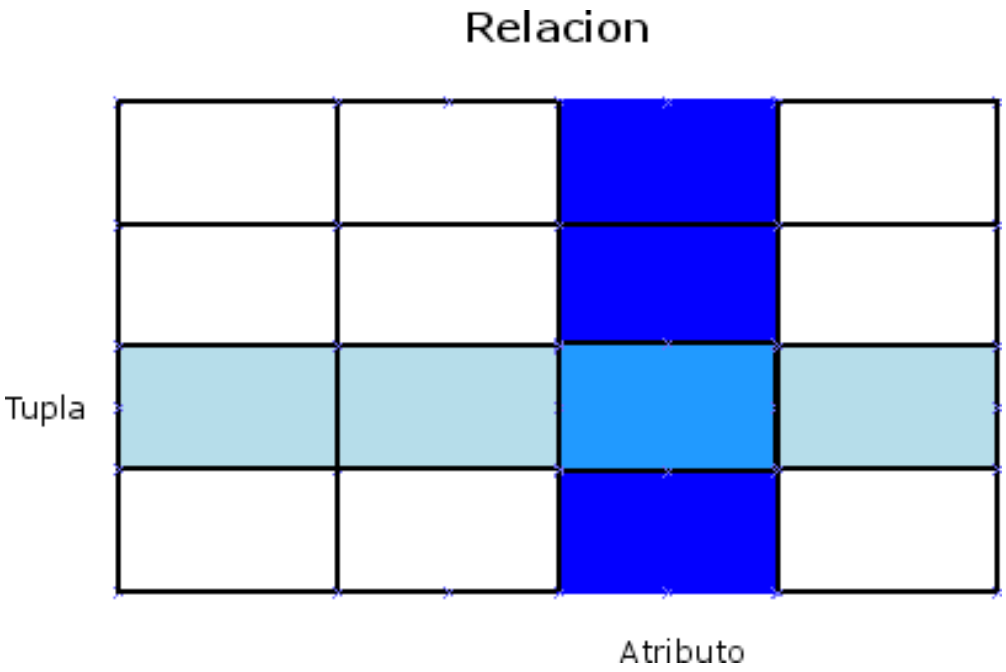
Alternativas populares de Sistemas de gestión de bases de datos relacionales

El modelo de datos relacionales, que organiza los datos en tablas de filas y columnas, predomina en las herramientas de gestión de bases de datos. Hoy en día existen otros modelos de datos, incluidos NoSQL y NewSQL, pero los sistemas de administración de bases de datos relacionales (RDBMS) siguen siendo dominantes para almacenar y administrar datos en todo el mundo.

Las bases de datos son grupos de información o datos modelados lógicamente. Un sistema de administración de bases de datos (DBMS), por otro lado, es un programa de computadora que interactúa con una base de datos. Un DBMS le permite controlar el acceso a una base de datos, escribir datos, ejecutar consultas y realizar cualquier otra tarea relacionada con la administración de la base de datos. Aunque los sistemas de gestión de bases de datos a menudo se denominan "bases de datos", los dos términos no son intercambiables. Una base de datos puede ser cualquier colección de datos, no solo uno almacenado en una computadora, mientras que un DBMS es el software que le permite interactuar con una base de datos.

Todos los sistemas de gestión de bases de datos tienen un modelo subyacente que estructura cómo se almacenan y se accede a los datos. Un sistema de gestión de bases de datos relacionales es un DBMS que emplea el modelo de datos relacionales. En este modelo, los datos se organizan en tablas, que en el contexto de los RDBMS se

denominan más formalmente relaciones. Una relación es un conjunto de tuplas o filas en una tabla, y cada tupla comparte un conjunto de atributos o columnas:



Como vimos en el módulo 4, la mayoría de las bases de datos relacionales utilizan un lenguaje de consulta estructurado (SQL) para administrar y consultar datos. Sin embargo, muchos RDBMS usan su propio dialecto particular de SQL, que puede tener ciertas limitaciones o extensiones. Estas extensiones suelen incluir características adicionales que permiten a los usuarios realizar operaciones más complejas de las que podrían realizar con SQL estándar. A cada columna se le asigna un tipo de datos que dicta qué tipo de entradas se permiten en esa columna. Los diferentes RDBMS implementan diferentes tipos de datos, que no siempre son directamente intercambiables. Algunos tipos de datos comunes incluyen fechas (dates), cadenas (strings), enteros (integer) y booleanos (booleans).

Los tipos de datos numéricos pueden tener signo (**+** o **-**, signed), lo que significa que pueden representar números positivos y negativos, o sin signo (unsigned), lo que significa que solo pueden representar números positivos. Por ejemplo, el tipo de datos **tinyint** de MySQL puede contener 8 bits de datos, lo que equivale a 256 valores posibles. El rango con signo de este tipo de datos es de **-128** a **127**, mientras que el rango sin signo es de **0** a **255**.

A veces, un administrador de base de datos impondrá una restricción en una tabla para limitar los valores que se pueden ingresar en ella. Normalmente, una restricción se aplica a una columna en particular, pero algunas restricciones también se pueden aplicar a una tabla completa. A continuación, se muestran algunas restricciones que se utilizan comúnmente en SQL:

- **UNIQUE**: la aplicación de esta restricción a una columna garantiza que no haya dos entradas en esa columna idénticas.

- **NOT NULL**: esta restricción asegura que una columna no tenga entradas **NULL**.

- **PRIMARY KEY**: "Llave primaria" es una combinación de **UNIQUE** y **NOT NULL**, la restricción **PRIMARY KEY** asegura que ninguna entrada en la columna sea **NULL** y que cada entrada sea distinta.

- **FOREIGN KEY**: una "Llave Externa" **FOREIGN KEY** es una columna en una tabla que se refiere a la "Llave primaria" de otra tabla. Esta restricción se usa para vincular dos tablas juntas: las entradas a la columna **FOREIGN KEY** ya deben existir en la columna principal **PRIMARY KEY** para que el proceso de escritura sea exitoso.

- **CHECK**: Esta restricción limita el rango de valores que se pueden ingresar en una columna. Por ejemplo, si su aplicación está destinada solo a residentes de la región de Antofagasta, puede agregar una restricción **CHECK** en una columna de teléfonos para permitir solo entradas que contengan 55 como prefijo.

- **DEFAULT**: proporciona un valor predeterminado para una columna determinada. A menos que se especifique otro valor, SQLite ingresa el valor predeterminado automáticamente.

- **INDEX** se utiliza para ayudar a recuperar datos de una tabla más rápidamente, esta restricción es similar a un índice en un libro de texto: en lugar de tener que revisar cada entrada en una tabla, una consulta solo tiene que revisar las entradas de la columna indexada para encontrar la deseada resultados.

Hemos repasado y ampliado algunos conceptos relativos a los sistemas de administración de bases de datos relacionales en general, con el fin de explicar características de tres bases de datos relacionales de código abierto populares y a las que usted podría enfrentarse por su popularidad de uso.

SQLite

SQLite es un RDBMS autónomo, basado en archivos y completamente de código abierto conocido por su portabilidad, confiabilidad y sólido rendimiento incluso en entornos de poca memoria. Sus transacciones son compatibles con ACID (sigla en inglés para Atomicidad, Consistencia, Aislamiento y Durabilidad) incluso en los casos en que el sistema falla o sufre un corte de energía.

El sitio web del proyecto SQLite la describe como una base de datos "sin servidor". La mayoría de los motores de bases de datos relacionales se implementan como un proceso de servidor en el que los programas se comunican con el servidor host a través de una comunicación entre procesos que transmite solicitudes. Sin embargo, con SQLite, cualquier proceso que acceda a la base de datos lee y escribe en el archivo de disco de la base de datos directamente. Esto simplifica el proceso de configuración de SQLite, ya que elimina cualquier necesidad de configurar un proceso de servidor. Asimismo, no es necesaria ninguna configuración para los programas que utilizarán la base de datos SQLite: todo lo que necesitan es acceso al disco.

SQLite es software gratuito y de código abierto, y no se requiere una licencia especial para usarlo. Sin embargo, el proyecto ofrece varias extensiones, cada una por una tarifa única, que ayudan con la compresión y el cifrado. Además, el proyecto ofrece varios paquetes de soporte comercial, cada uno por una tarifa anual.
Si ha puesto atención a la creación de proyectos django es la base de datos configurada por defecto en **settings**.

Tipos de datos compatibles con SQLite: SQLite permite una variedad de tipos de datos, organizados en las siguientes clases de almacenamiento:

- **null** Incluye cualquier valor **NULL.**
- **integer**: Enteros con signo, almacenados en 1, 2, 3, 4, 6 u 8 bytes según la magnitud del valor.
- **real**: Números reales, o valores de coma flotante, almacenados como números de coma flotante de 8 bytes.
- **text**: Cadenas de texto almacenadas utilizando la codificación de la base de datos, que puede ser **UTF-8**, **UTF-16BE** o **UTF-16LE**.
- **blob**: Cualquier blob de datos, con cada blob almacenado exactamente como se ingresó. (binary large object)

En el contexto de SQLite, los términos "clase de almacenamiento" y "tipo de datos" se consideran intercambiables. Si desea obtener más información sobre los tipos de datos de SQLite y la afinidad de tipos de SQLite, consulte la documentación oficial de SQLite sobre el tema.

Ventajas de SQLite

- Tamaño reducido: como su nombre lo indica, la librería SQLite es muy ligera. Aunque el espacio que utiliza varía según el sistema donde está instalado, puede ocupar menos de 600 KB de espacio. Además, es completamente autónoma, lo que significa que no hay dependencias externas que deba instalar en su sistema para que SQLite funcione.

- Fácil de usar: SQLite a veces se describe como una base de datos de "configuración cero" que está lista para usar de inmediato. SQLite no se ejecuta como un proceso de servidor, lo que significa que nunca es necesario detenerlo, iniciarlo o reiniciarlo, y no viene con ningún archivo de configuración que deba administrarse. Estas características ayudan a agilizar la ruta desde la instalación de SQLite hasta la integración con una aplicación.

- Portátil: a diferencia de otros sistemas de administración de bases de datos, que generalmente almacenan datos como un gran lote de archivos separados, una base de datos SQLite completa se almacena en un solo archivo. Este archivo se puede ubicar en cualquier lugar de una jerarquía de directorios y se puede compartir mediante un medio extraíble o un protocolo de transferencia de archivos.

Desventajas de SQLite

- Concurrencia limitada: aunque varios procesos pueden acceder y consultar una base de datos SQLite al mismo tiempo, solo un proceso puede realizar cambios en la base de datos en un momento dado. Esto significa que SQLite admite una mayor simultaneidad que la mayoría de los otros sistemas de administración de bases de datos integrados, pero no tanto como los RDBMS de cliente/servidor como MySQL o PostgreSQL.

- Sin administración de usuarios: los sistemas de bases de datos a menudo vienen con soporte para usuarios o conexiones administradas con privilegios de acceso predefinidos a la base de datos y las tablas. Debido a que SQLite lee y escribe directamente en un archivo de disco normal, los únicos permisos de acceso aplicables son los permisos de acceso típicos del sistema operativo subyacente. Esto hace que SQLite sea una mala elección para aplicaciones que requieren varios usuarios con permisos de acceso especiales.

- Seguridad: un motor de base de datos que utiliza un servidor puede, en algunos casos, brindar una mejor protección contra errores en la aplicación cliente que una base de datos sin servidor como SQLite. Por ejemplo, los punteros perdidos en un cliente no pueden dañar la memoria del servidor. Además, debido a que un servidor es un único proceso persistente, una base de datos cliente-servidor puede controlar el acceso a los datos con más precisión que una base de datos sin servidor, lo que permite un bloqueo más detallado y una mejor concurrencia.

Cuándo usar SQLite

- Aplicaciones integradas: SQLite es una excelente opción de base de datos para aplicaciones que necesitan portabilidad y no requieren expansión futura. Los ejemplos incluyen aplicaciones locales de un solo usuario y aplicaciones o juegos móviles.

- Acceso al disco: en los casos en que una aplicación necesita leer y escribir archivos en el disco directamente, puede ser beneficioso usar SQLite para la funcionalidad adicional y la simplicidad que viene con el uso de SQL.

- Testing: para muchas aplicaciones puede resultar excesivo probar su funcionalidad con un DBMS que utiliza un proceso de servidor adicional. SQLite tiene un modo en memoria que se puede usar para ejecutar pruebas rápidamente sin la sobrecarga de las operaciones reales de la base de datos, lo que lo convierte en una opción ideal para las pruebas/tests.

Cuándo No usar SQLite

- Trabajar con una gran cantidad de datos: SQLite puede admitir técnicamente una base de datos de hasta 140 TB de tamaño, siempre que la unidad de disco y el sistema de archivos también admitan los requisitos de tamaño de la base de datos. Sin embargo, el sitio web SQLite recomienda que cualquier base de datos que se acerque a 1TB se aloje en una base de datos cliente-servidor centralizada, ya que una base de datos SQLite de ese tamaño o mayor sería difícil de administrar.

- Grandes volúmenes de escritura: SQLite permite que solo se lleve a cabo una operación de escritura en un momento dado, lo que limita significativamente su rendimiento. Si su aplicación requiere muchas operaciones de escritura o varios escritores simultáneos, es posible que SQLite no sea adecuado para sus necesidades.

- Se requiere acceso a la red: debido a que SQLite es una base de datos sin servidor, no proporciona acceso directo a la red a sus datos. Este acceso está integrado en la aplicación, por lo que, si los datos en SQLite están ubicados en una máquina separada de la aplicación, se requerirá un enlace con alto ancho de banda a través de la red. Esta es una solución costosa e ineficaz y, en tales casos, un DBMS cliente-servidor puede ser una mejor opción.

MySql

Según el "DB-Engines Ranking", MySQL ha sido el RDBMS de código abierto más popular desde que el sitio comenzó a rastrear la popularidad de las bases de datos en 2012. Es un producto rico en funciones que impulsa a muchos de los sitios web y aplicaciones más grandes del mundo, incluidos Twitter, Facebook, Netflix y Spotify.
Comenzar con MySQL es relativamente sencillo, gracias en gran parte a su documentación exhaustiva y a su gran comunidad de desarrolladores, así como a la abundancia de recursos en línea relacionados con MySQL.

MySQL fue diseñado para brindar velocidad y confiabilidad, a expensas de la total adherencia al SQL estándar. Los desarrolladores de MySQL trabajan continuamente para lograr una adherencia más cercana al SQL estándar, pero aún está por detrás de otras implementaciones de SQL. Sin embargo, viene con varios modos y extensiones SQL que lo acercan al cumplimiento. A diferencia de las aplicaciones que usan SQLite, las aplicaciones que usan una base de datos MySQL acceden a ella a través de un proceso daemon separado. Debido a que el proceso del servidor se encuentra entre la base de datos y otras aplicaciones, permite un mayor control sobre quién tiene acceso a la base de datos.

MySQL ha inspirado una gran cantidad de aplicaciones, herramientas y librerías integradas de terceros que amplían su funcionalidad y ayudan a que sea más fácil trabajar con ellas. Algunas de las herramientas de terceros más utilizadas son phpMyAdmin, DBeaver y HeidiSQL.

Tipos de datos compatibles con MySQL: Los tipos de datos de MySQL se pueden organizar en tres categorías amplias: tipos numéricos, tipos de fecha y hora y tipos de cadenas.

Tipos numéricos:

- **tinyint**: Un número entero muy pequeño. El rango con signo para este tipo de datos numéricos es de **-128** a **127**, mientras que el rango sin signo es de **0** a **255**.

- **smallint**: Un pequeño entero. El rango con signo para este tipo numérico es de **-32768** a **32767**, mientras que el rango sin signo es de **0** a **65535**.

- **mediumint**: Un número entero de tamaño mediano. El rango con signo para este tipo de datos numéricos es de **-838860**8 a **8388607**, mientras que el rango sin signo es de **0** a **16777215**.

- **int o integer**: Un entero de tamaño normal. El rango con signo para este tipo de datos numéricos es de **-2147483648 a 2147483647**, mientras que el rango sin signo es de **0 a 4294967295**.

- **bigint**: Un entero grande. El rango con signo para este tipo de datos numéricos es de **-9223372036854775808** a **9223372036854775807**, mientras que el rango sin signo es de **0** a **18446744073709551615**.

- **float**: Un pequeño número de punto flotante (precisión simple).

- **double, double precision o real**: Un número de coma flotante de tamaño normal (doble precisión).

- **dec, decimal, fixed o numeric**: Un número de punto fijo empaquetado. La longitud de visualización de las entradas para este tipo de datos se define cuando se crea la columna, y cada entrada se adhiere a esa longitud.

- **bool o boolean**: Un booleano es un tipo de datos que solo tiene dos valores posibles, generalmente **true** o **false**.

- **bit**: Un tipo de valor de bit para el que puede especificar el número de bits por valor, de **1** a **64**.

Tipos de fecha y hora:

- **date**: Una fecha representada como **AAAA-MM-DD**.

- **datetime**: Una marca de tiempo (timestamp) que muestra la fecha y la hora, mostrada como **AAAA-MM-DD HH:MM:SS**.

- **timestamp**: Una marca de tiempo (timestamp) que indica la cantidad de tiempo desde la época Unix (**00:00:00** el 1 de enero de 1970).

- **time**: Hora del día, que se muestra como **HH:MM:SS.**

- **year**: Un año expresado en un formato de 2 o 4 dígitos, siendo 4 dígitos el valor predeterminado.

Tipos de cadenas (string):

- **char**: Una cadena de longitud fija; las entradas de este tipo se rellenan a la derecha con espacios para cumplir con la longitud especificada cuando se almacenan.

- **varchar**: Una cadena de longitud variable.

- **binary**: Similar al tipo **char**, pero una cadena de bytes binarios de una longitud especificada en lugar de una cadena de caracteres no binarios.

- **varbinary**: Similar al tipo **varchar**, pero una cadena de bytes binarios de longitud variable en lugar de una cadena de caracteres no binarios.

- **blob**: Una cadena binaria con una longitud máxima de **65535** (**2^16 - 1**) bytes de datos.

- **tinyblob**: Una columna de blobs con una longitud máxima de **255** (**2^8 - 1**) bytes de datos.

- **mediumblob**: Una columna de blobs con una longitud máxima de **16777215** (**2^24 - 1**) bytes de datos.

- **longblob**: Una columna de blob con una longitud máxima de **4294967295** (**2^32 - 1**) bytes de datos.

- **texto**: Una cadena con una longitud máxima de **65535** (**2^16 - 1**) caracteres.

- **tinytext**: Una columna de texto con una longitud máxima de **255** (**2^8 - 1**) caracteres.

- **mediumtext**: Una columna de texto con una longitud máxima de **16777215** (**2^24 - 1**) caracteres.

- **longtext**: Una columna de texto con una longitud máxima de **4294967295** (**2^32 - 1**) caracteres.

- **enum**: Una enumeración, que es un objeto cadena que toma un solo valor de una lista de valores que se declaran cuando se crea la tabla.

- **set**: Similar a una enumeración, un objeto cadena que puede tener cero o más valores, cada uno de los cuales debe elegirse de una lista de valores permitidos que se especifican cuando se crea la tabla.

Advantages of MySQL

- Popularidad y facilidad de uso: como uno de los sistemas de bases de datos más populares del mundo, no hay escasez de administradores de bases de datos que tengan experiencia trabajando con MySQL. Del mismo modo, existe una gran cantidad de documentación impresa y en línea sobre cómo instalar y administrar una base de datos MySQL, así como una serie de herramientas de terceros, como **phpMyAdmin**, que tienen como objetivo simplificar el proceso de comenzar con la base de datos.

- Seguridad: MySQL viene instalado con un script que le ayuda a mejorar la seguridad de su base de datos estableciendo el nivel de seguridad de la contraseña de la instalación, definiendo una contraseña para el usuario root, eliminando cuentas anónimas y eliminando bases de datos de prueba que son, por defecto, accesibles para todos los usuarios. Además, a diferencia de SQLite, MySQL admite la administración de usuarios y le permite otorgar privilegios de acceso usuario por usuario.

- Velocidad: al optar por no implementar ciertas características de SQL, los desarrolladores de MySQL pudieron priorizar la velocidad. Si bien las pruebas de referencia más recientes muestran que otros RDBMS como PostgreSQL pueden igualar o al menos acercarse a MySQL en términos de velocidad, MySQL aún tiene la reputación de ser una solución de base de datos extremadamente rápida.

- Replicación: MySQL admite varios tipos diferentes de replicación, que es la práctica de compartir información entre dos o más hosts para ayudar a mejorar la confiabilidad, disponibilidad y tolerancia a fallas. Esto es útil para configurar una solución de respaldo de base de datos o escalar horizontalmente una base de datos.

Desventajas de MySQL

- Limitaciones conocidas: debido a que MySQL fue diseñado para velocidad y facilidad de uso en lugar del cumplimiento total de SQL, viene con ciertas limitaciones funcionales. Por ejemplo, carece de soporte para cláusulas **FULL JOIN.**

- Licencias y características patentadas: MySQL es un software de doble licencia, con una edición comunitaria gratuita y de código abierto con licencia GPLv2 y varias ediciones comerciales pagas publicadas bajo licencias patentadas.

Debido a esto, algunas funciones y complementos solo están disponibles para las ediciones propietarias.

- Desarrollo lento: desde que Sun Microsystems adquirió el proyecto MySQL en 2008, y luego por Oracle Corporation en 2009, ha habido quejas de los usuarios de que el proceso de desarrollo del DBMS se ha ralentizado significativamente, ya que la comunidad ya no tiene la agencia para reaccionar rápidamente a los problemas e implementar cambios.

Cuándo usar MySQL

- Operaciones distribuidas: el soporte de replicación de MySQL lo convierte en una excelente opción para configuraciones de bases de datos distribuidas como arquitecturas primaria-secundaria o primaria-primaria.

- Sitios web y aplicaciones web: MySQL impulsa muchos sitios web y aplicaciones en Internet. Esto se debe, en gran parte, a lo fácil que es instalar y configurar una base de datos MySQL, así como a su velocidad general y escalabilidad a largo plazo.

- Crecimiento futuro esperado: el soporte de replicación de MySQL puede ayudar a facilitar el escalado horizontal. Además, es un proceso relativamente sencillo actualizar a un producto MySQL comercial, como MySQL Cluster, que admite la fragmentación automática, otro proceso de escalado horizontal.

Cuándo NO usar MySQL

- El cumplimiento de SQL es necesario: dado que MySQL no intenta implementar el estándar SQL completo, esta herramienta no es completamente compatible con SQL. Si el cumplimiento de SQL completo o casi completo es imprescindible para su caso de uso, es posible que desee utilizar un DBMS más compatible.

- Concurrencia y grandes volúmenes de datos: aunque MySQL generalmente funciona bien con operaciones de lectura intensa, las lecturas y escrituras simultáneas pueden ser problemáticas. Si su aplicación tendrá muchos usuarios escribiendo datos a la vez, otro RDBMS como PostgreSQL podría ser una mejor opción de base de datos.

PostgreSql

PostgreSQL, también conocido como Postgres, se anuncia a sí misma como "la base de datos relacional de código abierto más avanzada del mundo". Fue creado con el objetivo de ser altamente extensible y compatible con los estándares. PostgreSQL es

una base de datos relacional de objetos, lo que significa que, aunque es principalmente una base de datos relacional, también incluye características, como la herencia de tablas y la sobrecarga de funciones, que se asocian más a menudo con las bases de datos de objetos.

Postgres es capaz de manejar de manera eficiente múltiples tareas al mismo tiempo, una característica conocida como concurrencia. Lo logra sin bloqueos de lectura gracias a su implementación de Multiversion Concurrency Control (MVCC), que asegura la atomicidad, consistencia, aislamiento y durabilidad de sus transacciones, también conocido como cumplimiento ACID ((sigla en inglés para Atomicidad, Consistencia, Aislamiento y Durabilidad)).

PostgreSQL no se usa tan ampliamente como MySQL, pero todavía hay una serie de herramientas y bibliotecas de terceros diseñadas para simplificar el trabajo con PostgreSQL, incluidos **pgAdmin** y **Postbird**.

Tipos de datos soportados por PostgreSQL

PostgreSQL admite tipos de datos numéricos, de cadena y de fecha y hora como MySQL. Además, admite tipos de datos para formas geométricas, direcciones de red, cadenas de bits, búsquedas de texto y entradas JSON, así como varios tipos de datos especiales.

Tipos numéricos:

- **bigint** Un entero de 8 bytes con signo.
- **bigserial** Un entero de 8 bytes que aumenta automáticamente.
- **double precision**: Un número de coma flotante de precisión doble de 8 bytes.
- **integer**: Un entero de 4 bytes con signo.
- **numeric o decimal**: Un número de precisión seleccionable, recomendado para su uso en casos donde la exactitud es crucial, como cantidades monetarias.
- **real**: Un número de coma flotante de precisión simple de 4 bytes.
- **smallint**: Un entero de 2 bytes con signo.
- **smallserial**: Un entero de 2 bytes que aumenta automáticamente.

- **serial**: Un entero de 4 bytes que aumenta automáticamente.

Tipos de caracteres:

- **character**: Una cadena de caracteres con una longitud fija especificada.
- **character varying o varchar**: Una cadena de caracteres con una longitud variable pero limitada.
- **text**: Una cadena de caracteres de una longitud ilimitada variable.

Tipos de fecha y hora:

- **date**: Una fecha del calendario que consta de día, mes y año.
- **interval**: Un intervalo de tiempo.
- **time o time without time zone**: Una hora del día, sin incluir la zona horaria.
- **time with time zone**: Una hora del día, incluida la zona horaria.
- **timestamp o timestamp without time zone**: Una fecha y hora, sin incluir la zona horaria.
- **timestamp with time zone**: Una fecha y hora, incluida la zona horaria.

Tipos geométricos:

- **box**: Una caja rectangular en un plano.
- **circle**: Un círculo en un plano.
- **líne**: Una línea infinita en un plano.
- **lseg**: Un segmento de línea en un plano.
- **path**: Una trayectoria geométrica en un plano.
- **point**: Un punto geométrico en un plano.

- **polygon**: Una trayectoria geométrica cerrada en un plano.

Tipos de direcciones de red:

- **cidr**: Una dirección de red IPv4 o IPv6.
- **inet**: Una dirección de host IPv4 o IPv6.
- **macaddr**: Una dirección de control de acceso a medios (MAC).

Tipos de cadenas de bits:

- **bit**: Una cadena de bits de longitud fija.
- **bit varying**: Una cadena de bits de longitud variable.

Tipos de búsqueda de texto:

- **tsquery**: Una consulta de búsqueda de texto.
- **tsvector**: Un documento de búsqueda de texto.

Tipos de JSON:

- **json**: Datos JSON textuales.
- **jsonb**: Datos JSON binarios descompuestos.

Otros tipos de datos:

- **boolean**: Un booleano lógico, que representa verdadero o falso.
- **bytea**: Abreviatura de "arreglo de bytes", este tipo se utiliza para datos binarios.
- **money**: Una cantidad de moneda.
- **pg_lsn**: Un número de secuencia de registro de PostgreSQL.

- **txid_snapshot**: Una ID de transacción a nivel de usuario.

- **uuid**: Un identificador universalmente único.

- **xml**: datos XML.

Ventajas de PostgreSQL

- Cumplimiento de SQL: más que SQLite o MySQL, PostgreSQL tiene como objetivo adherirse estrechamente a los estándares SQL. De acuerdo con la documentación oficial de PostgreSQL, PostgreSQL admite 160 de las 179 funciones necesarias para el pleno cumplimiento de SQL: 2011, además de una larga lista de funciones opcionales.

- Código abierto e impulsado por la comunidad: un proyecto totalmente de código abierto, el código fuente de PostgreSQL es desarrollado por una comunidad grande y dedicada. De manera similar, la comunidad de Postgres mantiene y contribuye con numerosos recursos en línea que describen cómo trabajar con el DBMS, incluida la documentación oficial, la wiki de PostgreSQL y varios foros en línea.

- Extensible: los usuarios pueden extender PostgreSQL programáticamente y sobre la marcha a través de su operación basada en catálogos y su uso de carga dinámica. Se puede designar un archivo de código de objeto, como una biblioteca compartida, y PostgreSQL lo cargará según sea necesario.

Desventajas de PostgreSQL

- Rendimiento de la memoria: para cada nueva conexión de cliente, PostgreSQL bifurca un nuevo proceso. A cada nuevo proceso se le asignan aproximadamente 10 MB de memoria, que pueden acumularse rápidamente para bases de datos con muchas conexiones. En consecuencia, para operaciones simples de lectura pesada, PostgreSQL suele ser menos eficaz que otros RDBMS, como MySQL.

- Popularidad: aunque se ha utilizado más ampliamente en los últimos años, PostgreSQL históricamente se quedó atrás de MySQL en términos de popularidad. Una consecuencia de esto es que todavía hay menos herramientas de terceros que pueden ayudar a administrar una base de datos PostgreSQL. De manera similar, no hay tantos administradores de bases de datos con experiencia en la gestión de una base de datos de Postgres en comparación con aquellos con experiencia en MySQL.

Cuándo usar PostgreSQL

- La integridad de los datos es importante: PostgreSQL ha sido totalmente compatible con ACID desde 2001 e implementa el control de moneda de múltiples versiones para garantizar que los datos permanezcan consistentes, lo que lo convierte en una opción sólida de RDBMS cuando la integridad de los datos es crítica.

- Integración con otras herramientas: PostgreSQL es compatible con una amplia gama de lenguajes y plataformas de programación. Esto significa que, si alguna vez necesita migrar su base de datos a otro sistema operativo o integrarla con una herramienta específica, probablemente será más fácil con una base de datos PostgreSQL que con otro DBMS.

- Operaciones complejas: Postgres admite planes de consulta que pueden aprovechar múltiples CPUs para responder consultas con mayor velocidad. Esto, junto con su fuerte soporte para múltiples escrituras concurrentes, lo convierte en una excelente opción para operaciones complejas como almacenamiento de datos y procesamiento de transacciones en línea.

Cuándo no usar PostgreSQL

- La velocidad es imperativa: a expensas de la velocidad, PostgreSQL se diseñó teniendo en cuenta la extensibilidad y la compatibilidad. Si su proyecto requiere las operaciones de lectura más rápidas posibles, es posible que PostgreSQL no sea la mejor opción de DBMS.

- Configuraciones simples: debido a su gran conjunto de características y su fuerte adherencia al SQL estándar, Postgres puede ser excesivo para configuraciones simples de bases de datos. Para operaciones de lectura intensiva donde se requiere velocidad, MySQL suele ser una opción más práctica.

- Replicación compleja: aunque PostgreSQL proporciona un sólido soporte para la replicación, sigue siendo una característica relativamente nueva y algunas configuraciones, como una arquitectura primaria-primaria, solo son posibles con extensiones. La replicación es una característica más madura en MySQL y muchos usuarios ven que la replicación de MySQL es más fácil de implementar, particularmente para aquellos que carecen de la experiencia necesaria en administración de sistemas y bases de datos.

Hoy en día, SQLite, MySQL y PostgreSQL son los tres sistemas de administración de bases de datos relacionales de código abierto más populares del mundo. Cada uno tiene sus propias características y limitaciones únicas, y sobresale en escenarios particulares. Hay bastantes variables en juego cuando se decide por un RDBMS, y la elección rara vez es tan simple como elegir el más rápido o el que tiene más funciones. La próxima vez que necesite una solución de base de datos relacional, asegúrese de investigar estas y otras herramientas en profundidad para encontrar la que mejor se adapte a sus necesidades.

Oracle

Mencionamos acá por su popularidad a la base de datos Oracle que es una base de datos patentada producida y comercializada por Oracle Corporation (no es open source). Hay varias versiones diferentes, desde la basada en la nube hasta una libre para desarrollar/implementar/distribuir; Edición Oracle Database Express se puede conseguir libremente. Pero Oracle Express Edition tiene características muy limitadas en comparación con, por ejemplo, MySQL. Para obtener funciones amplias, debe comprar Oracle Standard Edition u Oracle Enterprise Edition. Usualmente empresas multinacionales, bancos, grandes empresas industriales, etc. usan Oracle dado su costo, porque además ya existía en el mercado mucho antes de las alternativas open-source, asi que muchos quedaron dependiendo de esta alternativa. Por otro lado, también está el soporte dedicado que da Oracle, mediante técnicos especialistas, call centers, etc.

El concepto de ORM

Si uno es relativamente nuevo en el mundo de la programación, términos como Object-Relational-Mapper pueden sonar intimidantes. Lo bueno de los ORM es que en realidad facilitan la escritura de código una vez que se dominan (normalmente).
Antes de hablar sobre lo que es un mapeador relacional de objetos (ORM), sería mejor hablar primero del mapeo relacional de objetos como concepto.

A menos que haya trabajado exclusivamente con bases de datos NoSQL, es probable que haya escrito una buena cantidad de consultas SQL. Por lo general, se ven así:

```
SELECT * FROM usuarios WHERE email = 'test@test.com';
```

El mapeo relacional de objetos es la idea de poder escribir consultas como la anterior, así como otras mucho más complicadas, utilizando el paradigma orientado a objetos en un lenguaje de programación, en nuestro caso Python. En pocas palabras, estamos

tratando de interactuar con nuestra base de datos utilizando nuestro lenguaje de elección en lugar de SQL.

Aquí es donde entra el mapeador relacional de objetos. Cuando la mayoría de la gente dice "ORM", se está refiriendo a una biblioteca que implementa esta técnica. Por ejemplo, la consulta anterior ahora se vería así en javascript:

```
var orm = require('generic-orm-libarry');
var user = orm("users").where({ email: 'test@test.com' });
```

Como se puede ver, estamos usando una biblioteca ORM imaginaria para ejecutar exactamente la misma consulta, excepto que podemos escribirla en JavaScript (o en cualquier idioma que estemos usando). Podemos usar los lenguajes que conocemos, y también abstraernos de la complejidad de interactuar con una base de datos. Como con cualquier técnica, existen ventajas y desventajas que deben tenerse en cuenta al utilizar un ORM.

Ventajas

- Puede escribir en el lenguaje que ya está utilizando. Si somos honestos, probablemente no seamos los mejores escribiendo declaraciones SQL. SQL es un lenguaje ridículamente poderoso, pero la mayoría de nosotros no escribimos en él con frecuencia. Sin embargo, tendemos a ser mucho más fluidos en un lenguaje u otro y ser capaces de aprovechar esa fluidez es increíble.

- Hace abstracción del sistema de base de datos para que cambiar de MySQL a PostgreSQL por ejemplo, o cualquier versión que prefiera, sea fácil.

- Dependiendo del ORM, obtendrá muchas funciones avanzadas listas para usar, como soporte para transacciones, agrupación de conexiones, migraciones, poblar tablas y todo tipo de otras ventajas.

- Muchas de las consultas que escriba funcionarán mejor que si las escribiera usted mismo en SQL.

Desventajas

- Si es un maestro en SQL, probablemente pueda obtener consultas más eficaces escribiéndolas usted mismo.

- Hay que invertir cierto tiempo en aprender a usar cualquier ORM dado.

- La configuración inicial de un ORM puede ser difícil.

- Como desarrollador, es importante comprender lo que está sucediendo "bajo el capó". Dado que los ORM pueden servir como una muleta para evitar comprender las bases de datos y SQL, puede convertirlo en un desarrollador más débil en esa parte del montón de habilidades necesarias para ser un buen desarrollador.

Se puede aprovechar mucho el tiempo usando un ORM. Como desarrollador full-stack trabajando en equipos pequeños donde no hay un gurú de bases de datos dedicado a eso específicamente, simplifica enormemente el trabajo con la capa de datos.

Aunque algunas personas piensan en el proceso de configuración de un ORM como un problema, existen algunas alternativas que se pueden configurar con diferentes bases de datos para diferentes entornos utilizando una interfaz coherente.

Migraciones en Django

Desde la versión 1.7, Django viene con soporte integrado para migraciones de bases de datos. En Django, las migraciones de bases de datos suelen ir de la mano de los modelos: cada vez que codifica un nuevo modelo, también genera una migración para crear la tabla necesaria en la base de datos. Sin embargo, las migraciones pueden hacer mucho más.

Si usted es nuevo en Django o en el desarrollo web en general, es posible que no esté familiarizado con el concepto de migraciones de bases de datos y que no parezca obvio por qué son una buena idea.

Como hemos visto Django está diseñado para trabajar con una base de datos relacional, almacenada en un sistema de administración de base de datos relacional como PostgreSQL, MySQL o SQLite. La descripción de todas las tablas con sus columnas y sus respectivos tipos de datos se denomina usualmente esquema de base de datos (schema).

Adelantándonos un poco veremos que un "modelo", que en términos prácticos puede ser analogado a una tabla en la base datos se puede definir, de la siguiente forma:

```
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)
```

Pero simplemente definir una clase modelo en un archivo Python no hace que una tabla de base de datos aparezca mágicamente de la nada. Crear las tablas de la base de datos para almacenar sus modelos Django es el trabajo de una "migración de base de datos". Además, cada vez que realiza un cambio en sus modelos, como agregar un campo, la base de datos también debe cambiarse. Las migraciones también se encargan de eso.

Sin migraciones, tendría que conectarse a su base de datos y escribir un montón de comandos SQL o usar una herramienta gráfica como pgAdmin para modificar el esquema de la base de datos cada vez que quisiera cambiar la definición de su modelo.

En Django, las migraciones se escriben principalmente en Python, por lo que no es necesario trabajar con SQL a menos que se le presenten casos donde realmente lo requiera. Crear un modelo y luego escribir SQL para crear las tablas de la base de datos sería repetitivo. Las migraciones se generan a partir de sus modelos, asegurándose de que no se repita (DRY).

Por lo general, cuando se trabaja con un equipo de desarrollo existen varias instancias de la base de datos, por ejemplo, una base de datos para cada desarrollador de su equipo, una base de datos para pruebas y una base de datos con datos en vivo. Sin migraciones, tendrá que realizar cualquier cambio de esquema en cada una de sus bases de datos y tendrá que realizar un seguimiento de los cambios que ya se han realizado en qué base de datos. Con Django Migrations, puede mantener fácilmente varias bases de datos sincronizadas con sus modelos.

Un sistema de control de versiones, como Git, es excelente para el código, pero no tanto para los esquemas de bases de datos. Como las migraciones son simples en Python en Django, puede ponerlas en un sistema de control de versiones como cualquier otro código. A estas alturas, es de esperar que esté convencido de que las migraciones son una herramienta útil y poderosa.

Pronto vamos a crear un nuevo proyecto django, para ilustrar cumchos de los conceptos de esta clase. En relación a la migraciones, usted verá dos comandos típicos para realizarlas:

- `$ python manage.py makemigrations [opcional tabla]`
- `$ python manage.py migrate`

Esto crea el archivo de migraciones que le indica a Django cómo crear las tablas de la base de datos para los modelos definidos en su aplicación. Echemos otro vistazo al árbol de directorios del proyecto que hemos estado trabajando:

```
.
└── sitio
    ├── app
    │   ├── __init__.py
    │   ├── admin.py
    │   ├── apps.py
    │   ├── migrations
    │   │   └── 0001_initial.py
    │   ...
    ...
```

Como puede ver, el directorio de migraciones contiene un archivo: **0001_initial.py**. Tal archivo se generó porque aplicamos los comandos para migraciones descritos más arriba.

Ahora usaremos la utilidad **dbshell** para ver qué pasa en la base de datos misma. Como no hemos configurado los **settings** de la base de datos Django asume SQLite que es la alternativa por default.

Nota: estamos inspeccionando estos conceptos, pero pronto vamos a crear un proyecto nuevo para ilustrar el trabajo con Postgres, modelos, y trabajo en base de datos de forma más profunda.

```
$ python manage.py dbshell
SQLite version 3.33.0 2020-08-14 13:23:32
Enter ".help" for usage hints.
sqlite> .tables
sqlite>
```

Naturalmente no tenemos tablas ni información relevante en nuestra base de datos. Apliquemos las migraciones

```
$ python manage.py makemigrations
Migrations for 'app':
  app/migrations/0003_auto_20200906_0738.py
    - Alter field nombre on app


$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, app, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying     auth.0001_initial...     OK
  Applying admin.0001_initial... OK
  Applying     admin.0002_logentry_remove_auto_add...     OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying app.0001_initial... OK
  Applying app.0002_auto_20200901_1702... OK
  Applying app.0003_auto_20200906_0738... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
```

```
  Applying  auth.0004_alter_user_username_opts...  OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying   auth.0008_alter_user_username_max_length...   OK
  Applying   auth.0009_alter_user_last_name_max_length...   OK
  Applying     auth.0010_alter_group_name_max_length...     OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
```

Muchas cosas pasaron. Según el resultado, la migración se ha aplicado correctamente. Pero, ¿de dónde vienen todas las demás migraciones?

¿Recuerda la configuración **INSTALLED_APPS**? Algunas de las otras aplicaciones enumeradas allí también vienen con migraciones, y el comando de administración de migraciones aplica las migraciones para todas las aplicaciones instaladas de forma predeterminada.

Miremos la base de datos una vez más:

```
o$ python manage.py dbshell
SQLite version 3.33.0 2020-08-14 13:23:32
Enter ".help" for usage hints.
sqlite> .tables
app_app
auth_user_user_permissions
auth_group                                 django_admin_log
auth_group_permissions         django_content_type
auth_permission            django_migrations
auth_user                  django_session
auth_user_groups
```

Ahora hay varias tablas. Sus nombres te dan una idea de su propósito. La migración que generó en el paso anterior ha creado la tabla **app_app**. Inspeccionémosla con el comando **.schema**:

```
sqlite>   .schema   --indent   app_app
CREATE TABLE IF NOT EXISTS "app_app"(
  "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
  "email" varchar(254) NOT NULL,
  "nombre" varchar(64) NULL
);
```

El comando .schema imprime la instrucción CREATE que ejecutaría para crear la tabla. El parámetro --indent lo formatea muy bien. Incluso si no está familiarizado con la sintaxis SQL, puede ver que el esquema de la tabla refleja los campos del modelo:

```
class App(models.Model):
    nombre = models.CharField(max_length=64, blank=True,
                              null=True,
validators=[validate_estimada])        email = models.EmailField()
```

Hay una columna para cada campo y un **id** de columna adicional para la llave primaria, que Django crea automáticamente a menos que especifique explícitamente una clave principal en su modelo.

Esto es lo que sucede si vuelve a ejecutar el comando migrate:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, app, auth, contenttypes, sessions
Running migrations:
  No migrations to apply.
```

Nada, Django recuerda qué migraciones ya se han aplicado y no intenta volver a ejecutarlas.

Vale la pena señalar que también puede limitar el comando de administración de migraciones a una sola aplicación:

```
$ python manage.py migrate app
Operations to perform:
  Apply all migrations: app
Running migrations:
  No migrations to apply.
```

Como puede ver, Django ahora solo aplica migraciones para la aplicación **app.**

Cuando ejecute las migraciones por primera vez, es una buena idea aplicar todas las migraciones para asegurarse de que su base de datos contenga las tablas necesarias para las funciones que usted podría considerar como obvias, autenticación de usuarios y las sesiones.

Sus modelos no están escritos para nunca cambiar. Sus modelos cambiarán a medida que su proyecto Django obtenga más funciones. Puede agregar o eliminar campos o cambiar sus tipos y opciones.

Cuando cambia la definición de un modelo, las tablas de la base de datos utilizadas para almacenar estos modelos también deben cambiarse. Si las definiciones de su modelo no coinciden con el esquema de su base de datos actual, lo más probable es que se encuentre con un **django.db.utils.OperationalError.**

Entonces, cómo se cambian las tablas de la base de datos: creando y aplicando una migración.

Supongamos que mientras probábamos nuestra **app** nos dimos cuenta de que **64** caracteres no eran suficiente para almacenar nombre, por lo mismo decidimos ampliar tal campo a **128**, en el modelo, cambiamos la **max_length** de `nombre.

```
class App(models.Model):
    nombre = models.CharField(max_length=128, blank=True,
                              null=True,
validators=[validate_estimada])
    email = models.EmailField()
```

Sin migraciones, usted tendría que buscar la forma y la sintaxis adecuada para ampliar ese campo con instrucciones en SQL. Pero mejor dejémoslo a Django.

```
$ python manage.py makemigrations
Migrations for 'app':
  app/migrations/0004_auto_20200906_0815.py
    - Alter field nombre on app
```

Ahora apliquemos esta migración a su base de datos:

```
$ python manage.py migrate app
Operations to perform:
  Apply all migrations: app
Running migrations:
  Applying app.0004_auto_20200906_0815... OK
```

La migración se aplicó correctamente, por lo que puede usar **dbshell** para verificar que los cambios surtieron efecto:

```
$ python manage.py dbshell
SQLite version 3.33.0 2020-08-14 13:23:32
Enter ".help" for usage hints.
sqlite> .schema --indent app_app
CREATE TABLE IF NOT EXISTS "app_app"(
  "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
  "email" varchar(254) NOT NULL,
  "nombre" varchar(128) NULL
);
```

Si compara este nuevo esquema con el que vimos más arriba se dará cuenta de que el campo **nombre** ahora acepta **128** caracteres en lugar de **64**

Si desea saber qué migraciones existen en un proyecto de Django, no tiene que buscar en los directorios de migraciones de sus aplicaciones instaladas. Puede utilizar el comando **showmigrations**:

```
o$ python manage.py showmigrations
Admin
 [X] 0001_initial
 [X] 0002_logentry_remove_auto_add
 [X] 0003_logentry_add_action_flag_choices
app
 [X] 0001_initial
 [X] 0002_auto_20200901_1702
 [X] 0003_auto_20200906_0738
 [X] 0004_auto_20200906_0815
auth
 [X] 0001_initial
...
Contenttypes
 [X] 0001_initial
 [X] 0002_remove_content_type_name
Sessions
 [X] 0001_initial
```

Aquí se enumeran todas las aplicaciones del proyecto y las migraciones asociadas con cada aplicación. Además, pondrá una **X** junto a las migraciones que ya se han aplicado.

Para nuestro pequeño ejemplo, el comando **showmigrations** no es particularmente interesante, pero es útil cuando comienza a trabajar en una base de código existente o trabaja en un equipo donde no es la única persona que agrega migraciones.

Anular la aplicación de migraciones

Ahora sabe cómo realizar cambios en el esquema de su base de datos creando y aplicando migraciones. En algún momento, es posible que desee deshacer los cambios y volver a un esquema de base de datos anterior porque:

- Quiere probar una migración que escribió un colega.
- Darse cuenta de que un cambio que hizo fue una mala idea.
- Trabajar en múltiples funciones con diferentes cambios en la base de datos en paralelo.
- Desea restaurar una copia de seguridad que se creó cuando la base de datos aún tenía un esquema anterior.

Afortunadamente, las migraciones no tienen por qué ser una calle de un solo sentido. En muchos casos, los efectos de una migración se pueden deshacer si no se aplica una migración. Para anular la aplicación de una migración, debe llamar a **migrate** con el

nombre de la aplicación y el nombre de la migración antes de la migración que desea anular la aplicación.

Supongamos que ahora nos dimos cuenta, que **64** caracteres era suficiente para el campo, y no deberíamos haber cambiado el campo **nombre** de nuestra tabla.

Si mira más arriba, el listado de migraciones nos damos cuenta de que previo a la última migración **(0004_auto_20200906_0815**) teníamos la migración **0003_auto_20200906_0738**, entonces hacemos:

```
$ python manage.py migrate app 0003_auto_20200906_0738
Operations to perform:
  Target specific migration: 0003_auto_20200906_0738, from app
Running migrations:
  Rendering model states... DONE
  Unapplying app.0004_auto_20200906_0815... OK
```

Podemos volver a revisar nuestra tabla con **dbshell**

```
 python manage.py dbshell
SQLite version 3.33.0 2020-08-14 13:23:32
Enter ".help" for usage hints.
sqlite> .schema --indent app_app
CREATE TABLE IF NOT EXISTS "app_app"(
  "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
  "nombre" varchar(64) NULL,
  "email" varchar(254) NOT NULL
);
```

Una vez más **nombre** acepta solo **64** caracteres. La migración no se aplicó, lo que significa que los cambios en la base de datos se han revertido.

Importante: Anular la aplicación de una migración no elimina su archivo de migración. La próxima vez que ejecute el comando **migrate**, la migración se aplicará nuevamente. No confunda la anulación de la aplicación de migraciones con la operación de deshacer a la que está acostumbrado desde su editor de texto favorito. No todas las operaciones de la base de datos se pueden revertir por completo. Si elimina un campo de un modelo, crea una migración y la aplica, Django eliminará la columna correspondiente de la base de datos. Si no aplica esa migración, se volverá a crear la columna, pero no se recuperarán los datos que se almacenaron en esa columna.

Nombrando las migraciones

En el ejemplo anterior, Django creó un nombre para la migración basado en la marca de tiempo, algo como ***0004_auto_20200906_0815.py**. Si no está satisfecho con eso,

puede usar el parámetro --name para proporcionar un nombre personalizado (sin la extensión .**py**).

Para probarlo, primero debe eliminar la migración anterior. Ya la ha desactivado, por lo que puede eliminar el archivo de forma segura:

```
$ rm app/migrations/0004_auto_20200906_0815.py
```

Ahora podemos recrearla con un nombre más descriptivo (recuerde que en nuestro modelo el campo **nombre** sigue especificando **128** caracteres:

```
$ python manage.py makemigrations app --name cambio_nombre_128
Migrations for 'app':
  app/migrations/0004_cambio_nombre_128.py
    - Alter field nombre on app
```

Si revisamos nuestro directorio **migrations** encontraremos un archivo con un nombre como **0004_cambio_nombre_128.py**

6.1.2.- Sintaxis de consultas en ORM

Por ahora no hemos ingresado datos a nuestra DB, lo podemos comprobar con **dbshell**

```
sqlite> SELECT * FROM app_app;
sqlite>
```

Ingresemos algunos datos utilizando la consola interactiva. Importante **shell** and **dbshell** son y cumplen funciones completamente diferentes.

Primero importamos el modelo, luego creamos el objeto y por último lo guardamos en la base de datos (a veces se usa el término **persistence** para este paso final).

```
$ python manage.py shell
>>> from app.models import App
>>>      sujeto01      =      App(nombre='Juanito      Escarcha',
email='juanito@escarcha.com')>>> sujeto01.save()
```

Examinemos ahora con dbshell que tenemos en nuestra base de datos:

```
$ python manage.py dbshell
...
sqlite>    SELECT    *    FROM    app_app;
1|Juanito Escarcha|juanito@escarcha.com
...
```

Vemos que J**uanito Escarcha** ya está en nuestra base de datos. Pero que sucede si tuvimos un problema con 'Juanito' y preferimos tratar con su hermano 'Rodrigo Escarcha', simplemente podemos actualizar la base de datos, usando **shell:**

```
>>> from app.models import App
>>> sujeto01 = App.objects.filter(nombre='Juanito Escarcha')
>>> sujeto01[0].nombre'Juanito Escarcha'
>>>                                    App.objects.filter(nombre='Juanito
Escarcha').update(nombre='Rodrigo Escarcha')
```

Veamos que pasó a nivel de base de datos en **dbshell**, conservamos el email, pero cambiamos el nombre

```
sqlite>    SELECT    *    FROM    app_app;
1|Rodrigo Escarcha|juanito@escarcha.com
```

Qué sucede si no logramos llegar a nada productivo con Juanito ni Rodrigo, ya que Juanito y Rodrigo entraron en conflictos por el email. Como todo se puso muy complicado, nosotros ya no queremos tener nada que ver con los Escarcha, tendremos que borrar la columna en **shell**:

```
>>> App.objects.filter(nombre='Rodrigo Escarcha').delete()
(1, {'app.App': 1})
```

En **dbshell**

```
sqlite> SELECT * FROM app_app;
sqlite>
```

Acá solo mostramos un barniz de lo que se puede hacer manipulando objetos y como se transmiten esos cambios en la base de datos real. Más adelante veremos el tema con más profundidad y con los ejercicios debería quedar más claro de forma práctica. Enumerar todas las posibilidades es imposible, pero la documentación en este sentido es muy buena https://docs.djangoproject.com/en/3.1/topics/db/queries/

6.1.2.1. Querys SQL en Django

Ahora bien, qué sucede si en realidad no encontramos la función o método adecuado para realizar lo que queremos en Django, no obstante, en SQL sabemos muy bien lo que hay que hacer, y lidiar con objetos y métodos que no conocemos nos tomaría a la larga más tiempo, en ese caso podemos hacer las consultas directamente en SQL dentro de Django.

Supongamos que luego de una conversación bastante profunda decidimos que Juanito Escarcha es un buen tipo y lo queremos re-incorporar a nuestra base de datos. Vamos a **shell** para ver esta situación:

```
>>> from app.models import App
>>> from django.db import connection
>>> cursor = connection.cursor()
>>> cursor.execute("INSERT into app_app (nombre, email) VALUES
('Juanito Escarcha', 'familia@escarcha.com')")
```

En **dbshell**

```
sqlite>   SELECT   *   FROM   app_app;
5|Juanito Escarcha|familia@escarcha.com
sqlite>
```

Una vez más tuvimos problemas con Juanito, esta vez se pasó... pero necesitamos arreglar el entuerto y Rodrigo que parece más razonable, con los mismos **imports** anteriores:

```
>>> cursor.execute("UPDATE app_app SET nombre = 'Rodrigo Escarcha'
WHERE nombre = 'Juanito Escarcha'")
```

En **dbshell**

```
sqlite>   SELECT   *   FROM   app_app;
5|Rodrigo Escarcha|familia@escarcha.com
```

Definitivamente los Escarcha son unos estafadores, así que los eliminaremos de una buena vez de nuestros datos, con ellos ya no se puede negociar sin salir magullados, no queremos tener nada que ver con algún Escarcha nunca jamás:

```
>>> cursor.execute("DELETE FROM app_app WHERE nombre LIKE
'%Escarcha%'")
```

En **dbshell**

```
sqlite> SELECT * FROM app_app;
sqlite>
```

6.1.2.2. Soporte para bases NoSql

Según la documentación oficial Django no soporta oficialmente bases de datos NoSQL, no obstante, existen proyectos paralelos en tal dirección: https://docs.djangoproject.com/en/3.1/faq/models/#does-django-support-nosql-databases

Para este apartado vamos a trabajar con MongoDB https://www.mongodb.com/ así es que, si eventualmente quiere seguirnos, necesitará instalar tal software. Esta

sección solo da un pequeño barniz acerca de NoSQL (not only SQL). Las bases de datos NoSQL utilizan una variedad de modelos de datos para acceder y administrar datos. Estos tipos de bases de datos están optimizados específicamente para aplicaciones que requieren un gran volumen de datos, baja latencia y modelos de datos flexibles, que se logran al relajar algunas de las restricciones de coherencia de datos de otras bases de datos. Las bases de datos NoSQL no son relacionales.

Si instaló correctamente mongo, debería ser capaz de ver algo similar a esto:

```
$ mongo
MongoDB shell version v4.4.0
connecting to: mongodb://127.0.0.1:27017/
...
```

Como verá estamos usando la versión 4.4.0 para este ejemplo. Para probar su instalación, puede tratar algunos comandos útiles (ya dentro de mongodb) como **db.help()**; **db.stats()**

Como Django no soporta oficialmente bases de datos NoSQL, usaremos un conector no oficial llamado **djongo** https://github.com/nesdis/djongo para instalarlo basta con ejecutar:

```
$ pip install djongo
```

Una vez que sjongo está instalado vamos a cambiar los settings de django para usar otra base de datos (recuerde que estábamos usando SQLite), así en **settings.py** modificamos **DATABASE**

```
...
DATABASES = {
    'default': {
        'ENGINE': 'djongo',
        'NAME': 'mymongodb',
        'CLIENT': {
            'host': '127.0.0.1'
        }
    }
}
...
```

Y ahora vamos a aplicar las migraciones para que sean captadas por nuestra nueva base de datos llamada **mymongodb**

```
django01/sitio$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, app, auth, contenttypes, sessions
Running migrations:
```

```
This version of djongo does not support "NULL, NOT NULL column
validation check" fully. Visit https://www.patreon.com/nesdis
  Applying contenttypes.0001_initial...This version of djongo does
not support "schema validation using CONSTRAINT" fully. Visit
https://www.patreon.com/nesdis
 OK
  Applying auth.0001_initial...
This version of djongo does not support "schema validation using KEY"
fully. Visit https://www.patreon.com/nesdisThis version of djongo
does not support "schema validation using REFERENCES" fully. Visit
https://www.patreon.com/nesdis
 OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
...
...
```

La información mostrada en pantalla va a ser bastante extensa, arriba solo mostramos un fragmento.

En el shell de mongo para usar nuestra base datos hacemos:

```
> use mymongodb
switched to db mymongodb
```

Ahora examinemos que tablas tiene nuestra base de datos con **show collections**:

```
> show collections
__schema
app_app
auth_group
auth_group_permissions
auth_permission
auth_user
auth_user_groups
auth_user_user_permissions
django_admin_log
django_content_type
django_migrations
django_session
```

Tenemos los documentos (un equivalente a tablas en bases de datos relacionales) creados por Djago y nuestro **app_app**

Observemos que hay en nuestro **app_app**

```
> db.app_app.find().pretty()
>
```

Naturalmente está vacío.

Utilicemos la consola de Django para crear un objeto y persistirlo a nuestra base de datos MongoDB. Hacemos exactamente lo mismo que veníamos haciendo con SQLite (que es relacional)

```
$ python manage.py shell
Python 3.8.5 (default, Jul 21 2020, 10:42:08)
...
>>> from app.models import App
>>> sujeto01 = App(nombre='Juanito Escarcha',
email='juanito@escarcha.com')
>>> sujeto01.save()
```

Una vez más hemos creado a Juanito Escarcha, si ahora nos movemos al shell de mongo para inspeccionar veremos

```
> db.app_app.find().pretty()
{
        "_id" : ObjectId("5f5607e503f6ada861e468e0"),
        "id" : 1,
        "nombre" : "Juanito Escarcha",
        "email" : "juanito@escarcha.com"
}
```

Este es el documento JSON en el cual se guarda nuestra información. Como dijimos esto es solo una pequeñísima muestra de NoSQL, ya que, nuestro foco (y la mayoría de los desarrollos web) utilizan bases de datos relacionales.

6.1.3. 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.