


Open · Opened 2 days ago by  **Rubén Montero**

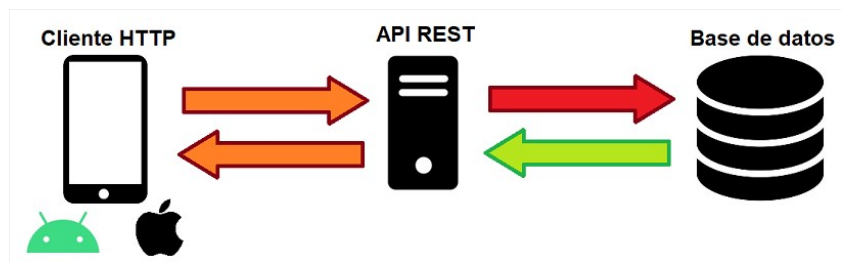
Ladrillos para el muro

Resumen

- Hablaremos de *mapeo objeto-relacional* (ORM)
- Crearemos nuestro primer *modelo* en `models.py`

Descripción

Como ya hemos visto con anterioridad, estamos trabajando en la creación de APIs REST que *reciben* peticiones de los usuarios y las *traducen* a modificaciones en una base de datos:



¿Otra vez SQL?

¡No!

¿No?

No.

Algo nuevo.

Mapeo objeto-relacional (ORM)

Es una técnica de acceso a datos implementada por *frameworks* como [Django](#), [Hibernate](#), [Laravel](#) y [muchos otros...](#)

¿En qué consiste?

En usar **clases** de *programación orientada a objetos* (POO), como **sinónimos** de las **tablas** de una base de datos SQL.

Veámoslo con un ejemplo.

En nuestro muro

Queremos tener **dos** tablas en la base de datos *relacional*.

Podrían verse, por ejemplo, así:

- **Entry** : Contiene *Entradas* publicadas en el muro

id	title	content	publication_date
1	Primera pintada en el muro	¡Ahí va esta obra de arte!	2022-07-13 10:25:00
2	Pole	Espero no haber fallado en hacer la primera publicación	2022-07-13 10:25:01
...

- **Comment** : Contiene *Comentarios* de las entradas.

id	content	entry
1	¡Pues sí que fallaste!	2

id	content	entry
2	Ohhhh... Sí que fallaste, lo siento	2
3	Bonita primera pintada :)	1

Entonces, el *mapeo objeto-relacional*...

Consistirá en tener **dos** clases. Cada una *representa* una tabla.

```
class Entry:
    # ...

class Comment:
    # ...
```

¿Para qué?

La idea es que **modificaremos la base de datos con código orientado a objetos** únicamente, sin usar SQL.

Por ejemplo, **crear** una nueva *instancia* de `Comment` será como hacer un SQL `INSERT`.

Pero, ¿para qué?

Para que el desarrollador **no** tenga que lidiar con **código SQL**.

En otras palabras, **trabajaremos contra una base de datos SQL sin escribir una línea de SQL**.

¡Suenan excelente!

Lo es.

Comencemos

[models.py](#)

Cada una de esas *clases* de las que hemos hablado arriba se denominan **modelo**.

En un proyecto Django, estas clases se encuentran **todas** en el archivo `models.py` de cada *app*.

Si abres `WallAPI/wallrest04app/models.py` verás:

```
from django.db import models

# Create your models here.
```

La documentación cuenta con un [ejemplo muy descriptivo](#) de cómo empezar.

Entry

Comenzaremos creando la clase `Entry`. **Debe** indicar `models.Model` entre los *paréntesis* (`()`), que es la sintaxis de Python para indicar [herencia](#).

```
from django.db import models

class Entry(models.Model):
    # ...
```

Los dos primeros **VARCHAR**

Comenzaremos especificando *dos* primeros atributos de tipo `models.CharField`, que es la *clase* Django correspondiente a un **VARCHAR**:

```
from django.db import models

class Entry(models.Model):
    title = models.CharField(max_length=160)
    content = models.CharField(max_length=5500)
```

¿Qué es `max_length`?

Un parámetro que debemos pasar en el constructor. Como sabes, los `VARCHAR` en SQL deben indicar un *tamaño máximo* (e.g.: `VARCHAR(100)`). Dicha restricción llega hasta nosotros de esta manera.

¿Los atributos no se declaraban en `__init__`?

Los *atributos de instancia* se declaran en `__init__`.

Los *atributos de clase* se declaran fuera. Por limitaciones de Python, los modelos de Django debemos plantearlos usando *atributos de clase*. No hay que preocuparse. Django se encargará de que cada *instancia* cuente con sus atributos *independientes*.

Resumen, en `models.py` los atributos debemos declararlos así.

Un campo para una fecha

¡Así que completemos nuestro `Entry` con un último *atributo* para la **fecha de publicación**!

```
from django.db import models

class Entry(models.Model):
    title = models.CharField(max_length=160)
    content = models.CharField(max_length=5500)
    publication_date = models.DateTimeField(auto_now=True)
```

Con el parámetro `auto_now=True` indicamos que, cuando se **guarde** una nueva instancia de `Entry`, queremos que la fecha/hora del servidor se asigne automáticamente.

Si quisiéramos almacenar únicamente una fecha (sin hora), podríamos usar `models.DateField`

Un momento, ¿y el `id`?

En Django **no necesitamos** indicar el ID de las tablas manualmente.

Entonces, ¿no hay `id`?

Sí que lo hay.

Django se encargará de que haya un ID tipo numérico autoincremental, **automáticamente**.

¿Modelo terminado?

¡Sí!

¡Enhorabuena! Has creado tu primer model ORM con Django.

Por último

Comprueba que tu código pasa el *test* asociado a la tarea.

Haz `commit` y `push` para subir los cambios al repositorio.



[Rubén Montero @ruben.montero](#) changed milestone to [%Sprint 4](#) 2 days ago