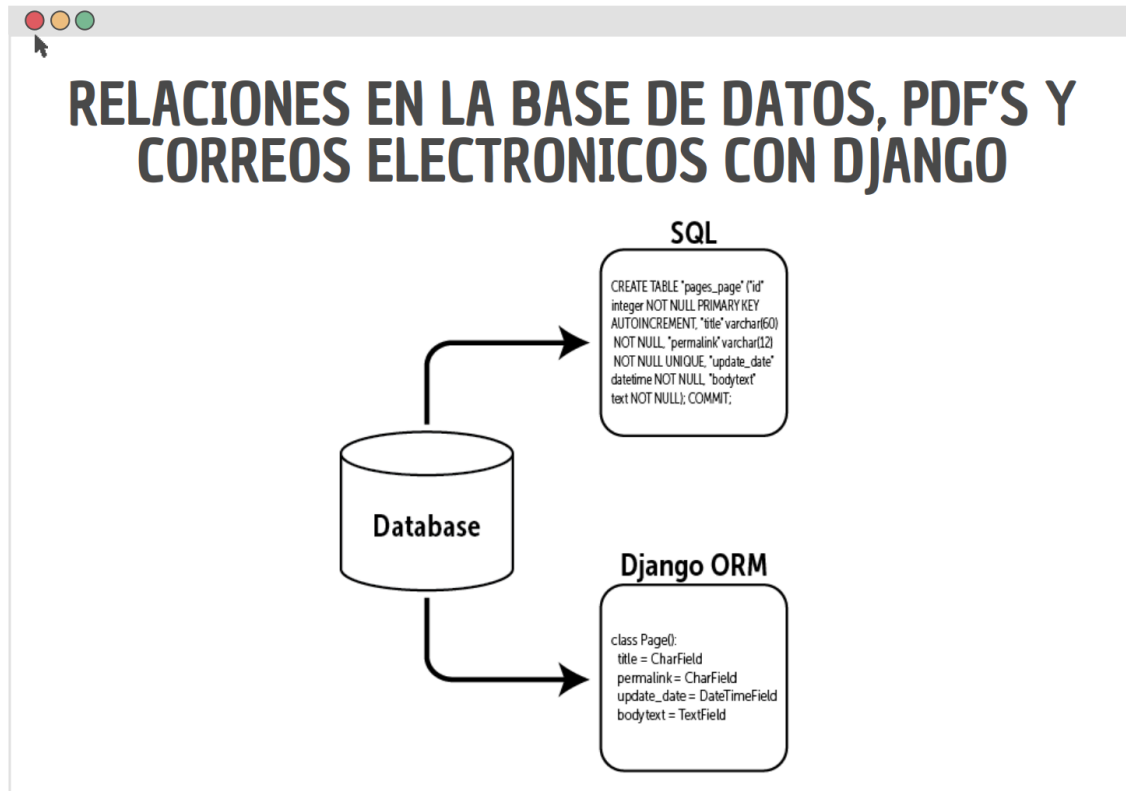




PROGRAMACIÓN WEB 2



Profesor(a):

Carlo Jose Luis Corrales Delgado

Estudiante:

Jorge Luis Mamani Huarsaya

Email:

jmamanihuars@unsa.edu.pe

Repositorio GitHub:

<https://github.com/jorghee/django-projects>

15 de junio, 2024

Tipos de relaciones en la base de datos con Django

El sistema de gestión de base de datos que se está utilizando en esta oportunidad es PostgreSQL, por lo tanto empezamos creando un proyecto `django_relationships` Django y dentro de este proyecto creamos la aplicación `relationships`. Antes de enfocarnos en la creación de modelos Django, nos dirigimos al archivo `settings.py` del proyecto donde modificamos la base de datos por defecto.

```
1 DATABASES = {
2     'default': {
3         'ENGINE': 'django.db.backends.postgresql',
4         'NAME': env('DB_NAME'),
5         'USER': env('DB_USER'),
6         'PASSWORD': env('DB_PASSWORD'),
7         'HOST': env('DB_HOST'),
8         'PORT': env('DB_PORT'),
9     }
10 }
```

Relación uno a muchos

Hemos creado 2 modelos en Django: `Language` y `Framework` donde podemos apreciar esta relación, pues un `Language` puede tener varios `Frameworks`, sin embargo un `Framework` solo puede tener un `Language`. Para proporcionar esta característica, Django nos permite usar el método `ForeignKey` donde indica que esta variable es una clave foránea.

```
1 class Language(models.Model):
2     name = models.CharField(max_length=10)
3
4     def __str__(self):
5         return self.name
6
7 class Framework(models.Model):
8     name = models.CharField(max_length=10)
9     language = models.ForeignKey(Language, on_delete=models.CASCADE)
10
11     def __str__(self):
12         return self.name
```

Para realizar las pruebas usando la shell de Python, necesitamos informarle a Django de la existencia de la aplicación `relationships` en el archivo de configuración `settings.py` y luego crear y realizar las migraciones en nuestra base de datos, en este caso estamos usando en sistema de gestión de base de datos PostgreSQL.

```
1 INSTALLED_APPS = [
2     'relationships',
```

Ahora sí podemos realizar las pruebas usando la shell de python, por ejemplo vamos a crear objetos de los modelos `Language` y `Framework`.

```
via v3.12.3 (program) > python manage.py shell
Python 3.12.3 (main, Apr 23 2024, 09:16:07) [GCC 13.2.1 20240417] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from relationships.models import Language, Framework
>>> python = Language(name='Python')
>>> python.save()
>>> django = Framework(name='Django')
>>> django.language = python
>>> flask = Framework(name='Flask', language=python)
>>> django.save()
>>> flask.save()
>>> quit()
```

Figure 1: Filas en la base de datos

La funcionalidad de Django se ve en las siguientes operaciones como son filtrar de acuerdo a un campo específico. Primeramente veremos en nuestra base de datos que el campo `language` en el modelo `Framework`, se creará automáticamente como `language_id` en la tabla `Framework` en la base de datos. Este campo almacena el `id` del `Language` relacionado.

```
via v3.12.3 (program) > python manage.py shell
Python 3.12.3 (main, Apr 23 2024, 09:16:07) [GCC 13.2.1 20240417] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from relationships.models import Language, Framework
>>> Framework.objects.all()
<QuerySet [<Framework: Django>, <Framework: Flask>]>
>>> java = Language(name="Java")
>>> java.save()
>>> spring = Framework(name="Spring", language=java)
>>> spring.save()
>>> Framework.objects.all()
<QuerySet [<Framework: Django>, <Framework: Flask>, <Framework: Spring>]>
>>> Framework.objects.filter(language__name="Python")
<QuerySet [<Framework: Django>, <Framework: Flask>]>
>>> Framework.objects.filter(language__name="Java")
<QuerySet [<Framework: Spring>]>
```

Figure 2: Uso del doble guión bajo

Como vemos y somos observadores, nos preguntaremos a qué referencia `language__name`. Esta es una funcionalidad de Django para seguir las relaciones entre los modelos y realizar consultas a través de esas relaciones.

- **language:** Este es el nombre del campo de clave foránea en el modelo `Framework` que hace referencia al modelo `Language`. Este campo establece una relación entre los modelos `Framework` y `Language`.
- **__name:** La notación de doble guion bajo `__` indica a Django que queremos realizar una consulta a través de la relación establecida por el campo `language`. El campo `language` es una instancia del modelo `Language`, y queremos acceder a su campo `name`.

Ahora nos surge otra duda, si decimos que `language` es el campo en Django de clave foránea que referencia a una instancia `Language`, entonces, **por qué se puede realizar la siguiente consulta si en `Language` no hay ningún campo foráneo que referencia a `Framework`?**

```
1 Language.objects.filter(framework__name="Spring")
```

Lo que pasa es que en Django, cuando definimos una relación de clave foránea en un modelo automáticamente crea una relación inversa en el modelo relacionado. Esto nos permite realizar consultas desde ambos lados de la relación.

Relacion muchos a muchos

Para mostrar esta relación, creamos las clases **Movie** y **Character** como modelos en Django, que establecerá la relación entre películas y personajes. Podemos intuir la necesidad de esta relación para este caso ya que una película puede tener varios personajes y viceversa, un personaje puede participar en varias películas..

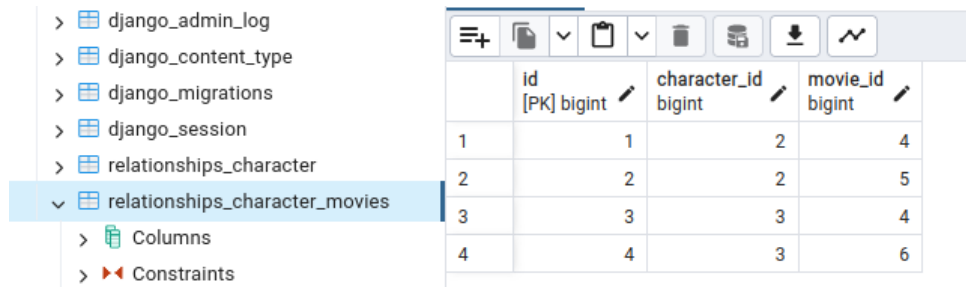
```
1 class Movie(models.Model):
2     name = models.CharField(max_length=100)
3
4     def __str__(self):
5         return self.name
6
7 class Character(models.Model):
8     name = models.CharField(max_length=100)
9     movies = models.ManyToManyField(Movie)
10
11     def __str__(self):
12         return self.name
```

Ahora podemos realizar las siguiente operaciones para poder observar cómo es que Django maneja este tipo de relación y qué es lo que hace en la base datos.

```
via v3.12.3 (program) > python manage.py shell
Python 3.12.3 (main, Apr 23 2024, 09:16:07) [GCC 13.2.1 20240417] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from relationships.models import Movie, Character
>>> Character.objects.filter(movies__name="Civil War")
<QuerySet [<Character: Captain America>]>
>>> Movie.objects.filter(character__name="Captain America")
<QuerySet [<Movie: Avengers>, <Movie: Civil War>]>
>>> captain_america = Character.objects.get(name="Captain America")
>>> captain_america
<Character: Captain America>
>>> captain_america.movies.all()
captain_america.movies.alast() captain_america.movies.alias(
captain_america.movies.allatest( captain_america.movies.all()
>>> captain_america.movies.all()
<QuerySet [<Movie: Avengers>, <Movie: Civil War>]>
>>> avengers = Movie.objects.get(name="Avengers")
>>> avengers
<Movie: Avengers>
>>> avengers.character_set.all()
<QuerySet [<Character: Captain America>, <Character: Thor>]>
>>> quit()
```

Figure 3: Relación muchos a muchos

En la base de datos, Django maneja las relaciones de muchos a muchos utilizando una tabla intermedia que vincula las claves primarias de ambos modelos relacionados. Por ejemplo, para las relaciones anteriores, tenemos una tabla intermedia que se ve así:



	id [PK] bigint	character_id bigint	movie_id bigint
1	1	2	4
2	2	2	5
3	3	3	4
4	4	3	6

Figure 4: Tabla intermedia generada automáticamente

Generación de PDFs con Django

Para evidenciar esta funcionalidad, solo se ha creado un nuevo proyecto en el cual directamente implementamos la lógica para generar el pdf. Sin embargo tuvimos que hacer algunas configuraciones importantes en el archivo `settings.py`, como es habilitar el directorio `templates` para crear plantillas y también modificar el idioma y la hora de la zona.

```

1  TEMPLATES = [
2      {
3          'BACKEND': 'django.template.backends.django.DjangoTemplates',
4          'DIRS': [os.path.join(BASE_DIR, 'templates')],
  
```

```

1  LANGUAGE_CODE = 'es'
2  TIME_ZONE = 'America/Lima'
  
```

Ahora, se ha creado el archivo `renderers.py` que se encarga de parsear el HTML a PDF, para esta funcionalidad se ha tenido que instalar el paquete `xhtml2pdf` de Python.

```

1  from io import BytesIO
2  from django.http import HttpResponse
3  from django.template.loader import get_template
4
5  from xhtml2pdf import pisa
6
7  def render_to_pdf(template_src, context_dict={}):
8      template = get_template(template_src)
9      html = template.render(context_dict)
10     result = BytesIO()
11     pdf = pisa.pisaDocument(BytesIO(html.encode("ISO-8859-1")), result)
12     if pdf.err:
13         return HttpResponse("Invalid PDF", status_code=400, content_type='text/plain')
14     return HttpResponse(result.getvalue(), content_type='application/pdf')
  
```

Después de crear este archivo, vamos a crear un vista que se encargue de manejar la solicitud del cliente.

```
1 from django.http import Http404
2 import datetime
3 from . import renderers
4
5 def pdf_view(request, *args, **kwargs):
6     data = {
7         'pdf_title': "Factura Rovistar",
8         'today': datetime.date.today(),
9         'amount': 39.99,
10        'customer_name': 'Jorge Luis Mamani Huarsaya',
11        'invoice_number': 1233434,
12    }
13    return renderers.render_to_pdf('pdf/invoice.html', data)
```

Finalmente realizamos el direccionamiento modificando el archivo `urls.py`

```
1 from django.contrib import admin
2 from django.urls import path
3
4 from .views import pdf_view
5
6 urlpatterns = [
7     path('', pdf_view, name="pdf_view"),
8     path('admin/', admin.site.urls),
9 ]
```

Lanzamos el servidor local y probamos la funcionalidad. Nos daremos cuenta que accediendo desde la URL de nuestro navegador, se verá el pdf generado.

Envío de correos electronicos con Django

Comenzamos realizando las configuraciones en el archivo `settings.py` para que se pueda usar el servicio de envío de correos electrónicos.

```
1 EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
2 EMAIL_HOST = 'smtp.gmail.com'
3 EMAIL_PORT = 587
4 EMAIL_HOST_USER = env('EMAIL_USER')
5 EMAIL_HOST_PASSWORD = env('EMAIL_PASSWORD')
6 EMAIL_USE_TLS = True
7 EMAIL_USE_SSL = False
```

Ahora, creamos una aplicación denominada `send` y creamos una vista que se encarga de la lógica de manejar la solicitud del cliente. En este caso solo enviaremos directamente el email cuando se acceda a este sitio web.

```
1 from django.shortcuts import render
2 from django.core.mail import send_mail
3 from django.conf import settings
4
5 # Create your views here.
6 def index(request):
7
```

```
8 subject = "Hello, is Jorghee"
9 message = "I am programmer, I like the apples and penguins"
10 email_from = settings.EMAIL_HOST_USER
11 recipient_list = ['mamannihuarsaya4b@gmail.com']
12
13 report = "Correo enviado exitosamente"
14 try:
15     send_mail(subject, message, email_from, recipient_list, fail_silently=False)
16 except Exception as e:
17     report = "Error al enviar el correo"
18     print(e)
19
20 return render(request, 'send/index.html', { "report":report })
```

Como vemos, estamos renderizando una plantilla. Esta plantilla nos indicará si se realizó correctamente el envío del correo electrónico. Luego de hacer estas modificaciones, tenemos que direccionar las solicitudes modificando el archivo `urls.py` del proyecto y creando el archivo `urls.py` de la aplicación.