Django: modelos

Modelos

Los modelos son clases que definen la estructura de los datos que se almacenan en base de datos, incluyendo los tipos de datos, restricciones como valores máximos o mínimos, valores por defecto, listas de opciones posibles, etc.

La definición de un modelo es independiente de la tecnología de base de datos utilizada, esto quiere decir que los modelos de una aplicación seguirán intactos aunque cambiemos a otra tecnología. Esto se debe a que no tenemos que programar cómo comunicarnos con la base de datos ni las sentencias necesarias para realizar las operaciones requeridas, es django quien se encarga de todo esto.

Definimos los modelos en el archivo models.py de nuestra aplicación. Un modelo será una subclase de **django.db.models.Model** y puede incluir campos, métodos y metadatos. Cada campo en nuestro modelo representará un campo en la base de datos.

Modelos: ejemplo

Ejemplo de creación de un modelo para representar personas.

El modelo se compone de campos donde cada uno de ellos es una instancia de la clase **Field** correspondiente. Hay diferentes tipos de Field, los más utilizados son:

CharField para textos, **DateField** para fechas, **DateTimeField** para fechas con tiempo, **IntegerField** para números enteros, **BooleanField** para booleanos, **FloatField** para números de coma flotante… etc, ver la lista completa en **este enlace**.

```python
class Book(models.Model):
    ''' Modelo libro'''
    title = models.CharField(max_length=200)
    summary = models.TextField(max_length=1000)
    isbn = models.CharField(max_length=13)
    pages = models.IntegerField(default=0)
    year = models.IntegerField(default=1970)

    class Meta:
        db_table = 'book'
        ordering = ['title']

    def get_absolute_url(self):
        ''' reverse permite devolver una url a partir de su nombre definido en urls.py,
        detalle de un libro'''
        return reverse('book_detail', args=[str(self.id)])

    def __str__(self):
        """String for representing the Model object."""
        return self.isbn
```

Nota: dentro de nuestras clases de modelo se suele definir una clase Meta que permite especificar cosas como el nombre que tendrá la tabla en base de datos, el orden en que deben recuperarse los registros, etc. Documentación **meta**.

Modelos: ejemplo

Cada **campo en el modelo** corresponde a una columna en la tabla, la clase completa corresponde a una tabla.

Podemos definir los nombres de campo que queramos a excepción de las palabras clean, save o delete (exclusivas para django).

Nota: En caso de necesitar un tipo Field que no esté definido entre los ya ofrecidos por django existe la posibilidad de crear tipos propios, **guía para hacerlo**.

```python
class Book(models.Model):
    ''' Modelo libro'''
    title = models.CharField(max_length=200)
    summary = models.TextField(max_length=1000)
    isbn = models.CharField(max_length=13)
    pages = models.IntegerField(default=0)
    year = models.IntegerField(default=1970)

    class Meta:
        db_table = 'book'
        ordering = ['title']

    def get_absolute_url(self):
        ''' reverse permite devolver una url a partir de su nombre definido en urls.py,
        detalle de un libro'''
        return reverse('book_detail', args=[str(self.id)])

    def __str__(self):
        """String for representing the Model object."""
        return self.isbn
```

Nota: dentro de nuestras clases de modelo se suele definir una clase Meta que permite especificar cosas como el nombre que tendrá la tabla en base de datos, el orden en que deben recuperarse los registros, etc. Documentación **meta**.

Modelos: ejemplo

Al no especificar un campo con la opción primary_key=True django genera automáticamente un campo id que tiene en cuenta cuando se trabaja con los modelos, por eso en base de datos vemos la columna id pero en nuestra clase no vemos el campo id. Por defecto tiene esta configuración:

id = models.AutoField(primary_key=True)

AutoField es internamente un **IntegerField** pero que se autoincrementa en función de los IDs disponibles.

```python
class Person(models.Model):

    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    created_date = models.DateField()
    age = models.IntegerField()
    married = models.BooleanField()
    money = models.FloatField()
```

Nota: dentro de nuestras clases de modelo se suele definir una clase Meta que permite especificar cosas como el nombre que tendrá la tabla en base de datos, el orden en que deben recuperarse los registros, etc. Documentación **meta**.

Modelos: trasladar cambios a base de datos

Una vez se crea un modelo o se edita uno existente, será necesario llevar los cambios a la base de datos haciendo uso de migraciones.

Primero ejecutamos el comando que creará las migraciones:

py -3 manage.py makemigrations (Windows)

python3 manage.py makemigrations (Linux/Mac)

Después ejecutamos la migración:

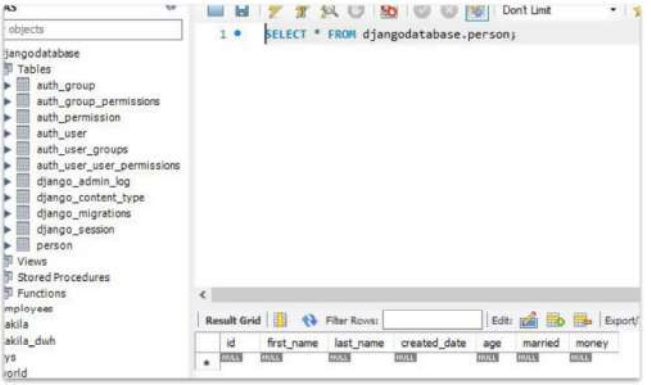
py -3 manage.py migrate (Windows)

python3 manage.py migrate (Linux/Mac)

Comprobamos que la base de datos ahora tiene una tabla persona con los campos que hemos definido en nuestro modelo desde python.

```
PS C:\Users\Alan\python\proyectos_django\proyecto1> py -3 .\manage.py makemigrations
Migrations for 'app1':
  app1\migrations\0001_initial.py
    - Create model Person
```

```
PS C:\Users\Alan\python\proyectos_django\proyecto1> py -3 .\manage.py migrate
Operations to perform:
  Apply all migrations: admin, app1, auth, contenttypes, sessions
Running migrations:
  Applying app1.0001_initial... OK
```



Modelos: utilizar los modelos, Create

Django nos proporciona una api para trabajar con los modelos, es decir, una serie de métodos que podremos utilizar para realizar operaciones con base de datos, lo que nos ahorra tener que programar nosotros ese comportamiento.

Los modelos son utilizados en las **vistas**, generalmente para realizar operaciones **CRUD** y cualquier otro proceso que necesitemos.

Crearemos un nuevo método en el archivo **views.py** que creará un registro en base de datos, esto se hace mediante el método create().

Archivo: views.py

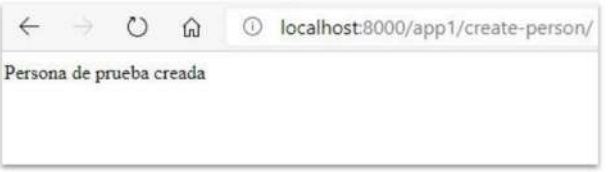
```python
import datetime
from django.shortcuts import render
from django.http import HttpResponse
from .models import Person

# Create your views here.

def create_person(request):
    ''' Método create para crear registros en db '''
    person_result = Person.objects.create(
        first_name = 'John',
        last_name = 'Doe',
        age = 34,
        created_date = datetime.date.today(),
        married = False,
        money = 100677.54,
        )
    print(person_result)
    return HttpResponse('Persona de prueba creada')
```

El método **create** lo que hace es crear y guardar el registro persona en base de datos.

Modelos: utilizar los modelos, Create

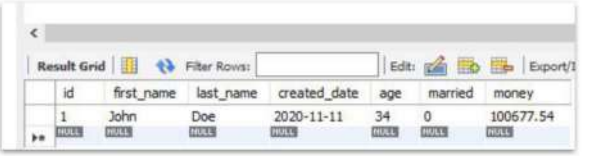
Añadimos una url para acceder a ese nuevo método.

Como no hemos especificado el campo id, django lo genera automáticamente en base de datos.




```
Django version 3.1.3, using settings 'proyecto1.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
Person(first_name:John, last_name:Doe, created_date:2020-11-09, age:34, married:False, money:100677.54)
```

```python
from django.urls import path
from . import views

urlpatterns = [
    path('saludo/', views.saludo, name='saludo'),
    path('despedida/', views.despedida, name='despedida'),
    path('create-person/', views.create_person, name='create-person'),
    path('persons/', views.find_persons, name='persons'),
]
```



Modelos: utilizar los modelos, Create

Opcional: inicialmente pylint mostrará un error en Person porque no es capaz de detectar el atributo objects.

Para evitar esto podemos añadir django a pylint para que pueda detectarlo y no lo muestre como un error.

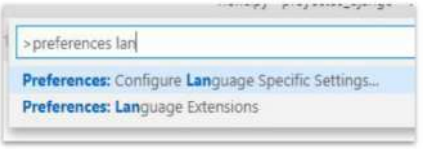
1 - Para ello ejecutamos **Ctrl+shift + p** y buscamos la opción **Preferences: Configure Language Specific Settings**.

2 - Después escribiremos **Python**.

3 - Por último añadimos el código que se muestra en la imagen al archivo **settings.json**.

4 - **Guardamos**.

Paso 1



Paso 2

Paso 3

Modelos: recuperar todos los registros

El mismo atributo objects heredado en los modelos ofrece el **método all()** para recuperar todos los registros de una tabla:

<model_name>.objects.all()

1 - Archivo views.py: creamos nueva vista para recuperar

```
27  def find_persons(request):
28      ''' Recupera todas las personas de db'''
29      persons = Person.objects.all()
30      return HttpResponse(persons)
```

2 - Archivo urls.py de app1, añadimos url para la nueva vista

```
1  from django.urls import path
2  from . import views
3
4  urlpatterns = [
5      path('saludo/', views.saludo, name='saludo'),
6      path('despedida/', views.despedida, name='despedida'),
7      path('create-person/', views.create_person, name='create-person'),
8      path('persons/', views.find_persons, name='persons'),
9  ]
10
```

3 - Llamada a la url y verificación de que recupera los datos

Nota: podemos crear una plantilla y vincularla en la nueva vista **find_persons** para que muestre un html elaborado con los datos de las personas

Modelos: filtrar

Para realizar **filtrados** por campos utilizamos el método **filter()**. Permite filtrar por cualquier campo y obtendrá los resultados que coincidan, pudiendo haber más de uno o incluso ninguno.

El método filter() también permite afinar más el tipo de coincidencia qué debe hacer mediante el uso de [lookups].

```
32  def get_person(request):
33
34      ''' Recupera una persona mediante filtro (hardcoded)'''
35      john = Person.objects.filter(first_name="John")
36      print(john)
37      return HttpResponse(john)
```

```
Django version 3.1.3, using settings 'proyecto1.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
<QuerySet [<Person: Person(first_name:John, last_name:Doe, created_date:2020-11-11, age:
ey:100677.54)>, <Person: Person(first_name:John, last_name:Doe, created_date:2020-11-09,
```

Modelos: get

Si sabemos que solo hay **un resultado** que queremos obtener, como por ejemplo un cliente a partir de su nif o cualquier otro modelo a partir de su id único, tiene sentido utilizar el **método get()**, que nos devolverá un objeto en lugar de un QuerySet.

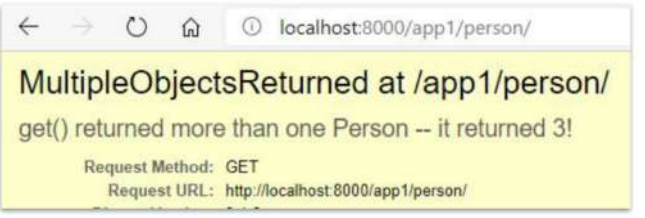
```
def get_person(request):

    ''' Recupera una persona mediante get (hardcoded)'''
    john = Person.objects.get(id=1)
    print(john)
    return HttpResponse(john)
```

Modelos: get

Si sabemos que solo hay un resultado que queremos obtener, como por ejemplo un cliente a partir de su nif o cualquier otro modelo a partir de su id único, tiene sentido utilizar el **método get()**, que nos devolverá un objeto en lugar de un QuerySet.

Nota: si no hay ningún registro en base de datos se producirá una excepción de tipo **DoesNotExist**, por otro lado, si hay más de un registro se producirá una excepción de tipo **MultipleObjectsReturned**.

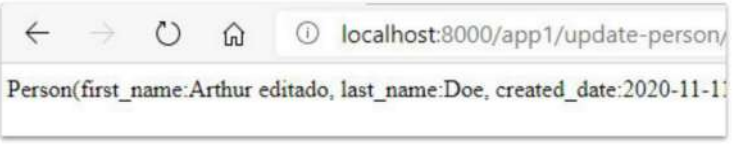
```
def get_person(request):

    ''' Recupera una persona mediante get (hardcoded)'''
    john = Person.objects.get(first_name="John")
    print(john)
    return HttpResponse(john)
```



Modelos: modificar/actualizar

Una operación muy común es tener que **actualizar un registro** de la base de datos. Para ello, utilizamos el método **save()** sobre el modelo. No es necesario realizar commit.

```python
def update_person(request):

    ''' Modifica el campo nombre de un objeto persona recuperado
    de la base de datos'''
    arthur = Person.objects.get(first_name="Arthur")
    arthur.first_name = "Arthur editado"
    arthur.save()
    return HttpResponse(arthur)
```

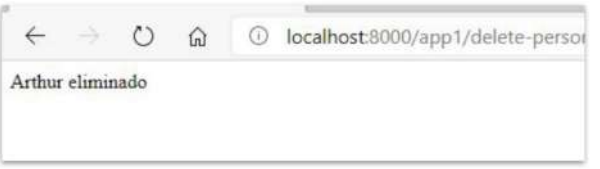


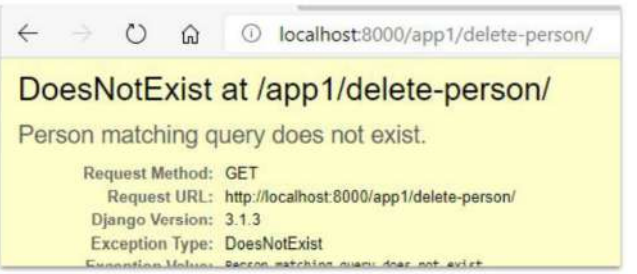
Modelos: eliminar

Para **eliminar un registro** de la base de datos utilizamos el método **delete()** sobre un modelo. No es necesario realizar commit.

Nota: si intentamos borrar un modelo que no existe o que ya fue borrado previamente se producirá una excepción de tipo DoesNotExist.

```
def delete_person(request):

    ''' Elimina el registro persona que hemos obtenido primero
    de la base de datos'''
    arthur = Person.objects.get(first_name="Arthur")
    arthur.delete()
    return HttpResponse("Arthur eliminado")
```





Modelos y relaciones

A la hora de conceptualizar los datos de la aplicación y definir el esquema o modelos de datos nos encontramos la existencia de relaciones. Por ejemplo: libro y autor, libro y género, taller y coche, coche y fabricante, etc.

Existen clases para representar todas estas relaciones en los modelos de django:

- Relaciones **uno a uno**: OneToOneField
- Relaciones **uno a muchos**: ForeignKey
- Relaciones **muchos a uno**: ForeignKey
- Relaciones **muchos a muchos**: ManyToManyField

Modelos y relaciones: OneToOneField

Utilizamos **OneToOneField** para representar las relaciones de **uno a uno**, por ejemplo, un cliente tiene una dirección, y esa dirección sólo pertenece a ese cliente.

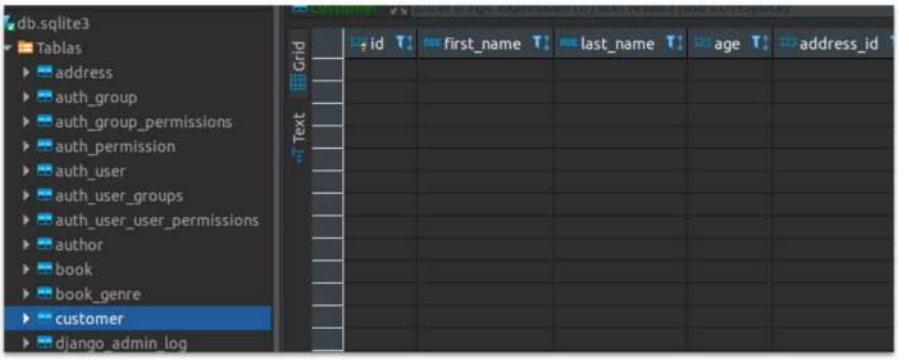
Con el argumento on_delete y la opción models.CASCADE configuramos que si se borra un objeto address se borre también el customer.

```python
class Address(models.Model):
    street = models.CharField(max_length=200)
    postal_code = models.IntegerField(default=0)
    country = models.CharField(max_length=200)

    class Meta:
        db_table = 'address'
        ordering = ['street']
```

```python
class Customer(models.Model):
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)
    age = models.IntegerField(default=0)
    address = models.OneToOneField(Address, on_delete=models.CASCADE)

    class Meta:
        db_table = 'customer'
        ordering = ['first_name']
```

Modelos y relaciones: OneToOneField

A nivel de base de datos tendremos dos tablas una por modelo y en una de ellas habrá una nueva columna que apunte a la clave primaria de la otra.



Modelos y relaciones: ForeignKey

Relación de **uno a muchos o de muchos a uno**.

Un objeto tiene una lista de otros objetos, por ejemplo: un autor tiene varios libros pero un libro pertenece a un solo autor, un taller tiene varios coches, etc.

Para ello utilizamos el tipo de campo **ForeignKey**, en el cual especificamos la entidad que se relaciona.

La opción **CASCADE** permite configurar si queremos que cuando se borre un objeto se pueda borrar también el objeto relacionado.

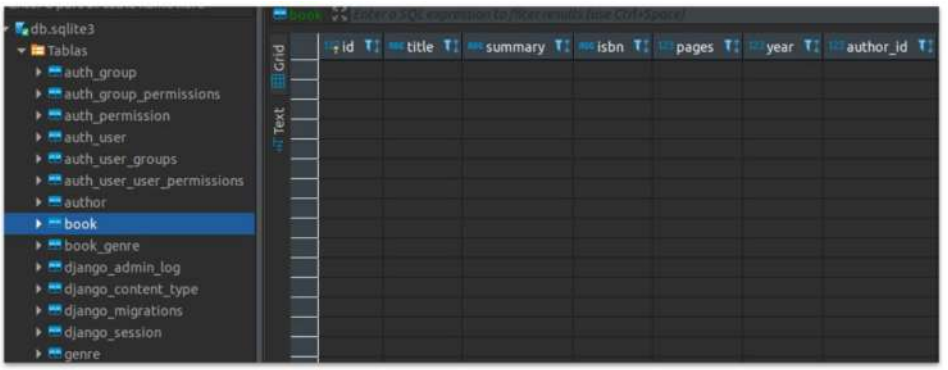
```python
class Author(models.Model):
    name = models.CharField(max_length=200)
    age = models.IntegerField(default=0)

    class Meta:
        db_table = 'author'
        ordering = ['name']
```

```python
class Book(models.Model):

    title = models.CharField(max_length=200)
    summary = models.TextField(max_length=1000)
    isbn = models.CharField(max_length=13)
    pages = models.IntegerField(default=0)
    year = models.IntegerField(default=1970)
    author = models.ForeignKey(Author, on_delete=models.SET_NULL, null=True)
```

Modelos y relaciones: ForeignKey

Un libro pertenece a un autor, pero un autor puede tener múltiples libros.

Esta relación crea una nueva columna con el nombre del campo más el sufijo _id para referenciar que apunta a la clave primaria del otro modelo, autor en este caso.



Modelos y relaciones

El otro tipo de relación un poco más compleja es aquel en el que un objeto tiene múltiples objetos en relación de **muchos a muchos**, por ejemplo:

Un libro tiene múltiples categorías, una misma categoría puede estar en múltiples libros.

Relación de muchos a muchos.

Para implementarlo utilizamos el tipo ManyToMany en el modelo.

No importa cuál de los dos modelos tenga el campo ManyToMany, lo importante es que solo lo tenga uno de los dos, no ambos. Se suele poner en el modelo que más se suela utilizar en los formularios de las plantillas.

```python
class Genre(models.Model):
    name = models.CharField(max_length=200)

    class Meta:
        db_table = 'genre'
        ordering = ['name']
```

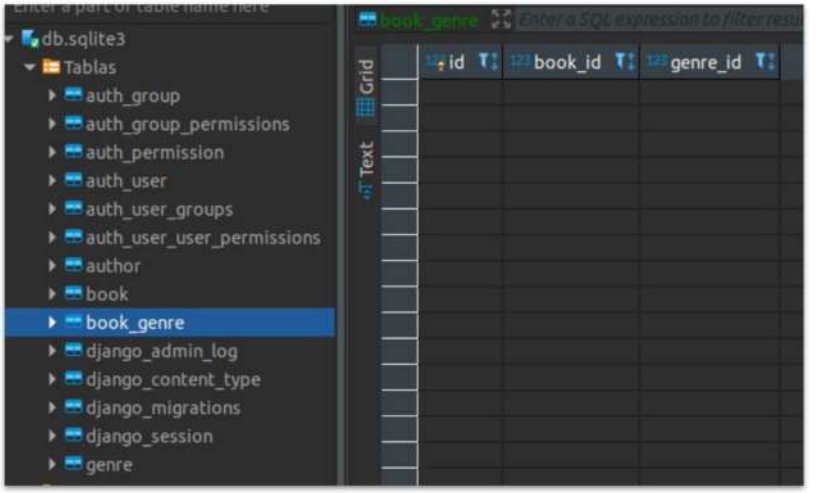
```python
class Book(models.Model):

    title = models.CharField(max_length=200)
    summary = models.TextField(max_length=1000)
    isbn = models.CharField(max_length=13)
    pages = models.IntegerField(default=0)
    year = models.IntegerField(default=1970)
    author = models.ForeignKey(Author, on_delete=models.SET_N
    genre = models.ManyToManyField(Genre)
```

Modelos y relaciones

Un libro puede pertenecer a más de un género, mientras que un género puede estar en más de un libro.

Esta relación crea una nueva tabla en la base de datos para almacenar las claves primarias de ambos modelos.

De esta forma un mismo libro puede estar asociado a más de un género, cosa que no podría reflejarse con una nueva columna en la tabla libro como ocurría en la relación uno a muchos.



Urls dinámicas

Es común utilizar **parámetros en las urls**, por ejemplo cuando necesitamos recuperar un modelo a partir de su id u otros campos.

En este ejemplo creamos una url para recuperar una persona a partir de su id.

Para conocer a fondo los **converters** que podemos utilizar en las urls ver la [documentación oficial](#).

```python
urlpatterns = [
    path('saludo/', views.saludo, name='saludo'),
    path('despedida/', views.despedida, name='despedida'),
    path('create-person/', views.create_person, name='create-person'),
    path('persons/', views.find_persons, name='persons'),
    path('person/', views.get_person, name='person'),
    path('update-person/', views.update_person, name='update-person'),
    path('delete-person/', views.delete_person, name='delete-person'),
    path('person/<int:person_id>/', views.find_person, name='find-person'),
]
```

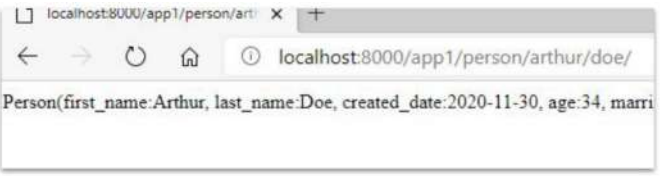
```python
from django.http import HttpResponse, Http404
from .models import Person

# Create your views here.

def find_person(request, person_id):
    ''' Recupera una persona mediante get a partir de su id'''
    try:
        person = Person.objects.get(id=person_id)
    except Person.DoesNotExist as person_not_exist:
        raise Http404('La persona solicitada no existe') from person_not_exist

    return HttpResponse(person)
```

Urls dinámicas

También podemos añadir más de un parámetro, y de otro tipo de dato como por ejemplo una cadena de texto.

```
path('delete-person/', views.delete_person, name='delete-person'),
path('person/<int:person_id>/', views.find_person, name='find-person'),
path('person/<str:first_name>/<str:last_name>/', views.find_person_by_name, name='find-person-name'),
```

```
def find_person_by_name(request, first_name, last_name):
    ''' Recupera una persona mediante get a partir de su id'''
    try:
        person = Person.objects.get(first_name=first_name, last_name=last_name)
    except Person.DoesNotExist as person_not_exist:
        raise Http404('La persona solicitada no existe') from person_not_exist

    return HttpResponse(person)
```



Vistas basadas en clase (vistas genéricas)

Django permite acelerar el desarrollo en **modelos** y **plantillas** gracias al ahorro de código que ofrece al gestionar automáticamente las conexiones a base de datos y el reemplazo de datos de forma dinámica en las plantillas.

En cuanto a las vistas, django ofrece más formas de crear vistas a mayores de mediante funciones con el fin de poder acelerar su desarrollo también. Esta forma es mediante **clases** de manera que podamos aprovechar los mecanismos de la programación orientada a objetos para la reutilización de código, como es el caso de la **herencia**.

Django ofrece una serie de clases vista que heredan de la **clase base View** y que proveen diferentes funcionalidades, ejemplos: **RedirectView**, **TemplateView**, **ListView**, **DetailView**, etc.

Vistas basadas en clase (vistas genéricas)

Para cargar una vista basada en clase en el archivo urls.py utilizamos la siguiente sintaxis:

path(books/', views.BookListView.as_view(), name='books')

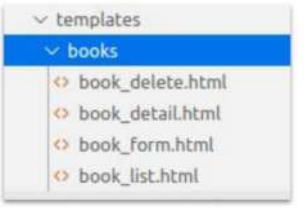
Implementaremos previamente la clase **BookListView** en nuestro archivo views.py.

En caso de querer especificar la plantilla desde urls.py en lugar de en la propia clase dentro de views.py podemos hacerlo:

path(books/', views.BookListView.as_view(template_name='book_list.html'), name='books')

Si no especificamos el argumento **template_name** django infiere a partir del nombre del modelo y el tipo de vista, en este caso el modelo es Book y la vista es una lista por lo que infiere book_list.html y lo busca en el directorio **templates/<nombreapp>/**, al especificar el argumento template_name podemos vincularlo a otro archivo.

Vistas basadas en clase (vistas genéricas)



Ejemplo de cómo crear la **vista listado** de libros **basada en clases**. Partiendo del ejemplo biblioteca:

1. Primero creamos la clase BookListView en views.py.
2. Después la vinculamos en urls.py

Resultado: el listado sigue funcionando exactamente igual, pero nos hemos ahorrado mucho código.

```python
from django.views import generic

# Create your views here.

class BookListView(generic.ListView):
    """Generic class-based view for a list of books."""
    model = Book
```


```python
urlpatterns = [
    path('', views.index, name='index'), # Página home
    path('books/', views.BookListView.as_view(), name='books'), #
    path('books/create/', views.book_create, name='book_create'),
    path('books/<int:id>/', views.book_detail, name='book_detail'
```

Vistas basadas en clase (vistas genéricas)

La operaciones más comunes y para las cuales hay una clase ya definida en django para utilizar como vista son:

- **ListView**: listado, vista para la operación CRUD recuperar objetos.
- **DetailView**: detalle, vista para la operación CRUD recuperar objeto.
- **CreateView**: creación, vista para la operación CRUD crear nuevo objeto.
- **UpdateView**: actualización, vista para la operación CRUD actualizar un objeto.
- **DeleteView**: borrado, vista para la operación CRUD borrar un objeto.

Ver lista completa de **vistas basadas en clase**. En el ejemplo 2 subido a la plataforma se implementa la biblioteca haciendo uso de vistas basadas en clase.

Vistas basadas en clase: implementación

Para implementar las vistas basadas en clase crearemos **nuevas clases** dentro de views.py que hereden de las clases vista propias de django.

Ver lista completa de **vistas basadas en clase**.

```python
from django.views.generic.edit import CreateView, UpdateView, DeleteView

# ********** Vistas para modelo Book **********

class BookListView(generic.ListView):
    ''' Vista genérica basada en clase para el listado de libros '''
    model = Book

class BookDetailView(generic.DetailView):
    ''' Vista genérica basada en clase para el detalle de un libro '''
    model = Book

class BookCreate(CreateView):
    ''' Vista genérica basada en clase para crear un libro '''
    model = Book
    fields = ['title', 'summary', 'isbn', 'pages', 'year', 'author','genres']

class BookUpdate(UpdateView):
    ''' Vista genérica basada en clase para actualizar un libro '''
    model = Book
    fields = ['title', 'summary', 'isbn', 'pages', 'year', 'author','genres']

class BookDelete(DeleteView):
    ''' Vista genérica basada en clase para borrar un libro '''
    model = Book
    success_url = reverse_lazy('books')
```

Urls dinámicas

Ejemplos de urls dinámicas cuando usamos vistas genéricas basadas en clase.

Ahora referenciamos a la clase creada seguido del método **as_view()**.

```
path('books/', views.BookListView.as_view(), name='books'),
path('book/<int:pk>', views.BookDetailView.as_view(), name='book_detail'),
path('book/create/', views.BookCreate.as_view(), name='book_create'),
path('book/<int:pk>/update/', views.BookUpdate.as_view(), name='book_update'),
path('book/<int:pk>/delete/', views.BookDelete.as_view(), name='book_delete'),

path('authors/', views.AuthorListView.as_view(), name='authors'),
path('author/<int:pk>',views.AuthorDetailView.as_view(), name='author_detail'),
path('author/create/', views.AuthorCreate.as_view(), name='author_create'),
path('author/<int:pk>/update/', views.AuthorUpdate.as_view(), name='author_update'),
path('author/<int:pk>/delete/', views.AuthorDelete.as_view(), name='author_delete'),

path('addresses/', views.AddressListView.as_view(), name='addresses'),
path('address/<int:pk>',views.AddressDetailView.as_view(), name='address_detail'),
path('address/create/', views.AddressCreate.as_view(), name='address_create'),
path('address/<int:pk>/update/', views.AddressUpdate.as_view(), name='address_update'),
path('address/<int:pk>/delete/', views.AddressDelete.as_view(), name='address_delete'),


path('genres/', views.GenreListView.as_view(), name='genres'),
path('genre/<int:pk>',views.GenreDetailView.as_view(), name='genre_detail'),
path('genre/create/', views.GenreCreate.as_view(), name='genre_create'),
path('genre/<int:pk>/update/', views.GenreUpdate.as_view(), name='genre_update'),
path('genre/<int:pk>/delete/', views.GenreDelete.as_view(), name='genre_delete'),
```


Telefónica
EDUCACIÓN DIGITAL