INGENIERÍA | DEPARTAMENTO
UNIVERSIDAD DE LOS ANDES | DE COMPUTACIÓN
MÉRIDA VENEZUELA | ESCUELA DE SISTEMAS

# Laboratorio de Ingeniería del Software
# Práctica Nro. 7
# Modelos en Django

Prof. Solazver Solé
Prep. Yenifer Ramírez
Semestre B-2018

## 1. Modelo

Un modelo es la representación de los datos a través de objetos. Los modelos definen la estructura de los datos almacenados, incluidos los tipos de campo y los atributos de cada campo, como su tamaño máximo, valores predeterminados, lista de selección de opciones, texto de ayuda para la documentación, texto de etiqueta para formularios, etc.

### 1.1. Diseñando un modelo

Antes de iniciar a codificar un modelo en cualquier Framework es importante tomarnos unos minutos para pensar que necesitamos almacenar y cuáles son las relaciones que puedan existir entre los datos.
Para efectos de este y el siguiente laboratorio trabajaremos con la elaboración de un Proyecto para la *gestión de libros de una Biblioteca* , por lo que sabemos que tenemos que almacenar información sobre libros (título, resumen, autor, idioma escrito, categoría, ISBN) y que podríamos tener varias copias disponibles (con id único, estado de disponibilidad, etc.). Es posible que necesitemos almacenar más información sobre el autor que solo su nombre, y puede haber varios autores con el mismo nombre o nombres similares. Queremos poder ordenar la información según el título del libro, el autor, el idioma escrito y la categoría.
Al diseñar nuestros modelos, tiene sentido tener modelos separados para cada objeto[1]. En este caso, los objetos obvios son libros, instancias de libros y autores.

[1] Grupo de información relacionada

También es posible que desee utilizar modelos para representar las opciones de la lista de selección (por ejemplo, como una lista desplegable de opciones), en lugar de codificar las opciones en el sitio web en sí; como el género del libro (por ejemplo, ciencia ficción, poesía francesa, etc.) y el idioma (inglés, francés, japonés).
Una vez que hayamos decidido cuáles serán nuestros modelos y sus campos, debemos pensar en la relación que existe entre ellos, relaciones de *uno a uno, de uno a muchos y de muchos a muchos*.
Con todo esto en mente procedemos a representar nuestro modelo con un Diagrama UML (como programadores es importante que nos familiaricemos con este diagrama puesto que en la mayoría de equipos de desarrollo el diseñador de base de datos es alguien distinto al programador, y según sea nuestro rol, tendremos un diagrama UML como entrada o salida de nuestra labor).

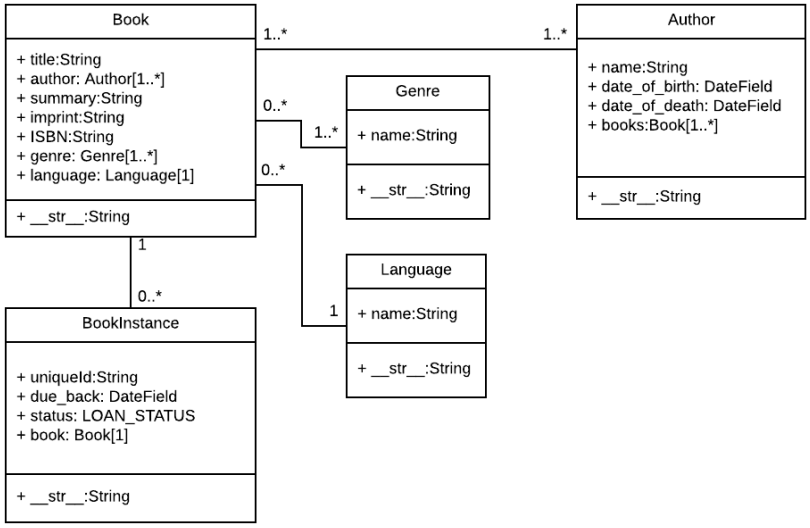


Figura 1: Diagrama UML del modelo de una Biblioteca

En el diagrama hemos creado modelos para el libro (los detalles genéricos del libro), instancia del libro (estado de copias físicas específicas del libro disponible en el sistema) y autor. También hemos decidido tener un modelo para el género. Hemos decidido no tener un modelo para el `BookInstance:status`, en su lugar, especificaremos directamente, en el código, los valores (`LOAN_STATUS`). Dentro de cada uno de los cuadros, puede ver el nombre del modelo, los nombres y tipos de campo, y también los métodos y sus tipos de devolución como se expresa en estos tipos de diagramas.
El diagrama también muestra las relaciones entre los modelos, incluida su car-

dinalidad[2]. Los números cercanos al modelo `Book` muestran que un libro debe tener uno o más `Genres` (tantos como desee), mientras que los números al otro lado de la línea al lado de `Genre` muestran que puede tener cero o más libros asociados.

## 1.2. Uso de modelos en Django

La definición del *modelo es independiente de la base de datos* subyacente, puede elegir una de entre varias como parte de la configuración de su proyecto. Una vez que hayamos elegido la base de datos que deseamos usar, no necesitamos hablar directamente con ella. Simplemente escribimos la estructura del modelo y algo de código, y Django se encargará de todo el trabajo, al comunicarse con la base de datos por nosotros.

### 1.2.1. Definición de Modelo

Los modelos están definidos, normalmente, en el archivo **models.py** de la aplicación. Son implementados como subclases de `django.db.models.Model`, y pueden incluir campos, métodos y metadata. El fragmento de código más abajo muestra un modelo "típico", llamado `MyModelName`:

```
from django.db import models

class MyModelName(models.Model):
    """
    Una clase tipica definiendo un modelo, derivado
    desde la clase Model.
    """

     # Campos
     my_field_name =
     models.CharField (max_length=20,
                       help_text="Field documentation")

     # Metadata
     class Meta:
         ordering = ["-my_field_name"]

     # Metodos
     def get_absolute_url(self):
          """
          Devuelve la url para acceder a una instancia
          particular de MyModelName.
```

[2]La cardinalidad expresa la cantidad de instancias (máximo y mínimo) de cada modelo que pueden estar presentes en la relación.

```
        """
        return reverse('model-detail-view',
                       args=[str(self.id)])

    def __str__(self):
        """
        Cadena para representar el objeto MyModelName
        (en el sitio de Admin, etc.)
        """
        return self.field_name
```

1. **Campos**

   Un modelo puede tener un número arbitrario de campos, de cualquier tipo. Cada uno representa una columna de datos que queremos guardar en nuestras tablas de la base de datos. Cada registro de la base de datos (fila) consistirá en uno de cada posible valor del campo.

```
my_field_name =
models.CharField (max_length=20,
                  help_text="Field documentation")
```

   El código anterior tiene un único campo llamado `my_field_name`, de tipo[3] `models.CharField` lo que significa que este campo contendrá una cadena de caracteres alfanuméricos. Los tipos de campo son asignados usando clases específicas, que determinan el tipo de registro que se usa para guardar el dato en la base, junto con un criterio de evaluación que se usará cuando se reciban los valores de un formulario HTML (es decir, qué constituye un valor válido). Los tipos de campo pueden también tomar argumentos[4] que especifican además cómo se guarda o cómo se puede usar. En este caso le damos a nuestro campo dos argumentos:

   - `max_length=20`: establece que la longitud máxima del valor de este campo es 20 caracteres.
   - `help_text="Field documentation"`: proporciona una etiqueta de texto para mostrar que ayuda a los usuarios a saber qué valor proporcionar cuando un usuario ha de introducirlo vía un formulario HTML.

   El nombre del campo se usa para referirnos a él en consultas (*queries*) y plantillas (*templates*). Los campos también tienen una etiqueta, que puede ser especificada como argumento (`verbose_name`) o inferida automáticamente, a partir del nombre de variable que identifica al campo, capitalizando la primera letra y reemplazando los guiones bajos por espacios (por ejemplo `my_field_name` tendría la etiqueta por defecto de *My field name*).

[3] Tipos de Campo: https:/docs.djangoproject.com/en/1.10/ref/models/fields/#field-types
[4] Argumentos: https://docs.djangoproject.com/en/1.10/ref/models/fields/#field-options

El orden en que los campos son declarados afectará su orden por defecto si un modelo es renderizado en un formulario, aunque este comportamiento se puede anular.

2. **Metadatos** Podemos declarar metadatos a nivel de modelo para nuestro Modelo declarando `class Meta`.

```
class Meta:
    ordering = ["-my_field_name"]
    ...
```

Una de las características más útiles de estos metadatos es controlar el orden por defecto de los registros que se devuelven cuando se consulta el tipo de modelo. Se hace especificando el orden de comprobación en una lista de nombres de campo en el atributo `ordering`, como se muestra en el código. La ordenación dependerá del tipo de campo (los campos de caracteres se ordenan alfabéticamente, mientras que los campos de fechas están clasificados por orden cronológico). Como se muestra, se puede invertir el orden de clasificación añadiendo el símbolo (-) como prefijo del nombre del campo. Por ejemplo, si elegimos clasificar los libros de esta forma por defecto:

```
ordering = ["title", "-pubdate"]
```

Los libros serán clasificados alfabéticamente por título, de la A al a Z, y luego por fecha de publicación dentro de cada título, desde el más reciente al más antiguo.

Otro atributo común es `verbose_name`, un nombre descriptivo para la clase en forma singular y plural:

```
verbose_name = "BetterName"
```

Otros atributos útiles te permiten crear y aplicar nuevos "permisos de acceso"para el modelo (los permisos por defecto se aplican automáticamente), te permiten la ordenación basado en otro campo, o declarar que la clase es .ªbstracta"[5].

Muchas de las otras opciones de metadatos[6] controlan qué base datos debe usarse para el modelo y cómo se guardan los datos (éstas son realmente útiles si necesitas mapear un modelo a una base datos existente).

3. **Métodos**

Un modelo puede tener también métodos, como mínimo, en cada modelo deberíamos definir el método estándar de las clases de Python `__str__()`

[5]Una clase base para la que no se crean registros, y que en cambio se derivará para crear otros modelos

[6]Opciones de metadatos: https:/docs.djangoproject.com/en/2.0/ref/models/options

para devolver una cadena de texto legible para cada objeto. Esta cadena se usa para representar registros individuales en el sitio de administración (y en cualquier otro lugar donde necesitemos referirnos a una instancia del modelo). Con frecuencia éste devolverá un título o nombre de campo del modelo.

```
def __str__(self):
        return self.field_name
```

Otro método común a incluir en los modelos de Django es `get_absolute_url()`, que devuelve un URL para presentar registros individuales del modelo en el sitio web (si definimos este método, Django añadirá automáticamente un botón "Ver en el sitio", en la ventana de edición del registro del modelo en el sitio de Administración).

```
def get_absolute_url(self):
    """
     Devuelve la url para acceder a una instancia
     particular del modelo.
    """
    return reverse('model-detail-view',
                   args=[str(self.id)])
```

### 1.2.2. Gestión de modelos

Una vez que hemos definido nuestras clases de modelos pueden usarse para crear, actualizar o borrar registros, y ejecutar consultas para obtener todos los registros o subconjuntos particulares de registros. Generalmente este tipo de gestión se comienza a realizar cuando definamos nuestras vistas, pero por efectos de este laboratorio explicaremos algunas, que se pueden ejecutar desde la terminal de Django.

```
(eV)usuario@pc:~$ python manage.py shell
```

1. **Creación y modificación de registros**

   Para crear un registro podemos definir una instancia del modelo y llamar a `save()`.

```
"""
Creacion de un nuevo registro usando el constructor
 del modelo.
"""
a_record = MyModelName(my_field_name="Instancia#1")

# Guardar el objeto en la base de datos.
a_record.save()
```

Podemos acceder a los campos de este nuevo registro usando la sintaxis de puntos y cambiar los valores. Tenemos que llamar a `save()` para almacenar los valores modificados en la base de datos.

```
"""
 Acceso a los valores de los campos del modelo
 usando atributos Python.
"""
print(a_record.id) # Deberia devolver 1.
print(a_record.my_field_name) # Imprime Instancia#1

"""
Cambio de un registro modificando los campos
llamando a save() a continuacion.
"""
a_record.my_field_name="Nuevo Nombre de Instancia"
a_record.save()
```

**Nota:** Si no hemos declarado ningún campo como `primary_key`, al nuevo registro se le proporcionará una automáticamente, con el nombre de campo `id`. Podemos consultar este campo después de guardar el registro anterior y debería tener un valor de 1.

2. **Búsqueda de registros**

Podemos buscar registros que coincidan con un cierto criterio usando el atributo `objects` del modelo (proporcionado por la clase base). Incluso podemos obtener todos los registros de un modelo como `QuerySet`[7], usando `objects.all()`.

```
all_books = Book.objects.all()
```

El método de Django `filter()` nos permite filtrar el `QuerySet` devuelto para que coincida un campo de texto o numérico con un criterio particular. Por ejemplo, para filtrar libros que contengan la palabra "wild", en el título y luego contarlos, podemos hacer lo siguiente:

```
wild_books = Book.objects.filter
             (title__contains='wild')

number_wild_books = Book.objects.filter
                    (title__contains='wild').count()
```

Los campos a buscar y el tipo de coincidencia son definidos en el nombre del parámetro del filtro, usando el formato: `field_name__match_type`

[7]El QuerySet es un objeto iterable, significando que contiene un número de objetos por los que podemos iterar o hacer bucle.

(debemos tener en cuenta el doble guión bajo entre `title` y `contains` anterior). En el ejemplo anterior estamos filtrando `title` por un valor sensible a mayúsculas y minúsculas. Podemos hacer otros muchos tipos de coincidencias [8]

En algunos casos, necesitaremos filtrar por un campo que define una relación *uno-a-muchos* con otro modelo (por ejemplo, una ForeignKey). En estos casos podemos referenciar", a campos dentro del modelo relacionado con un doble guión bajo adicional. Así, por ejemplo, para filtrar los libros de un género específico tienes que referenciar el `name` a través del campo `genre` como se muestra:

```
books_containing_genre =
  Book.objects.filter(genre__name__icontains=
                      'fiction')
# Will match on: Fiction, Science fiction, etc.
```

**Nota:** explicar cómo buscar regsitros usando un modelo y nombres de campos .ªbstractos puede resultar un poco confuso. En los códigos expuestos nos referomos a un modelo `Book` con campos `title` y `genre`, donde `genre` (género) es también un modelo con un solo campo `name`. Por otra parte podemos usar guiones bajos (___) para navegar por tantos niveles de relaciones (*ForeignKeyManyToManyField*) como queramos. Por ejemplo, un `Book` que tuviera diferentes "`types`", definidos usando una relación adicional "`cover`", podría tener un nombre de parámetro: `type__cover__name__exact='hard'`.

### 1.2.3. Definiendo los Modelos de la Biblioteca

En esta sección del laboratorio comenzaremos a definir los modelos para nuestra biblioteca. En nuestra aplicación **catalog**, creada en el laboratorio anterior, en **models.py** importaremos el módulo `models` que contiene la clase `models.Model`, que servirá de base para nuestros modelos (para efectos del laboratorio solo definiremos un modelo).

1. **Modelo "Libro"**

   El modelo Libro representa la información que se tiene sobre un libro, en sentido general, pero no sobre un libro particular que se encuentre disponible en la biblioteca. Este modelo utiliza campos de tipo `CharField` para representar el título (`title`) y el `isbn` del libro (notemos que el campo isbn especifica su etiqueta como ÏSBNütilizando el primer parámetro posicional, ya que la etiqueta por defecto hubiera sido Ïsbn"). Además tenemos un campo para la sinopsis (`summary`), de tipo `TextField`, ya que este texto podría ser bastante largo.

[8] Tipos de Coincidencias: https://docs.djangoproject.com/en/1.10/ref/models/querysets/#field-lookups

El género es un campo de tipo `ManyToManyField`, de manera tal que un mismo libro podrá abarcar varios géneros y un mismo género podrá abarcar muchos libros. El autor es declarado como `ForeignKey`, de modo que cada libro podrá tener un sólo autor, pero un autor podrá tener muchos libros (en la vida real, un mismo libro puede tener varios autores, pero en nuestra implementación no).

En la declaración de ambos campos, el modelo relacionado se ingresa como primer parámetro posicional, usando el nombre de la clase que lo implementa o, bien, el nombre del modelo como `string`, si éste no ha sido implementado en el archivo, antes de la declaración del campo. Otros parámetros interesantes que podemos observar, en el campo `author`, son `null=True`, que permite a la base de datos almacenar `null` si el autor no ha sido seleccionado, y `on_delete=models.SET_NULL`, que pondrá en `null` el campo si el registro del autor relacionado es eliminado de la base de datos.

El modelo también define `__str__()`, usando el campo `title` para representar un registro de la clase `Book`. El último método, `get_absoulte_url()` devuelve un URL que puede ser usado para acceder al detalle de un registro particular (para que esto funcione, debemos definir un mapeo de URL que tenga el nombre `book-detail` y una vista y una plantilla asociadas a él).

```
1  from django.db import models
2  from django.urls import reverse #Generar URL
3
4  class Book(models.Model):
5      """
6      Modelo que representa un libro (pero no un
7      Ejemplar especifico).
8      """
9
10     title = models.CharField(max_length=200)
11
12     author =
13      models.ForeignKey('Author',
14                        on_delete=models.SET_NULL,
15                        null=True)
16     """
17     ForeignKey ya que un libro tiene un solo autor,
18     pero el mismo autor puede haber escrito muchos
19     libros.
20     'Author' es un string, en vez de un objeto,
21     porque la clase Author aun no ha sido declarada.
22     """
23
```

```
    summary =
     models.TextField(max_length=1000,
                      help_text="Descripcion")

    isbn =
     models.CharField('ISBN',max_length=13,
                      help_text='13 Caracteres
                      <a href="https://www.isbn-
                      international.org/content/
                      what-isbn">ISBN number</a>')

    genre =
     models.ManyToManyField(Genre,
                            help_text="Genero")
    """
    ManyToManyField, porque un genero puede
    contener muchos libros y un libro puede
    cubrir varios generos.

    Si La clase Genre ya ha sido definida, entonces
    podemos especificar el objeto arriba, si no se
    aplica igual que Author.
    """

    def __str__(self):
        """
        String que representa al objeto Book
        """
        return self.title


    def get_absolute_url(self):
        """
        Devuelve el URL a una instancia particular
        de Book
        """
        return reverse('book-detail',
                       args=[str(self.id)])
```

### Actividades Sugeridas

- Codifique cada uno de los modelos presentados en la Figura 1.
- Investigue como hacer una vista y realice dos vistas de alguno de los modelos anteriores, use: `https://developer.mozilla.org/es/docs/Learn/Server-side/Django/Generic_views`.