Django Tutorial Part 3: Using models

Created by 이 윤 준 (yoonjoon.lee@gmail.com)

May, 2019

We'll

- show how to define models for the LocalLibrary website,
- explain what a model is, how it is declared, and some of the main field types.
- show briefly a few of the main ways you can access model data.

Overview

Django web applications access and manage data through Python objects referred to as models.

Models define the structure of stored data, including the field types and possibly also their maximum size, default values, selection list options, help text for documentation, label text for forms, etc.

The definition of the model is independent of the underlying database.

Once we've chosen what database you want to use, we don't need to talk to it directly at all.

Designing the LocalLibrary models

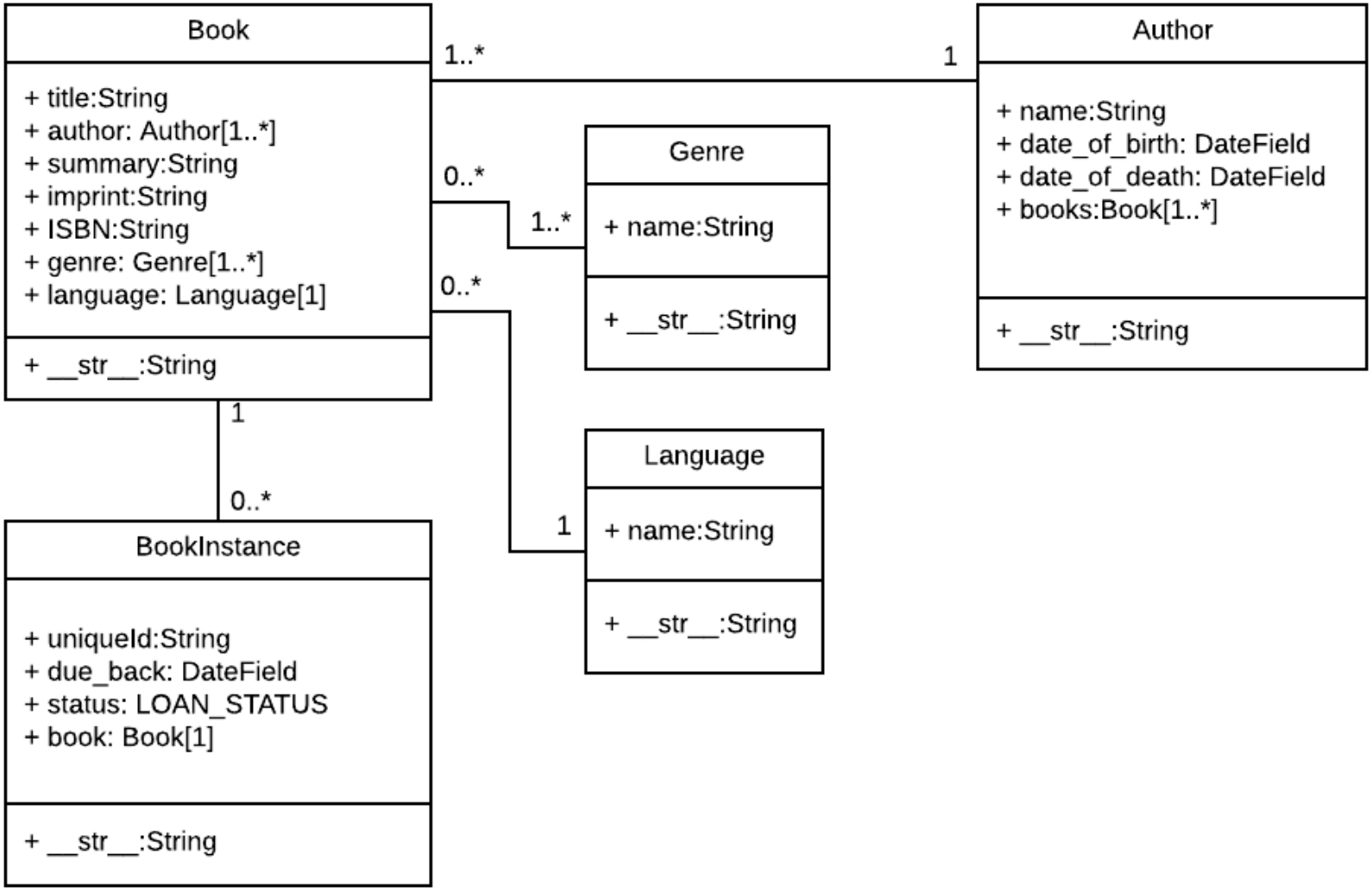
Let's think about what data we need to store and the relationships between the different objects.

- store information about books (title, summary, author, written language, category, ISBN) and that we might have multiple copies available.
- store more information about the author than just their name, and there might be multiple authors with the same or similar names.
- sort information based on book title, author, written language, and category.

When designing our models it makes sense to have separate models for every "object". In this case, the obvious objects are books, book instances, and authors.

We might also want to use models to represent selection-list options. In this case, include the book genre (e.g. Science Fiction, French Poetry, etc.) and language (English, French, Japanese).

Django allows you to define relationships that are one to one (`OneToOneField`), one to many (`ForeignKey`) and many to many (`ManyToManyField`).


Book
+ title:String
+ author: Author[1..*]
+ summary:String
+ imprint:String
+ ISBN:String
+ genre: Genre[1..*]
+ language: Language[1]
+ __str__:String
1..*
1
Author
+ name:String
+ date_of_birth: DateField
+ date_of_death: DateField
+ books:Book[1..*]
+ __str__:String
0..*
1..*
Genre
+ name:String
+ __str__:String
0..*
1
Language
+ name:String
+ __str__:String
1
0..*
BookInstance
+ uniqueId:String
+ due_back: DateField
+ status: LOAN_STATUS
+ book: Book[1]
+ __str__:String

We've created models for the book (the generic details of the book), book instance (status of specific physical copies of the book available in the system), and author.

We have also decided to have a model for the genre so that values can be created/selected through the admin interface.

We've decided not to have a model for the `BookInstance:status` — we've hardcoded the values (`LOAN_STATUS`) because we don't expect these to change.

The diagram also shows the relationships between the models, including their *multiplicities*.

The multiplicities are the numbers on the diagram showing the numbers of each model that may be present in the relationship. For example, the connecting line between the boxes shows that Book and a Genre are related.

The numbers close to the Genre model show that a book must have one or more Genres, while the numbers on the other end of the line next to the Book model show that a Genre can have zero or many associated books.

Model primer

Explain briefly a overview of how a model is defined and some of the more important fields and field arguments.

Model definition

Models are usually defined in an application's **models.py** file. They are implemented as subclasses of `django.db.models.Model`, and can include fields, methods and metadata. The code fragment shows a "typical" model, named `MyModelName`:

```python
from django.db import models

class MyModelName(models.Model):
    """A typical class defining a model, derived from the Model class."""

    # Fields
    my_field_name = models.CharField(max_length=20,
      help_text='Enter field documentation')
    ...

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

    # Methods
    def get_absolute_url(self):
        """Returns the url to access a particular
           instance of MyModelName."""
        return reverse('model-detail-view',
          args=[str(self.id)])

    def __str__(self):
        """String for representing the MyModelName object
           (in Admin site etc.)."""
        return self.my_field_name
```

Fields

A model can have an arbitrary number of fields, of any type — each one represents a column of data that we want to store in one of our database tables

```
my_field_name = models.CharField(max_length=20,
                                 help_text='Enter field documentation')
```

Our above example has a single field called `my_field_name`, of `type models.CharField`.

The field types are assigned using specific classes, which determine the type of record that is used to store the data in the database, along with validation criteria to be used when values are received from an HTML form.

We are giving our field two arguments:

- `max_length=20` - states that the maximum length of a value in this field is 20 characters.
- `help_text='Enter field documentation'` - provides a text label to display to help users know what value to provide when this value is to be entered by a user via an HTML form.

The field name is used to refer to it in queries and templates. Fields also have a label, which is either specified as an argument (`verbose_name`) or inferred by capitalising the first letter of the field's variable name and replacing any underscores with a space (for example `my_field_name` would have a default label of *My field name*).

The order that fields are declared will affect their default order if a model is rendered in a form (e.g. in the Admin site), though this may be overridden.

Common field arguments

The following common arguments can be used:

- **help_text**: Provides a text label for HTML forms (e.g. in the admin site), as described above.
- **verbose_name**: A human-readable name for the field used in field labels. If not specified, Django will infer the default verbose name from the field name.
- **default**: The default value for the field. This can be a value or a callable object, in which case the object will be called every time a new record is created.
- **null**: If `True`, Django will store blank values as `NULL` in the database for fields where this is appropriate (a `CharField` will instead store an empty string). The default is `False`.

- **blank**: If `True`, the field is allowed to be blank in our forms. The default is `False`, which means that Django's form validation will force us to enter a value. This is often used with `null=True` , because if we're going to allow blank values, we also want the database to be able to represent them appropriately.
- **choices**: A group of choices for this field. If this is provided, the default corresponding form widget will be a select box with these choices instead of the standard text field.
- **primary_key**: If `True`, sets the current field as the primary key for the model (A primary key is a special database column designated to uniquely identify all the different table records). If no field is specified as the primary key then Django will automatically add a field for this purpose.

Common field types

The more commonly used types of fields

- **CharField** is used to define short-to-mid sized fixed-length strings. We must specify the `max_length` of the data to be stored.
- **TextField** is used for large arbitrary-length strings. We may specify a `max_length` for the field, but this is used only when the field is displayed in forms (it is not enforced at the database level).
- **IntegerField** is a field for storing integer (whole number) values, and for validating entered values as integers in forms.
- **DateField** and **DateTimeField** are used for storing/representing dates and date/time information (as Python `datetime.date` and `datetime.datetime` objects, respectively). These fields can additionally declare the (mutually exclusive) parameters `auto_now=True` (to set the field to the current date every time the model is saved), `auto_now_add` (to only set the date when the model is first created) and `default` (to set a default date that can be overridden by the user).

- **EmailField** is used to store and validate email addresses.
- **FileField** and **ImageField** are used to upload files and images respectively (the `ImageField` simply adds additional validation that the uploaded file is an image). These have parameters to define how and where the uploaded files are stored.
- **AutoField** is a special type of `IntegerField` that automatically increments. A primary key of this type is automatically added to our model if we don't explicitly specify one.
- **ForeignKey** is used to specify a one-to-many relationship to another database model. The "one" side of the relationship is the model that contains the "key" (models containing a "foreign key" referring to that "key", are on the "many" side of such a relationship).
- **ManyToManyField** is used to specify a many-to-many relationship. In our library app we will use these very similarly to ForeignKeys, but they can be used in more complicated ways to describe the relationships between groups. These have the parameter `on_delete` to define what happens when the associated record is deleted (e.g. a value of `models.SET_NULL` would simply set the value to `NULL`).

Metadata

We can declare model-level metadata for our Model by declaring `class Meta`.

```
    class Meta:
        ordering = ['-my_field_name']
```

One of the most useful features of this metadata is to control the *default ordering* of records returned when we query the model type.

If we chose to sort books like this by default:

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

Another common attribute is `verbose_name`, a verbose name for the class in singular and plural form:

```
        verbose_name = 'BetterName'
```

Other useful attributes allow

- to create and apply new "access permissions" for the model.
- ordering based on another field,
- to declare that the class is "abstract"

Many of the other metadata options control what database must be used for the model and how the data is stored.

Methods

A model can also have methods.

Minimally, in every model we should define the standard Python class method `__str__()` to return a human-readable string for each object.

This string is used to represent individual records in the administration site.

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

Another common method to include in Django models is `get_absolute_url()`, which returns a URL for displaying individual model records on the website.

```
def get_absolute_url(self):
    """Returns the url to access a particular
       instance of the model."""
    return reverse('model-detail-view',
                   args=[str(self.id)])
```

Model management

Once we've defined our model classes we can use them to create, update, or delete records, and to run queries to get all records or particular subsets of records.

Creating and modifying records

To create a record we can define an instance of the model and then call `save()`.

```
    # Create a new record using the model's
    # constructor.
    record = MyModelName(my_field_name="Instance #1")

    # Save the object into the database.
    record.save()
```

We can access the fields in this new record using the dot syntax, and change the values. We have to call `save()` to store modified values to the database.

```
    # Access model field values using Python attributes.
    print(record.id) # should return 1 for the first record.
    print(record.my_field_name) # should print 'Instance #1'

    # Change record by modifying the fields,
    # then calling save().
    record.my_field_name = "New Instance Name"
    record.save()
```

Searching for records

We can search for records that match certain criteria using the model's objects attribute.

We can get all records for a model as a `QuerySet`, using `objects.all()`. The QuerySet is an iterable object, meaning that it contains a number of objects that we can iterate/loop through.

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

Django's `filter()` method allows us to filter the returned `QuerySet` to match a specified text or numeric field against particular criteria.

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

The fields to match and the type of match are defined in the filter parameter name, using the format: `field_name__match_type`.

In some cases we'll need to filter on a field that defines a one-to-many relationship to another model (e.g. a `ForeignKey`).

In this case we can "index" to fields within the related model with additional double underscores.

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

Defining the LocalLibrary Models

In this section we will start defining the models for the library. Open *models.py (in /locallibrary/catalog/)*.

```
from django.db import models

# Create your models here.
```

Genre model

This model is used to store information about the book category.

We've created the Genre as a model rather than as free text or a selection list so that the possible values can be managed through the database rather than being hard coded.

```
class Genre(models.Model):
    """Model representing a book genre."""
    name = models.CharField(max_length=200,
           help_text='Enter a book genre (e.g. Science Fiction)')

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

At the end of the model we declare a `__str__()` method, which simply returns the name of the genre defined by a particular record.

Book model

The book model represents all information about an available book in a general sense, but not a particular physical "instance" or "copy" available for loan.

```
# Used to generate URLs by reversing the URL patterns
from django.urls import reverse

class Book(models.Model):
    """Model representing a book (but not a specific copy of a book)."""
    title = models.CharField(max_length=200)

    """Foreign Key used because book can only have one author,  multiple books.
       but authors can have Author as a string rather than
       object because it hasn't been declared yet in the file."""
    author = models.ForeignKey('Author',
                               on_delete=models.SET_NULL,
                               null=True)
    summary = models.TextField(max_length=1000,
                               help_text='Enter a brief description of the book')
    isbn = models.CharField('ISBN', max_length=13,
        help_text='13 Character <a href="https://www.isbn-international.org/conten
```

```
    """ManyToManyField used because genre can contain
       many books. Books can cover many genres.
       Genre class has already been defined so we can
       specify the object above."""
    genre = models.ManyToManyField(Genre,
                                   help_text='Select a genre for this book')

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

    def get_absolute_url(self):
        """Returns the url to access a detail record for this book."""
        return reverse('book-detail', args=[str(self.id)])
```

The genre is a `ManyToManyField`, so that a book can have multiple genres and a genre can have many books. The author is declared as `ForeignKey`, so each book will only have one author, but an author may have many books.

In both field types the related model class is declared as the first unnamed parameter using either the model class or a string containing the name of the related model.

We must use the name of the model as a string if the associated class has not yet been defined in this file before it is referenced.

The other parameters of interest in the `author` field are `null=True`, which allows the database to store a `Null` value if no author is selected, and `on_delete=models.SET_NULL`, which will set the value of the author to `Null` if the associated author record is deleted.

The model also defines `__str__()`, using the book's `title` field to represent a `Book` record. The method, `get_absolute_url()` returns a URL that can be used to access a detail record for this model.

BookInstance model

The `BookInstance` represents a specific copy of a book that someone might borrow, and includes information about whether the copy is available or on what date it is expected back, "imprint" or version details, and a unique id for the book in the library.

The model uses

- `ForeignKey` to identify the associated `Book`.
- `CharField` to represent the imprint (specific release) of the book.

```python
import uuid # Required for unique book instances

class BookInstance(models.Model):
    """Model representing a specific copy of a book
       (i.e. that can be borrowed from the library)."""
    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        help_text='Unique ID for this particular book across whole library')
    book = models.ForeignKey(
        'Book',
        on_delete=models.SET_NULL,
        null=True)
    imprint = models.CharField(max_length=200)
    due_back = models.DateField(null=True, blank=True)

    LOAN_STATUS = (
        ('m', 'Maintenance'),
        ('o', 'On loan'),
        ('a', 'Available'),
        ('r', 'Reserved'),
    )

    status = models.CharField(
        max_length=1,
        choices=LOAN_STATUS,
        blank=True,
        default='m',
        help_text='Book availability',
    )
```

```
    class Meta:
        ordering = ['due_back']

    def __str__(self):
        """String for representing the Model object."""
        return f'{self.id} ({self.book.title})'
```

We additionally declare a few new types of field:

- UUIDField
- DateField
- status

The model `__str__()` represents the `BookInstance` object using a combination of its unique id and the associated `Book`'s title.

Author model

```
class Author(models.Model):
    """Model representing an author."""
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    date_of_birth = models.DateField(
        null=True,
        blank=True)
    date_of_death = models.DateField(
        'Died',
        null=True,
        blank=True)

    class Meta:
        ordering = ['last_name', 'first_name']

    def get_absolute_url(self):
        """Returns the url to access a particular author
           instance."""
        return reverse('author-detail', args=[str(self.id)])

    def __str__(self):
        """String for representing the Model object."""
        return f'{self.last_name}, {self.first_name}'
```

Re-run the database migrations

Now re-run your database migrations to add them to your database.

```
py -3 manage.py makemigrations
py -3 manage.py migrate
```

Language model — challenge

Imagine a local benefactor donates a number of new books written in another language. The challenge is to work out how these would be best represented in our library website, and then to add them to the models.

Some things to consider:

- Should "language" be associated with a `Book`, `BookInstance`, or some other object?
- Should the different languages be represented using model, a free text field, or a hard-coded selection list?

After you've decided, add the field.