

# Models with Django Admin

## Chapter 4

COMP222-Chapter 4

1

## Objectives

- In this chapter we will use a database for the first time to build a basic Message Board application where users can post and read short messages.
- We'll explore Django's powerful built-in admin interface which provides a visual way to make changes to our data.
- An example to use generic class-based [ListView](#) which contains the logic to get all the records in the model.
- Introduce the built-in generic class-based [DetailView](#)

COMP222-Chapter 4

2

## Introduction

- For Django's MTV framework, where

- **M** stands for "Model,"
- **T** stands for "Template,"
- **V** stands for "View,"

we have covered the views function (Chapter 2) and templates (Chapter 3).

- In this Chapter, we will discuss Models for making database-driven websites

3

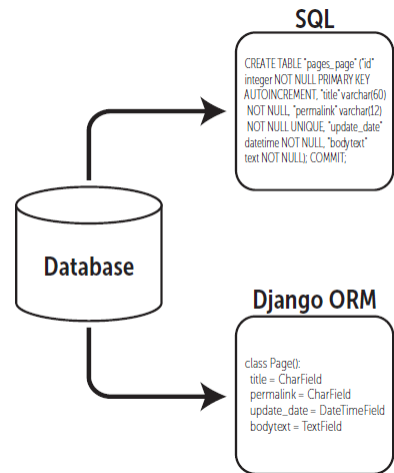
## ORM: Object Relational Mapper

- When you think of databases, you will usually think of the *Structured Query Language (SQL)*, with which we query the database for the required data.
- With Django, querying an underlying database is taken care of by the **Object Relational Mapper (ORM)**.
- A model is a Python object that describes your database table's data. Instead of directly working on the database via SQL, you only need to manipulate the corresponding Python model object.
- ORMs also make it theoretically possible to switch an application between various relational databases. For example, a developer could use SQLite for local development and MySQL in production. A production application could be switched from MySQL to PostgreSQL with minimal code modifications.

4

## Django Models

- Django's models provide an Object-relational Mapping (ORM) to the underlying database.
- Most common databases are programmed with some form of Structured Query Language (SQL), however each database implements SQL in its own way.
- An ORM tool on the other hand, provides a simple mapping between an *object* (the 'O' in ORM) and the underlying database, without the programmer needing to know the database structure, or requiring complex SQL to manipulate and retrieve data.



5

## Database-driven websites

- Most modern web applications often involves interacting with a database.
- Behind the scenes, a database-driven website connects to a database server, retrieves some data out of it, and displays that data on a web page.
- The site might also provide ways for site visitors to populate the database on their own.

6

## RAW SQL queries

- In this example view, we use the MySQLdb library to connect to a MySQL database, retrieve some records, and feed them to a template for display as a web page:

```
from django.shortcuts import render
import MySQLdb

def book_list(request):
    db = MySQLdb.connect(user='me', db='mydb', passwd='secret',
        host='localhost')
    cursor = db.cursor()
    cursor.execute('SELECT name FROM books ORDER BY name')
    names = [row[0] for row in cursor.fetchall()]
    db.close()
    return render(request, 'book_list.html', {'names': names})
```

7

## Problems of raw SQL queries

This approach works, but some problems should jump out at you immediately:

- We're hard-coding the database connection parameters. Ideally, these parameters would be stored in the Django configuration.
- We're writing a fair bit of boilerplate code: creating a connection, creating a cursor, executing a statement, and closing the connection. Ideally, all we'd have to do is specify which results we wanted.
- It ties us to MySQL. If, down the road, we switch from MySQL to PostgreSQL, we'll most likely have to rewrite a large amount of our code. Ideally, the database server we're using would be abstracted, so that a database server change could be made in a single place. (This feature is particularly relevant if you're building an open-source Django application that you want to be used by as many people as possible.)

8

## Defining Models in Python

- A Django model is a description of the data in your database, represented as Python code.
- It's your data layout—the equivalent of your SQL CREATE TABLE statements—except it's in Python instead of SQL, and it includes more than just database column definitions.
- SQL is inconsistent across database platforms. If you're distributing a web application, for example, it's much more pragmatic to distribute a Python module that describes your data layout than separate sets of CREATE TABLE statements for MySQL, PostgreSQL, and SQLite.

9

## Django: built-in support for database backends

- Django provides built-in support for several types of database backends.
- With just a few lines in our [settings.py](#) file it can support PostgreSQL, MySQL, Oracle, or SQLite.

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': BASE_DIR / 'db.sqlite3',  
    }  
}
```

settings.py

- But the simplest—by far—to use is SQLite because it runs off a single file and requires no complex installation.
- Django uses SQLite by default for this reason and it's a perfect choice for small projects

## Creating `posts` app in the existing project

To use Django's models, you must create a Django app. Models must live within apps. Hence, we'll create a new app called `posts` for a Message Board application.

Our initial setup involves the following steps:

Step 1: Inside the project folder with virtual environment activated, create a new app called `posts`:

```
python manage.py startapp posts
```

Step 2: Update settings.py

Add the `posts` app to our project under `INSTALLED_APPS`.

COMP222-Chapter 3

11

## Step 3: Creating the database for the app

- Execute the `migrate` command to create an initial database based on Django's default settings.

```
(django) 05:34 ~/django_projects/myTestSite $ python manage.py migrate
```

- If you look inside the directory with the `ls` command, there's now a `db.sqlite3` file representing SQLite database.

- To make sure the database reflects the current state of your project, run migrate (and also makemigrations) each time you update a model.

```
(django) 02:05 ~/django_projects/myTestSite $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
(django) 02:07 ~/django_projects/myTestSite $ ls
catalog  db.sqlite3  manage.py  mbPosts  myTestSite  templates
```

COMP222

## Step 4: Create a database model

- Create a database model where we can store and display posts from our users. Creating a Django model creates a corresponding table in the database.
- Open the [posts/models.py](#) file and look at the default code which Django provides.

```
# posts/models.py
from django.db import models
# Create your models here
```

Django imports a module models to build new database models, which will “model” the characteristics of the data in our database.

- We want to create a model to store the textual content of a message board post, which we can do so as follows:

```
class Post(models.Model):
    text = models.TextField()
```

- This created a new database model called Post which has the database field text and the type of content it will hold is TextField().

COMP222-Chapter 4

13

## Field name restrictions

### Field name restrictions

- A field name cannot be a Python reserved word.
- A field name cannot contain more than one underscore in a row.

14

# Common field types

Field	Default Widget	Description
AutoField	N/A	An IntegerField that automatically increments according to available IDs.
BigIntegerField	NumberInput	A 64-bit integer, much like an IntegerField except that it is guaranteed to fit numbers from -9223372036854775808 to 9223372036854775807
BinaryField	N/A	A field to store raw binary data. It only supports bytes assignment. Be aware that this field has limited functionality.
BooleanField	CheckboxInput	A true/false field. If you need to accept null values then use NullBooleanField instead.
CharField	TextInput	A string field, for small- to large-sized strings. For large amounts of text, use TextField. CharField has one extra required argument: max_length. The maximum length (in characters) of the field.
DateField	DateInput	A date, represented in Python by a datetime.date instance. Has two extra, optional arguments: auto_now which automatically set the field to now every time the object is saved, and auto_now_add which automatically set the field to now when the object is first created.

<https://docs.djangoproject.com/en/3.1/ref/models/fields/>

15

# Common field types (cont'd)

Field	Default Widget	Description
FloatField	NumberInput	A floating-point number represented in Python by a float instance. Note when field.localize is False, the default widget is TextInput
DecimalField	TextInput	A fixed-precision decimal number, represented in Python by a Decimal instance. Has two required arguments: max_digits and decimal_places.
IntegerField	NumberInput	An integer. Values from -2147483648 to 2147483647 are safe in all databases supported by Django.
PositiveIntegerField	NumberInput	An integer. Values from 0 to 2147483647 are safe in all databases supported by Django.
SmallIntegerField	NumberInput	Like an IntegerField, but only allows values under a certain point. Values from -32768 to 32767 are safe
NullBooleanField	NullBooleanSelect	Like a BooleanField, but allows NULL as one of the options.
TextField	Textarea	A large text field. If you specify a max_length attribute, it will be reflected in the Textarea widget of the auto-generated form field.

16



# Common field types with field options

```
from django.db import models

class NewTable(models.Model):
    bigint_f = models.BigIntegerField()
    bool_f = models.BooleanField()
    date_f = models.DateField(auto_now=True)
    char_f = models.CharField(max_length=20, unique=True)
    datetime_f = models.DateTimeField(auto_now_add=True)
    decimal_f = models.DecimalField(max_digits=10, decimal_places=2)
    float_f = models.FloatField(null=True)
    int_f = models.IntegerField(default=2010)
    text_f = models.TextField()
```

COMP222-Chapter 4

## Python Indentation

Most of the programming languages like C and Java use braces { } to define a block of code. Python, however, uses indentation.

A code block (body of a function, loop, etc.) starts with indentation and ends with the first unindented line.

The amount of indentation is up to you, but it must be consistent throughout that block.

`auto_now`: fields are updated to the current timestamp every time an object is saved and are therefore perfect for tracking when an object was last modified.

`auto_now_add`: field is saved as the current timestamp when a row is first added to the database, and is therefore perfect for tracking when it was created.

## Field.choices

- A sequence consisting itself of iterables of exactly two items (e.g. [(A, B), (A, B) ...]) to use as choices for this field.
- If choices are given, the default form widget will be a select box with these choices instead of the standard text field.
- The first element in each tuple is the actual value to be set on the model, and the second element is the human-readable name. For example:

```
YEAR_IN_SCHOOL_CHOICES = [
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
]
```

COMP222-Chapter 4

18

## Field.choices: Django Best Practice Preferred Way

- Generally, it's best to define a suitably-named constant for each value.
- Single leading underscore: weak "internal use" indicator. Indicates that it is for internal use only, and is not supposed to be imported and used publicly.

```
class Book(models.Model):
    AVAILABLE = 'available'
    BORROWED = 'borrowed'
    ARCHIVED = 'archived'

    STATUS = (
        (AVAILABLE, _('Available to borrow')),
        (BORROWED, _('Borrowed by someone')),
        (ARCHIVED, _('Archived - not available anymore')),
    )
    # [...]
    status = models.CharField(
        max_length=32,
        choices=STATUS,
        default= AVAILABLE,
    )
```

COMP222-Chapter 4

19

## Step 5 & 6: Activating models

- Now that our new model is created, we need to activate it.
- Whenever we create or modify an existing model, we'll need to update Django in a two-step process.
  1. First we create a migration file **for an app** with the **makemigrations** command which generate the SQL commands.

**(django) 05:34 ~/django\_projects/myTestSite \$**  
**python manage.py makemigrations posts**

```
Migrations for 'posts':
posts/migrations/0001_initial.py
- Create model Post
```

2. Second we build the actual database with **migrate** which executes the instructions in our migrations file.

**(django) 05:34 ~/django\_projects/myTestSite \$**  
**python manage.py migrate posts**

```
Operations to perform:
  Apply all migrations: posts
Running migrations:
  Applying posts.0001_initial... OK
```

COMP222-Chapter 4

20

## Makemigrations vs Migrate

- Makemigrations will create the migration.
- A **migration** basically tells your database how it's being changed (i.e. new column added, new table, dropped tables etc.).
  - Create the migrations: generate the SQL commands
- **Migrate** is what pushes your changes to your database. It will run all the migrations created (or the ones that haven't been pushed yet).
  - Run the migrations: execute the SQL commands
- It is necessary to run both the commands to complete the migration of the database tables to be in sync with your models.

21

## Make Migrations vs Migrate (cont'd)

- You should think of migrations as a version control system for your database schema.
- **makemigrations** is responsible for packaging up your model changes into individual migration files - analogous to commits - and **migrate** is responsible for applying those to your database.
- The migration files for each app live in a "migrations" directory inside of that app.
- You can use **showmigrations** command to list a project's migrations and their status.
- You can revert back by migrating to the previous migration, if needed.

COMP222-Chapter 4

22

## Note on Make Migrations & Migrate

- Note that you don't have to include the app name after either `makemigrations` or `migrate`.
- If you simply run the commands then they will apply to all available changes.
- But it's a good habit to be specific. If we had two separate apps in our project, and updated the models in both, and then ran `makemigrations`, it would generate a migrations file containing data on both changes.
- This makes debugging harder in the future.
- You want each migration file to be as small and isolated as possible. That way if you need to look at past migrations, there is only one change per migration rather than one that applies to multiple apps.

COMP222-Chapter 4

23

## sqlmigrate command

- After running the command `python manage.py makemigrations yourAppName`, a file called `"0001_initial.py"` will be created in your migrations folder of your app `yourAppName`.
- You can run the `sqlmigrate` command to display SQL statements for a given migration.

(django)05:34~/django\_projects/myTestSite \$ python manage.py sqlmigrate yourAppName 0001\_initial

```
(django2) 05:58 ~/django_projects/myTestSite $ python manage.py sqlmigrate posts 0001_initial
BEGIN;
--
-- Create model Post
--
CREATE TABLE "posts_post" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "text" text NOT NULL);
COMMIT;
(django2) 05:58 ~/django_projects/myTestSite $
```

COMP222-Chapter 4

24

## inspectdb command

- When you already have a database file to work with and you want to build a Django app to use it,  
`python manage.py inspectdb > models.py`
- The manage.py command `inspectdb`, looks at the DB defined in the settings.py file and then proceeds to do its best to automatically create the right mapping from tables and columns to Django model objects.
- We pipe the output from that command to models.py which is the expected file name for the object relational mappings in a Django application.

COMP222-Chapter 4

25

## Primary key

- By default, Django automatically gives every model an auto-incrementing integer primary key field called `id`, which is used as the **primary key** for that **model** (refer to slide 22).
- You can create your own **primary key** field by adding the keyword `primary_key=True` to a field.
- `primary_key=True` implies `null=False` and `unique=True`.
- If you add your own **primary key** field, the automatic one will not be added.
- The primary key field is read-only. If you change the value of the primary key on an existing object and then save it, a new object will be created alongside the old one.

COMP222-Chapter 4

26

## Single-column primary key

- Each Django model is required to have a single-column primary key.
- A workaround:

```
class Hop(models.Model):
    migration = models.ForeignKey('Migration')
    host = models.ForeignKey(User, related_name='host_set')
    class Meta:
        constraints = [models.UniqueConstraint(fields= ['migration', 'host']),
                        name='migration_and_host_uniq' ]
```

COMP222-Chapter 4

27

## Step 7: Django Admin: create superuser

- Django provides us with a robust admin interface for interacting with our database.
- To use the Django admin, we first need to create a superuser.
- In your command line console, type the following command and respond to the prompts for a username, email, and password:  
 (django) 05:34 ~/django\_projects/myTestSite \$ **python manage.py createsuperuser**
- Login to Django Admin on the browser via [yourusername.pythonanywhere.com/admin/](http://yourusername.pythonanywhere.com/admin/) by entering the username and password you just created.
- You will see the Django admin homepage next.
- By default, this URL route is in the project-level urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
]
```

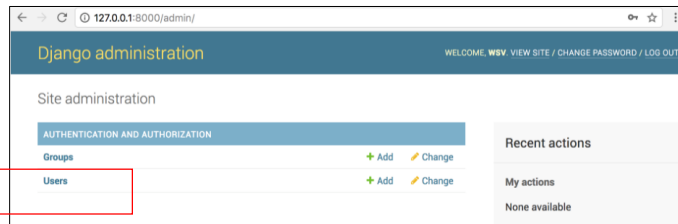
COMP222-Chapter 4

28

## Step 8: Django Admin: edit admin.py

- Our posts app is not displayed on the main admin page!

Referring back to slide 12, the auth model is automatically created to handle Users.



- We need to explicitly tell Django what to display in the admin.
- Each model that you want Django to represent in the admin interface needs to be registered.
- Edit `posts/admin.py` file to look like this:

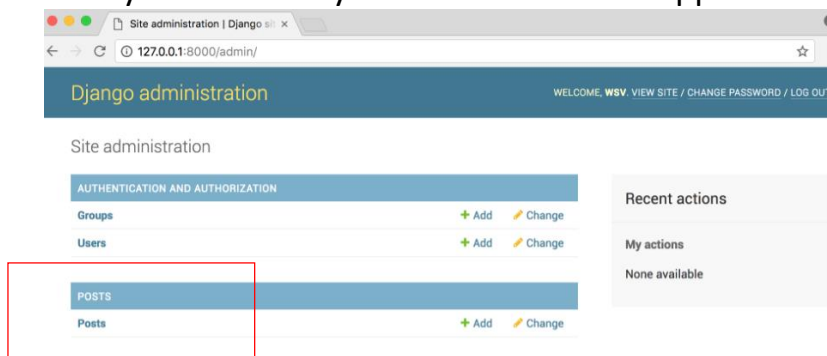
```
from django.contrib import admin
from .models import Post
admin.site.register(Post)
```

COMP222-Chapter 4

29

## Admin Homepage updated

- Django now knows that it should display our posts app and its database model Post on the admin page.
- If you refresh your browser you'll see that it now appears.

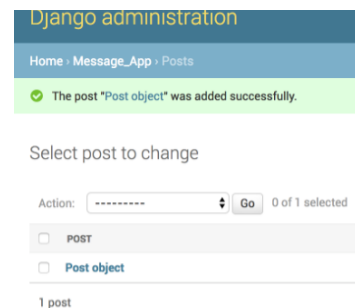


COMP222-Chapter 4

30

## Django Admin: Adding a new entry

- Now let's create our first message board post for our database.
- Click on the **+ Add button** opposite Posts. Enter your own text in the Text form field.
- Then click the "Save" button, which will redirect you to the main Post page.
- However, if you look closely, our new entry is called "Post object", which isn't very helpful.



COMP222-Chapter 4

31

## str() method to improve readability of models

- Within the `posts/models.py` file, add a new method `__str__()` as follows:

Remember to pay attention to proper indentation.

```
class Post(models.Model):
    text = models.TextField()
    def __str__(self):
        """A string representation of the model."""
        return self.text[:50]
```

Double underscore: Name mangling is intended to give classes an easy way to define "private" instance variables and methods.

- Implementing `__str__()` is a quick way to change the representation of an object from a meaningless string to understandable data.
- Note that `text` without the `self` would cause a `NameError`, as it would be referencing a local or global variable which doesn't exist.
  - `text` is an attribute of the model instance, and can only be referenced via `self`.
- Reload on the Web tab page. Refresh your Admin page in the browser, it has changed to a much more descriptive representation of our database entry.

COMP222-Chapter 4

32



## Python 3's f-Strings (formatted string literal)

- f-strings let you include the value of Python expressions inside a string by prefixing the string with f or F and writing expressions as {expression}

```
def __str__(self):
```

```
    """String for representing the Model object."""
```

```
    return f'{self.id} ({self.book.title}) '
```

Strings in python are surrounded by either single quotation marks, or double quotation marks.

- A possible representation will be: 3 (Master Django)

A docstring is the first statement after a class or function declaration.  
single-line docstrings: """This is a single line docstring"""

The docstring becomes the `__doc__` special attribute for the object. `__doc__` is used by many tools and applications (including Django's admin documentation tool) to create documentation for your code. For more information on docstrings, see **PEP257** (Python Enhancement Proposal)

COMP222-Chapter 4

33

## Django Admin Security

- Django admin is an *admin interface* limited to trusted site administrators, that enables the adding, editing and deletion of site content.
- The admin site is not intended to be a public interface to data, nor is it intended for sophisticated sorting and searching of your data.
- The admin site is for trusted site administrators. Keeping this sweet spot in mind is the key to effective admin-site usage.
- One of the most important thing is to make Django admin secure.
- Before you deploy your application you must change **admin/** path to **something only you know**. Otherwise, someone can easily type /admin in url and access to administrator login page.
- Now, work on Lab 4.1 for practice.

34

## Display database content on our webpage: Views/Templates/URLs

- In order to display our database content on our homepage, we have to wire up our views, templates, and URLConfs.
- In Chapter 3, we used the built-in generic [TemplateView](#) to display a template file on our homepage.
- Now we want to list the contents of our database model with the generic class-based [ListView](#).

COMP222-Chapter 4

35

## Views with generic class-based ListView

In the posts/views.py file enter the code below:

```
from django.views.generic import ListView
from .models import Post
class HomePageView(ListView):
    model = Post
    template_name = 'home.html'
```

```
<!-- templates/home.html -->
<h1>Message board homepage</h1>
<ul>
    {% for post in object_list %}
        <li>{{ post }}</li>
    {% endfor %}
</ul>
```

Can add the `{% empty %}` template tag to handle the case when there are no objects returned by `ListView`.

- First, import `ListView`
- In the second line we define which model we're using.
- In the view, we subclass the generic `ListView`, specify the model name and template reference.
- Internally, `ListView` returns an object called `object_list`, that contains all the post objects that we want to display in our template.
- The idea is similar to executing the SQL statement `"SELECT * FROM Post"`

COMP222-Chapter 4

36

## Filtering with queryset

- Instead of listing all the posts, we can use queryset to filter to only display posts with a published flag set to True.

```
class PostDetailView(ListView):
    model = Post
    queryset = Post.objects.filter(published=True)
```

OR

```
class PostDetailView(ListView):
    ...
    def get_queryset(self):
        return Post.objects.filter(published=True)
```

As a matter of fact, in our views, specifying `model = Post` is actually shorthand for saying `queryset = Post.objects.all()`

`filter()` method returns a `QuerySet`, which is like a list.

The idea is similar to executing the SQL statement `"SELECT * FROM Post WHERE published = True"`

COMP222-Chapter 5

37

## `get_object_or_404()` method in Django Models

- Consider a scenario with a drop-down list containing the list of publishers. Based on the user's selection, only books of the chosen publisher are listed (instead of all the books).
- Using `get_queryset()` to filter books of the chosen publisher.

```
class PublisherBookList(ListView):
    template_name = 'books_by_publisher.html'
    def get_queryset(self):
        self.publisher = get_object_or_404(Publisher, name=self.kwargs['publisher'])
        return Book.objects.filter(publisher=self.publisher)
```

Check the Publisher table to see if it contains the value of keyword argument named publisher (see slide 45).

COMP222-Chapter 4

38

## class-based generic views

- When building a web application, there are certain kind of views that we build again and again, such as a view that displays all records in the database (e.g., displaying all books in the books table), etc.
- These kinds of views perform the same functions and lead to repeated code.
- As **DRY (Don't Repeat Yourself)** principle, Django uses class-based generic views.
- When using generic views, all we have to do is inherit the desired class from `django.views.generic` module and provide some information like [model, template name, context object name](#), etc.

COMP222-Chapter 3

39

## Templates (cont'd)

- In our templates file `home.html` we can use the Django Templating Language's **for loop** to list all the objects in `object_list` returned by `ListView`.

```
<!-- templates/home.html -->
<h1>Message board homepage</h1>
<ul>
  {% for post in object_list %}
    <li>{{ post }}</li>
  {% endfor %}
</ul>
```

While `object_list` works just fine, it isn't all that "friendly" to template authors: they have to "just know" that they're dealing with posts here.

When you are dealing with an object or queryset, Django is able to populate the context using the lowercased version of the model class' name. This is provided in addition to the default `object_list` entry, but contains exactly the same data, i.e. `post_list`

COMP222-Chapter 4

40

## context\_object\_name

- You can manually set the name of the context variable via the `context_object_name` attribute.
- Rewrite our `HomePageView` as follows by adding “`context_object_name`”.

```
class HomePageView (ListView):
    model = Post
    template_name = 'home.html'
    context_object_name = 'posts_PostList'
```

- Our template tag now can use more meaningful name to access

```
<!-- templates/ home.html -->
<h1> Message board homepage </h1>
<ul>
    {% for post in posts_PostList %}
        <li> {{post }} </li>
    {% empty %}
        <p> No messages yet.
    {% endfor %}
</ul>
```

COMP222-Chapter 4

41

## URLConfs

- The last step is to set up our URLConfs. In the `project-level urls.py` file, include our posts and add include on the second line.
- Then `create an application-level urls.py` file (`posts/urls.py`) to include the following code:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('posts.urls')),
]
```

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

urlpatterns = [
    path('', views.HomePageView.as_view(), name='home'),
]
```

- What is the URL that will map to the view function `HomePageView`?
- Now work on lab 4.2 for practice.

COMP222-Chapter 4

42

## Class-based DetailView

- **DetailView** should be used when you want to present detail of a single model instance, which is similar to executing the SQL statement “`SELECT * FROM Table WHERE id = ....`”.
- **DetailView** expects a primary key passed to it as the identifier, via the URL.
- Hence, our URL pattern for “post\_detail” needs a primary key to locate a specific record:

```
path('post/<int:pk>/', views.BlogDetailView.as_view(), name='post_detail'),
```

- That means in order for this route to work, we must pass in an argument with the primary key of the object. So the route for the first entry will be at post/1.

COMP222-Chapter 5

43

## Class-based DetailView (cont'd)

- By default, DetailView provides a context object we can use in our template called either **object** or **the lowercase name of our model**.

```
# blog/views.py
from django.views.generic import DetailView
from .models import Post
class BlogDetailView (DetailView):
    model = Post
    template_name = 'post_detail.html'
```

```
1 <!-- templates/post_detail.html -->
2 |
3 <h2>{{ post.title }}</h2>
4 <p>{{ post.body }}</p>
```

- Since DetailView is to return the details of a selected post given its id, we do NOT need a for loop to iterate over the objects.

COMP222-Chapter 5

44

## DetailView returns 404 error

- How to redirect from DetailView when the specified object does not exist?
- If the object does not exist, there is an `Http404` Page Not Found exception.
- You can catch exceptions in the `get` method of the view.

```
from django.http import Http404, HttpResponseRedirect
from django.shortcuts import redirect
from django.urls import reverse
from django.views.generic import DetailView
class MyDetailView(DetailView):
    def get(self, request, *args, **kwargs):
        try:
            return super().get(request, *args, **kwargs)
        except Http404:
            return redirect(reverse('my_list_view_name'))
```

- The `super()` function represents the parent class of the current one (in this case, this means `DetailView`).
- `get()` is for method = 'get'
- `*args` (Non-Keyword Arguments), with `*` meaning variable-length argument list
- `**kwargs` (Keyword Arguments), allows you to **handle named arguments not being defined in advance**.

Either use a URL named pattern, or simply display a message.  
`return HttpResponseRedirect('<h1>Page not found</h1>')`

COMP222-Chapter 5

45

## Django's DetailView - `get_object()`

- Django's class-based **DetailView** relies on `SingleObjectMixin`, which provides a `get_object()` method that figures out the **object** based on the URL of the request.
- By `SingleObjectMixin`, `DetailView` refers to a view (logic) to display one instance of a table in the database.
- By default, it expects the slug or `pk` as argument for the generic view.
- We can override `get_object()` to replace this default behavior, so that it gets the single desired object from the database, e.g.

```
class TicketDetail(DetailView):
    model = Ticket

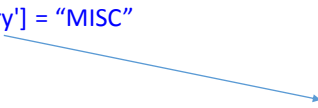
    def get_object(self, queryset=None):
        return Ticket.objects.get(uuid=self.kwargs.get("uuid"))
```

46

## Returning extra data with a generic view

- Over-riding `get_context_data()`

```
class AppDetailView(generic.DetailView):  
    model = Application  
    def get_context_data(self, **kwargs):  
        context = super().get_context_data(**kwargs)  
        context['category'] = "MISC"  
        return context
```



```
<h1>{{ object.title }}</h1>  
<p>{{ object.description }}</p>  
<p>{{ category }}</p>
```

Now work on Labs 5.1 (a quick review of Labs 1 to 4) and 5.2 for practice.

## More on Django Models



## Forging relationships

Django provides three types of fields for forging relationships between models in your database.

- `ForeignKey`, a field type that allows us to create a one-to-many relationship;
- `OneToOneField`, a field type that allows us to define a strict one-to-one relationship; and
- `ManyToManyField`, a field type which allows us to define a many-to-many relationship.
  - It doesn't matter which model has the [ManyToManyField](#), but only put it in one of the models – not both.
  - It is suggested, but not required, that the name of a [ManyToManyField](#) be a plural describing the set of related model objects.

```
models.py
from django.db import models

# Create your models here.
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

The Publisher model is equivalent to the following table

```
CREATE TABLE "books_publisher" (
  "id" serial NOT NULL PRIMARY KEY,
  "name" varchar(30) NOT NULL,
  "address" varchar(50) NOT NULL,
  "city" varchar(60) NOT NULL,
  "state_province" varchar(30) NOT NULL,
  "country" varchar(50) NOT NULL,
  "website" varchar(200) NOT NULL
);
```

COMP222-Chapter 4

49

## The book model

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

- In our example models, Book has a `ManyToManyField` called `authors`.
- This designates that a book has one or many authors
  - the Book table doesn't get an authors column.
  - Rather, Django creates an additional table - a many-to-many *join table* – that handles the mapping of books to authors.
  - The Book table has a publisher column added though (`ForeignKey`).
- However, there is an API to access the authors via `Book.authors.all()`
  - The model that defines the [ManyToManyField](#) uses the attribute name of that field itself, whereas the “reverse” model uses the lowercased model name of the original model, plus `'_set'`
  - So, to list all the books of an author in a template file, use `author.book_set.all`

50

## Extra fields on many-to-many relationships

- For `ManyToManyField`, sometimes you may need to associate data with the relationship between two models.
- For these situations, Django allows you to specify the model that will be used to govern the many-to-many relationship.
  - You can then put extra fields on the intermediate model.
  - The intermediate model is associated with the `ManyToManyField` using the `through` argument to point to the model that will act as an intermediary.

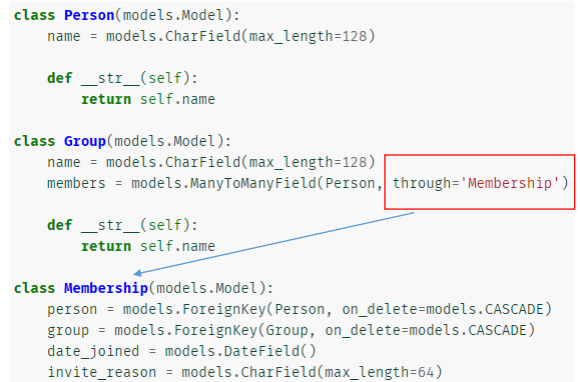
```
class Person(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

    def __str__(self):
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

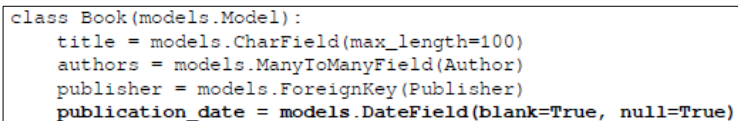


51

## Field options: `null`, `blank`

- Let's change our Book model to allow a blank `publication_date`.

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField(blank=True, null=True)
```



- `null=True` changes the semantics of the database – that is, it changes the CREATE TABLE statement to remove the NOT NULL from the `publication_date` field.
- `blank=True` determines whether the field will be required in Django admin and custom forms.
- The combo of the two is so frequent because if you're going to allow a field to be blank in your form, you're going to also need your database to allow NULL values for that field. The exception is `CharFields` and `TextFields`, which in Django are never saved as NULL. Blank values are stored in the DB as an empty string ("")
- Django does not automate changes to database schemas, so remember to `run makemigrations & migrate` command whenever you make such a change to a model.

52

## Customizing field labels – `verbose_name`

- On the admin site's edit forms, each field's label is generated from its model field name.
- To customize a label, specify `verbose_name` in the appropriate model field.
- Here's how to change the label of the `Author.email` field to **e-mail**, with a hyphen:

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(blank=True, verbose_name='e-mail')
```

- Note that you shouldn't capitalize the first letter of a `verbose_name` unless it should always be capitalized (for example "USA state").
- Django will automatically capitalize it when it needs to, and it will use the exact `verbose_name` value in other places that don't require capitalization.

53

## Retrieving related data of a foreign key

- In the template file for the details of a publisher, we want to display all the books published by a specific publisher.
- In our case, working with a foreign key, we want to follow a relationship **backward**: for each Publisher look up related Book model.
- Django has a built-in syntax known as `FOO_set` where FOO is the lowercased source model name.
- So for `Publisher` model, we can use `book_set` to access all instances of the model.

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

COMP222-Chapter 4

54

## Retrieving related data of a foreign key: example

- To access each book, use `publisher.book_set.all` which means first look at the publisher model, then the book model, and select all included. It can take a little while to become accustomed to this syntax for referencing foreign key data in a template!

```
<h2>{{ publisher.name }}</h2>
<p></p>
<ul>
  {% for book in publisher.book_set.all %}
    <li>{{ book.title }}</li>
  {% empty %}
    <p> no books under this publisher</p>
  {% endfor %}
</ul>
```

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

COMP222-Chapter 4

55

## Following relationships “backward” - override the FOO\_set name by setting the related\_name

- You can override the FOO\_set name by setting the related\_name parameter in the ForeignKey definition.
- For example, if the Book model was altered to `publisher = models.ForeignKey(Publisher, related_name = 'books')`, then the code would look like this:

```
{% for book in publisher.books.all %}
```

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher, related_name='books')
    publication_date = models.DateField()
```

COMP222-Chapter 4

56

## Adding Extra Manager Methods

- Adding extra manager methods can add “table-level” functionality to your models.
- For example, let’s give our Book model a manager method `title_count()` that takes a keyword and returns the number of books that have a title containing that keyword.
- This encapsulates commonly executed queries so that we don’t have to duplicate code.

```
# models.py
from django.db import models
# ... Author and Publisher models here ...
class BookManager(models.Manager):
    def title_count(self, keyword):
        return self.filter(title__icontains=keyword).count()

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
    num_pages = models.IntegerField(blank=True, null=True)
    objects = BookManager()
```

COMP222-Chapter 4

- We’ve assigned `BookManager()` to the `objects` attribute on the model.
- This has the effect of replacing the “default” manager for the model, which is called `objects` and is automatically created if you don’t specify a custom manager.

With this manager in place, we can now do this:

```
>>> Book.objects.title_count('django')
4
>>> Book.objects.title_count('python')
18
```

57

## Bend the Django ORM to your will, but accept the limitations

- By default the Django ORM does not do any joins.
- When application code accesses related entities or child sets, you might need to write manager method with raw queries.
- With this example, use `OpinionPoll.objects.with_counts()` to return that list of `OpinionPoll` objects with `num_responses` attributes.

```
class PollManager(models.Manager):
    def with_counts(self):
        from django.db import connection
        cursor = connection.cursor()
        cursor.execute("""SELECT p.id, p.question, p.poll_date, COUNT(*)
            FROM polls_opinionpoll p, polls_response r
            WHERE p.id = r.poll_id
            GROUP BY p.id, p.question, p.poll_date""")
        result_list = []
        for row in cursor.fetchall():
            p = self.model(id=row[0], question=row[1], poll_date=row[2])
            p.num_responses = row[3]
            result_list.append(p)
        return result_list

class OpinionPoll(models.Model):
    question = models.CharField(max_length=200)
    poll_date = models.DateField()
    objects = PollManager()

class Response(models.Model):
    poll = models.ForeignKey(OpinionPoll)
    response = models.TextField()
```

COMP222-Chapter 4

58

## Creating SEO (Search Engine Optimization) Friendly URL using SlugField

- For example, in a typical URL:  
<https://stackoverflow.com/questions/427102/what-is-a-slug-in-django/>
  - Here, the format is `https://stackoverflow.com/questions/{id}/{slug}`
  - the last data [what-is-a-slug-in-django is the slug](#).
  - SlugField doesn't ensure uniqueness. So, this is a better idea to include the id in the URL, and is what's actually used to do the query (also faster than querying on the slug).
  - It is a way of generating a valid URL, generally using data already obtained.

COMP222-Chapter 4

59

## SlugField

- SlugField in Django is like a CharField, where you can specify `max_length` attribute.  
`field_name = models.SlugField(max_length=200, null = True, blank = True, allow_unicode=True)`
- If `max_length` is not specified, Django will use a default length of 50.
- It also implies setting **Field.db\_index** to **True**.
- It is useful to automatically prepopulate a SlugField based on the value of some other value.
- A Slug is basically a short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs.

COMP222-Chapter 4

60

## How to use SlugField ?

- We can convert the title into a slug automatically.
- We want this to be triggered every time a new instance of *Post* model is created.

```
from django.db import models
from django.urls import reverse
from django.template.defaultfilters import slugify

class Post(models.Model):
    title = models.CharField(max_length = 250)
    slug = models.SlugField(max_length = 250, null = True, blank = True, allow_unicode=True)
    text = models.TextField()
```

```
def save(self, *args, **kwargs):
    self.slug = slugify(self.title)
    super(Post, self).save(*args, **kwargs)
```

This means that when a Post object is saved, it will automatically populate the slug field using the title field content.

```
def get_absolute_url(self):
    return reverse('post_detail', args=[self.slug, self.id])
```

This is to pass the values of slug and id to the url pattern named 'post\_detail'.

COMP222-Chapter 4

61

## URL pattern with both slug and pk

- The URL pattern is:
 

```
path('post/<slug:slug>-<int:pk>/', views.BlogDetailView.as_view(),
      name='post_detail'),
```
- URL template tag in template files:
 

```
<a href="{% url 'post_detail' slug=post.slug pk=post.pk %}">{{ post.title }}</a>
```

This returns an absolute path reference matching a given view with optional parameters.

COMP222-Chapter 4

62

## Using the Django interactive shell (>>>)

- To use the Django interactive shell, you must be running the virtual environment, and then run the following command from inside your project root folder created with startproject command.

```
(django)05:34~/django_projects/myTestSite $ python manage.py shell
```

- Your terminal output should look like this:

```
>>>
```

```
>>> from yourAppName.models import modelName
```

- To exit the interactive shell, type exit() or "Ctrl-D".

COMP222-Chapter 4

63

## Ordering data

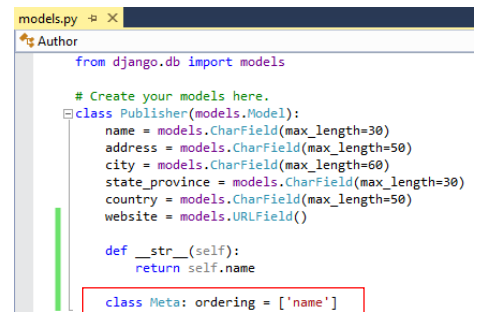
```
>>> Publisher.objects.order_by("state_province", "address")
```

- You can also specify reverse ordering by prefixing the field name with a "-"

```
>>> Publisher.objects.order_by("-name")
```

- To specify a default ordering in the model:

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



```
models.py  Author
from django.db import models

# Create your models here.
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __str__(self):
        return self.name

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

64



## Slicing data

- To display only the first one

`Publisher.objects.order_by('name')[0]`

- To retrieve a specific subset of data using range-slicing syntax

`Publisher.objects.order_by('name')[0:2]`

This returns two objects

- Negative slicing is not supported

`Publisher.objects.order_by('name')[-1]`

```
Traceback (most recent call last):
...
AssertionError: Negative indexing is not supported.
```

- To get around, just change the `order_by()` statement

`Publisher.objects.order_by('-name')[0]`

65

## Retrieving single objects

- `filter()` method returns a `QuerySet`, which is like a list.
- `get()` method fetches only a single object, as opposed to a list.

- With `get()`, a query resulting in multiple objects will cause an exception, so does a query that returns no objects.

```
>>> Publisher.objects.get(country="U.S.A.")
Traceback (most recent call last):
...
MultipleObjectsReturned: get() returned more than one Publisher -- it
returned 2! Lookup parameters were {'country': 'U.S.A.'}
```

- To trap these exceptions,

```
try:
    p = Publisher.objects.get(name='Apress')
except Publisher.DoesNotExist:
    print ("Apress isn't in the database yet.")
else:
    print ("Apress is in the database.")
```

66

## Conclusion

- We've now built our first database-driven app. While it's deliberately quite basic, now we know how to create a database model, update it with the admin panel, and then display the contents on a web page. But something is missing.....
- In the real-world, users need forms to interact with our site. After all, not everyone should have access to the admin panel.
- In the next chapter, we'll build a blog application that uses forms to handle the CRUD operations so that users can create, read, update, and ddelete posts.

## Summary: what have you learnt?

- Create a database model for the app
- The `makemigrations` and `migrate` command
- Django's powerful built-in admin interface
- For views, use the built-in generic class-based `ListView` which contains the logic to get all the records in the model.
- Introduce the built-in generic class-based `DetailView` which contains the logic to get the records meeting the condition in the model.
- For template files, use `for loop` to list all the objects in `object_list` returned by `ListView`.

## Wrap up of Django Admin

- An **admin interface** is a web-based interface, limited to trusted site administrators, that enables the adding, editing and deletion of site content.
- Add a superuser with the command:  
`python manage.py createsuperuser`
- Edit `admin.py` to register the model(s) to be displayed in the Django Admin