

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 (called mbposts) 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.

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

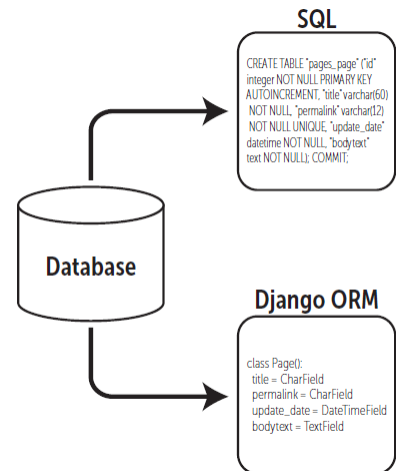
ORM: Object Relational Mapper

- When you think of databases, you will usually think of the *Structured Query Language (SQL)*, the common means with which we query the database for the data we require.
- With Django, querying an underlying database - which can store all sorts of data, such as your website's user details – is taken care of by the **Object Relational Mapper (ORM)**.
- In essence, data stored within a database table can be encapsulated within a *model*.
- 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.
- This chapter walks you through the basics of data management with Django and its ORM.

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

“Dumb” way to do database queries in views

- 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 the previous approach

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': os.path.join(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: 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.
(django2) 05:34 ~/django_projects/mysite \$ `python manage.py migrate`
- If you look inside the directory with the `ls` command, you'll see there's now a `db.sqlite3` file representing our SQLite database.

```
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 sessions.0001_initial... OK
```

COMP222-Chapter 4

12

Step 4: Create a database model

- Our first task is to 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()
```

- We’ve 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.

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 – an example

```
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()
```

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.

COMP222-Chapter 4

17

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 with the **makemigrations** command which generate the SQL commands. Note that **Migration files** do not execute those commands on our database file, rather they are a reference of all new changes to our models. This approach means that we have a record of the changes to our models over time.

(django2) 05:34 ~/django_projects/mysite \$
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.

(django2) 05:34 ~/django_projects/mysite \$
python manage.py migrate posts

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

COMP222-Chapter 4

18

0001_initial.py

- 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 following command to see how Django uses the model to generate SQL.

(django2)05:34~/django_projects/mysite \$ 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

19

Make Migrations vs Migrate

- Makemigration 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.

20

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 revert back by migrating to the previous migration, if needed.

COMP222-Chapter 4

21

Note on Make Migrations & Migrate

- Note that you don't have to include a 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

22

Step 7: Django Admin: create superuser

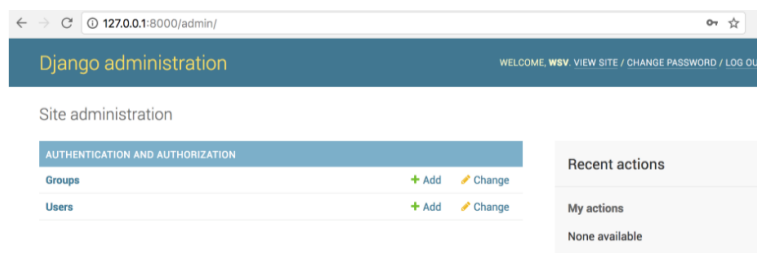
- Django provides us with a robust admin interface for interacting with our database.
- To use the Django admin, we first need to create a superuser.
- In your command line console, type the following command and respond to the prompts for a username, email, and password:
 (django2) 05:34 ~/django_projects/mysite \$ `python manage.py createsuperuser`
- Login to Django Admin on the browser via yourusername.pythonanywhere.com/admin/ by entering the username and password you just created. You will see the Django admin homepage next.

COMP222-Chapter 4

23

Step 8: Django Admin: edit admin.py

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



- We need to explicitly tell Django what to display in the admin.
- Edit posts/admin.py file to look like this:

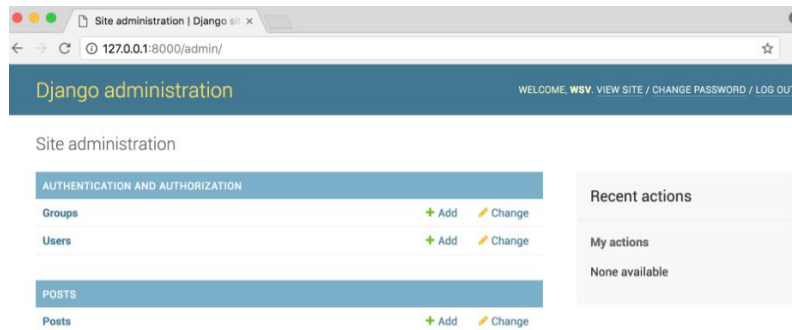
```
from django.contrib import admin
from .models import Post
admin.site.register(Post)
```

COMP222-Chapter 4

24

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.

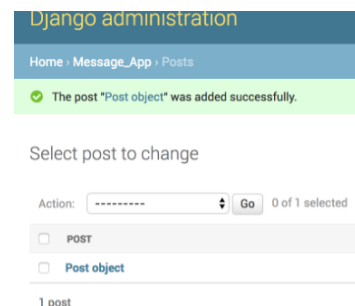


COMP222-Chapter 4

25

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

26

str() method to improve readability of models

- Within the posts/models.py file, add a new function `__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]
```

- In Python, if a function is included inside a class, it's called a *method*. If a function is inside another function, it's called a *sub-function*.
- The primary purpose of classes and functions is to group pieces of related code. The major difference between the two is a function *does* something whereas a class *is* something. For example, if Person was a class, walk() and eat() would be functions.
- Refresh your Admin page in the browser, it has changed to a much more descriptive representation of our database entry.

COMP222-Chapter 4

27

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

28

Views with generic class-based ListView

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

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

Remember to pay attention to proper indentation.

- 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

29

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

30

context_object_name

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

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

Remember to pay attention to proper indentation.

- 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>
    {% endfor %}
</ul>
```

COMP222-Chapter 4

31

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`?

COMP222-Chapter 4

32

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.
- Coming next, we'll build a blog application that uses forms to handle the CRUD operations so that users can create, read, update, and delete 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
- Use the built-in generic class-based `ListView` which contains the logic to get all the records in the model
- 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

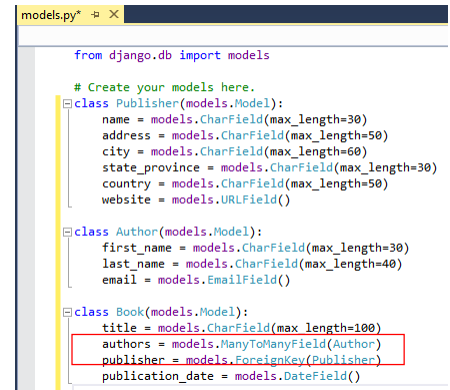
35

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.



```

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

37

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, but the Book database 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.
- Finally, note we haven't explicitly defined a primary key in any of these models. Django automatically gives every model an auto-incrementing integer primary key field called `id`.
- Each Django model is required to have a single-column primary key.

38

Making date and numeric fields optional

- 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 the migrate command](#) whenever you make such a change to a model.

39

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.

40

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.

```
(django2)05:34~/django_projects/mysite $ 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

41

Ordering data

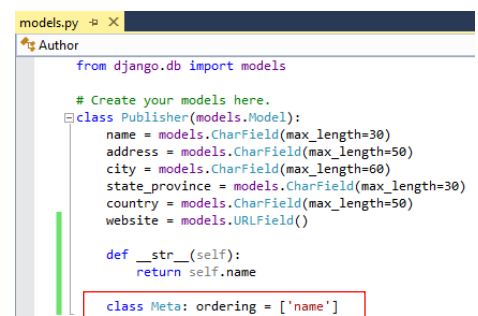
```
>>> Publisher.objects.order_by("state_province", "address")
```

- 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']
```



42

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]`

43

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.
- 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.")
```

>>> 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.'}

44

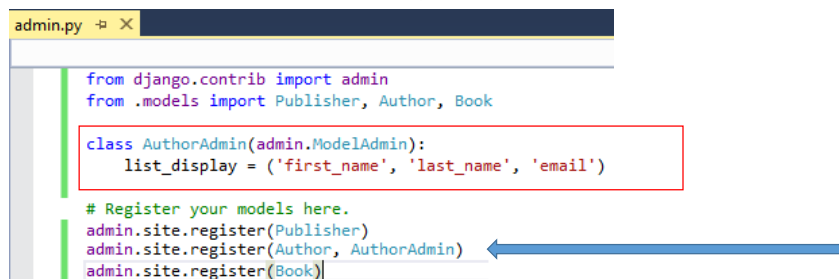
More on Django Admin

COMP222-Chapter 4

45

Customizing change lists – list_display

- By default, the change list displays the result of `__str__()` for each object.
- To add other fields to the change list to display, define a [ModelAdmin](#) class for the Author model.
- Edit [admin.py](#) to make these changes:



```
admin.py  [icon] [icon]
from django.contrib import admin
from .models import Publisher, Author, Book

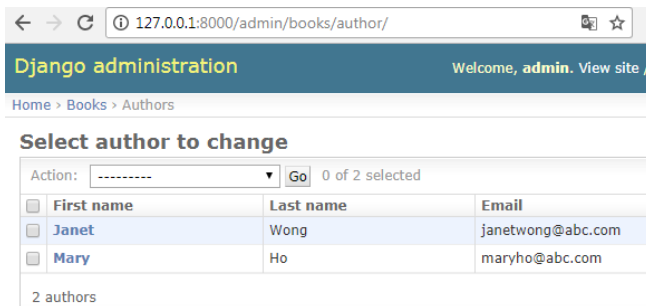
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')

# Register your models here.
admin.site.register(Publisher)
admin.site.register(Author, AuthorAdmin)
admin.site.register(Book)
```

46

Customizing change lists – list_display (cont'd)

- Reload the author change list page, and three columns are displayed – the first name, last name and e-mail address.
- In addition, each of those columns is sortable by clicking on the column header.



47

Adding search bar – search_fields

- To add a simple search bar, add `search_fields` to the `AuthorAdmin`,

```
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
    search_fields = ['first_name', 'last_name']
```

Remember to pay attention
to proper indentation.

- Note the use of `[]` and the search is case-insensitive and searches both fields.

48

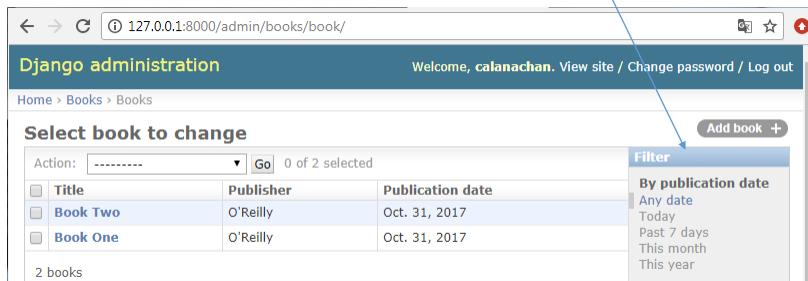
Adding date filters – list_filter

- To add date filters to our Book model's change list page:

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
```

Remember to pay attention to proper indentation.

- list_filter is used to create filters along the right side of the change list page.
- For date fields, Django provides shortcuts to filter the list to **Today**, **Past 7 days**, **This month**, and **This year**.
- list_filter also works on fields of other types. It shows up as long as there are at least two values to choose from.

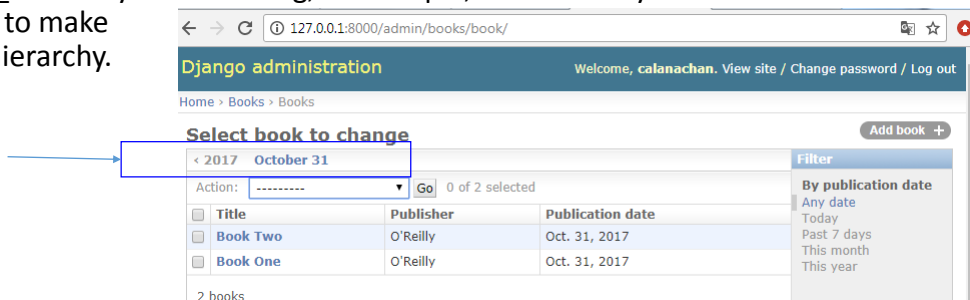


Adding date filters – date_hierarchy

- Another way to offer date filters is to use the `date_hierarchy` admin option.

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
```

- The change list page gets a date drill-down navigation bar at the top of the list as a result.
- `date_hierarchy` takes a string, not a tuple, because only one date field can be used to make the hierarchy.




50

Ordering

- By default, the change list orders objects according to their model's ordering within class Meta. If not specified, then the ordering is undefined.
- To change the default ordering so that books on the change list page are always ordered descending by their publication date.

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
```




Remember to pay attention to proper indentation.

51

Customizing edit forms

- Just as the change list can be customized, edit forms can be customized in many ways.
- By default, the order of fields in an edit form corresponds to the order they are defined in the model.
- We can change that using the [fields option](#) in our ModelAdmin subclass

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    fields = ('title', 'authors', 'publisher', 'publication_date')
```



52

fields option

- Another useful thing the fields option lets you do is to exclude certain fields from being edited entirely.
- Just leave out the field(s) you want to exclude, e.g. `publication_date`.
- You might use this if your admin users are only trusted to edit a certain segment of your data, or if some of your fields are changed by some outside, automated process.
- When a user uses this incomplete form to add a new book, Django will simply set the `publication_date` to `None` – so make sure that field has `null=True`.

53

filter_horizontal option

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    filter_horizontal = ('authors',)
```

- The **Authors** section now uses a fancy JavaScript filter interface that lets you search through the options dynamically and move specific authors from **Available authors** to the **Chosen authors** box, and vice versa.

The screenshot shows the Django administration interface for adding a new book. The 'Authors' field is highlighted with a red box in the code above, and the corresponding UI element is shown in the screenshot. The 'Available authors' box contains a search filter and a list of authors: Mary Ho and Janet Wong. The 'Chosen authors' box is currently empty. Below the boxes are buttons for 'Choose all' and 'Remove all'. The form also includes fields for 'Title', 'Publisher', and 'Publication date'.

54

filter_vertical option

- ModelAdmin classes also support a filter_vertical option.
- This works exactly as filter_horizontal, but the resulting JavaScript interface stacks the two boxes vertically instead of horizontally.
- filter_horizontal and filter_vertical only work on ManyToManyField fields, not ForeignKey fields.

55

raw_id_fields


- By default, the admin site uses simple <select> boxes for ForeignKey fields.
- But, as for ManyToManyField, sometimes you don't want to incur the overhead of having to select all the related objects to display in the drop-down.
- For example, if our book database grows to include thousands of publishers, the **Add book** form could take a while to load, because it would have to load every publisher for display in the <select> box.
- The way to fix this is to use an option called [raw_id_fields](#).

56

raw_id_fields (cont'd)

- Set this to a tuple of ForeignKey field names, and those fields will be displayed in the admin with a simple text input box (<input type="text">) instead of a <select>.

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    filter_horizontal = ('authors',)
    raw_id_fields = ('publisher',)
```



Remember to pay attention
to proper indentation.

57

Note on admin site

- The admin site is not intended to be a public interface to data, nor is it intended to allow 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.

58