Why should I use Django ORM when I know SQL?

Chapter 11

1

Advantages of Django ORM

- The main benefits of using Django ORM instead of SQL is a huge improvement in development speed, code maintenance, security and ease of development in general.
- It supports seamless schema generation and migration, and queries with them, data validation and integrity, switching between various databases without rewriting code, etc.
- With ORM it is much easier and less error-prone to modify the queries than with raw SQL.
- It is much easier to introduce security vulnerabilities when writing raw SQL. By using ORM exclusively you are guaranteed to be safe from SQL injections.

2

ORM vs raw SQL queries

- Django core developer Malcolm Tredinnick said (paraphrased):
  "*The ORM can do many wonderful things, but sometimes SQL is the right answer. The rough policy for the Django ORM is that it's a storage layer that happens to use SQL to implement functionality. If you need to write advanced SQL you should write it. I would balance that by cautioning against overuse of the raw() and extra() methods.*"
- Django project co-leader Jacob Kaplan-Moss says (paraphrased):
  "*If it's easier to write a query using SQL than Django, then do it. extra() is nasty and should be avoided; raw() is great and should be used where appropriate.*"

  Sourced from the book Two Scoops of Django
- This doesn't mean replace Django's ORM with SQL. This means use it as a last resort, when it's absolutely necessary. What's the point of using Django if you're not going to use Django?

3

(1) Raw SQL queries

- You can fall back to writing raw SQL.
- Django gives you two ways of performing raw SQL queries:
  1. Use Manager.raw() to perform raw queries and return model instances, or
  2. Avoid the model layer entirely and execute custom SQL directly.
- A model manager's raw() method returns results are structured as a RawQuerySet class instance, which is very similar to the QuerySet class instances produced by Django model queries.

4

(1) Examples of Django model manager raw() method

- By default, Django figures out a database table name by joining the model's "app label" – name used in manage.py startapp – to the model's class name, with an underscore between them.
- Performing raw SQL queries in interactive shell

```
>>>for p in Person.objects.raw('SELECT * FROM myapp_person'): print p
```

It's exactly the same as running Person.objects.all()

- Raw SQL query with aggregate function added as extra model field

```
>>> items_with_inventory = Item.objects.raw("SELECT *, sum(price*stock) as assets from items_item");
```

- raw() supports indexing, so if you need only the first result you can write:

```
>>>first_person = Person.objects.raw('SELECT * FROM myapp_person')[0]
>>>items_with_inventory[0].assets
```

5

(1) Passing parameters into raw()

- We have executed only static queries so far.
- To create SQL queries dynamically, we need to pass user-supplied data into our queries.
- You can use the params argument to pass parameters into raw():

```
lname = 'Doe'
Person.objects.raw('SELECT * FROM myapp_person WHERE last_name = %s', [lname])

user = Users.objects.raw('SELECT * FROM main_users WHERE mobile = %s OR email = %s', [mobile, email])
```

- You cannot quote the placeholder, otherwise, it does not work.

6

(2) Executing custom SQL directly

- You can access the database directly, routing around the model layer entirely.
- The object django.db.connection represents the default database connection.
- To use the database connection, call connection.cursor() to get a cursor object.
- Then, call cursor.execute(sql, [params]) to execute the SQL and cursor.fetchone() or cursor.fetchall() to return the resulting rows.

7

cursor.fetchone() or cursor.fetchall()

- **fetchone():** Fetch the next row of a query result set, returning a single tuple, or None when no more data is available

```
>>> cursor.execute("SELECT * FROM test WHERE id = %s", (3,))
>>> cursor.fetchone()
 (3, 42, 'bar')
```

- **fetchall()**: Fetch all (remaining) rows of a query result, returning them as a list of tuples. An empty list is returned if there is no more record to fetch.

```
>>> cursor.execute("SELECT * FROM test;")
>>> cursor.fetchall()
[(1, 100, "abc'def"), (2, None, 'dada'), (3, 42, 'bar')]
```

8

(2) Executing custom SQL directly – an example

- Django figures out a database table name by joining the model's "app label" – name used in manage.py startapp – to the model's class name, with an underscore between them
- To protect against injection, do not include quotes around the %s placeholders in the SQL string.

```
from django.db import connection
def my_custom_sql(self):
  with connection.cursor() as cursor:
    cursor.execute("UPDATE myapp_modelName SET foo = 1 WHERE baz = %s", [self.baz])
    cursor.execute("SELECT foo FROM myapp_modelName WHERE baz = %s", [self.baz])
    row = cursor.fetchone()
  return row
```

- imports django.db.connection represents the default database connection defined in the DATABASES variable in settings.py.
- With a connection reference to the Django database, you can make use of the Python DB API, which starts with the use of the cursor() method.
- Executes the cursor.execute() method to perform an operation.

9

Returning results as a dictionary (dict)

- By default, the Python DB API returns results without their field names, which means you end up with a list of values, rather than a dict.
- A dictionary is a set of *key: value* pairs, the keys are unique within one dictionary. A pair of braces creates an empty dictionary: {}.
- To return results as a dict by using something like this:

```
def dictfetchall(cursor):
    columns = [col[0] for col in cursor.description]
    return [
      dict(zip(columns, row)) for row in cursor.fetchall()
    ]
```

- Returns a tuple describing each column in the result row.
- The information about each column is a tuple of (column_name, declared_column_type), e.g. (('id', 'string'), ('parent_id', 'string'))

```
>>> list_1 = ['pie', 'cake', 'tart']
>>> list_2 = ['apple', 'banana', 'lemon']
>>> dict(zip(list_2,list_1))
{'apple': 'pie', 'banana': 'cake', 'lemon': 'pie'}
```

the zip command turns two lists into one list, then the dict command returns it as a dictionary.

```
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> cursor.fetchall()
[(54360982, None), (54360880, None)]

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> dictfetchall(cursor)
[{'parent_id': None, 'id': 54360982}, {'parent_id': None, 'id': 54360880}]
```

10

Django raw SQL queries with connection() and low-level DB API methods

```
from django.db import connection

# Delete record
target_id = 1
with connection.cursor() as cursor:
    cursor.execute("DELETE from items_item where id = %s", [target_id])

# Select one record
salad_item = None
with connection.cursor() as cursor:
    cursor.execute("SELECT * from items_item where name='Red Fruit Salad'")
    salad_item = cursor.fetchone()
# DB API fetchone produces a tuple, where elements are accessible by index
salad_item[0] # id
salad_item[1] # name
salad_item[2] # description

# Select multiple records
all_drinks = None
with connection.cursor() as cursor:
    cursor.execute("SELECT * from items_drink")
    all_drinks = cursor.fetchall()
# DB API fetchall produces a list of tuples
all_drinks[0][0] # first drink id
```

- imports django.db.connection which represents the default database connection defined in the DATABASES variable in settings.py.
- With a connection reference to the Django database, you can make use of the Python DB API, which starts with the use of the cursor() method.
- Executes the cursor.execute() method to perform an operation.
- Django figures out a database table name by joining the model's "app label" – name used in manage.py startapp – to the model's class name, with an underscore between them

11

Warning! SQL injection

- **Do not use string formatting and quote placeholders in your SQL strings**

  query = "SELECT * FROM myapp_person WHERE last_name = '**%s**'" % lname
  Person.objects.raw(query)
- Using the params argument and leaving the placeholders unquoted protects you from SQL injection attacks, a common exploit where attackers inject arbitrary SQL into your database.
- If you use string interpolation (%) and quote the placeholder, you are at risk for SQL injection.

12

SQL injection: an example

```
query = "SELECT * FROM users WHERE username = '%s' " % username
```

- What happens with input: "'; select true; --"

```
>>> print("select * from users where username = '%s'" % "'; select true; --")
select * from users where username = ''; select true; --'
```

- The resulting text contains three statements and it displays all the records of the users table.
  - select * from users where username = '';
  - select true;
  - --' (This snippet defuses anything that comes after it. The intruder added the comment symbol (--) to turn everything you might have put after the last placeholder into a comment.)
- However, for sqlite3, the above query will result in the message: "you can only execute ONE statement".

13

General concepts for preventing SQLi

- The fundamental rules of preventing SQL injection:
  - **Never** trust any data submitted by the user
  - **Always** use "parameterized statements" when directly constructing SQL queries
- Here's a search function with parameterized statements:

```
def search(request):
    ...
    cursor.execute("SELECT * FROM blog_post WHERE title LIKE %s", ['%' + query + '%'])
```

- Notice the %s in the SQL string, and the second parameter to execute.
- This second argument is the parameter list; items in this list are *safely* injected into the query to replace the placeholder.
- The above query with input '; DELETE FROM blog_post will result in the message:
  "You searched for: **'; DELETE FROM blog_post**"

14

An example: RAW query vs ORM

```
<form action="/blog/post/search/" method="GET">
<input type = "text" name="q">
<input type = "submit" value = "search">
</form>
```

```
def searchRAW(request):
    if 'q' in request.GET and request.GET['q']:
        query = request.GET['q']
        from django.db import connection
        with connection.cursor() as cur:
            #the wildcard character with the params argument not in SQL
            cur.execute("SELECT * FROM blog_post WHERE title LIKE %s",
                        ['%' + query + '%'])
            posts = dictfetchall(cur)   # this returns with their field names
            return render(request, 'search_results.html', {'posts': posts,
                    'query': query})
    else:
        return HttpResponse('Please submit a search term.')
```

```
def search(request):
    if 'q' in request.GET and request.GET['q']:
        query = request.GET['q']
        posts = Post.objects.filter(title__icontains=query)
        return render(request, 'search_results.html',
                        {'posts': posts, 'query': query})
    else:
        return HttpResponse('Submit a search term.')
```

```
path('post/search/', views.searchRAW, name='search'),
```

```
path('post/search/', views.search, name='search'),
```

15

Request vs Response

- HTTP messages are how data is exchanged between a server and a client.
- There are two types of messages:
  - *requests* sent by the client to trigger an action on the server, and
  - *responses*, the answer from the server.
- HTTP requests are messages sent by the client to initiate an action on the server. Their *start-line* contain three elements:
  - An *HTTP method*, a verb (like GET, PUT or POST) or a noun (like HEAD or OPTIONS), that describes the action to be performed.
  - The *request target*, usually a URL
  - The *HTTP version*, which defines the structure of the remaining message.

16

Post vs Get

- GET indicates that a resource should be fetched.
  - Use GET when the act of submitting the form is just a request to get data.
- POST means that data is pushed to the server (creating or modifying a resource, or generating a temporary document to send back).
  - Use POST whenever the act of submitting the form will have some side effect–*changing* data, or sending an e-mail, or something else that's beyond simple *display* of data.

17

A simple form-handling example: search-form

With models of books, authors and publishers, let's create a view that lets users search our book database by title.

1. Set up a search_form view that displays a search form.

```
from django.shortcuts import render

def search_form(request):
    return render(request, 'search_form.html')
```

2. Create the template search_form.html.

   The HTML <form> defines a variable q. When it's submitted, the value of q is sent via GET (method="get") to the URL /search/.

   The Django view that handles the URL /search/ (search()) has access to the q value in request.GET.

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

3. path('search-form/', views.search_form, name='search_form'),

18

A form-handling example: search view

4. Write the search view as indicated in step 2 before.

```
from django.http import HttpResponse
from django.shortcuts import render
from books.models import Book

def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render(request, 'search_results.html',
                      {'books': books, 'query': q})
    else:
        return HttpResponse('Please submit a search term.')
```

Besides checking that 'q' exists in request.GET, also make sure that request.GET['q'] is a non-empty value before passing it to the database query.

Get the records from the Book model that matches the search term.

Pass the data of books and q to search_results.html, which contains the template variables named 'books' and 'query'.

if a user visits /search/ with no GET parameters, it simply displays a message.

- The next step is to write the template 'search_results.html' to display the list of books that matches the search term with the template variables 'books' and 'query'.

19

search_results.html

- Note the usage of the pluralize template filter, which outputs an "s" if appropriate, based on the number of books found.
- Can you locate the template variables query and books?
- Next, let's see how the page works.

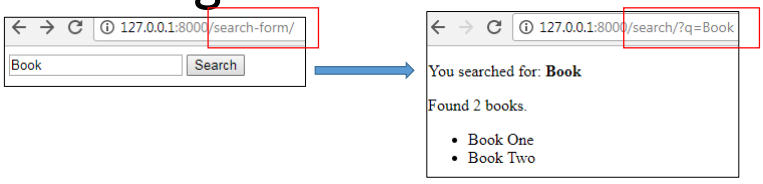
search_results.html

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Book Search</title>
</head>
<body>
    <p>You searched for: <strong>{{ query }}</strong></p>
    {% if books %}
        <p>Found {{ books|length }}
        book{{ books|pluralize }}.</p>
        <ul>
        {% for book in books %}
        <li>{{ book.title }}</li>
        {% endfor %}
        </ul>
    {% else %}
        <p>No books matched your search criteria.</p>
    {% endif %}
</body>
</html>
```

20

Searching books interface



Result of searching for "Book". Look at the URL: /search/?q=Book

Clicking the "Search" button without any input. This requires the user to hit the browser's back button to go back to the search form for input. Let's have an improved version, to be discussed in the next slide.

```
def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render(request, 'search_results.html',
                      {'books': books, 'query': q})
    else:
        return HttpResponse('Please submit a search term.')
```

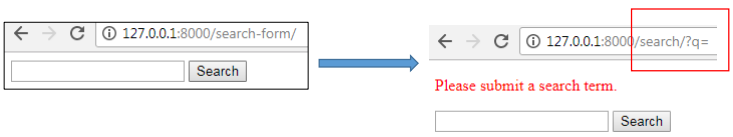

An improved version for the search

```
def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render(request, 'search_results.html',
        {'books': books, 'query': q})
    else:
        return render (request, 'search_form.html', {'error': True})
```

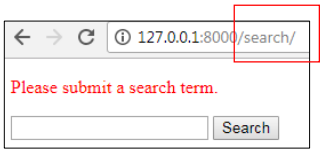
Now, if the query is empty, we pass a template variable error = True to search_form.html. As a result, we have to update search_form.html to read the template variable.

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    {% if error %}
        <p style="color: red;">Please submit a search term.</p>
    {% endif %}
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

Result of the improved version for the search



Clicking the "Search" button without any input will load the search_form.html with an error message. So, it is not necessary to press the BACK button to go to the search interface.



As it stands, a request to the URL /search/ (without any GET parameters) will display the empty form (but with an error). We can change search() to hide the error message when somebody visits /search/ with no GET parameters. See next slide for discussion.

23

Updated view: /search/ with no GET parameters

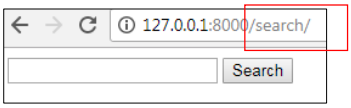
- In this updated view, if a user visits /search/ with no GET parameters, they'll see the search form with no error message.
- If a user submits the form with an empty value for 'q', they'll see the search form with an error message.
- And, finally, if a user submits the form with a non-empty value for 'q', they'll see the search results.

```
def search(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        else:
            books = Book.objects.filter(title__icontains=q)
            return render(request, 'search_results.html', {'books': books, 'query': q})
    return render (request, 'search_form.html', {'error': error})
```

So, if there is no 'q' in request.GET, it will jump to the last line of code with error = False. As a result, search_form.html is displayed without any error message.

24

Updated view: /search/ with no GET parameters (cont'd)



Compared with slide 22. Now, a request to the URL /search/ (without any GET parameters) will display the empty form without error!

- Now that search() hides the error message when somebody visits /search/ with no GET parameters, we can remove the search_form() view, along with its associated URLpattern.
- Since the two views and URLs are merged into one and /search/ handles both search–form display and result display, the HTML <form> in search_form.html doesn't have to hard-code a URL.

  <form action="" method="get">
- The action="" means *Submit the form to the same URL as the current page.*

25

Simple validation

- Whenever we have user input, we need to handle validation to make sure that the data input is what we expected.
- For example, we can add a requirement that the search term is less than or equal to 20 characters long.

```
def search(request):
    errors = []
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            errors.append('Enter a search term.')
        elif len(q) > 20:
            errors.append('Please enter at most 20 characters.')
        else:
            books = Book.objects.filter(title__icontains=q)
            return render(request, 'search_results.html', {'books': books, 'query': q})
    return render (request, 'search_form.html', {'errors': errors})
```

a list of error message strings

- Now, error is no longer of Boolean type, but contains a list of error message strings.
- Next, we have to update the search_form.html template to reflect that it's now passed an errors list instead of an error Boolean value.

26

Search_form.html updated

- Updated search_form.html template to reflect that it's now passed an errors list instead of an error Boolean value.
- We used a for loop to display the list of errors.

search_form.html

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Search</title>
</head>
<body>
    {% if errors %}
        <ul>
        {% for error in errors %}
            <li>{{ error }}</li>
        {% endfor %}
        </ul>
    {% endif %}
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```


Another example on function-based view

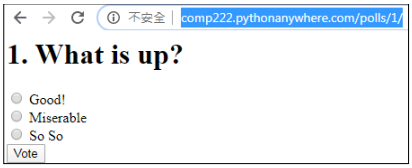
```
from django.db import models

# Create your models here
class Question (models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published', null=True, blank=True)

class Choice (models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

```
def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    try:
      selected_choice=question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
      # Redisplay the question voting form.
      return render(request, 'detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",})
    else:
      selected_choice.votes += 1
      selected_choice.save()
      # Always return an HttpResponseRedirect after successfully
      # dealing with POST data. This prevents data from being posted
      # twice if a user hits the Back button.
      return HttpResponseRedirect(reverse('results', args=(question.id,)))
```

path('<int:question_id>/vote/', views.vote, name='vote'),



polls/templates/detail.html

```
<h1>{{ question.question_text }}</h1>


{% if error_message %}<p><strong>{{ error_message }}</strong></p>{%
endif %}


<form action="{% url 'vote' question.pk %}" method="post">
{% csrf_token %}

{% for choice in question.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}"
       value="{{ choice.id }}">
    <label id="choice{{ forloop.counter }}"> {{ choice.choice_text }}
       </label> <br>
{% endfor %}

<input type="submit" value="Vote">

</form>
```


#results.html

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{
choice.votes|pluralize }}</li>
{% endfor %}
</ul>
<a href="{% url 'detail' question.pk %}">Vote again?</a>
```