Custom Python and Django for AbbVie Part II

John Strickler

Version 1.0, November 2020

Table of Contents

About this course

Course Outline

Day 1

Chapter 1 Querying Django Models
Chapter 2 Migrating Django Data
Chapter 3 Loading Django Data
Chapter 4 About REST

Day 2

Chapter 5 Django REST Framework Basics
Chapter 6 Django REST Viewsets
Chapter 7 Django REST Filters
Chapter 8 Django REST Documentation

Day 3

Chapter 9 Django User Authentication
Chapter 10 Django REST Authentication
Chapter 11 Django Unit Testing

If time permits

Chapter 12 Django REST Final Project

NOTE	The actual schedule varies with circumstances. Time permitting, the last day may include a larger (half-day) project to reinforce Django REST skills learned during class.

Appendices

Chapter 1: Querying Django Models

Objectives

- Understand data managers
- Learn about QuerySets
- Use database filters
- Define field lookups
- Chain filters
- Get aggregate data

Object Queries

- Model class is table, instance is row
- model.objects is a manager
- QuerySet is a collection of objects

In the Django ORM, a model class represents a table, and a model instance represents a row. You have already seen some of this in the chapter on views.

To query your data, you will usually use a Manager. The default manager is named objects, and is available from the Model class.

From the object manager, you can get all rows, filter rows, or exclude rows. You can also implement relational operations, such as greater-than or less-than.

NOTE

For this chapter, there is just one view function that uses the builtin function eval() to evaluate a list of strings containing queries, in order to avoid duplicating the queries as labels.

Access this view at http://localhost:8000/superheroes/heroqueries

Example

django2/djsuper/superheroes/viewsqueries.py

```
from django.shortcuts import get_object_or_404, render
from django.db.models import Min, Max, Count, FloatField, Q
from .models import Superhero

q_hulk = Q(name__icontains="hulk")
q_woman = Q(name__icontains="woman")

def hero_queries(request):

    queries = [
        'Superhero.objects.all()',
        'Superhero.objects.filter(name="Superman")',
        'Superhero.objects.filter(name="Superman").first()',
        'Superhero.objects.filter(name="Spider-Man").first()',
        'Superhero.objects.filter(name="Spider-Man").first().secret_identity',
        'Superhero.objects.filter(name="Superman").first().enemies.all',
        'Superhero.objects.filter(name="Spider-Man").first().powers.all',
        'Superhero.objects.filter(name="Batman").first().powers.all',
        'Superhero.objects.exclude(name="Batman")',
        'Superhero.objects.order_by("name")',
        'Superhero.objects.count()',
        'Superhero.objects.aggregate(Count("name"))',
        'Superhero.objects.aggregate(Min("name"))',
        'Superhero.objects.aggregate(Max("name"))',
        'Superhero.objects.aggregate(Min("name"),Max("name"))',
        'Superhero.objects.filter(name__contains="man").count()',
        '''Superhero.objects.filter(
                name__contains="man").exclude(name__contains="woman")''',
        '''Superhero.objects.filter(
                name__contains="man").exclude(
                name__contains="woman").count()''',
        'Superhero.objects.all()[:3]',
        'Superhero.objects.filter(name__contains="man")[:2]',
        '''Superhero.objects.filter(
            enemies__name__icontains="Luthor").first().name''',
        'Superhero.objects.filter(q_hulk | q_woman)',
    ]

    query_pairs = [
        (query, eval(query)) for query in queries
    ]
    context = {
        'page_title': 'Query Examples',
        'query_pairs': query_pairs,
    }
    return render(request, 'hero_queries.html', context)
```

Example

django2/djsuper/superheroes/templates/hero_queries.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ hero_name }}</title>
</head>
<body>
<h1>{{ hero_name }}</h1>
<h2>Secret Identity: {{ secret_identity }}</h2>
<h2>Real Name: {{ real_name }}</h2>
</body>
</html>
```

Opening a Django shell

- Convenient for quick sanity checks
- Sets up Django environment
- Starts iPython (enhanced interpreter)

To interactively work with your models, you can open a shell. This opens a Python interpreter (nowadays iPython) with the project's configuration loaded.

This makes it easy to manipulate database objects.

To start the shell, type

```
python manage.py shell
```

NOTE If iPython (recommended) is installed, the Django shell will use it instead of the builtin interpreter.

QuerySets

- Iterable collection of objects
- Roughly equivalent to "SELECT ..."
- Each row contains fields
- Use manager to create

A QuerySet is a collection of objects from a model. QuerySets can have any number of filters, which control which objects are in the result set. Filters correspond to the "WHERE ..." clause of a SQL query.

all(), filter(), exclude(), sortby(), and other functions return a QuerySet object. A QuerySet can itself be filtered, sorted, sliced, etc.

ORM queries are deferred (AKA lazy). The actual database query does not happen until the QuerySet is evaluated.

```
Superhero.objects.all()
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>, <Superhero: Wonder
Woman>, <Superhero: Hulk>]

Superhero.objects.filter(name="Superman")
[<Superhero: Superman>]

Superhero.objects.filter(name="Superman").first()
Superman

Superhero.objects.filter(name="Spiderman").first()
Spiderman

Superhero.objects.filter(name="Spiderman").first().secret_identity
Peter Parker

Superhero.objects.filter(name="Superman").first().enemies.all
[<Enemy: Lex Luthor>, <Enemy: General Zod>]

Superhero.objects.filter(name="Spiderman").first().powers.all
[<Power: Super strength>, <Power: Spidey-sense>, <Power: Intellect>]

Superhero.objects.filter(name="Batman").first().powers.all
[<Power: Detective ability>]

Superhero.objects.exclude(name="Batman")
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Wonder Woman>, <Superhero:
Hulk>]

Superhero.objects.order_by("name")
[<Superhero: Batman>, <Superhero: Hulk>, <Superhero: Spiderman>, <Superhero: Superman>,
<Superhero: Wonder Woman>]
```

Query Functions

Returning QuerySets	Returning Objects
filter()	get()
exclude()	create()
annotate()	get_or_create()
order_by()	update_or_create()
reverse()	latest()
distinct()	earliest()
values()	first()
values_list()	last()
dates()	
datetimes()	**Returning other values**
none() (empty)	bulk_create() (None)
all()	count() (int)
union()	in_bulk() (dict)
intersection() (1.11+ only)	iterator() (iterator)
difference() (1.11+ only)	aggregate() (dict)
select_related() (1.11+ only)	exists() (bool)
prefetch_related()	update() (int)
extra()	delete() (int)
defer()	as_manager() (Manager)
only()	
using()	
select_for_update()	
raw()	

Field lookups

- Field comparisons
- Use *field*__operator
- Work with filter(), exclude(), distinct(), etc.

In SQL, the WHERE clause allows you to compare columns using relational operators. To do this Django, you can append special operator_suffixes to field names, such as name__greaterthan or secret_identity__contains. Note that the operators are preceded by two underscores.

Example

```
Superhero.objects.filter( name__contains="man").exclude(name__contains="woman")
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>]

Superhero.objects.filter( name__contains="man").exclude( name__contains="woman").count()
3
```

You can also create custom lookups for model fields

Field Lookup operator suffixes

Use in **filter()**, **exclude()**, and **get()**.

```
__exact
__iexact
__contains
__icontains
__in
__gt
__gte
__lt
__lte
__startswith
__istartswith
__endswith
__iendswith
__range
__date
__year
__month
__day
__weekday
__hour
__minute
__second
__isnull
__regex
__iregex
```

Aggregation Functions

- Calculate values over result set
- Use aggregate() plus calls to Count, Min, Max, etc
- Correspond to SQL COUNT(), MIN(), MAX(), etc

Some database tasks require calculations that use the entire result set (or subset). These are usually called aggregates. Common aggregation functions include count(), min(), and max().

To do this in Django, call the aggregate() function on a QuerySet, passing in one or more aggregation function. Each class is passed at least the name of the field to aggregate over.

aggregate() returns a dictionary where the keys are fieldname__aggregation_class, such as name__min. Parameters to aggregation functions may include a field name (positional), and the named parameter output_field, which specifies which field to return.

You can also generate aggregate values via the annotate() clause on a QuerySet

Aggregation Functions

```
Avg()
Count()
Min()
Max()
StdDev()
Sum()
Variance()
```

Example

```
Superhero.objects.aggregate(Count("name"))
{'name__count': 5}

Superhero.objects.aggregate(Min("name"))
{'name__min': 'Batman'}

Superhero.objects.aggregate(Max("name"))
{'name__max': 'Wonder Woman'}

Superhero.objects.aggregate(Min("name"),Max("name"))
{'name__max': 'Wonder Woman', 'name__min': 'Batman'}
```

Chaining filters

- Anything returning QuerySet can be chained
- Other methods are terminal (can’t be chained)

Any QuerySet returned by some method can then have another method called on it; thus, you can chain filter methods to fine-tune what you need.

For instance, you could chain filter() and exclude(), then call count() on the result.

Example

```
Superhero.objects.filter(name__contains="man").count()
4

Superhero.objects.filter( name__contains="man").exclude(name__contains="woman")
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>]

Superhero.objects.filter( name__contains="man").exclude( name__contains="woman").count()
3
```

Slicing QuerySets

- Slice operator [start:stop:step] works on QuerySets
- Slice is lazy – only retrieves data for slice
- Use to fetch first N objects, etc.
- Negative indices are not supported

While a QuerySet is not an actual list, it can be sliced like most builtin sequences. Furthermore, the slice returns another QuerySet, so it doesn't execute the actual query until the data is accessed.

One difference from normal slicing is that negative indices and ranges are not supported.

Example

```
Superhero.objects.all()[:3]
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>]

Superhero.objects.filter(name__contains="man")[:2]
[<Superhero: Superman>, <Superhero: Spiderman>]
```

Related fields

- Search models via related fields
- Use column__foreign_column

To search models using values in related fields, use column__foreign_column__lookup. This lets you apply field lookups to fields in the related column.

Example

```
Superhero.objects.filter(enemies__name__icontains="Luthor").first().name
Superman
```

Q objects

- Encapsulates SQL expression in Python objects
- Only needed for complex queries

By default, chained queries are AND-ed together. If you need OR conditions, you can us the Q object. A Q object encapsulates (but does not execute) a SQL expression. Q objects can be combined with the | (OR) or & (AND) operators. Arguments to the Q object are the same as for filters – field lookups.

Example

superheroes/queries/views.py

```
q_hulk = Q(name__icontains="hulk")
q_woman = Q(name__icontains="woman")
...
Superhero.objects.filter(q_hulk | q_woman)
[<Superhero: Wonder Woman>, <Superhero: Hulk>]
```

Chapter 1 Exercises

Exercise 1-1 (dogs/*)

In this exercise, you will start a project that you will use throughout the rest of the class.

- Create a project named **dogs** using cookiecutter-rest
- In the dogs project, create an app named dogs_core
- Configure the app
- Create models for Dog and Breed
    - Dog fields
        - name
        - breed (foreign key)
        - sex (m or f)
        - is_neutered
    - Breed fields
        - name
- Update the database
    - Create migrations
    - Migrate
- Add models to admin

Now use the Django shell to add some records for dogs and breeds. Add at least 6 dogs and at least 2 breeds. Be sure to add lots of variations so you can use them for searching.

Once you have created the records and saved them, use the query methods to find records as follows (vary the queries to match your own data):

1. All dogs
2. All breeds
3. Dog with specified name
4. All female dogs
5. All dogs of a selected breed
6. All female dogs whose name begin with 'B' *etc*

NOTE

As an optional enhancement, you could add a Category model, with categories such as working, herding, companion, sporting, etc. See https://www.akc.org/public-education/resources/general-tips-information/dog-breeds-sorted-groups/ for a list of which dogs belong in which categories. Category would be a foreign key to Breed.

Chapter 2: Migrating Django Data

Objectives

- Understanding migrations
- Using manage.py to migrate data
- Reverting (squashing) migrations

About migrations

- Changes to your database schema
- Speed up database changes
- Use scc on database schemas

After creating your initial database schema (table and column definitions), you will frequently need to add tables, as well as adding, deleting, and modifying columns.

Doing this manually is slow and error-prone.

Django 1.7 added migrations. A migration is a change to your data. Django tracks migrations through several utilities that can be called via manage.py. These tools make it much easier to propagate changes to your models into your database schema.

Migrations can be thought of as version control for your models.

Separating schema from data

- Schema is infrastructure (tables and columns)
- Data is … *data*

While your database contains your data, it also contains tables and columns. This metadata is called the schema. When you modify a model, you will will be modifying your database schema when you migrate.

Django migration tools

- migrate
- makemigrations
- sqlmigrate
- showmigrations

The four commands provided for migration are shown above. They are called from manage.py.

migrate applies migrations, unapplies migrations, and lists status.

makemigrations checks for changes to your models and creates new migrations as needed.

sqlmigrate displays the SQL statements that will be executed for a migration.

showmigrations lists all of the migrations for a project.

NOTE	migrations are applied project-wide by default.

Migration workflow

- Make changes to models
- Run manage.py makemigrations
- Run manage.py migrate

The normal workflow for working with migrations is:

1. Make changes to models as needed.
2. Run manage.py makemigrations This will create a set of migrations, which you can view with manage.py sqlmigrate.
3. Run manage.py migrate

This will apply any pending migrations.

NOTE	Migrations are per-app, but run from the project perspective.

Adding non-nullable fields

- Add default value to model
- Add default value via `manage.py migrate`
- Migrate twice
  - Add field as nullable
  - Update values
  - Make field non-nullable

If you are adding a non-nullable field, the migrate command will need to know how to handle that field for existing records.

```
You are trying to add a non-nullable field '<new_field>' to <model> without a default; we
can't do that (the database needs something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for
this column)
 2) Quit, and let me add a default in models.py
Select an option:
```

You can either add a default that will be applied to existing rows, or you can go back and add a default value to the field in the model.

If you need to add a field, but you can't add the same value to all existing fields, then you can migrate twice:

1. Create the field as nullable.
2. Create and run the first migration
3. Update the field as needed, making sure all rows have a value
4. Change the field to be non-nullable
5. Create and run the second migration

Migration files

- Normal python scripts
- Stored in the migrations folder

Migrations are implemented as normal Python scripts. They are stored in the migrations folder within the app.

Example

```
from django.db import migrations, models

class Migration(migrations.Migration):

    dependencies = [("migrations", "0001_initial")]

    operations = [
        migrations.DeleteModel("Tribble"),
        migrations.AddField("Author", "rating", models.IntegerField(default=0)),
    ]
```

Typical migration workflow

Typically, the workflow to update a production server might look like this:

LOCAL DEVELOPMENT ENV:

- Change your model locally
- Create new migrations (`manage.py makemigrations`)
- Migrate models (`manage.py migrate`)
- Test your changes locally
- Commit & push your changes to (git) server

ON THE PRODUCTION SERVER:

- Set ENV parameters
- Pull new code from **git**
- Update any new python dependencies (e.g. `pip install -r requirements.txt`)
- Migrate (`manage.py migrate`)
- Update static files (python manage.py collectstatic)
- Restart server

Squashing migrations

- Reduce set of migrations
- Does not remove old migrations
- Mostly automated

To cut down on the number of migrations needed, the **squashmigrations** command can be called from manage.py.

This will try to reduce all of the migrations up to a specified point into a smaller set of changes. It does not remove the original migrations, but when you run migrate commands, it will automatically used the squashed version.

Example

```
manage.py squashmigrations registry 0006
```

Reverting to previous migrations

- Select target migration
- Migrate to target
- Delete previous migrations
  - Use migration **zero** to unapply all

To revert to a previous migration, use the **migrate** command from **manage.py** with the latest migration to keep. In otherwords, if you have magration 0012, 0013, and 0014, and you want to revert to 0012, then say `manage.py migrate my_app 0001`.

To unapply all migrations, say `manage.py migrate my_app zero`.

Data Migrations

- Not automated
- Same as schema migrations
- Typically put in migrations folder

While Django does not have tools to automate data migrations, they are easy to create with the same tools that are used for schemas. You can write them as normal Python scripts and keep them in the migrations folder.

You have to follow the same format as schema migrations.

NOTE	You could also add new commands to **manage.py**

Chapter 2 Exercises

Exercise 2-1 (dogs/dogs_core/models.py)

Add an an integer column "weight" to the Dog model. Use migration tools to update the database from the model.

HINT: Remember to do the migration in two steps (or use the option to provide a default).

Chapter 3: Loading Django Data

Objectives

- Load data three ways
  - Create a data migration
  - Use **loaddata**
  - Create a new management command

Where do I start?

- Project database starts empty
- May need to add data for testing

Once you've created and migrated your models, your database is ready to use. And empty.

For a few records, you can add data via the admin interface or the Django shell.

However, you may want to add a number of records, for testing, or to migrate data from a previous incarnation of the project.

There are at least three ways to do this:

1. Create a data migration using the migration tools
2. Use **loaddata**
3. Create a new management command

NOTE	A fourth way is to write a script that is not part of your Django project, and that directly uses the database, but that is not recommended. It might create data that is incompatible with your app.

Creating a data migration

- Create an empty migration
- Implement code in migration file
- Call `manage.py migrate`
- Preferred approach

This is the preferred way to load data, as it becomes part of your data migration, so migrating on the deployment server is simpler.

To use migration tools, first create an empty migration with `manage.py makemigrations --empty <appname>`. This adds a migration file in the migrations folder, with no content. This will create a minimal migration script:

```
from django.db import migrations


class Migration(migrations.Migration):

    dependencies = [
        ('myapp', '0004_auto_20201031_2202'),
    ]

    operations = [
    ]
```

The **dependencies** list is pre-populated with the previous dependency

To perform a data migration, create a function that takes two arguments, an app registry (typically called `apps`) that manages previous versions of the app, and a schema editor. For normal use, you won't need the schema editor.

The app registry will be used to fetch objects from the database that match the current migration. Use `apps.get_model(<appname>, <model>)`. In the function, perform whatever tasks are needed for your data migration. Remember to save all modified objects. You may want to write helper methods so that your main function doesn't get unwieldy.

Call your function with `migrations.RunPython(<function>)`.

```
from django.db import migrations

def copy_data(apps, schema_editor):
    Model1 = apps.get_model('myapp', 'Model1')
    Model2 = apps.get_model('myapp', 'Model2')
    for model1 in Model1.objects.all():
        model1.some_field = "some value"
        model1.save()

class Migration(migrations.Migration):


    dependencies = [
        ('myapp', '0004_auto_20201031_2202'),
    ]

    operations = [
        migrations.RunPython(copy_data)
    ]
```

NOTE If you make a mistake and need to re-run your migration, use the `--fake` option to `manage.py migrate`. Use this option with the migration *before* the one you want to re-run. Then call `manage.py migrate` as usual and it will re-run the migration:

```
manage.py migrate --fake <previous migration>
manage.py migrate
```

Using loaddata

- Create JSON or YAML data file(s)
- Use `manage.py loaddata`

You can use the manage.py command **loaddata** to insert data into your database. The data must be in JSON or YAML format. To see the correct format, add at least one record using the Django shell or the admin interface, then call `manage.py dumpdata <APP.MODEL> <FILENAME>' for JSON, or `manage.py dumpdate --format yaml <APP.MODEL> <FILENAME>` for YAML.

If you have related fields, you will have to load one table to get IDs, then write code to use that table to build the next table, etc.

Creating a new management command

- Add command to manage.py
- Create commands in *app*/management/command
- Inherit from BaseCommand
  - Add arguments
  - Define **handle()**

If you need to do some specific tasks relative to your databases (or any part of your app), you can add new commands to manage.py for convenience. These commands have access to your project's Django configuration.

To add a new commmand, create a folder called `management` in an app, then create a subfolder called `commands`. In that folder, you can create any number of modules (scripts), each of which will be a separate command. For example, `validate_cities.py` could then be called as `manage.py validate_cities`.

In each command script, create a class named `Command` which subclasses `BaseCommand`. Add a class level variable named **help** which is set to a description of what the command does.

If the command needs arguments, define the method `add_arguments()`, which has a parameter `parser`. This parser object will parse arguments passed in after your command name when it is called via `manage.py`. Usage is

```
parser.add_argument('<argument name>', type=<argument type>)
```

You must define a method named `handler()`. This will be the code that runs when you call your command. It takes two arguments. `args` is not normally needed, and `options` contains the options parsed in `add_arguments()`.

If your command needs to work with the database, you can import your models at the top of the script.

Chapter 3 Exercises

Exercise 3-1 (dogs/dogs_core/*)

Add a new field, **abbr** to Breed to hold the breed abbreviation. It should be a character field with a maximum size of 4. It will be created from the first two letters of the breed if it is a single word, or the first letters of each word if more than one word. Make the field nullable and migrate the changes.

Create a custom migration to populate the **abbr** field.

Exercise 3-2 (dogs/dogs_core/*)

Use **loaddata** to load data from the files **dog_data.yaml** and **breed_data.yaml** into your database. The yaml files are in the DATA folder.

Load the breeds first, and you will then have to dump the breeds table, and then manually copy the breed IDs into the dog file before loading it. In real life, you would write a script to do this.

Exercise 3-3 (dogs/dogs_core/*)

Create a new manage.py command named **addsnoopy** to add a dog to your database with the following data:

Name: Snoopy
Breed: Beagle
Weight: 30 Sex: M Neutered: Y

Run the command and confirm that Snoopy is added to your database.

NOTE	Your command should check to see if "Beagle" is in the breed table, and if not, add it.

Chapter 4: About REST

Objectives

- Learning REST guidelines
- Applying HTTP verbs to REST
- Aligning REST with CRUD
- Examining RESTful URLs
- Discussing searching and filtering

The REST API

- Based on HTTP verbs
  - GET get all objects or details on one
  - POST add a new object
  - PUT update an object (all fields)
  - PATH update an object (some fields)
  - DEL delete an object

REST stands for *RE*presentational *S*tate *T*ransfer, first described (and named) by Roy Fielding in 2000. It is not a protocol or structure, but rather an architectural style resulting from a set of guidelines. It provides for loosely-coupled resource management over HTTP. REST does not enforce a particular implementation.

A RESTful site provides *resources*, which contain *records*. The same API typically contains more than one resource; each has a different *endpoint* (URL). For instance, `https://sandbox-api.brewerydb.com/v2/` has resources **beer**, **brewery**, and **ingredient**.

A RESTful API uses HTTP verbs to manipulate records. The same endpoint can be used for all access; what happens depends on a combination of which HTTP verb is used, plus whether there is more information on the URL.

If it is just the endpoint (e.g. *www.wombatlove.com/api/v1/wombats*): * GET retrieves a list of all resources. Query strings can be used to sort or filter the list. * POST adds a new resource

If it is the endpoint plus more information, typically a primary key (e.g. *www.wombatlove.com/api/v1/wombats/1*): * GET retrieves details for that resource * PUT updates the resource (replaces all fields) * PATCH updates the resource (replaces some fields) * DELETE removes the resource

NOTE	see https://restfulapi.net/resource-naming/ for more information on designing a RESTful interface

REST constraints

There are six *constraints*, or guidelines, that make up REST. They are designed to be flexible. Remember, REST is an architecture, not a protocol.

Uniform Interface

- The interface defines the interactions between the client and the server (i.e., the client application and the RESTful API).
- Representations of resources contain enough information to alter or delete the resource.
- Data is sent via JSON (or maybe XML or HTML), rather than its "native" format.
- Every message has information that tells how to process the message.
- HATEOAS — all interaction is done via Hypermedia, using the URI, body contents, request headers, and parameters (Hypermedia as the Engine of Application State). While REST does not *require* HTTP, there are very few REST APIs that do not use it. A consequence of this is that the returned data should have links to retrieve the data requested or related data.

Stateless

- No sessions
- Requests contain all state (data) needed by server to fulfil the request. If multiple requests are needed, the client must resend, including authorization details.
- No client context stored on server between requests (client manages the state of the application)

Cacheable

- Responses must return uniform data (no timestamps, etc.) so it can be cached.
- Caching is important to reduce load on servers and infrastructure.

Client-Server

- Clients and server are completely independent
- Client only knows API (resource URIs)
- Server does not know (or care) how client uses data

Layered System

- APIs may be deployed on one server, data stored on another server, and validation on a third server, e.g.
- Client does not know (or care) about any servers other than the one providing the API

Code on Demand (optional)

- A REST-compliant API may optionally return executable code (e.g. GUI widget)
- This is unusual.

REST data

- JSON most popular format
- Any format is allowable
- Specified via request header

The data provided by a RESTful endpoint is usually in JSON format, but it doesn't have to be. The API can provide CSV, YAML, or other formats.

While some sites use a query string or infer from a resource suffix (i.e., `/wombats.csv`), the correct way to ask for a format is to speciy a MIME type in the request header.

```
response = requests.get('http://www.wombats.com/api/wombats', header={'accept',
'text/csv')
```

When is REST not REST?

- REST is guidelines, not protocol
- Implementers are not consistent
- YMMV (your mileage may vary)

REST is a set of guidelines, not a specific protocol. Because of this, REST implementations vary widely. For instance, many APIs use more than one endpoint for the same resource. Many APIs do not return a list of links on a GET request to the endpoint, but the details for every resource. Many APIs misuse (from the strict REST point of view) extended URLs.

For those sites, you have to read the docs carefully for each individual API to see exactly what they expect, and what they provide.

To avoid that, follow standard REST conventions, and your entire API should be easily discoverable by both humans and programs. Of course, your API will vary in the details of whatever data you're providing.

Public APIs

A list of public RESTful APIs is located here: http://www.programmableweb.com/category/all/apis?data_format=21190

You can use these to examine what they did wrong or, hopefully, did right.

REST + HTTP details

GET

A GET request can either get a list of resourses or the details for a particular resource.

With ID

If an ID is provided as part of the URI, a GET request should return the specified detail record.

Without ID

If given with no ID, the GET request should return all records for that resource, subject to query strings or default pagination. Pagination can be requested by the client or configured at the site or resource level. A successful resource request returns HTTP status 200 (OK)

Returns

200 for success, 4xx if no resource available

POST

The POST request creates a new record. It should be accompanied by JSON data in the body of the request.

Returns

201 (created) on sucess, 4xx for invalid data

PUT/PATCH

The PUT and PATCH requests update a new record. PUT replaces data in a record, and should provide values for all fields. PATCH updates a record and need only provide one or more values.

They should be accompanied by JSON data in the body of the request.

Returns

200 on success, 4xx for invalid data

DELETE

The DELETE request removes a specified record.

Returns

200 on success, 4xx for missing ID

Table 1. RESTful requests

Verb	URL	Description
GET	/API/wombat	Get list of all wombats
POST	/API/wombat	Create a new wombat
GET	/API/wombat/{id}	Get details of wombat by id
PUT	/API/wombat/{id}	Update wombat by id
DELETE	/API/wombat/{id}	Delete wombat by "id"

REST best practices

Accept and provide JSON (not YAML, CSV, XML, etc.)

JSON is the standard. Use it.

Use nouns for endpoint paths (`/api/wombats`, not `/api/get_wombats`)

There is actually a lot of discussion on this point, but for most people, nouns sound most natural.

Use plural nouns (`/api/wombats`, not (`/api/wombat`)

See previous.

Use simple nesting that matches related resources (`/api/wombats/:id/sibling`), not (`/api/wombats/siblings?id=:id`)

While you can set up your URIs any way you like, structure them in a way that matches your related fields.

On error, return standard error codes

- 400 Bad Request — Client submitted incorrect data
- 401 Unauthorized — User is not authenticated
- 403 Forbidden — Authenticated user does not have permission for particular resource
- 404 Not found — Resource is not found (invalid ID, no results for query, etc)

Use secure IDs

Use UUIDs rather than sequential integers for ID fields, to prevent a hacker guessing ID numbers.

Use caching

Caching at any level will improve API throughput. Django has several builtin caching tools. For external tools you can use **Redis** and many others.

Keep verioning simple.

Some REST purists prefer to moving version information into the header. In this case, requests must put the key "Accept-version" in the request header, with a value such as "v1". This means that the URI will never change. However, it makes changing versions invisible to the client, who must know to put the right version in the header. In the absence of "Accept-version", sites usually return the latest version, which could break older client software.

The more common approach is to build the version into the URI:

```
https://www.wombats.com/api/v1/wombats
https://www.wombats.com/api/v2/wombats
```

and so forth. Keep the version part of the URL as simple as possible. It does not have to look like "v1", it can be any string — this will be handled in the URL config in each app, or in the project.

The OpenAPI Spec

- Originally called Swagger
- Describes site + resources
- YAML or JSON

The OpenAPI specification is a standard way of describing a RESTful API.

It grew out of a spec and toolset called **Swagger** originally created in 2010. The company **SmartBear** acquired the Swagger project in 2015, and donated the spec to the Linux foundation. The spec was renamed OpenAPI and is now open source. SmartBear provides commercial API tools under the Swagger brand.

The spec can be used to document an API before it is code, and (spoiler alert!) the Django REST framework can generate an OpenAPI spec from your models, serializers, and filters.

See the spec here: https://swagger.io/specification/

You can use the free API editor from Swagger to create or edit a spec.

https://swagger.io/tools/swagger-editor/

You can download the editor and install it using **npm, or use their online version. If you have trouble with the installation, then just open *index.html** with a browser.

Chapter 5: Django REST Framework Basics

Objectives

- Learn basics of the Django REST framework
- Define serializers for models
- RESTful routing
- Create function-based API views
- Create class-based API views

NOTE	This chapter assumes a Django project is already created, with some available models.

About Django REST framework

- Flexible package for building REST APIs
- Includes OAuth authentication
- Supports ORM and non-ORM data
- Very customizable

The Django REST framework is a comprehensive package for creating RESTful APIs. It leverages the existing Django configuration and infrasructure.

The basic idea is to provide serializers that transform your models (or other data) into JSON (or possibly other formats) that can be returned via your application's REST API.

REST Framework provides class-based views, viewsets, and filters that make it easy to create REST apps.

A big feature of REST Framework is the browsable API.

REST Framework includes OAuth and other kinds of authentication, and supports both ORM and non-ORM data sources. It is extremely customizable.

Django REST Framework Architecture

- Serializers
- Filters
- Class-based views
- Viewsets

The crux of REST Framework, like a page-based Django app, is the model. REST Framework uses normal Django models. What it provides beyond that are serializers, filters, class-based views (CBVs), and viewsets. These work together to let you build apps with less coding.

For each model, you define a serializer. This is a class that converts the data in the object to native Python datatypes that can then be rendered into JSON (or some other format). In the serializer, you can control which fields are exposed to the API as well as performing conversions, etc.

Each model can also have a *filter*. This is a class that decribes query strings for the model. Filters allow you to fetch data with a URL like `⋯/api/contacts?name=Fred`.

To save writing a large number of similar view functions, REST Framework provides two kinds of classes that abstract away common tasks.

The first kind of classes are API views, which provides responses to HTTP requests such as GET or POST. They save the trouble of writing individual view functions for each different HTTP request. There are several basic class-based API views, plus mixins for customization.

The second kind are *viewsets*. Viewset further abstract the data from individual view functions. They work in conjunction with *router*, to automatically set up URL paths.

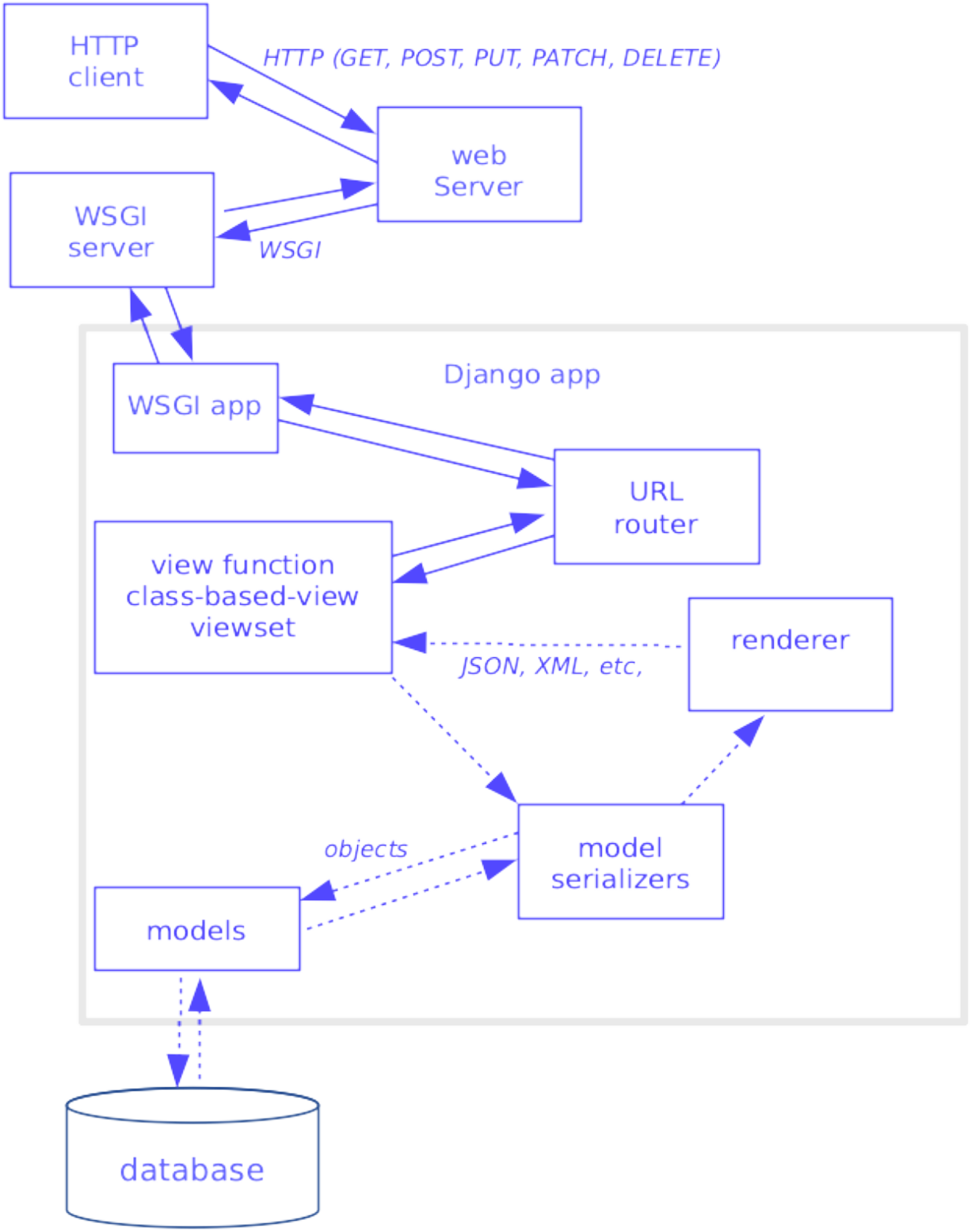

Figure 1. Django REST Architecture

Initial setup

- Add Django REST Framework to settings.py
- Create models
- Create serializers

To get started add **rest_framework** to INSTALLED_APPS in settings.

add a configuration dictionary to the project's settings.py file. This will contain all of the global settings for the REST API.

Serializing the hard way

- Use generic Serializer
- Specify all fields
- Define create and update methods

One way to create serializers is to start with a class that inherits from rest_framework.serializers.Serializer. Then add class variables that map to the fields in your models, or at least all the fields you want to expose in in your REST app.

Once this is done, the serializer class can be used with the matching model to serialize and deserialize the data.

To test serializers, they can be used in the Django shell:

```
In [20]: from contacts_core.models import City, Contact
In [21]: contact = Contact.objects.all().first()
In [22]: contact
Out[22]: <Contact: Contact object (99a5ba00-65bb-4ec0-8400-a7876fe6155c)>
In [23]: serializer = ContactSerializerPlain(contact)
In [24]: serializer.data
Out[24]: {'id': '99a5ba00-65bb-4ec0-8400-a7876fe6155c', 'first_name': 'John',
'last_name': 'Strickler', 'street_address': '4110 Talcott Dr.', 'postcode': '27705',
'dob': '1956-10-31', 'city': OrderedDict([('id', '7dc9cfb0-243f-46ea-b9a1-f1e29807308f'),
('name', 'Durham'), ('admindiv', 'NC'), ('country', 'US')])}
In [25]: from rest_framework.renderers import JSONRenderer
In [27]: json_data = JSONRenderer().render(serializer.data)
In [29]: json_data
Out[29]: b'{"id":"99a5ba00-65bb-4ec0-8400-
a7876fe6155c","first_name":"John","last_name":"Strickler","street_address":"4110 Talcott
Dr.","postcode":"27705","dob":"1956-10-31","city":{"id":"7dc9cfb0-243f-46ea-b9a1-
f1e29807308f","name":"Durham","admindiv":"NC","country":"US"}}'
```

Serializing the easier way

- Use ModelSerializer
- Reads metadata from models
- Only needs nested Meta class

The easy way to create serializers is to let the ModelSerializer class do all the work. It can read the metadata from your models and generate the serializer fields.

Example

django2/contacts/contacts_core/api/serializers.py

```
from rest_framework import serializers
from contacts_core.models import City, Contact

class CitySerializerPlain(serializers.Serializer):

    id = serializers.UUIDField()
    name = serializers.CharField(max_length=32)
    admindiv = serializers.CharField(max_length=32)
    country = serializers.CharField(max_length=2)

class ContactSerializerPlain(serializers.Serializer):

    id = serializers.UUIDField()
    first_name = serializers.CharField(max_length=32)
    last_name = serializers.CharField(max_length=32)
    street_address = serializers.CharField(max_length=32)
    postcode = serializers.CharField(max_length=16)
    dob = serializers.DateField()
    city = CitySerializerPlain()

class CitySerializer(serializers.ModelSerializer):

    class Meta:
        model = City
        fields = ('id', 'name', 'admindiv', 'country')

class ContactSerializer(serializers.ModelSerializer):
    city = serializers.HyperlinkedRelatedField(view_name='contacts_core:api:cbcities-
detail', read_only=True)

    class Meta:
        model = Contact
        fields = ('id', 'first_name', 'last_name', 'street_address', 'city', 'postcode',
'dob')
```

Implementing RESTful views

- Define JSON response from HttpResponse
- Import csrf_exempt decorator
- Create normal views
- Return JSON rather than HTML

For really simple cases, you can create more or less normal Django function-based views. However, you'll want to decorate them with the csrf_exempt decorator which protects them from cross-site scripting.

And of course the views should return JSON, not HTML, so use the serializers that were created earlier plus the JSON renderer.

Example

django2/contacts/contacts_core/api/fb_views.py

```
# not needed for REST CBVs or Viewsets
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework.renderers import JSONRenderer
from contacts_core.models import Contact
from .serializers import ContactSerializerPlain

# Create your RESTful views here.

# example without template (only used in class -- always use templates in real life):
@api_view(['GET'])
def hello(request):
     message = {"message": "Welcome to Contacts API Core"}
     return Response(JSONRenderer.render(message), 200)

@api_view(['GET'])
def contacts(request):
     contacts = Contact.objects.all()
     serializer = ContactSerializerPlain(contacts, many=True)
     contacts_json = JSONRenderer().render(serializer.data)
     return Response(contacts_json, 200)
```

Configuring RESTful routes

- Add views as normal to *app*/urls.py
- Allow for variable parts of URL
- Use named regular expression groups.

Add route configuration to the app's urls.py, and register them in the project's urls.py.

Example

django2/contacts/contacts/urls.py

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

urlpatterns = [
    path('admin', admin.site.urls),
    path('', include('contacts_core.urls', namespace="contacts_core")),

]

# include Django Debug toolbar if DEBUG is set
if settings.DEBUG:
    import debug_toolbar
    urlpatterns = [
        path('__debug__', include(debug_toolbar.urls)),
    ] + urlpatterns
```

Example

django2/contacts/contacts_core/urls.py

```
"""
URL Configuration for contacts_core
"""
from django.urls import path, include
from . import views   # import views from app

app_name = 'contacts_core'

urlpatterns = [
    path('api/', include('contacts_core.api.urls', namespace="api")),

]
```

Example

django2/contacts/contacts_core/api/urls.py

```
from django.urls import path, include
from rest_framework import routers

from . import fb_views
from . import cb_views
from . import viewsets

app_name = 'api'

router = routers.DefaultRouter()
router.register('contacts', viewsets.ContactViewSet)
router.register('cities', viewsets.CityViewSet)

urlpatterns = [
    path('fbv/hello', fb_views.hello, name="hello"),
    path('fbv/contacts', fb_views.contacts, name="fbcontacts"),
    path('cbv/contacts', cb_views.ContactsList.as_view(), name="contacts"),
    path('cbv/contacts/<str:pk>', cb_views.ContactsDetail.as_view(), name="cbcontacts-
detail"),
    path('cbv/cities', cb_views.CitiesList.as_view(), name="cities"),
    path('cbv/cities/<str:pk>', cb_views.CitiesDetail.as_view(), name="cbcities-detail"),
    path('', include(router.urls)),
]
```

Class-based Views

- Builtin view functions
- Just need object and serializer
- Take care of rendering

While you *can* create function-based views, most developers use either viewsets or class-based views. We'll take a look at class-based views (CBVs) here, and viewsets get their own chapter.

To create a class-based view, import the app's models, and import classes from rest_framework.generics.

All that's needed for simple cases is to specify the queryset — the list of objects that the class represents, and the serializer class for those objects.

To use the class-based view, add it to a URL config. Since the URL config needs a *callable* (normally a function or method), CBVs have a method `as_view()` which returns a callable object that will implement the view.

Example URL config

```
path('cbv/contacts/<str:pk>', cb_views.ContactsDetail.as_view(), name="cbcontacts-
detail"),
```

Example

django2/contacts/contacts_core/api/cb_views.py

```
# not needed for REST CBVs or Viewsets
from contacts_core.models import Contact, City
from rest_framework import generics
from  .serializers import ContactSerializer, CitySerializer

# class-based views (aka CBVs)
class ContactsList(generics.ListCreateAPIView):
    queryset = Contact.objects.all()
    serializer_class = ContactSerializer

class ContactsDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Contact.objects.all()
    serializer_class = ContactSerializer

class CitiesList(generics.ListCreateAPIView):
    queryset = City.objects.all()
    serializer_class = CitySerializer

class CitiesDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = City.objects.all()
    serializer_class = CitySerializer
```

Chapter 5 Exercises

Exercise 5-1: (dogs/dogs_core/*)

Create a simple (non model-based) serializer for the Dog model, and a simple function-based view to display it.

Exercise 5-2: (dogs/dogs_core/*)

Create a model-based serializer for the Dog model, and a class-based view to display it.

Chapter 6: Django REST Viewsets

Objectives

- Understand viewsets
- Expose models using viewsets
- Provide pagination

What are viewsets?

- Higher level of abstraction
- Tools for exposing models
- "Easier than CBVs"

viewsets are REST Framework classes that go beyond class-based views to a higher level of abstraction. They provide methods such as `list()` and `create()` rather than mapping directly to the HTTP verbs.

In practice, they mean that you have even less code to write. The combine all the logic for the views for a model in a ingle class.

Another advantage of viewsets is that they define their own routes, so you typically only have to add one line to your URL config for each model, and don’t have to worry about naming the routes individually.

Creating Viewsets

- Import `viewsets`
- Define class
- Specify queryset and serializer

In many ways, using a viewset is similar to using a class-based view. You import a viewset, create a class to subclass it, and specify the model and serializer needed.

There are serveral base viewset classes. Which one to choose depends on how much customization there is. The most generic viewset is ViewSet, which requires you to write your own methods as needed.

The most commonly used viewset is ModelViewset, which exposes the data from a model, using the model's serializer.

NOTE you can customize viewsets in the same ways you can customize CBVs.

Example

django2/contacts/contacts_core/api/viewsets.py

```
from rest_framework import viewsets
from contacts_core.api.serializers import ContactSerializer, CitySerializer
from contacts_core.api.filters import * # (optional)  change to only needed serializers

class ContactViewSet(viewsets.ModelViewSet):
    queryset = Contact.objects.all()
    serializer_class = ContactSerializer
    # filter_backends = [DjangoFilterBackend]
    # filterset_class = MyFirstModelFilter


class CityViewSet(viewsets.ModelViewSet):
    queryset = City.objects.all()
    serializer_class = CitySerializer
    # filter_backends = [DjangoFilterBackend]
    # filterset_class = MySecondModelFilter
```

Table 2. Available ViewSets

Viewset	Decription
ViewSet	Minimal viewset — add attributes (auth classes, permission classes) to define behavior
GenericViewSet	Inherits from GenericAPIView, but doesn't define actions
ModelViewSet	Includes implementations of standard actions.
ReadOnlyModelViewSet	like ModelViewSet, but read-only

Setting up routes

- Create a **router**
- Register viewsets with routers
- Add router.urls to URL config

To set up routes for a viewset, import **routers** from `rest_framework`.

Create a DefaultRouter (or other router) and register one or more viewsets with it. The name a viewset is registered with is incorporated into its route. These names are the resource endpoints.

In `urlpatterns`, use `include()` to delegate to the router-generated urls.

NOTE Routers, like everything else in Django, can be customized.

Example

django2/contacts/contacts_core/api/urls.py

```
from django.urls import path, include
from rest_framework import routers

from . import fb_views
from . import cb_views
from . import viewsets

app_name = 'api'

router = routers.DefaultRouter()
router.register('contacts', viewsets.ContactViewSet)
router.register('cities', viewsets.CityViewSet)

urlpatterns = [
    path('fbv/hello', fb_views.hello, name="hello"),
    path('fbv/contacts', fb_views.contacts, name="fbcontacts"),
    path('cbv/contacts', cb_views.ContactsList.as_view(), name="contacts"),
    path('cbv/contacts/<str:pk>', cb_views.ContactsDetail.as_view(), name="cbcontacts-
detail"),
    path('cbv/cities', cb_views.CitiesList.as_view(), name="cities"),
    path('cbv/cities/<str:pk>', cb_views.CitiesDetail.as_view(), name="cbcities-detail"),
    path('', include(router.urls)),
]
```

Customizing viewsets

- Add class-level attributes
- Useful attributes
  - permission classes
  - lookup_field
  - pagination_class
  - filter_backends

In addition to `queryset` and `serializer_class`, there are many attributes than can be defined on a viewset.

`permission_classes` allows you to specify what permissions are required to access data in the view.

`lookup_field` specifies a field other than **pk** (the default) for item lookup.

`pagination_class` specifies a class for controlling pagination.

`filter_backends` provides filters as HTTP query strings to search the models.

See https://www.django-rest-framework.org/api-guide/generic-views/#genericapiview for a complete list of available attributes.

Adding pagination

- Controls number of items retrieved
- Can be set
  - App-wide
  - Per model

If your database has ten million wombat records and you make a generic request to list the wombat resource, it may overwhelm your client application. A well-designed API should allow limits and pagination.

REST Framework provides the **LimitOffsetPagination** class, which lets you provide a limit and an offset. The offset is the (0-based) first object to retrieve, and limit is the number of items to retrieve.

Also, using pagination adds **next** and **previous** fields to your result so that a client app can use them for navigation.

Normally, you want all views to have the same pagination, so you set it up in settings.py as one of the entries in REST_FRAMEWORK:

```
'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination',
```

Assign to the `pagination_class` attribute of a CBV or viewset if you want to change the pagination scheme for that particular model.

NOTE set `pagination_class=None` to disable pagination.

Chapter 6 Exercises

Exercise 6-1 (dogs/dogs_core/*)

Add viewsets to your dogs app.

(Optional): Add pagination.

Chapter 8: Django REST Documentation

Objectives

- Generate browsable API docs
- Generate OpenAPI schemas

Documentation?

- Useful before coding
- Useful after coding
- Critical for client apps

Of course, it is important to provide documentation for applications. In the case of REST, it is crucial for anyone writing client code for your REST endpoints.

You can generate a schema from your existing API using various tools.

Generating an OpenAPI schema

- Install **drf-spectactular**
- Add routes

To generate an OpenAPI schema, you can install the **drf-spectacular** extension to the REST Framework.

Then you just need to add routes to your URL config.

To install, use

```
pip install drf-spectacular
```

Add the following to **REST_FRAMEWORK** in `settings.py`.

```
'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
```

Then, in your project-level URL config, import views from `drf_spectacular.views`.

You can define route the three views as follows:

```
path('api/schema/', SpectacularAPIView.as_view(), name='schema'),
# Optional UI:
path('api/schema/swagger-ui/', SpectacularSwaggerView.as_view(url_name='schema'),
name='swagger-ui'),
path('api/schema/redoc/', SpectacularRedocView.as_view(url_name='schema'), name='redoc'),
]
```

Now just visit the defined URLs.

- api/schema *download the schema as YAML*
- api/schema/swagger-ui *browsable Swagger/OpenAPI schema*
- api/schema/redoc *browsable ReDoc schema*

The ReDoc schema grew out of the Swagger schema, and provides a 3-panel layout.

Chapter 8 Exercises

Exercise 8-1 (dogs/dogs/*)

Use drf-spectacular to generate the OpenAPI spec for the dogs API.

Generate and test Python client code.

Chapter 9: Django User Authentication

Objectives

- Understand Django's builtin authentication system
- Create and use User objects
- Add authentication to web requests
- Customize users

Users

- Core of auth system
- Represent app users
- All users created equal

User objects represent the users of your application. Via Users, you can set access, keep profiles, tag content with its creator, and other tasks. There is only one type of User; elevated permissions ("root", "admin", etc.) are controlled by User attributes.

Users have the following attributes:

- username
- password
- email
- first_name
- last_name

Creating Users

- Use the admin tool
- User.objects.create_user()
- manage.py createsuperuser …
- manage.py changepassword …

If you have enabled the Django admin system, then you can easily create users through that interface.

You can also create users with the create_user() helper function.

As with most auth systems, the password itself is not stored in the database, but just the hash value of the password. One result of this is that you should not directly modify the password field in the database.

You can create a superuser (admin user) with

```
manage.py createsuperuser --username=username email=email
```

You can change a password with

```
manage.py changepassword username
```

Alternatively, you can change the password with the set_password() method on a User object.

Authenticating Users (low-level)

- django.contrib.auth.authenticate
- Pass in username/password
- Returns non-None value on success

You can authenticate credentials with the authenticate() function from django.contrib.auth. It takes a username and password as parameters, and returns None if the authentication fails. It returns a non-None (i.e., True) value when the authentication is successful.

However, most of the time you will want to use the login_required() decorator, which calls the above for you.

Example

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    # A backend authenticated the credentials
else:
    # No backend authenticated the credentials
```

Permissions

- Kinds of permission: add, change, delete
- Set permissions on object types and instances
- user.user_permissions.set(), .add(), .remove(), .clear()

You can set permissions on objects in general, and on specific instances of object.

You can also create custom permissions

Groups

- Users sharing permissions
- User gets all permissions of group

You can create any number of groups that represent a shared set of permissions.

Create groups via the Django admin page, or programmatically with django.contrib.auth.models.Group. Use

```
Group.objects.get_or_crate(name='group name')
```

To create a new group. It returns a tuple containing the new group object (if created) and a Boolean value that says whether the group was successfully created. After the group has been created, you can refer to it by name when working with users and permissions.

Web request authentication

- Django has builtin sessions
- request object has user attribute
- Check request.user.is_authenticated

Django uses sessions and builtin middleware to provide authentication for request objects.

A request.user attribute on every request corresponds to the current user. If the current user has not logged in, this attribute will be set to an instance of AnonymousUser, otherwise it will be an instance of User.

To tell whether the user is authenticated, use:

```
if request.user.is_authenticated:
    # Do something for authenticated users.
    ...
else:
    # Do something for anonymous users.
    ...
```

login shortcuts

- login_required() decorator
- LoginRequiredMixin
- Decorator for view functions

To simplify working with authentication, there are two shortcuts – one for view functions, and one for class-based views.

For view functions, use the @login_required decorator. When a user visits a url served by the decorated view function, it redirects to settings.LOGIN_URL (a path to your app's login page) if the user is not logged in.

For class-based views, you can add LoginRequiredMixin to the list of base classes.

from django.contrib.auth.mixins import LoginRequiredMixin

Example

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

or

```
from django.contrib.auth.mixins import LoginRequiredMixin

class MyView(LoginRequiredMixin, View):
    login_url = '/login/'
    redirect_field_name = 'redirect_to'
```

Customizing users

- Proxy user
- One-to-one
- Extend builtin user

There are three primary ways to customize a Django user.

Proxy model

A proxy model can be used when you need to modify the behavior of the user model without actually changing the database schema. You could, for instance, specify a different manager for queries, or add custom methods.

To do this,

1. import the User model from `django.contrib.auth.models`.
2. define a new model that inherits from User
3. modify the model
    - add a different manager
    - define custom methods
    - add Meta fields

Think of a proxy model as a *view* of the "real" User model.

One-to-one model

A one-to-one model can be used when you want to store extra data for users, but don't need to modify the authorization process. without the complexity of extending the user model.

This is most useful when there is profile data to be kept for each user.

To do this,

1. define a new model (do not inherit from `django.contrib.auth.models.User`)
2. add a field of type OneToOneField.
3. use *signals* to synchronize the new model with User
    - import `post_save` or other signals from `django.db.models.signals`
    - import `receiver` from `django.dispatch`
    - define methods for saving, updating, etc.

- decorate methods with `@receiver(post_save, sender=User)`
- methods will be called when User is updated

NOTE Using a one-to-one model may result in extra queries to the database, which could have an adverse effect on throughput.

Custom model

The most flexible (and complex) way to customize a user is to extend the user model. In this case, you're creating a new model that inherits from User.

This can be used to completely customize the behavior of a user. If you do this, you must update the settings files with:

```
AUTH_USER_MODEL = '<myapp>.User'
```

Chapter 9 Exercises

Exercise 9-1 (dogs/*)

Add a simple login page to your dogs app. Use a form with Username/Password fields and a Submit button. You can use Django's builtin auth system.

Chapter 12: Django REST Final Project

Django REST Final Project

Design and develop a REST API to access artists and artworks from the Tate Galleries in London, UK. The data is availble as two CSV files, in `DATA/tate_data`.

- Create a data migration to load the data (or one of the other suggested forms of loading data)
- Use best practices to create an app with two resource endpoints:

```
/api/artists
/api/artworks
```

- Provide links for related fields
  - Artist resource should provide links to artwork details
  - Artwork should provide link to artist
- Allow for all normal actions on the resources: GET, POST, PUT, PATCH, and DELETE. These should have their usual RESTful meanings.
- Implement token-based authentication
- Configure offset/limit pagination
- Provide case-insensitive search terms for both artists and artworks.
  - Artists
    - Name
    - Birthplace
    - Year of birth (max/min)
    - Gender
  - Artworks
    - Title
    - Medium
- Provide a browsable OpenAPI schema at `/api/schema`
- Set up the admin interface for DB maintenance
- Create unit tests for all area of the app
  - Create test fixtures
  - Some things you might test for:
    - Invalid UUID returns 404
    - DELETE removes item from database
    - Search terms retrieve

- Create an admin user with permission to modify records; normal users will have only read-write access.

This is your project. Feel free to vary any of the above, especially in areas of real life interest.

NOTE	The project will not be graded *per se*, but you are welcome to submit a zip file or GitHub link of your project by email to the instructor for constructive criticism.

Appendix A: Using Cookiecutter

About cookiecutter

- Alternate startup tool
- Creates "real-life" Django setup
- Different from **startproject**
- Authors
  - Audrey Roy Greenfeld
  - Daniel Roy Greenfeld

cookiecutter is a utility written by Audrey and Roy Greenfeld to make it easy to replicate a standard setup for Django. The cookiecutter command prompts you for information, then creates the project folder.

It uses a cookiecutter *template*, which is a folder, to create the new project. There are several templates on **github** to choose from. The standard template, cookiecutter-django, is a bit advanced ("bleeding edge", in the authors' words) for class, so two basic templates (one for projects and one for apps) are provided with the class files.

The utility copies the template layout (all folders and files) to a new folder which is the "slug" (short name) of your project. It inserts your project name in the appropriate places.

CAUTION	Not all templates are templates! When you see 'template' above, it refers to a cookiecutter source folder and its files, *not* a Django or Jinja2 HTML template.

cookiecutter home page: https://github.com/audreyr/cookiecutter
cookiecutter docs: https://cookiecutter.readthedocs.io

Using cookiecutter

- cookiecutter *cookiecutter_name*
- creates folder under current

To use **cookiecutter**, execute the cookiecutter command with one argument, the name of the cookiecutter template folder. This can be a folder on the local hard drive, or it can be online. cookiecutter supports Github, Mercurial, and GitLab repositories, but you can also just give the URL to any online file.

cookiecutter will ask a few configuration questions. (The standard template is much more elaborate). One of the questions is the name of the project "slug". This is the short name that will be used to name files and folders, so it should be short and only contain letters, digits, and underscores. The default slug is created from the project name, but can be anything you like. The project slug will be used as the name of the main project folder.

The new folder, which is your Django project, is created in the current folder.

Example local usage

```
cookiecutter ../SETUP/cookiecutter-{django-version}
project_name [Project Name]: My Wonderful Project
project_slug [my_wonderful_project]: wonderful
project_description [Short Description of the project]: A Wonderful Django Project for
Class
time_zone [America/New_York]:

Congratulations! you have created project wonderful

Now, cd into the wonderful folder.

Execute this command to set up Django's admin and user databases:

    python manage.py migrate

At this point, you can start creating apps.
```

tree wonderful

```
wonderful
├── manage.py
└── wonderful
    ├── _\_init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Example online usage

```
cookiecutter https://github.com/audreyr/cookiecutter-pypackage.git
cookiecutter git+ssh://git@github.com/audreyr/cookiecutter-pypackage.git
cookiecutter hg+ssh://hg@bitbucket.org/audreyr/cookiecutter-pypackage
```

Django app creation checklist

The following is the general workflow when creating an app using the cookiecutter layout provided with the class. You can leave out the steps involving models and migrations if you are not using a database.

- ▢ Create the project using cookiecutter (or other layout creator)

```
cookiecutter ../SETUP/cookiecutter-{django-version}
```

- ▢ Navigate to the project folder

```
cd project_name
```

- ▢ Run the first migration (optional for now)

```
python manage.py migrate
```

- ▢ Create the app using cookiecutter (or other layout creator)

```
cookiecutter ../SETUP/cookiecutter-{django-version}-app
```

- ▢ Add the app to INSTALLED_APPS in *project_name.project_name*.settings.py
- ▢ Create one or more models in *project_name.app_name*.models.py
- ▢ Create migrations and run them

```
python manage.py makemigrations
python manage.py migrate
```

- ▢ Create one or more views in *project_name.app_name*.views.py
- ▢ Create one or more templates in *project_name.app_name*.templates
- ▢ Add the views to the app's URL config in *project_name.app_name*.urls.py
- ▢ Add the app's URL config to the project's URL config in *project_name.project_name*.urls.py
- ▢ Start the development server

```
python manage.py runserver
```

Appendix B: Python Bibliography

Title	Author	Publisher
Data Science		
Building machine learning systems with Python	William Richert, Luis Pedro Coelho	Packt Publishing
High Performance Python	Mischa Gorlelick and Ian Ozsvald	O'Reilly Media
Introduction to Machine Learning with Python	Sarah Guido	O'Reilly & Assoc.
iPython Interactive Computing and Visualization Cookbook	Cyril Rossant	Packt Publishing
Learning iPython for Interactive Computing and Visualization	Cyril Rossant	Packt Publishing
Learning Pandas	Michael Heydt	Packt Publishing
Learning scikit-learn: Machine Learning in Python	Raúl Garreta, Guillermo Moncecchi	Packt Publishing
Mastering Machine Learning with Scikit-learn	Gavin Hackeling	Packt Publishing
Matplotlib for Python Developers	Sandro Tosi	Packt Publishing
Numpy Beginner's Guide	Ivan Idris	Packt Publishing
Numpy Cookbook	Ivan Idris	Packt Publishing
Practical Data Science Cookbook	Tony Ojeda, Sean Patrick Murphy, Benjamin Bengfort, Abhijit Dasgupta	Packt Publishing
Python Text Processing with NLTK 2.0 Cookbook	Jacob Perkins	Packt Publishing
Scikit-learn cookbook	Trent Hauck	Packt Publishing
Python Data Visualization Cookbook	Igor Milovanovic	Packt Publishing
Python for Data Analysis	Wes McKinney	O'Reilly & Assoc.
Design Patterns		
Design Patterns: Elements of Reusable Object-Oriented Software	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides	Addison-Wesley Professional
Head First Design Patterns	Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra	O'Reilly Media
Learning Python Design Patterns	Gennadiy Zlobin	Packt Publishing

Title	Author	Publisher
Mastering Python Design Patterns	Sakis Kasampalis	Packt Publishing
General Python development		
Expert Python Programming	Tarek Ziadé	Packt Publishing
Fluent Python	Luciano Ramalho	O'Reilly & Assoc.
Learning Python, 2nd Ed.	Mark Lutz, David Asher	O'Reilly & Assoc.
Mastering Object-oriented Python	Stephen F. Lott	Packt Publishing
Programming Python, 2nd Ed.	Mark Lutz	O'Reilly & Assoc.
Python 3 Object Oriented Programming	Dusty Phillips	Packt Publishing
Python Cookbook, 3nd. Ed.	David Beazley, Brian K. Jones	O'Reilly & Assoc.
Python Essential Reference, 4th. Ed.	David M. Beazley	Addison-Wesley Professional
Python in a Nutshell	Alex Martelli	O'Reilly & Assoc.
Python Programming on Win32	Mark Hammond, Andy Robinson	O'Reilly & Assoc.
The Python Standard Library By Example	Doug Hellmann	Addison-Wesley Professional
Misc		
Python Geospatial Development	Erik Westra	Packt Publishing
Python High Performance Programming	Gabriele Lanaro	Packt Publishing
Networking		
Python Network Programming Cookbook	Dr. M. O. Faruque Sarker	Packt Publishing
Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers	T J O'Connor	Syngress
Web Scraping with Python	Ryan Mitchell	O'Reilly & Assoc.
Testing		
Python Testing Cookbook	Greg L. Turnquist	Packt Publishing
Learning Python Testing	Daniel Arbuckle	Packt Publishing

Title	Author	Publisher
Learning Selenium Testing Tools, 3rd Ed.	Raghavendra Prasad MG	Packt Publishing
Web Development		
Building Web Applications with Flask	Italo Maia	Packt Publishing
Django 1.0 Website Development	Ayman Hourieh	Packt Publishing
Django 1.1 Testing and Development	Karen M. Tracey	Packt Publishing
Django By Example	Antonio Melé	Packt Publishing
Django Design Patterns and Best Practices	Arun Ravindran	Packt Publishing
Django Essentials	Samuel Dauzon	Packt Publishing
Django Project Blueprints	Asad Jibran Ahmed	Packt Publishing
Flask Blueprints	Joel Perras	Packt Publishing
Flask by Example	Gareth Dwyer	Packt Publishing
Flask Framework Cookbook	Shalabh Aggarwal	Packt Publishing
Flask Web Development	Miguel Grinberg	O'Reilly & Assoc.
Full Stack Python (e-book only)	Matt Makai	Gumroad (or free download)
Full Stack Python Guide to Deployments (e-book only)	Matt Makai	Gumroad (or free download)
High Performance Django	Peter Baumgartner, Yann Malet	Lincoln Loop
Instant Flask Web Development	Ron DuPlain	Packt Publishing
Learning Flask Framework	Matt Copperwaite, Charles O Leifer	Packt Publishing
Mastering Flask	Jack Stouffer	Packt Publishing
Two Scoops of Django: Best Practices for Django 1.11	Daniel Roy Greenfeld, Audrey Roy Greenfeld	Two Scoops Press
Web Development with Django Cookbook	Aidas Bendoraitis	Packt Publishing

Appendix C: Django Caching

About caching

- Web apps serve same pages over and over
- Each page needs work
- Caching skips redoing the work

A typical web app serves the same pages over and over, as users go to the same places and request the same data. Depending on the app, serving the page may involving fetching data from the database, performing some business logic on the data, and then passing the data to a template rendering function, which parses the template and fills in the data.

It is redundant to do this every time a user asks for a certain URL, so many web sites use caching, which stores returned pages and retrieves them directly without going through the view.

Caches may be set up to only cache for a certain amount of time, to only grow to a certain size, or both.

Python comes with **memcached,** which is robust and easy to set up. Other caching systems can be used, as well.

Types of caches

- In-memory
- Database
- File system
- Dummy

There are several locations to store cached pages. Many caches store pages in memory, although for huge sites this may not be feasible. For those sites, pages can be stored in a database or on the filesystem.

Django provides a dummy cache that does nothing. It is for use during development when you don't yet care about caching. It is then easy to swap out the dummy backend for the real backend.

Setting up the cache

- Set CACHES in settings.py
- set up default cache unless using more than one
- Set BACKEND to Python package providing cache
- set LOCATION to IP/port or other as needed by cache
- Run manage.py createcachetable

To set up the cache, define the CACHES option in settings.py.

Unless you are using more than one cache system, you can use the default label.

Set BACKEND to the Python package that implements the cache, and set LOCATION to the connection information needed by the cache. This may be an IP/port or other data.

With most caching backends, LOCATION can be a list of multiple locations to spread the load over several machines.

Once caching is configured, run

```
manage.py createcachetable
```

To set up a table in the database to support caching.

Example

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

Cache options

- CACHES takes many optional arguments
- Control size and timeout
- Key (data) manipulation

CACHES has many optional values to fine-tune how your pages are cached.

Example

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
        'TIMEOUT': 60,
        'OPTIONS': {
            'MAX_ENTRIES': 1000
        }
    }
}
```

Table 3. Cache Options

Option	Description
TIMEOUT	Default timeout, in seconds
OPTIONS	Options passed through tocache backend
MAX_ENTRIES	Maximum entries allowed
CULL_FREQUENCY	Fraction culled at MAX_ENTRIES
KEY_PREFIX	Prefix for all cache keys
VERSION	Version number for cache keys
KEY_FUNCTION	Path to a function composing key

Per-site and per-view caching

- Default is per-site
- Add apps to MIDDLEWARE in settings.py

To enable site-wide caching, add middleware apps to settings.py like this:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
    ...
]
```

These apps most be in the order specified.

To enable caching only on selected views, use the cache_page() decorator from django.views.decorators.cache. This decorator takes one argument, an integer timeout in seconds.

While you can use this decorator in the modules that define your views (e.g.., views.py), this ties your app closely to the cache.

You can also wrap the decorator around views passed to url() in your URLconf:

```
from django.views.decorators.cache import cache_page

urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', cache_page(60 * 15)(my_view)),
]
```

Low-level API

- Access cache directly
- Store any picklable object (most builtin types)

django.core.cache.caches is a dictionary-like object that provides access to the caches. You can access keys via caches[key] lookup, or set key/value using caches.set(key, value).

Appendix D: Django Database Connections

Database configuration

- DATABASES in settings
- Dictionary of connections
- Required: name, engine
- Optional: *many*

Databases are configured with the **DATABASES** setting. This is a dictionary with an entry for each connection. The key is the database alias for use within Django.

The alias "default" is special. It is used when a database is not specified. If you are only using one database, it is sufficient.

The value for each connection is a dictionary of per-connection settings. At least **name** and **engine** are required.

DATABASES options

Table 4. DATABASES options

Setting	Default	Description
ATOMIC_REQUESTS	False	Wrap each view in a transaction
AUTOCOMMIT	True	Use builtin transaction management.
ENGINE	'' (Empty string)	Database backend to use
HOST	'' (Empty string)	Database host (empty means localhost)
NAME	'' (Empty string)	Database name. For SQLite, it's full path database
CONN_MAX_AGE	0	Connection lifetime in seconds. 0 to close connections per request; None for unlimited persistent connections.
OPTIONS	{} (Empty dictionary)	Extra connection parameters
PASSWORD	'' (Empty string)	Database password (not used with SQLite)
PORT	'' (Empty string)	Database port (not used with SQLite)
USER	'' (Empty string)	Database user name (not used with SQLite)
TEST	{} (Empty dictionary)	Settings for test databases

Database backends

- Builtin
  - django.db.backends.postgresql_psycopg2
  - django.db.backends.mysql
  - django.db.backends.sqlite3
  - django.db.backends.oracle
- Available
  - MS SQL Server
  - SAP SQL Anywhere
  - IBM DB2
  - Microsoft SQL Server
  - Firebird
  - ODBC

Django ships with four backends (engines) for popular databases. Many backends are available for other databases.

The backend does the work of translating Django's generic models into the details for each database manager.

Multiple DBMS connections

- Load balancing
- Public/Private data
- If "default" empty must specify db
- Data is "sticky"

Many projects use more than one database. This might be for load balancing, or for keeping private/internal data on a separate database from public data.

The databases do not have to be on the same machine, and they do not have to be the same type. You might use SQLite for Django's internal admin databases while using Oracle for actual site data.

A common scenario for multiple databases is load balancing. There could be one primary server, and 3 replicated servers. Updates would happen on the primary, but read-only access would happen on a randomly selected clone server.

Migrating multiple DBMS

- Only "default" migrated by default
- Specify with --database=*database*
- All models will be synchronized

When you run **python manage.py migrate**, only the default database will be migrated. Use the **--database** option to specify others.

All models will be synchronized to each database.

The **makemigrations** command is not per-database, but only looks for changes to the models. Migrations will be covered in detail in a later chapter.

NOTE	Most database-related commands from manage.py use --database.

Selecting connection in views

- Default used if not specified
- *model*.objects.using("*dbalias*")
- *object*.save(using="*dbalias*")

When accessing data from multiple databases, querysets and **.save()** (or **.delete()**) will use the default database unless you specify another.

For general querying, the **.using** method is added to the normal manager (**objects**) to specify the database.

When saving or deleting, add the **using**= parameter.

NOTE	If you have a custom model manager, add **.db_manager(*database*) to *objects**.

Accessing connection directly

- Use django.db.connections
- Dictionary-like object.
- Can get a cursor

To access a connection directly, import **connections** from **django.db**. This can be used for raw queries.

```
from django.db import connections
conn = connections['wombat2']

cur = conn.cursor()
cur.execute('select * from wombats')
results = cur.fetchall()
```

Index