FREE eBook

LEARNING Django

Free unaffiliated eBook created from
Stack Overflow contributors.

#django

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: django

It is an unofficial and free Django ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Django.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Django

Remarks

Django advertises itself as "the web framework for perfectionists with deadlines" and "Django makes it easier to build better Web apps more quickly and with less code". It can be seen as an MVC architecture. At it's core it has:

- a lightweight and standalone web server for development and testing
- a form serialization and validation system that can translate between HTML forms and values suitable for storage in the database
- a template system that utilizes the concept of inheritance borrowed from object-oriented programming
- a caching framework that can use any of several cache methods support for middleware classes that can intervene at various stages of request processing and carry out custom functions
- an internal dispatcher system that allows components of an application to communicate events to each other via pre-defined signals
- an internationalization system, including translations of Django's own components into a variety of languages
- a serialization system that can produce and read XML and/or JSON representations of Django model instances
- a system for extending the capabilities of the template engine
- an interface to Python's built in unit test framework

Versions

Version	Release Date
1.11	2017-04-04
1.10	2016-08-01
1.9	2015-12-01
1.8	2015-04-01
1.7	2014-09-02
1.6	2013-11-06
1.5	2013-02-26
1.4	2012-03-23
1.3	2011-03-23

Version	Release Date
1.2	2010-05-17
1.1	2009-07-29
1.0	2008-09-03

Examples

Starting a Project

Django is a web development framework based on Python. Django **1.11** (the latest stable release) requires Python **2.7**, **3.4**, **3.5** or **3.6** to be installed. Assuming `pip` is available, installation is as simple as running the following command. Keep in mind, omitting the version as shown below will install the latest version of django:

```
$ pip install django
```

For installing specific version of django, let's suppose the version is django **1.10.5** , run the following command:

```
$ pip install django==1.10.5
```

Web applications built using Django must reside within a Django project. You can use the `django-admin` command to start a new project in the current directory:

```
$ django-admin startproject myproject
```

where `myproject` is a name that uniquely identifies the project and can consist of **numbers**, **letters**, and **underscores**.

This will create the following project structure:

```
myproject/
    manage.py
    myproject/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

To run the application, start the development server

```
$ cd myproject
$ python manage.py runserver
```

Now that the server's running, visit `http://127.0.0.1:8000/` with your web browser. You'll see the

following page:

By default, the `runserver` command starts the development server on the internal IP at port `8000`. This server will automatically restart as you make changes to your code. But in case you add new files, you'll have to manually restart the server.

If you want to change the server's port, pass it as a command-line argument.

```
$ python manage.py runserver 8080
```

If you want to change the server's IP, pass it along with the port.

```
$ python manage.py runserver 0.0.0.0:8000
```

Note that `runserver` is only for debug builds and local testing. Specialised server programs (such as Apache) should always be used in production.

Adding a Django App

A Django project usually contains multiple `apps`. This is simply a way to structure your project in smaller, maintainable modules. To create an app, go to your projectfolder (where `manage.py` is), and run the `startapp` command (change *myapp* to whatever you want):

```
python manage.py startapp myapp
```

This will generate the *myapp* folder and some necessary files for you, like `models.py` and `views.py`.

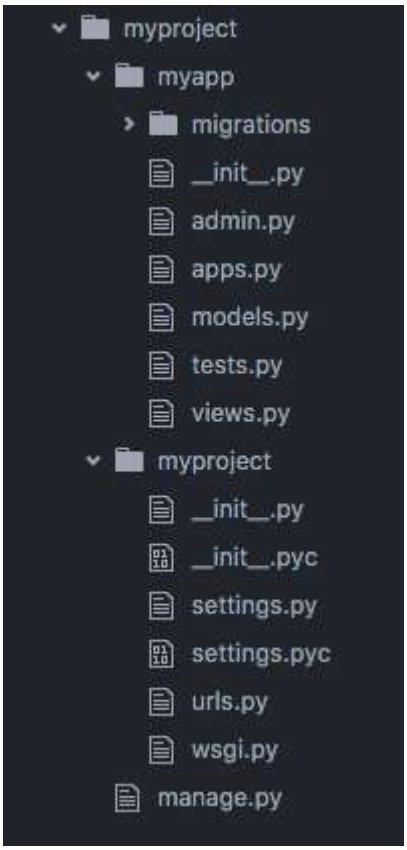
In order to make Django aware of *myapp*, add it to your `settings.py`:

```
# myproject/settings.py

# Application definition
INSTALLED_APPS = [
    ...
    'myapp',
]
```

The folder-structure of a Django project can be changed to fit your preference. Sometimes the

project folder is renamed to `/src` to avoid repeating folder names. A typical folder structure looks like this:



Django Concepts

django-admin is a command line tool that ships with Django. It comes with several useful commands for getting started with and managing a Django project. The command is the same as `./manage.py` , with the difference that you don't need to be in the project directory. The `DJANGO_SETTINGS_MODULE` environment variable needs to be set.

A **Django project** is a Python codebase that contains a Django settings file. A project can be created by the Django admin through the command `django-admin startproject NAME`. The project typically has a file called `manage.py` at the top level and a root URL file called `urls.py`. `manage.py` is a project specific version of `django-admin`, and lets you run management commands on that project. For example, to run your project locally, use `python manage.py runserver`. A project is made up of Django apps.

A **Django app** is a Python package that contains a models file (`models.py` by default) and other files such as app-specific urls and views. An app can be created through the command `django-admin startapp NAME` (this command should be run from inside your project directory). For an app to be part of a project, it must be included in the `INSTALLED_APPS` list in `settings.py`. If you used the standard configuration, Django comes with several apps of it's own apps preinstalled which will handle things like authentication for you. Apps can be used in multiple Django projects.

The **Django ORM** collects all of the database models defined in `models.py` and creates database tables based on those model classes. To do this, first, setup your database by modifying the `DATABASES` setting in `settings.py`. Then, once you have defined your database models, run `python`

`manage.py makemigrations` followed by `python manage.py migrate` to create or update your database's schema based on your models.

A complete hello world example.

Step 1 If you already have Django installed, you can skip this step.

```
pip install Django
```

Step 2 Create a new project

```
django-admin startproject hello
```

That will create a folder named `hello` which will contain the following files:

```
hello/
├── hello/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

Step 3 Inside the `hello` module (the folder containing the `__init.py__`) create a file called `views.py`:

```
hello/
├── hello/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   ├── views.py  <- here
│   └── wsgi.py
└── manage.py
```

and put in the following content:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse('Hello, World')
```

This is called a view function.

Step 4 Edit `hello/urls.py` as follows:

```
from django.conf.urls import url
from django.contrib import admin
from hello import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', views.hello)
```

```
]
```

which links the view function `hello()` to a URL.

Step 5 Start the server.

```
python manage.py runserver
```

Step 6

Browse to `http://localhost:8000/` in a browser and you will see:

> Hello, World

Virtual Environment

Although not strictly required, it is highly recommended to start your project in a "virtual environment." A virtual environment is a **container** (a directory) that holds a specific version of Python and a set of modules (dependencies), and which does not interfere with the operating system's native Python or other projects on the same computer.

By setting up a different virtual environment for each project you work on, various Django projects can run on different versions of Python, and can maintain their own sets of dependencies, without risk of conflict.

Python 3.3+

Python 3.3+ already includes a standard `venv` module, which you can usually call as `pyvenv`. In environments where the `pyvenv` command is not available, you can access the same functionality by directly invoking the module as `python3 -m venv`.

To create the Virtual environment:

```
$ pyvenv <env-folder>
# Or, if pyvenv is not available
$ python3 -m venv <env-folder>
```

Python 2

If using Python 2, you can first install it as a separate module from pip:

```
$ pip install virtualenv
```

And then create the environment using the `virtualenv` command instead:

```
$ virtualenv <env-folder>
```

Activate (any version)

The virtual environment is now set up. In order to use it, it must be *activated* in the terminal you want to use it.

To 'activate' the virtual environment (any Python version)

Linux like:

```
$ source <env-folder>/bin/activate
```

Windows like:

```
<env-folder>\Scripts\activate.bat
```

This changes your prompt to indicate the virtual environment is active. `(<env-folder>) $`

From now on, everything installed using `pip` will be installed to your virtual env folder, not system-wide.

To leave the virtual environment use `deactivate` :

```
(<env-folder>) $ deactivate
```

Alternatively: use virtualenvwrapper

You may also consider using virtualenvwrapper which makes virtualenv creation and activation very handy as well as separating it from your code:

```
# Create a virtualenv
mkvirtualenv my_virtualenv

# Activate a virtualenv
workon my_virtualenv

# Deactivate the current virtualenv
deactivate
```

Alternatively: use pyenv + pyenv-viritualenv

In environments where you need to handle multiple Python versions you can benefit from virtualenv together with pyenv-virtualenv:

```
# Create a virtualenv for specific Python version
pyenv virtualenv 2.7.10 my-virtual-env-2.7.10
```

```
# Create a vritualenv for active python verion
pyenv virtualenv venv34

# Activate, deactivate virtualenv
pyenv activate <name>
pyenv deactivate
```

When using virtualenvs, it is often useful to set your `PYTHONPATH` and `DJANGO_SETTINGS_MODULE` in the `postactivate` script.

```
#!/bin/sh
# This hook is sourced after this virtualenv is activated

# Set PYTHONPATH to isolate the virtualenv so that only modules installed
# in the virtualenv are available
export PYTHONPATH="/home/me/path/to/your/project_root:$VIRTUAL_ENV/lib/python3.4"

# Set DJANGO_SETTINGS_MODULE if you don't use the default `myproject.settings`
# or if you use `django-admin` rather than `manage.py`
export DJANGO_SETTINGS_MODULE="myproject.settings.dev"
```

Set your Project Path

It is often also helpful to set your project path inside a special `.project` file located in your base `<env-folder>`. When doing this, everytime you activate your virtual environment, it will change the active directory to the specified path.

Create a new file called `<env-folder>/.project`. The contents of the file should ONLY be the path of the project directory.

```
/path/to/project/directory
```

Now, initiate your virtual environment (either using `source <env-folder>/bin/activate` or `workon my_virtualenv`) and your terminal will change directories to `/path/to/project/directory`.

Single File Hello World Example

This example shows you a minimal way to create a Hello World page in Django. This will help you realize that the `django-admin startproject example` command basically creates a bunch of folders and files and that you don't necessarily need that structure to run your project.

1. Create a file called `file.py`.
2. Copy and paste the following code in that file.

```
import sys

from django.conf import settings
```

```
settings.configure(
    DEBUG=True,
    SECRET_KEY='thisisthesecretkey',
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)

from django.conf.urls import url
from django.http import HttpResponse

# Your code goes below this line.

def index(request):
    return HttpResponse('Hello, World!')

urlpatterns = [
    url(r'^$', index),
]

# Your code goes above this line

if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

3. Go to the terminal and run the file with this command `python file.py runserver`.

4. Open your browser and go to 127.0.0.1:8000.

Deployment friendly Project with Docker support.

The default Django project template is fine but once you get to deploy your code and for example devops put their hands on the project things get messy. What you can do is separate your source code from the rest that is required to be in your repository.

> You can find a usable Django project template on GitHub.

Project Structure

```
PROJECT_ROOT
├── devel.dockerfile
├── docker-compose.yml
├── nginx
│   └── project_name.conf
├── README.md
├── setup.py
└── src
    ├── manage.py
    └── project_name
        ├── __init__.py
```

```
        └── service
            ├── __init__.py
            ├── settings
            │   ├── common.py
            │   ├── development.py
            │   ├── __init__.py
            │   └── staging.py
            ├── urls.py
            └── wsgi.py
```

I like to keep the `service` directory named `service` for every project thanks to that I can use the same `Dockerfile` across all my projects. The split of requirements and settings are already well documented here:
Using multiple requirements files
Using multiple settings

Dockerfile

With the assumption that only developers make use of Docker (not every dev ops trust it these days). This could be a dev environment `devel.dockerfile`:

```
FROM python:2.7
ENV PYTHONUNBUFFERED 1

RUN mkdir /run/service
ADD . /run/service
WORKDIR /run/service

RUN pip install -U pip
RUN pip install -I -e .[develop] --process-dependency-links

WORKDIR /run/service/src
ENTRYPOINT ["python", "manage.py"]
CMD ["runserver", "0.0.0.0:8000"]
```

Adding only requirements will leverage Docker cache while building - you only need to rebuild on requirements change.

Compose

Docker compose comes in handy - especially when you have multiple services to run locally.
`docker-compose.yml`:

```
version: '2'
services:
  web:
    build:
      context: .
      dockerfile: devel.dockerfile
    volumes:
      - "./src/{{ project_name }}:/run/service/src/{{ project_name }}"
```

```
      - "./media:/run/service/media"
    ports:
      - "8000:8000"
    depends_on:
      - db
  db:
    image: mysql:5.6
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE={{ project_name }}
  nginx:
    image: nginx
    ports:
      - "80:80"
    volumes:
      - "./nginx:/etc/nginx/conf.d"
      - "./media:/var/media"
    depends_on:
      - web
```

Nginx

Your development environment should be as close to the prod environment as possible so I like using Nginx from the start. Here is an example nginx configuration file:

```
server {
    listen   80;
    client_max_body_size 4G;
    keepalive_timeout 5;

    location /media/ {
        autoindex on;
        alias /var/media/;
    }

    location / {
        proxy_pass_header Server;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Scheme $scheme;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Ssl on;
        proxy_connect_timeout 600;
        proxy_read_timeout 600;
        proxy_pass http://web:8000/;
    }
}
```

Usage

```
$ cd PROJECT_ROOT
$ docker-compose build web  # build the image - first-time and after requirements change
$ docker-compose up  # to run the project
```

```
$ docker-compose run --rm --service-ports --no-deps  # to run the project - and be able to use
PDB
$ docker-compose run --rm --no-deps <management_command>  # to use other than runserver
commands, like makemigrations
$ docker exec -ti web bash  # For accessing django container shell, using it you will be
inside /run/service directory, where you can run ./manage shell, or other stuff
$ docker-compose start  # Starting docker containers
$ docker-compose stop  # Stopping docker containers
```

Read Getting started with Django online: https://riptutorial.com/django/topic/200/getting-started-with-django

Chapter 2: Administration

Examples

Change list

Let's say you have a simple `myblog` app with the following model:

```
from django.conf import settings
from django.utils import timezone

class Article(models.Model):
    title = models.CharField(max_length=70)
    slug = models.SlugField(max_length=70, unique=True)
    author = models.ForeignKey(settings.AUTH_USER_MODEL, models.PROTECT)
    date_published = models.DateTimeField(default=timezone.now)
    is_draft = models.BooleanField(default=True)
    content = models.TextField()
```

Django Admin's "change list" is the page that lists all objects of a given model.

```
from django.contrib import admin
from myblog.models import Article

@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    pass
```

By default, it will use the `__str__()` method (or `__unicode__()` if you on python2) of your model to display the object "name". This means that if you didn't override it, you will see a list of articles, all named "Article object". To change this behavior, you can set the `__str__()` method:

```
class Article(models.Model):
    def __str__(self):
        return self.title
```

Now, all your articles should have a different name, and more explicit than "Article object".

However you may want to display other data in this list. For this, use `list_display`:

```
@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    list_display = ['__str__', 'author', 'date_published', 'is_draft']
```

`list_display` is not limited to the model fields and properties. it can also be a method of your `ModelAdmin`:

```
from django.forms.utils import flatatt
from django.urls import reverse
from django.utils.html import format_html
```

```
@admin.register(Article)
class ArticleAdmin(admin.ModelAdmin):
    list_display = ['title', 'author_link', 'date_published', 'is_draft']

    def author_link(self, obj):
        author = obj.author
        opts = author._meta
        route = '{}_{}_change'.format(opts.app_label, opts.model_name)
        author_edit_url = reverse(route, args=[author.pk])
        return format_html(
            '<a{}>{}</a>', flatatt({'href': author_edit_url}), author.first_name)

    # Set the column name in the change list
    author_link.short_description = "Author"
    # Set the field to use when ordering using this column
    author_link.admin_order_field = 'author__firstname'
```

Additional CSS styles and JS scripts for admin page

Suppose you have a simple `Customer` model:

```
class Customer(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    is_premium = models.BooleanField(default=False)
```

You register it in the Django admin and add search field by `first_name` and `last_name`:

```
@admin.register(Customer)
class CustomerAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name', 'is_premium']
    search_fields = ['first_name', 'last_name']
```

After you do this, the search fields appear in the admin list page with the default placeholder: "*keyword*". But what if you want to change that placeholder to "*Search by name*"?

You can do this by passing custom Javascript file into admin `Media`:

```
@admin.register(Customer)
class CustomerAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name', 'is_premium']
    search_fields = ['first_name', 'last_name']

    class Media:
        #this path may be any you want,
        #just put it in your static folder
        js = ('js/admin/placeholder.js', )
```

You can use browser debug toolbar to find what id or class Django set to this searchbar and then write your js code:

```
$(function () {
   $('#searchbar').attr('placeholder', 'Search by name')
```

```
})
```

Also `Media` class allows you to add css files with dictionary object:

```
class Media:
    css = {
        'all': ('css/admin/styles.css',)
         }
```

For example we need to display each element of `first_name` column in specific color.
By default Django create table column for every item in `list_display`, all `<td>` tags will have css class like `field-'list_display_name'`, in our case it will `field_first_name`

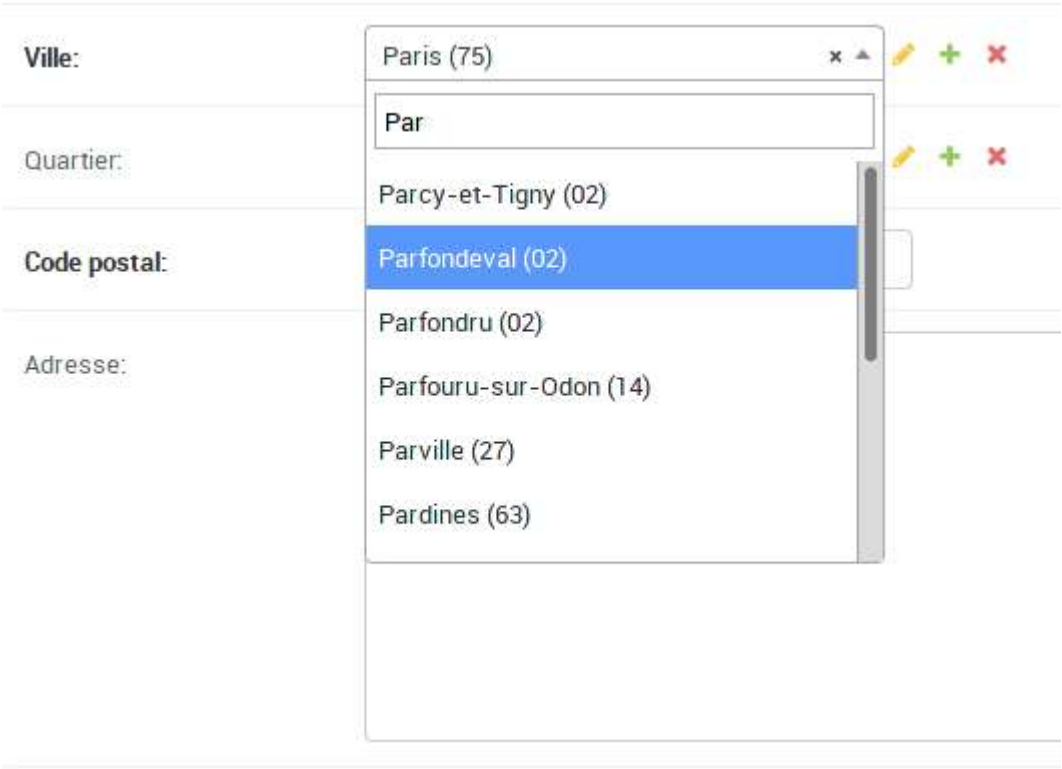
```
.field_first_name {
     background-color: #e6f2ff;
 }
```

If you want to customize other behavior by adding JS or some css styles, you can always check id`s and classes of elements in the browser debug tool.

Dealing with foreign keys referencing large tables

By default, Django renders `ForeignKey` fields as a `<select>` input. This can cause pages to be **load really slowly** if you have thousands or tens of thousand entries in the referenced table. And even if you have only hundreds of entries, it is quite uncomfortable to look for a particular entry among all.

A very handy external module for this is django-autocomplete-light (DAL). This enables to use autocomplete fields instead of `<select>` fields.



views.py

```
from dal import autocomplete

class CityAutocomp(autocomplete.Select2QuerySetView):
    def get_queryset(self):
        qs = City.objects.all()
        if self.q:
            qs = qs.filter(name__istartswith=self.q)
        return qs
```

urls.py

```
urlpatterns = [
    url(r'^city-autocomp/$', CityAutocomp.as_view(), name='city-autocomp'),
]
```

forms.py

```
from dal import autocomplete

class PlaceForm(forms.ModelForm):
    city = forms.ModelChoiceField(
        queryset=City.objects.all(),
        widget=autocomplete.ModelSelect2(url='city-autocomp')
    )

    class Meta:
        model = Place
```

```
        fields = ['__all__']
```

admin.py

```
@admin.register(Place)
class PlaceAdmin(admin.ModelAdmin):
    form = PlaceForm
```

Read Administration online: https://riptutorial.com/django/topic/1219/administration

Chapter 3: ArrayField - a PostgreSQL-specific field

Syntax

- from django.contrib.postgres.fields import ArrayField
- class ArrayField(base_field, size=None, **options)
- FooModel.objects.filter(array_field_name__contains=[objects, to, check])
- FooModel.objects.filter(array_field_name__contained_by=[objects, to, check])

Remarks

Note that although the `size` parameter is passed to PostgreSQL, PostgreSQL will not enforce it.

When using `ArrayField`s one should keep in mind this word of warning from the Postgresql arrays documentation.

> Tip: Arrays are not sets; searching for specific array elements can be a sign of database misdesign. Consider using a separate table with a row for each item that would be an array element. This will be easier to search, and is likely to scale better for a large number of elements.

Examples

A basic ArrayField

To create a PostgreSQL ArrayField, we should give ArrayField the type of data we want it to store as a field as its first argument. Since we'll be storing book ratings, we will use `FloatField`.

```
from django.db import models, FloatField
from django.contrib.postgres.fields import ArrayField

class Book(models.Model):
    ratings = ArrayField(FloatField())
```

Specifying the maximum size of an ArrayField

```
from django.db import models, IntegerField
from django.contrib.postgres.fields import ArrayField

class IceCream(models.Model):
    scoops = ArrayField(IntegerField()  # we'll use numbers to ID the scoops
                , size=6)  # our parlor only lets you have 6 scoops
```

When you use the size parameter, it's passed through to postgresql, which accepts it and then

ignores it! Thus it's quite possible to add 7 integers to the `scoops` field above using the postgresql console.

Querying for membership of ArrayField with contains

This query returns all cones with a chocolate scoop and a vanilla scoop.

```
VANILLA, CHOCOLATE, MINT, STRAWBERRY = 1, 2, 3, 4  # constants for flavors
choco_vanilla_cones = IceCream.objects.filter(scoops__contains=[CHOCOLATE, VANILLA])
```

Don't forget to import the `IceCream` model from your `models.py` file.

Also bear in mind that django will not create an index for `ArrayField`s. If you are going to search them, you are going to need an index and it will need to be manually created with a call to RunSQL in your migrations file.

Nesting ArrayFields

You can nest `ArrayField`s by passing another `ArrayField` as it's `base_field`.

```
from django.db import models, IntegerField
from django.contrib.postgres.fields import ArrayField

class SudokuBoard(models.Model):
    numbers = ArrayField(
        ArrayField(
            models.IntegerField(),
            size=9,
        ),
        size=9,
    )
```

Querying for all models who contain any item in a list with contained_by

This query returns all cones with either a mint scoop or a vanilla scoop.

```
minty_vanilla_cones = IceCream.objects.filter(scoops__contained_by=[MINT, VANILLA])
```

Read ArrayField - a PostgreSQL-specific field online:
https://riptutorial.com/django/topic/1693/arrayfield---a-postgresql-specific-field

Chapter 4: Async Tasks (Celery)

Remarks

Celery is a task queue which can run background or scheduled jobs and integrates with Django pretty well. Celery requires something known as **message broker** to pass messages from invocation to the workers. This message broker can be redis, rabbitmq or even Django ORM/db although that is not a recommended approach.

Before you get started with the example, You will have to configure celery. To configure celery, create a `celery_config.py` file in the main app, parallel to the `settings.py` file.

```
from __future__ import absolute_import
import os
from celery import Celery
from django.conf import settings

# broker url
BROKER_URL = 'redis://localhost:6379/0'

# Indicate Celery to use the default Django settings module
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'config.settings')

app = Celery('config')
app.config_from_object('django.conf:settings')
# if you do not need to keep track of results, this can be turned off
app.conf.update(
    CELERY_RESULT_BACKEND=BROKER_URL,
)

# This line will tell Celery to autodiscover all your tasks.py that are in your app folders
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

And in the main app's `__init__.py` file import the celery app. like this

```
# -*- coding: utf-8 -*-
# Not required for Python 3.
from __future__ import absolute_import

from .celery_config import app as celery_app  # noqa
```

To run celery worker, use this command at the level where manage.py is.

```
# pros is your django project,
celery -A proj worker -l info
```

Examples

Simple example to add 2 numbers

To get started:

1. Install celery `pip install celery`
2. configure celery (head to the remarks section)

```
from __future__ import absolute_import, unicode_literals

from celery.decorators import task


@task
def add_number(x, y):
    return x + y
```

You can run this asynchronously by using the `.delay()` method.

`add_number.delay(5, 10)`, where 5 and 10 are the arguments for the function `add_number`

To check if the async function has finished the operation, you can use the `.ready()` function on the async object returned by the `delay` method.

To fetch the result of the computation, you can use the `.result` attribute on the async object.

Example

```
async_result_object = add_number.delay(5, 10)
if async_result_object.ready():
    print(async_result_object.result)
```

Read Async Tasks (Celery) online: https://riptutorial.com/django/topic/5481/async-tasks--celery-

Chapter 5: Authentication Backends

Examples

Email Authentication Backend

Django's default authentication works on `username` and `password` fields. Email authentication backend will authenticate users based on `email` and `password`.

```
from django.contrib.auth import get_user_model

class EmailBackend(object):
    """
    Custom Email Backend to perform authentication via email
    """
    def authenticate(self, username=None, password=None):
        user_model = get_user_model()
        try:
            user = user_model.objects.get(email=username)
            if user.check_password(password): # check valid password
                return user # return user to be authenticated
        except user_model.DoesNotExist: # no matching user exists
            return None

    def get_user(self, user_id):
        user_model = get_user_model()
        try:
            return user_model.objects.get(pk=user_id)
        except user_model.DoesNotExist:
            return None
```

Add this authentication backend to the `AUTHENTICATION_BACKENDS` setting.

```
# settings.py
AUTHENTICATION_BACKENDS = (
    'my_app.backends.EmailBackend',
    ...
    )
```

Read Authentication Backends online: https://riptutorial.com/django/topic/1282/authentication-backends

Chapter 6: Class based views

Remarks

When using CBV we often need to know exactly what methods we can overwrite for each generic class. This page of the django documentation lists all the generic classes with all of their methods flattened and the class attributes we can use.

In addition, Classy Class Based View website provides the same information with a nice interactive interface.

Examples

Class Based Views

Class based views let you focus on what make your views special.

A static about page might have nothing special, except the template used. Use a TemplateView! All you have to do is set a template name. Job done. Next.

views.py

```
from django.views.generic import TemplateView


class AboutView(TemplateView):
    template_name = "about.html"
```

urls.py

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url('^about/', views.AboutView.as_view(), name='about'),
]
```

Notice how we don't use directly `AboutView` in the url. That's because a callable is expected and that's exactly what `as_view()` return.

Context data

Sometimes, your template need a bit more of information. For example, we would like to have the user in the header of the page, with a link to their profile next to the logout link. In these cases, use

the `get_context_data` method.

views.py

```
class BookView(DetailView):
    template_name = "book.html"

    def get_context_data(self, **kwargs)
        """ get_context_data let you fill the template context """
        context = super(BookView, self).get_context_data(**kwargs)
        # Get Related publishers
        context['publishers'] = self.object.publishers.filter(is_active=True)
        return context
```

You need to call get_context_data method on the super class and it will return the default context instance. Any item that you add to this dictionary will be available to the template.

book.html

```
<h3>Active publishers</h3>
<ul>
  {% for publisher in publishers %}
    <li>{{ publisher.name }}</li>
  {% endfor %}
</ul>
```

List and Details views

Template views are fine for static page and you could use them for everything with `get_context_data` but it would be barely better than using function as views.

Enter ListView and DetailView

app/models.py

```
from django.db import models

class Pokemon(models.Model):
    name = models.CharField(max_length=24)
    species = models.CharField(max_length=48)
    slug = models.CharField(max_length=48)
```

app/views.py

```
from django.views.generic import ListView, DetailView
```

```
from .models import Pokemon


class PokedexView(ListView):
    """ Provide a list of Pokemon objects """
    model = Pokemon
    paginate_by = 25

class PokemonView(DetailView):
    model = Pokemon
```

That's all you need to generate a view listing all your objects of a models and views of singular item. The list is even paginated. You can provide `template_name` if you want something specific. By default, it's generated from the model name.

app/templates/app/pokemon_list.html

```
<!DOCTYPE html>
<title>Pokedex</title>
<ul>{% for pokemon in pokemon_list %}
    <li><a href="{% url "app:pokemon" pokemon.pk %}">{{ pokemon.name }}</a>
        &ndash; {{ pokemon.species }}
</ul>
```

The context is populated with the list of object under two name, `object_list` and a second one build from the model name, here `pokemon_list`. If you have paginated the list, you have to take care of next and previous link too. The Paginator object can help with that, it's available in the context data too.

app/templates/app/pokemon_detail.html

```
<!DOCTYPE html>
<title>Pokemon {{ pokemon.name }}</title>
<h1>{{ pokemon.name }}</h1>
<h2>{{ pokemon.species }} </h2>
```

As before, the context is populated with your model object under the name `object` and `pokemon`, the second one being derived from the model name.

app/urls.py

```
from django.conf.urls import url
from . import views

app_name = 'app'
urlpatterns = [
    url(r'^pokemon/$', views.PokedexView.as_view(), name='pokedex'),
    url(r'^pokemon/(?P<pk>\d+)/$', views.PokemonView.as_view(), name='pokemon'),
```

```
]
```

In this snippet, the url for the detail view is built using the primary key. It's also possible to use a slug as argument. This gives a nicer looking url that's easier to remember. However it requires the presence of a field named slug in your model.

```
url(r'^pokemon/(?P<slug>[A-Za-z0-9_-]+)/$', views.PokemonView.as_view(), name='pokemon'),
```

If a field called `slug` is not present, you can use the `slug_field` setting in `DetailView` to point to a different field.

For pagination, use a page get parameters or put a page directly in the url.

Form and object creation

Writing a view to create object can be quite boring. You have to display a form, you have to validate it, you have to save the item or return the form with an error. Unless you use one of the generic editing views.

app/views.py

```
from django.core.urlresolvers import reverse_lazy
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from .models import Pokemon


class PokemonCreate(CreateView):
    model = Pokemon
    fields = ['name', 'species']


class PokemonUpdate(UpdateView):
    model = Pokemon
    fields = ['name', 'species']


class PokemonDelete(DeleteView):
    model = Pokemon
    success_url = reverse_lazy('pokedex')
```

`CreateView` and `UpdateView` have two required attribute, `model` and `fields`. By default, both use a template name based on the model name suffixed by '_form'. You can change only the suffix with the attribute template_name_suffix. The DeleteView show a confirmation message before deleting the object.

Both `UpdateView` and `DeleteView` need to fetch on object. They use the same method as `DetailView`, extracting variable from the url and matching the object fields.

app/templates/app/pokemon_form.html (extract)

```
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save" />
</form>
```

`form` contains the form with all needed fields. Here, it will be displayed with a paragraph for each field because of `as_p`.

app/templates/app/pokemon_confirm_delete.htm (extract)

```
<form action="" method="post">
    {% csrf_token %}
    <p>Are you sure you want to delete "{{ object }}"?</p>
    <input type="submit" value="Confirm" />
</form>
```

The `csrf_token` tag is required because of django protection against request forgery. The attribute action is left empty as the url displaying the form is the same as the one handling the deletion/save.

Two issues remain with the model, if using the same as with the list and detail exemple. First, create and update will complain about a missing redirection url. That can be solved by adding a `get_absolute_url` to the pokemon model. Second issue is the deletion confirmation not displaying meaningful information. To solve this, the easiest solution is to add a string representation.

app/models.py

```
from django.db import models
from django.urls import reverse
from django.utils.encoding import python_2_unicode_compatible


@python_2_unicode_compatible
class Pokemon(models.Model):
    name = models.CharField(max_length=24)
    species = models.CharField(max_length=48)

    def get_absolute_url(self):
        return reverse('app:pokemon', kwargs={'pk':self.pk})

    def __str__(self):
```

```
        return self.name
```

The class decorator will make sure everything work smoothly under python 2.

Minimal example

views.py:

```
from django.http import HttpResponse
from django.views.generic import View

class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
```

urls.py:

```
from django.conf.urls import url
from myapp.views import MyView

urlpatterns = [
    url(r'^about/$', MyView.as_view()),
]
```

Learn more on Django documentation »

Django Class Based Views: Example of CreateView

With the Class Based generic Views, it is very simple and easy to create the CRUD views from our models. Often, the built in Django admin is not enough or not preferred and we need to roll our own CRUD views. The CBVs can be very handy in such cases.

The `CreateView` class needs 3 things - a model, the fields to use and success url.

Example:

```
from django.views.generic import CreateView
from .models import Campaign

class CampaignCreateView(CreateView):
    model = Campaign
    fields = ('title', 'description')

    success_url = "/campaigns/list"
```

Once the creation success, the user is redirected to `success_url`. We can also define a method `get_success_url` instead and use `reverse` or `reverse_lazy` to get the success url.

Now, we need to create a template for this view. The template should be named in the format `<app name>/<model name>_form.html`. The model name must be in lower caps. For example, if my app

name is `dashboard`, then for the above create view, I need to create a template named `dashboard/campaign_form.html`.

In the template, a `form` variable would contain the form. Here's a sample code for the template:

```
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save" />
</form>
```

Now it's time to add the view to our url patterns.

```
url('^campaign/new/$', CampaignCreateView.as_view(), name='campaign_new'),
```

If we visit the URL, we should see a form with the fields we chose. When we submit, it will try to create a new instance of the model with the data and save it. On success, the user will be redirected to the success url. On errors, the form will be displayed again with the error messages.

One View, Multiple Forms

Here is a quick example of using multiple forms in one Django view.

```
from django.contrib import messages
from django.views.generic import TemplateView

from .forms import AddPostForm, AddCommentForm
from .models import Comment

class AddCommentView(TemplateView):

    post_form_class = AddPostForm
    comment_form_class = AddCommentForm
    template_name = 'blog/post.html'

    def post(self, request):
        post_data = request.POST or None
        post_form = self.post_form_class(post_data, prefix='post')
        comment_form = self.comment_form_class(post_data, prefix='comment')

        context = self.get_context_data(post_form=post_form,
                                        comment_form=comment_form)

        if post_form.is_valid():
            self.form_save(post_form)
        if comment_form.is_valid():
            self.form_save(comment_form)

        return self.render_to_response(context)

    def form_save(self, form):
        obj = form.save()
        messages.success(self.request, "{} saved successfully".format(obj))
        return obj
```

```
    def get(self, request, *args, **kwargs):
        return self.post(request, *args, **kwargs)
```

Read Class based views online: https://riptutorial.com/django/topic/1220/class-based-views

Chapter 7: Context Processors

Remarks

Use context processors to add variables that are accessible anywhere in your templates.

Specify a function, or functions that return `dict`s of the variables you want, then add those functions to `TEMPLATE_CONTEXT_PROCESSORS`.

Examples

Use a context processor to access settings.DEBUG in templates

in `myapp/context_processors.py`:

```
from django.conf import settings

def debug(request):
  return {'DEBUG': settings.DEBUG}
```

in `settings.py`:

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'myapp.context_processors.debug',
            ],
        },
    },
]
```

or, for versions < 1.9:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'myapp.context_processors.debug',
)
```

Then in my templates, simply:

```
 {% if DEBUG %} .header { background:#f00; } {% endif %}
 {{ DEBUG }}
```

Using a context processor to access your most recent blog entries in all templates

Assuming you have a model called `Post` defined in your `models.py` file that contains blog posts, and has a `date_published` field.

Step 1: Write the context processor

Create (or add to) a file in your app directory called `context_processors.py`:

```
from myapp.models import Post

def recent_blog_posts(request):
    return {'recent_posts':Post.objects.order_by('-date_published')[0:3],}  # Can change
numbers for more/fewer posts
```

Step 2: Add the context processor to your settings file

Make sure that you add your new context processor to your `settings.py` file in the `TEMPLATES` variable:

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'myapp.context_processors.recent_blog_posts',
            ],
        },
    },
]
```

(In Django versions before 1.9, this was set directly in `settings.py` using a `TEMPLATE_CONTEXT_PROCESSORS` variable.)

Step 3: Use the context processor in your templates

No need to pass recent blog entries through individual views anymore! Just use `recent_blog_posts` in any template.

E.g., in `home.html` you could create a sidebar with links to recent posts:

```
<div class="blog_post_sidebar">
    {% for post in recent_blog_posts %}
        <div class="post">
            <a href="{{post.get_absolute_url}}">{{post.title}}</a>
        </div>
    {% endfor %}
</div>
```

Or in `blog.html` you could create a more detailed display of each post:

```
<div class="content">
    {% for post in recent_blog_posts %}
        <div class="post_detail">
            <h2>{{post.title}}</h2>
            <p>Published on {{post.date_published}}</p>
            <p class="author">Written by: {{post.author}}</p>
            <p><a href="{{post.get_absolute_url}}">Permalink</a></p>
            <p class="post_body">{{post.body}}</p>
        </div>
    {% endfor %}
</div>
```

Extending your templates

Context processor to determine the template based on group membership(or any query/logic). This allows our public/regular users to get one template and our special group to get a different one.

myapp/context_processors.py

```
def template_selection(request):
    site_template = 'template_public.html'
    if request.user.is_authenticated():
        if request.user.groups.filter(name="some_group_name").exists():
            site_template = 'template_new.html'

    return {
        'site_template': site_template,
    }
```

Add the context processor to your settings.

In your templates, use the variable defined in the context processor.

```
{% extends site_template %}
```

Read Context Processors online: https://riptutorial.com/django/topic/491/context-processors

Chapter 8: Continuous Integration With Jenkins

Examples

Jenkins 2.0+ Pipeline Script

Modern versions of Jenkins (version 2.x) come with a "Build Pipeline Plugin" that can be used to orchestrate complex CI tasks without creating a multitude of interconnected jobs, and allow you to easily version-control your build / test configuration.

You may install this manually in a "Pipeline" type job, or, if your project is hosted on Github, you may use the "GitHub Organization Folder Plugin" to automatically set up jobs for you.

Here's a simple configuration for Django sites that require only the site's specified python modules to be installed.

```
#!/usr/bin/groovy

node {
  // If you are having issues with your project not getting updated,
  // try uncommenting the following lines.
  //stage 'Checkout'
  //checkout scm
  //sh 'git submodule update --init --recursive'

  stage 'Update Python Modules'
  // Create a virtualenv in this folder, and install or upgrade packages
  // specified in requirements.txt; https://pip.readthedocs.io/en/1.1/requirements.html
  sh 'virtualenv env && source env/bin/activate && pip install --upgrade -r requirements.txt'

  stage 'Test'
  // Invoke Django's tests
  sh 'source env/bin/activate && python ./manage.py runtests'
}
```

Jenkins 2.0+ Pipeline Script, Docker Containers

Here is an example of a pipeline script that builds a Docker container, then runs the tests inside of it. The entrypoint is assumed to be either `manage.py` or `invoke`/`fabric` with a `runtests` command available.

```
#!/usr/bin/groovy

node {
  stage 'Checkout'
  checkout scm
  sh 'git submodule update --init --recursive'

  imageName = 'mycontainer:build'
```

```
  remotes = [
    'dockerhub-account',
  ]

  stage 'Build'
  def djangoImage = docker.build imageName

  stage 'Run Tests'
  djangoImage.run('', 'runtests')

  stage 'Push'
  for (int i = 0; i < remotes.size(); i++) {
      sh "docker tag ${imageName} ${remotes[i]}/${imageName}"
      sh "docker push ${remotes[i]}/${imageName}"
   }
}
```

Read Continuous Integration With Jenkins online:
https://riptutorial.com/django/topic/5873/continuous-integration-with-jenkins

Chapter 9: CRUD in Django

Examples

Simplest CRUD example

If you find these steps unfamiliar, consider starting here instead. Note these steps come from Stack Overflow Documentation.

```
django-admin startproject myproject
cd myproject
python manage.py startapp myapp
```

myproject/settings.py Install the app

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'myapp',
]
```

Create a file called `urls.py` within the **myapp** directory and updated it with the following view.

```
from django.conf.urls import url
from myapp import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    ]
```

Update the other `urls.py` file with the following content.

```
from django.conf.urls import url
from django.contrib import admin
from django.conf.urls import include
from myapp import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^myapp/', include('myapp.urls')),
    url(r'^admin/', admin.site.urls),
]
```

Create a folder named `templates` within the **myapp** directory. Then create a file named `index.html` inside of the **templates** directory. Fill it with the following content.

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    <h3>Add a Name</h3>
    <button>Create</button>
</body>
</html>
```

We also need a view to show **index.html** which we can create by editing the **views.py** file like so:

```
from django.shortcuts import render, redirect

# Create your views here.
def index(request):
    return render(request, 'index.html', {})
```

You now have the base that you're going to work off of. The next step is create a Model. This is the simplest example possible so in your **models.py** folder add the following code.

```
from __future__ import unicode_literals

from django.db import models

# Create your models here.
class Name(models.Model):
    name_value = models.CharField(max_length=100)

    def __str__(self): # if Python 2 use __unicode__
        return self.name_value
```

This creates a model of a Name object which we'll add to the database with the following commands from the command line.

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

You should see some operations performed by Django. These setup up the tables and create a superuser that can access the admin database from a Django powered admin view. Speaking of which, lets register our new model with the admin view. Go to **admin.py** and add the following code.

```
from django.contrib import admin
from myapp.models import Name
# Register your models here.

admin.site.register(Name)
```

Back at the command line you can now spin up the server with the `python manage.py runserver`

command. You should be able to visit http://localhost:8000/ and see your app. Please then navigate to http://localhost:8000/admin so that you can add a name to your project. Log in and add a Name under the MYAPP table, we kept it simple for the example so ensure it's less than 100 characters.

In order to access the name you need to display it somewhere. Edit the index function within **views.py** to get all of the Name objects out of the database.

```
from django.shortcuts import render, redirect
from myapp.models import Name

# Create your views here.
def index(request):
    names_from_db = Name.objects.all()
    context_dict = {'names_from_context': names_from_db}
    return render(request, 'index.html', context_dict)
```

Now edit the **index.html** file to the following.

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    {% if names_from_context %}
        <ul>
            {% for name in names_from_context %}
                <li>{{ name.name_value }}  <button>Delete</button>
<button>Update</button></li>
            {% endfor %}
        </ul>
    {% else %}
        <h3>Please go to the admin and add a Name under 'MYAPP'</h3>
    {% endif %}
    <h3>Add a Name</h3>
    <button>Create</button>
</body>
</html>
```

That demonstrates the Read in CRUD. Within the **myapp** directory create a forms.py file. Add the following code:

```
from django import forms
from myapp.models import Name

class NameForm(forms.ModelForm):
    name_value = forms.CharField(max_length=100, help_text = "Enter a name")

    class Meta:
        model = Name
        fields = ('name_value',)
```

Update the **index.html** in the following manner:

```
<!DOCTYPE html>
<html>
<head>
    <title>myapp</title>
</head>
<body>
    <h2>Simplest Crud Example</h2>
    <p>This shows a list of names and lets you Create, Update and Delete them.</p>
    {% if names_from_context %}
        <ul>
            {% for name in names_from_context %}
                <li>{{ name.name_value }}  <button>Delete</button>
<button>Update</button></li>
            {% endfor %}
        </ul>
    {% else %}
        <h3>Please go to the admin and add a Name under 'MYAPP'</h3>
    {% endif %}
    <h3>Add a Name</h3>
    <form id="name_form" method="post" action="/">
        {% csrf_token %}
        {% for field in form.visible_fields %}
            {{ field.errors }}
            {{ field.help_text }}
            {{ field }}
        {% endfor %}
        <input type="submit" name="submit" value="Create">
    </form>
</body>
</html>
```

Next update the **views.py** in the following manner:

```
from django.shortcuts import render, redirect
from myapp.models import Name
from myapp.forms import NameForm

# Create your views here.
def index(request):
    names_from_db = Name.objects.all()

    form = NameForm()

    context_dict = {'names_from_context': names_from_db, 'form': form}

    if request.method == 'POST':
        form = NameForm(request.POST)

        if form.is_valid():
            form.save(commit=True)
            return render(request, 'index.html', context_dict)
        else:
            print(form.errors)

    return render(request, 'index.html', context_dict)
```

Restart your server and you should now have a working version of the app with the C in create

completed.

TODO add update and delete

Read CRUD in Django online: https://riptutorial.com/django/topic/7317/crud-in-django

Chapter 10: Custom Managers and Querysets

Examples

Defining a basic manager using Querysets and `as_manager` method

Django manger is an interface through which the django model queries the database. The `objects` field used in most django queries is actually the default manager created for us by django (this is only created if we don't define custom managers).

Why would we define a custom manager/queryset?

To avoid writing common queries all over our codebase and instead referring them using an easier to remember abstraction. Example: Decide for yourself which version is more readable :

- Only get all the active users : `User.objects.filter(is_active=True)` vs `User.manager.active()`
- Get all active dermatologists on our plaform : `User.objects.filter(is_active=True).filter(is_doctor=True).filter(specialization='Dermatology')` vs `User.manager.doctors.with_specialization('Dermatology')`

Another benefit is that if tomorrow we decide all `psychologists` are also `dermatologists`, we can easily modify the query in our Manager and be done with it.

Below is an example of creating a custom `Manager` defined by creating a `QuerySet` and using the `as_manager` method.

```
from django.db.models.query import QuerySet

class ProfileQuerySet(QuerySet):
    def doctors(self):
        return self.filter(user_type="Doctor", user__is_active=True)

    def with_specializations(self, specialization):
        return self.filter(specializations=specialization)

    def users(self):
        return self.filter(user_type="Customer", user__is_active=True)

ProfileManager = ProfileQuerySet.as_manager
```

We will add it to our model as below:

```
class Profile(models.Model):
    ...
    manager = ProfileManager()
```

NOTE : Once we've defined a `manager` on our model, `objects` won't be defined for the model anymore.

select_related for all queries

Model with ForeignKey

We will work with these models :

```
from django.db import models

class Book(models.Model):
 name= models.CharField(max_length=50)
 author = models.ForeignKey(Author)

class Author(models.Model):
 name = models.CharField(max_length=50)
```

Suppose we often (always) access `book.author.name`

In view

We could use the following, each time,

```
books = Book.objects.select_related('author').all()
```

But this is not DRY.

Custom Manager

```
class BookManager(models.Manager):

    def get_queryset(self):
        qs = super().get_queryset()
        return qs.select_related('author')

class Book(models.Model):
    ...
    objects = BookManager()
```

Note : the call to `super` must be changed for python 2.x

Now all we have to use in views is

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

and no additional queries will be made in template/view.

Define custom managers

Very often it happens to deal with models which have something like a `published` field. Such kind of fields are almost always used when retrieving objects, so that you will find yourself to write something like:

```
my_news = News.objects.filter(published=True)
```

too many times. You can use custom managers to deal with these situations, so that you can then write something like:

```
my_news = News.objects.published()
```

which is nicer and more easy to read by other developers too.

Create a file `managers.py` in your app directory, and define a new `models.Manager` class:

```
from django.db import models


class NewsManager(models.Manager):

    def published(self, **kwargs):
        # the method accepts **kwargs, so that it is possible to filter
        # published news
        # i.e: News.objects.published(insertion_date__gte=datetime.now)
        return self.filter(published=True, **kwargs)
```

use this class by redefining the `objects` property in the model class:

```
from django.db import models

# import the created manager
from .managers import NewsManager

class News(models.Model):
    """ News model
    """
    insertion_date = models.DateTimeField('insertion date', auto_now_add=True)
    title = models.CharField('title', max_length=255)
    # some other fields here
    published = models.BooleanField('published')

    # assign the manager class to the objects property
    objects = NewsManager()
```

Now you can get your published news simply this way:

```
my_news = News.objects.published()
```

and you can also perform more filtering:

```
my_news = News.objects.published(title__icontains='meow')
```

Read Custom Managers and Querysets online: https://riptutorial.com/django/topic/1400/custom-managers-and-querysets

Chapter 11: Database Routers

Examples

Adding a Database Routing file

To use multiple databases in Django, just specify each one in `settings.py`:

```
DATABASES = {
    'default': {
        'NAME': 'app_data',
        'ENGINE': 'django.db.backends.postgresql',
        'USER': 'django_db_user',
        'PASSWORD': os.environ['LOCAL_DB_PASSWORD']
    },
    'users': {
        'NAME': 'remote_data',
        'ENGINE': 'django.db.backends.mysql',
        'HOST': 'remote.host.db',
        'USER': 'remote_user',
        'PASSWORD': os.environ['REMOTE_DB_PASSWORD']
    }
}
```

Use a `dbrouters.py` file to specify which models should operate on which databases for each class of database operation, e.g. for remote data stored in `remote_data`, you might want the following:

```
class DbRouter(object):
    """
    A router to control all database operations on models in the
    auth application.
    """
    def db_for_read(self, model, **hints):
        """
        Attempts to read remote models go to remote database.
        """
        if model._meta.app_label == 'remote':
            return 'remote_data'
        return 'app_data'

    def db_for_write(self, model, **hints):
        """
        Attempts to write remote models go to the remote database.
        """
        if model._meta.app_label == 'remote':
            return 'remote_data'
        return 'app_data'

    def allow_relation(self, obj1, obj2, **hints):
        """
        Do not allow relations involving the remote database
        """
        if obj1._meta.app_label == 'remote' or \
           obj2._meta.app_label == 'remote':
           return False
```

```
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        """
        Do not allow migrations on the remote database
        """
        if model._meta.app_label == 'remote':
            return False
        return True
```

Finally, add your `dbrouter.py` to `settings.py`:

```
DATABASE_ROUTERS = ['path.to.DbRouter', ]
```

Specifying different databases in code

The normal `obj.save()` method will use the default database, or if a database router is used, it will use the database as specified in `db_for_write`. You can override it by using:

```
obj.save(using='other_db')
obj.delete(using='other_db')
```

Similarly, for reading:

```
MyModel.objects.using('other_db').all()
```

Read Database Routers online: https://riptutorial.com/django/topic/3395/database-routers

Chapter 12: Database Setup

Examples

MySQL / MariaDB

Django supports MySQL 5.5 and higher.

Make sure to have some packages installed:

```
$ sudo apt-get install mysql-server libmysqlclient-dev
$ sudo apt-get install python-dev python-pip              # for python 2
$ sudo apt-get install python3-dev python3-pip            # for python 3
```

As well as one of the Python MySQL drivers (`mysqlclient` beeing the recommended choice for Django):

```
$ pip install mysqlclient    # python 2 and 3
$ pip install MySQL-python   # python 2
$ pip install pymysql        # python 2 and 3
```

> The database encoding can not be set by Django, but needs to be configured on the database level. Look for `default-character-set` in my.cnf (or `/etc/mysql/mariadb.conf/*.cnf`) and set the encoding:

```
[mysql]
#default-character-set = latin1    #default on some systems.
#default-character-set = utf8mb4   #default on some systems.
default-character-set = utf8

...
[mysqld]
#character-set-server  = utf8mb4
#collation-server      = utf8mb4_general_ci
character-set-server  = utf8
collation-server      = utf8_general_ci
```

Database configuration for MySQL or MariaDB

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'DB_NAME',
        'USER': 'DB_USER',
        'PASSWORD': 'DB_PASSWORD',
        'HOST': 'localhost',   # Or an IP Address that your database is hosted on
        'PORT': '3306',
        #optional:
        'OPTIONS': {
```

```
            'charset' : 'utf8',
            'use_unicode' : True,
             'init_command': 'SET '
                'storage_engine=INNODB,'
                'character_set_connection=utf8,'
                'collation_connection=utf8_bin'
                #'sql_mode=STRICT_TRANS_TABLES,'    # see note below
                #'SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED',
        },
        'TEST_CHARSET': 'utf8',
        'TEST_COLLATION': 'utf8_general_ci',
    }
}
```

If you are using Oracle's MySQL connector your `ENGINE` line should look like this:

```
'ENGINE': 'mysql.connector.django',
```

When you create a database, make sure that to specify the encoding and collation:

```
CREATE DATABASE mydatabase CHARACTER SET utf8 COLLATE utf8_bin
```

> From MySQL 5.7 onwards and on fresh installs of MySQL 5.6, the default value of the sql_mode option contains **STRICT_TRANS_TABLES**. That option escalates warnings into errors when data is truncated upon insertion. Django highly recommends activating a *strict mode* for MySQL to prevent data loss (either STRICT_TRANS_TABLES or STRICT_ALL_TABLES). To enable add to */etc/my.cnf* `sql-mode = STRICT_TRANS_TABLES`

PostgreSQL

Make sure to have some packages installed:

```
sudo apt-get install libpq-dev
pip install psycopg2
```

Database settings for PostgreSQL:

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'myprojectDB',
        'USER': 'myprojectuser',
        'PASSWORD': 'password',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

In older versions you can also use the alias `django.db.backends.postgresql_psycopg2`.

When using Postresql you'll have access to some extra features:

Modelfields:

```
ArrayField         # A field for storing lists of data.
HStoreField        # A field for storing mappings of strings to strings.
JSONField          # A field for storing JSON encoded data.
IntegerRangeField  # Stores a range of integers
BigIntegerRangeField # Stores a big range of integers
FloatRangeField    # Stores a range of floating point values.
DateTimeRangeField # Stores a range of timestamps
```

sqlite

sqlite is the default for Django. *It should not be used in production since it is usually slow.*

```
#myapp/settings/settings.py

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'db/development.sqlite3',
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    },
}
```

Fixtures

Fixtures are initial data for the database. The most straightforward way when you have some existing data already is to use the command `dumpdata`

```
 ./manage.py dumpdata > databasedump.json                # full database
 ./manage.py dumpdata myapp > databasedump.json          # only 1 app
 ./manage.py dumpdata myapp.mymodel > databasedump.json  # only 1 model (table)
```

This will create a json file which can be imported again by using

```
./manage.py loaddata databasedump.json
```

When using the `loadddata` without specifying a file, Django will look for a `fixtures` folder in your app or the list of directories provided in the `FIXTURE_DIRS` in settings, and use its content instead.

```
/myapp
   /fixtures
        myfixtures.json
        morefixtures.xml
```

Possible file formats are: `JSON, XML or YAML`

Fixtures JSON example:

```
[
  {
    "model": "myapp.person",
    "pk": 1,
    "fields": {
      "first_name": "John",
      "last_name": "Lennon"
    }
  },
  {
    "model": "myapp.person",
    "pk": 2,
    "fields": {
      "first_name": "Paul",
      "last_name": "McCartney"
    }
  }
]
```

Fixtures YAML example:

```
- model: myapp.person
  pk: 1
  fields:
    first_name: John
    last_name: Lennon
- model: myapp.person
  pk: 2
  fields:
    first_name: Paul
    last_name: McCartney
```

Fixtures XML example:

```
<?xml version="1.0" encoding="utf-8"?>
<django-objects version="1.0">
    <object pk="1" model="myapp.person">
        <field type="CharField" name="first_name">John</field>
        <field type="CharField" name="last_name">Lennon</field>
    </object>
    <object pk="2" model="myapp.person">
        <field type="CharField" name="first_name">Paul</field>
        <field type="CharField" name="last_name">McCartney</field>
    </object>
</django-objects>
```

Django Cassandra Engine

- Install pip : `$ pip install django-cassandra-engine`
- Add Getting Started to INSTALLED_APPS in your settings.py file: `INSTALLED_APPS = ['django_cassandra_engine']`
- Cange DATABASES setting Standart:

  Standart

```
DATABASES = {
    'default': {
        'ENGINE': 'django_cassandra_engine',
        'NAME': 'db',
        'TEST_NAME': 'test_db',
        'HOST': 'db1.example.com,db2.example.com',
        'OPTIONS': {
            'replication': {
                'strategy_class': 'SimpleStrategy',
                'replication_factor': 1
            }
        }
    }
}
```

Cassandra create new user cqlsh :

```
DATABASES = {
'default': {
    'ENGINE': 'django_cassandra_engine',
    'NAME': 'db',
    'TEST_NAME': 'test_db',
    'USER_NAME'='cassandradb',
    'PASSWORD'= '123cassandra',
    'HOST': 'db1.example.com,db2.example.com',
    'OPTIONS': {
        'replication': {
            'strategy_class': 'SimpleStrategy',
            'replication_factor': 1
        }
    }
}
```

}

Read Database Setup online: https://riptutorial.com/django/topic/4933/database-setup

Chapter 13: Database transactions

Examples

Atomic transactions

Problem

By default, Django immediately commits changes to the database. When exceptions occur during a series of commits, this can leave your database in an unwanted state:

```
def create_category(name, products):
    category = Category.objects.create(name=name)
    product_api.add_products_to_category(category, products)
    activate_category(category)
```

In the following scenario:

```
>>> create_category('clothing', ['shirt', 'trousers', 'tie'])
--------------------------------------------------------------------------
ValueError: Product 'trousers' already exists
```

An exception occurs whilst trying to add the trousers product to the clothing category. By this point, the category itself has already been added, and the shirt product has been added to it.

The incomplete category and containing product would have to be manually removed before fixing the code and calling the `create_category()` method once more, as otherwise a duplicate category would be created.

Solution

The `django.db.transaction` module allows you to combine multiple database changes into an atomic transaction:

> [a] series of database operations such that either all occur, or nothing occurs.

Applied to the above scenario, this can be applied as a decorator:

```
from django.db import transaction

@transaction.atomic
def create_category(name, products):
    category = Category.objects.create(name=name)
    product_api.add_products_to_category(category, products)
    activate_category(category)
```

Or by using a context manager:

```
def create_category(name, products):
    with transaction.atomic():
        category = Category.objects.create(name=name)
        product_api.add_products_to_category(category, products)
        activate_category(category)
```

Now, if an exception occurs at any stage within the transaction, no database changes will be committed.

Read Database transactions online: https://riptutorial.com/django/topic/5555/database-transactions

Chapter 14: Debugging

Remarks

Pdb

Pdb can also print out all existing variables in global or local scope, by typing `globals()` or `locals()` in (Pdb) prompt respectively.

Examples

Using Python Debugger (Pdb)

Most basic Django debugging tool is pdb, a part of Python standard library.

Init view script

Let's examine a simple `views.py` script:

```
from django.http import HttpResponse


def index(request):
    foo = 1
    bar = 0

    bug = foo/bar

    return HttpResponse("%d goes here." % bug)
```

Console command to run server:

```
python manage.py runserver
```

It's obvious that Django would throw a `ZeroDivisionError` when you try to load index page, but if we'll pretend that the bug is very deep in the code, it could get really nasty.

Setting a breakpoint

Fortunately, we can set a *breakpoint* to trace down that bug:

```
from django.http import HttpResponse

# Pdb import
import pdb


def index(request):
    foo = 1
```

```
    bar = 0

    # This is our new breakpoint
    pdb.set_trace()

    bug = foo/bar

    return HttpResponse("%d goes here." % bug)
```

Console command to run server with pdb:

```
python -m pdb manage.py runserver
```

Now on page load breakpoint will trigger (Pdb) prompt in the shell, which will also hang your browser in pending state.

Debugging with pdb shell

It's time to debug that view by interacting with script via shell:

```
> ../views.py(12)index()
-> bug = foo/bar
# input 'foo/bar' expression to see division results:
(Pdb) foo/bar
*** ZeroDivisionError: division by zero
# input variables names to check their values:
(Pdb) foo
1
(Pdb) bar
0
# 'bar' is a source of the problem, so if we set it's value > 0...
(Pdb) bar = 1
(Pdb) foo/bar
1.0
# exception gone, ask pdb to continue execution by typing 'c':
(Pdb) c
[03/Aug/2016 10:50:45] "GET / HTTP/1.1" 200 111
```

In the last line we see that our view returned an `OK` response and executing as it should.

To stop pdb loop, just input `q` in a shell.

Using Django Debug Toolbar

First, you need to install django-debug-toolbar:

```
pip install django-debug-toolbar
```

settings.py:

Next, include it to project's installed apps, but be careful - it's always a good practice to use a different `settings.py` file for such development-only apps and middlewares as debug toolbar:

```
# If environment is dev...
DEBUG = True

INSTALLED_APPS += [
    'debug_toolbar',
]

MIDDLEWARE += ['debug_toolbar.middleware.DebugToolbarMiddleware']
```

Debug toolbar also relies on static files, so appropriate app should be included as well:

```
INSTALLED_APPS = [
    # ...
    'django.contrib.staticfiles',
    # ...
]

STATIC_URL = '/static/'

# If environment is dev...
DEBUG = True

INSTALLED_APPS += [
    'debug_toolbar',
]
```

In some cases, it's also required to set `INTERNAL_IPS` in `settings.py`:

```
INTERNAL_IPS = ('127.0.0.1', )
```

urls.py:

In `urls.py`, as official documentation suggests, the next snippet should enable debug toolbar routing:

```
if settings.DEBUG and 'debug_toolbar' in settings.INSTALLED_APPS:
    import debug_toolbar
    urlpatterns += [
        url(r'^__debug__/', include(debug_toolbar.urls)),
    ]
```

Collect toolbar's static after installation:

```
python manage.py collectstatic
```

That's it, debug toolbar will appear on you project's pages, providing various useful information about execution time, SQL, static files, signals, etc.

HTML:

Also, `django-debug-toolbar` requires a *Content-type* of `text/html`, `<html>` and `<body>` tags to render properly.

In case if you sure you've configured everything right, but debug toolbar is still not rendered: use this "nuclear" solution to try to figure it out.

Using "assert False"

While developing, inserting the following line to your code:

```
assert False, value
```

will cause django to raise an `AssertionError` with the value supplied as an error message when this line is executed.

If this occurs in a view, or in any code called from a view, and `DEBUG=True` is set, a full and detailed stacktrace with a lot of debugging information will be displayed in the browser.

Don't forget to remove the line when you are done!

Consider Writing More Documentation, Tests, Logging and Assertions Instead of Using a Debugger

Debugging takes time and effort.

Instead of chasing bugs with a debugger, consider spending more time on making your code better by:

- **Write and run Tests.** Python and Django have great builtin testing frameworks - that can be used to test your code much faster than manually with a debugger.
- **Writing proper documentation** for your functions, classes and modules. PEP 257 and Google's Python Style Guide supplies good practices for writing good docstrings.
- **Use Logging** to produce output from your program - during development and after deploying.
- **Add `assert`ions** to your code in important places: Reduce ambiguity, catch problems as they are created.

Bonus: Write doctests for combining documentation and testing!

Read Debugging online: https://riptutorial.com/django/topic/5072/debugging

Chapter 15: Deployment

Examples

Running Django application with Gunicorn

1. Install gunicorn

   ```
   pip install gunicorn
   ```

2. From django project folder (same folder where manage.py resides), run the following command to run current django project with gunicorn

   ```
   gunicorn [projectname].wsgi:application -b 127.0.0.1:[port number]
   ```

   You can use the `--env` option to set the path to load the settings

   ```
   gunicorn --env DJANGO_SETTINGS_MODULE=[projectname].settings [projectname].wsgi
   ```

   or run as daemon process using `-D` option

3. Upon successful start of gunicorn, the following lines will appear in console

   ```
   Starting gunicorn 19.5.0

   Listening at: http://127.0.0.1:[port number] ([pid])
   ```

   .... (other additional information about gunicorn server)

Deploying with Heroku

1. Download Heroku Toolbelt.

2. Navigate to the root of the sources of your Django app. You'll need tk

3. Type `heroku create [app_name]`. If you don't give an app name, Heroku will randomly generate one for you. Your app URL will be `http://[app name].herokuapp.com`

4. Make a text file with the name `Procfile`. Don't put an extension at the end.

   ```
   web: <bash command to start production server>
   ```

   If you have a worker process, you can add it too. Add another line in the format:
   `worker-name: <bash command to start worker>`

5. Add a requirements.txt.

- If you are using a virtual environment, execute `pip freeze > requirements.txt`
- Otherwise, *get a virtual environment!*. You can also manually list the Python packages you need, but that won't be covered in this tutorial.

6. **It's deployment time!**

    1. `git push heroku master`

       Heroku needs a git repository or a dropbox folder to do deploys. You can alternatively set up automatic reloading from a GitHub repository at `heroku.com`, but we won't cover that in this tutorial.

    2. `heroku ps:scale web=1`

       This scales the number of web "dynos" to one. You can learn more about dynos here.

    3. `heroku open` or navigate to `http://app-name.herokuapp.com`

       Tip: `heroku open` opens the URL to your heroku app in the default browser.

7. Add **add-ons**. You'll need to configure your Django app to bind with databases provided in Heroku as "add-ons". This example doesn't cover this, but another example is in the pipeline on deploying databases in Heroku.

Simple remote deploy fabfile.py

Fabric is a Python (2.5-2.7) library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks. It lets you execute arbitrary Python functions via the command line.

Install fabric via `pip install fabric`
Create `fabfile.py` in your root directory:

```
#myproject/fabfile.py
from fabric.api import *

@task
def dev():
    # details of development server
    env.user = # your ssh user
    env.password = #your ssh password
    env.hosts = # your ssh hosts (list instance, with comma-separated hosts)
    env.key_filename = # pass to ssh key for github in your local keyfile

@task
def release():
    # details of release server
    env.user = # your ssh user
    env.password = #your ssh password
    env.hosts = # your ssh hosts (list instance, with comma-separated hosts)
    env.key_filename = # pass to ssh key for github in your local keyfile

@task
def run():
    with cd('path/to/your_project/'):
        with prefix('source ../env/bin/activate'):
        # activate venv, suppose it appear in one level higher
            # pass commands one by one
```

```
            run('git pull')
            run('pip install -r requirements.txt')
            run('python manage.py migrate --noinput')
            run('python manage.py collectstatic --noinput')
            run('touch reload.txt')
```

To execute the file, simply use the `fab` command:

```
$ fab dev run  # for release server, `fab release run`
```

Note: you can not configure ssh keys for github and just type login and password manually, while fabfile runs, the same with keys.

Using Heroku Django Starter Template.

If you plan to host your Django website on Heroku, you can start your project using the Heroku Django Starter Template :

```
django-admin.py startproject --template=https://github.com/heroku/heroku-django-
template/archive/master.zip --name=Procfile YourProjectName
```

It has Production-ready configuration for Static Files, Database Settings, Gunicorn, etc and Enhancements to Django's static file serving functionality via WhiteNoise. This will save your time, it's All-Ready for hosting on Heroku, Just build your website on the top of this template

To deploy this template on Heroku:

```
git init
git add -A
git commit -m "Initial commit"

heroku create
git push heroku master

heroku run python manage.py migrate
```

That's it!

Django deployment instructions. Nginx + Gunicorn + Supervisor on Linux (Ubuntu)

Three basic tools.

1. nginx - free, open-source, high-performance HTTP server and reverse proxy, with high performance;
2. gunicorn - 'Green Unicorn' is a Python WSGI HTTP Server for UNIX (needed to manage your server);
3. supervisor - a client/server system that allows its users to monitor and control a number of processes on UNIX-like operating systems. Used when you app or system crashes, restarts your django / celery / celery cam, etc;

In order ot make it simple, let's assume your app is located in this directory: `/home/root/app/src/` and we're gonna use `root` user (but you should create separate user for your app). Also our virtualenvironment will be located in `/home/root/app/env/` path.

NGINX

Let's start with nginx. If nginx is not already on machine, install it with `sudo apt-get install nginx`. Later on you have to create a new config file in your nginx directory `/etc/nginx/sites-enabled/yourapp.conf`. If there is a file named `default.conf` - remove it.

Bellow code to a nginx conf file, which will try to run your service with using socket file; Later on there will be a configuration of gunicorn. Socket file is used here to communicate between nginx and gunicorn. It can also be done with using ports.

```
# your application name; can be whatever you want
upstream yourappname {
    server        unix:/home/root/app/src/gunicorn.sock fail_timeout=0;
}

server {
    # root folder of your application
    root        /home/root/app/src/;

    listen        80;
    # server name, your main domain, all subdomains and specific subdomains
    server_name   yourdomain.com *.yourdomain.com somesubdomain.yourdomain.com

    charset       utf-8;

    client_max_body_size                        100m;

    # place where logs will be stored;
    # folder and files have to be already located there, nginx will not create
    access_log        /home/root/app/src/logs/nginx-access.log;
    error_log         /home/root/app/src/logs/nginx-error.log;

    # this is where your app is served (gunicorn upstream above)
    location / {
        uwsgi_pass  yourappname;
        include     uwsgi_params;
    }

    # static files folder, I assume they will be used
    location /static/ {
        alias         /home/root/app/src/static/;
    }

    # media files folder
    location /media/ {
        alias         /home/root/app/src/media/;
    }

}
```

GUNICORN

Now our GUNICORN script, which will be responsible for running django application on server. First thing is to install gunicorn in virtual environment with `pip install gunicorn`.

```
#!/bin/bash

ME="root"
DJANGODIR=/home/root/app/src # django app dir
SOCKFILE=/home/root/app/src/gunicorn.sock # your sock file - do not create it manually
USER=root
GROUP=webapps
NUM_WORKERS=3
DJANGO_SETTINGS_MODULE=yourapp.yoursettings
DJANGO_WSGI_MODULE=yourapp.wsgi
echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd $DJANGODIR

source /home/root/app/env/bin/activate
export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
export PYTHONPATH=$DJANGODIR:$PYTHONPATH

# Create the run directory if it doesn't exist
RUNDIR=$(dirname $SOCKFILE)
test -d $RUNDIR || mkdir -p $RUNDIR

# Start your Django Gunicorn
# Programs meant to be run under supervisor should not daemonize themselves (do not use --
daemon)
exec /home/root/app/env/bin/gunicorn ${DJANGO_WSGI_MODULE}:application \
  --name root \
  --workers $NUM_WORKERS \
  --user=$USER --group=$GROUP \
  --bind=unix:$SOCKFILE \
  --log-level=debug \
  --log-file=-
```

in order to be able to run gunicorn start script it has to have execution mode enabled so

```
sudo chmod u+x /home/root/app/src/gunicorn_start
```

now you will be able to start your gunicorn server with just using `./gunicorn_start`

SUPERVISOR

As said in the beginning, we want our application to be restarted when fails by a supervisor. If supervisor not yet on server install with `sudo apt-get install supervisor`.

At first install supervisor. Then create a `.conf` file in your main directory
`/etc/supervisor/conf.d/your_conf_file.conf`

configuration file content:

```
[program:yourappname]
command = /home/root/app/src/gunicorn_start
user = root
stdout_logfile = /home/root/app/src/logs/gunicorn_supervisor.log
redirect_stderr = true
```

Quick brief, `[program:youappname]` is required at the beginning, it will be our identifier. also `stdout_logfile` is a file where logs will be stored, both access and errors.

Having that done we have to tell our supervisor that we have just added new configuration file. To do it, there is different process for different Ubuntu version.

For `Ubuntu version 14.04 or lesser` than it, simply run those commands:

`sudo supervisorctl reread` -> rereads all config files inside supervisor catalog this should print out: **yourappname: available**

`sudo supervisorctl update` -> updates supervisor to newly added config files; should print out **yourappname: added process group**

For `Ubuntu 16.04` Run:

```
sudo service supervisor restart
```

and in order to check if your app is running correctly just run

```
sudo supervisorctl status yourappname
```

This should display :

```
yourappname RUNNING pid 18020, uptime 0:00:50
```

To get live demonstration of this procedure, surf this **video**.

Deploying locally without setting up apache/nginx

Recommended way of production deployment calls for using Apache/Nginx for serving the static content. Thus, when `DEBUG` is false static and media contents fail to load. However, we can load the static content in deployment without having to setup Apache/Nginx server for our app using:

```
python manage.py runserver --insecure
```

This is only intended for local deployment(e.g LAN) and should never be used in production and is only available if the `staticfiles` app is in your project's `INSTALLED_APPS` setting.

Read Deployment online: https://riptutorial.com/django/topic/2792/deployment

Chapter 16: Django and Social Networks

Parameters

Setting	Does
Some Configurations	Handy basic settings that go with Django-Allauth (that I use most of the time). For more configuration options, see Configurations
ACCOUNT_AUTHENTICATION_METHOD (=”username” or “email” or “username_email”)	Specifies the login method to use – whether the user logs in by entering their username, e-mail address, or either one of both. Setting this to “email” requires ACCOUNT_EMAIL_REQUIRED=True
ACCOUNT_EMAIL_CONFIRMATION_EXPIRE_DAYS (=3)	Determines the expiration date of email confirmation mails (# of days).
ACCOUNT_EMAIL_REQUIRED (=False)	The user is required to hand over an e-mail address when signing up. This goes in tandem with the `ACCOUNT_AUTHENTICATION_METHOD` setting
ACCOUNT_EMAIL_VERIFICATION (=”optional”)	Determines the e-mail verification method during signup – choose one of "mandatory", "optional", or "none". When set to “mandatory” the user is blocked from logging in until the email address is verified. Choose “optional” or “none” to allow logins with an unverified e-mail address. In case of “optional”, the e-mail verification mail is still sent, whereas in case of “none” no e-mail verification mails are sent.
ACCOUNT_LOGIN_ATTEMPTS_LIMIT (=5)	Number of failed login attempts. When this number is exceeded, the user is prohibited from logging in for the specified ACCOUNT_LOGIN_ATTEMPTS_TIMEOUT seconds. While this protects the allauth login view, it does not protect Django’s admin login from being brute forced.
ACCOUNT_LOGOUT_ON_PASSWORD_CHANGE (=False)	Determines whether or not the user is automatically logged out after changing or

Setting	Does
	setting their password.
SOCIALACCOUNT_PROVIDERS (= dict)	Dictionary containing provider specific settings.

Examples

Easy way: python-social-auth

python-social-auth is a framework that simplifies the social authentication and authorization mechanism. It contains many social backends (Facebook, Twitter, Github, LinkedIn, etc.)

INSTALL

First we need to install the python-social-auth package with

```
pip install python-social-auth
```

or download the code from github. Now is a good time to add this to your `requirements.txt` file.

CONFIGURING settings.py

In the settings.py add:

```
INSTALLED_APPS = (
    ...
    'social.apps.django_app.default',
    ...
)
```

CONFIGURING BACKENDS

AUTHENTICATION_BACKENDS contains the backends that we will use, and we only have to put what's we need.

```
AUTHENTICATION_BACKENDS = (
    'social.backends.open_id.OpenIdAuth',
    'social.backends.google.GoogleOpenId',
    'social.backends.google.GoogleOAuth2',
    'social.backends.google.GoogleOAuth',
    'social.backends.twitter.TwitterOAuth',
    'social.backends.yahoo.YahooOpenId',
    ...
    'django.contrib.auth.backends.ModelBackend',
)
```

Your project `settings.py` may not yet have an `AUTHENTICATION_BACKENDS` field. If that is the case add the field. Be sure not to miss `'django.contrib.auth.backends.ModelBackend',` as it handles login by

username/password.

If we use for example Facebook and Linkedin Backends we need to add the API keys

```
SOCIAL_AUTH_FACEBOOK_KEY = 'YOURFACEBOOKKEY'
SOCIAL_AUTH_FACEBOOK_SECRET = 'YOURFACEBOOKSECRET'
```

and

```
SOCIAL_AUTH_LINKEDIN_KEY = 'YOURLINKEDINKEY'
SOCIAL_AUTH_LINKEDIN_SECRET = 'YOURLINKEDINSECRET'
```

Note: You can Obtain the nedded keys in Facebook developers and Linkedin developers and here you can see the full list and his respective way to especify the API key and the key Secret.

Note on Secret Keys: Secret keys should be kept secret. Here is a Stack Overflow explanation that is helpful. This tutorial is helpful for learning about enviromental variables.

TEMPLATE_CONTEXT_PROCESSORS will help to redirections, backends and other things, but at beginning we only need these:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'social.apps.django_app.context_processors.backends',
    'social.apps.django_app.context_processors.login_redirect',
    ...
)
```

In Django 1.8 setting up `TEMPLATE_CONTEXT_PREPROCESSORS` as shown above was deprecated. If this is the case for you you'll add it inside of the `TEMPLATES` dict. Yours should look something similar to this:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, "templates")],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'social.apps.django_app.context_processors.backends',
                'social.apps.django_app.context_processors.login_redirect',
            ],
        },
    },
]
```

USING A CUSTOM USER

If you are using a custom User Model and want to asociate with it, just add the following line (still

in **settings.py**)

```
SOCIAL_AUTH_USER_MODEL = 'somepackage.models.CustomUser'
```

`CustomUser` is a model which inherit or Abstract from default User.

CONFIGURING urls.py

```
# if you haven't imported inlcude make sure you do so at the top of your file
from django.conf.urls import url, include

urlpatterns = patterns('',
    ...
    url('', include('social.apps.django_app.urls', namespace='social'))
    ...
)
```

Next need to sync database to create needed models:

```
./manage.py migrate
```

Finally we can play!

in some template you need to add something like this:

```
    <a href="{% url 'social:begin' 'facebook' %}?next={{ request.path }}">Login with
Facebook</a>
    <a href="{% url 'social:begin' 'linkedin' %}?next={{ request.path }}">Login with
Linkedin</a>
```

if you use another backend just change 'facebook' by the backend name.

Logging users out

Once you have logged users in you'll likely want to create the functionality to log them back out. In some template, likely near where the log in template was shown, add the following tag:

```
<a href="{% url 'logout' %}">Logout</a>
```

or

```
<a href="/logout">Logout</a>
```

You'll want to edit your `urls.py` file with code similar to:

```
url(r'^logout/$', views.logout, name='logout'),
```

Lastly edit your views.py file with code similar to:

```
def logout(request):
```

```
    auth_logout(request)
    return redirect('/')
```

Using Django Allauth

For all my projects, Django-Allauth remained one that is easy to setup, and comes out of the box with many features including but not limited to:

- Some 50+ social networks authentications
- Mix signup of both local and social accounts
- Multiple social accounts
- Optional instant-signup for social accounts – no questions asked
- E-mail address management (multiple e-mail addresses, setting a primary)
- Password forgotten flow E-mail address verification flow

If you're interested in getting your hands dirty, Django-Allauth gets out of the way, with additional configurations to tweak the process and use of your authentication system.

The steps below assume you're using Django 1.10+

Setup steps:

```
pip install django-allauth
```

In your `settings.py` file, make the following changes:

```
# Specify the context processors as follows:
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                # Already defined Django-related contexts here

                # `allauth` needs this from django. It is there by default,
                # unless you've devilishly taken it away.
                'django.template.context_processors.request',
            ],
        },
    },
]

AUTHENTICATION_BACKENDS = (
    # Needed to login by username in Django admin, regardless of `allauth`
    'django.contrib.auth.backends.ModelBackend',

    # `allauth` specific authentication methods, such as login by e-mail
    'allauth.account.auth_backends.AuthenticationBackend',
)

INSTALLED_APPS = (
# Up here is all your default installed apps from Django
```

```
# The following apps are required:
'django.contrib.auth',
'django.contrib.sites',

'allauth',
'allauth.account',
'allauth.socialaccount',

# include the providers you want to enable:
'allauth.socialaccount.providers.google',
'allauth.socialaccount.providers.facebook',
)

# Don't forget this little dude.
SITE_ID = 1
```

Done with the changes in `settings.py` file above, move onto the `urls.py` file. It can be your `yourapp/urls.py` or your `ProjectName/urls.py`. Normally, I prefer the `ProjectName/urls.py`.

```
urlpatterns = [
    # other urls here
    url(r'^accounts/', include('allauth.urls')),
    # other urls here
]
```

Simply adding the `include('allauth.urls')`, gives you these urls for free:

```
^accounts/ ^ ^signup/$ [name='account_signup']
^accounts/ ^ ^login/$ [name='account_login']
^accounts/ ^ ^logout/$ [name='account_logout']
^accounts/ ^ ^password/change/$ [name='account_change_password']
^accounts/ ^ ^password/set/$ [name='account_set_password']
^accounts/ ^ ^inactive/$ [name='account_inactive']
^accounts/ ^ ^email/$ [name='account_email']
^accounts/ ^ ^confirm-email/$ [name='account_email_verification_sent']
^accounts/ ^ ^confirm-email/(?P<key>[-:\w]+)/$ [name='account_confirm_email']
^accounts/ ^ ^password/reset/$ [name='account_reset_password']
^accounts/ ^ ^password/reset/done/$ [name='account_reset_password_done']
^accounts/ ^ ^password/reset/key/(?P<uidb36>[0-9A-Za-z]+)-(?P<key>.+)/$
[name='account_reset_password_from_key']
^accounts/ ^ ^password/reset/key/done/$ [name='account_reset_password_from_key_done']
^accounts/ ^social/
^accounts/ ^google/
^accounts/ ^twitter/
^accounts/ ^facebook/
^accounts/ ^facebook/login/token/$ [name='facebook_login_by_token']
```

Finally, do `python ./manage.py migrate` to commit the migrates of Django-allauth into Database.

As usual, to be able to log into your app using any social network you've added, you'll have to add the social account details of the network.

Login to the Django Admin (`localhost:8000/admin`) and under `Social Applications` in the add your social account details.

You might need accounts at each auth provider in order to obtain details to fill in at the Social

Applications sections.

For detailed configurations of what you can have and tweak, see the Configurations page.

Read Django and Social Networks online: https://riptutorial.com/django/topic/4743/django-and-social-networks

Chapter 17: Django from the command line.

Remarks

While Django is primarily for web apps it has a powerful and easy to use ORM that can be used for command line apps and scripts too. There are two different approaches that can be used. The first being to create a custom management command and the second to initialize the Django environment at the start of your script.

Examples

Django from the command line.

Supposing you have setup a django project, and the settings file is in an app named main, this is how you initialize your code

```
import os, sys

# Setup environ
sys.path.append(os.getcwd())
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "main.settings")

# Setup django
import django
django.setup()

# rest of your imports go here

from main.models import MyModel

# normal python code that makes use of Django models go here

for obj in MyModel.objects.all():
    print obj
```

The above can be executed as

```
python main/cli.py
```

Read Django from the command line. online: https://riptutorial.com/django/topic/5848/django-from-the-command-line-

Chapter 18: Django Rest Framework

Examples

Simple barebones read-only API

Assuming you have a model that looks like the following, we will get up an running with a simple barebones **read-only** API driven by Django REST Framework ("DRF").

models.py

```
class FeedItem(models.Model):
    title = models.CharField(max_length=100, blank=True)
    url = models.URLField(blank=True)
    style = models.CharField(max_length=100, blank=True)
    description = models.TextField(blank=True)
```

The serializer is the component that will take all of the information from the Django model (in this case the `FeedItem`) and turn it into JSON. It is very similar to creating form classes in Django. If you have any experience in that, this will be very comfortable for you.

serializers.py

```
from rest_framework import serializers
from . import models

class FeedItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = models.FeedItem
        fields = ('title', 'url', 'description', 'style')
```

views.py

DRF offers many view classes to handle a variety of use cases. In this example, we are only going to have a **read-only** API, so, rather than using a more comprehensive viewset, or a bunch of related generic views, we will use a single subclass of DRF's `ListAPIView`.

The purpose of this class is to link the data with the serializer, and wrap it all together for a response object.

```
from rest_framework import generics
from . import serializers, models

class FeedItemList(generics.ListAPIView):
    serializer_class = serializers.FeedItemSerializer
    queryset = models.FeedItem.objects.all()
```

urls.py

Make sure you point your route to your DRF view.

```
from django.conf.urls import url
from . import views

urlpatterns = [
    ...
    url(r'path/to/api', views.FeedItemList.as_view()),
]
```

Read Django Rest Framework online: https://riptutorial.com/django/topic/7341/django-rest-framework

Chapter 19: django-filter

Examples

Use django-filter with CBV

`django-filter` is generic system for filtering Django QuerySets based on user selections. The documentation uses it in a function-based view as a product model:

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField()
    description = models.TextField()
    release_date = models.DateField()
    manufacturer = models.ForeignKey(Manufacturer)
```

The filter will be as follows:

```
import django_filters

class ProductFilter(django_filters.FilterSet):
    name = django_filters.CharFilter(lookup_expr='iexact')

    class Meta:
        model = Product
        fields = ['price', 'release_date']
```

To use this in a CBV, override `get_queryset()` of the ListView, then return the filtered `querset`:

```
from django.views.generic import ListView
from .filters import ProductFilter

class ArticleListView(ListView):
    model = Product

    def get_queryset(self):
        qs = self.model.objects.all()
        product_filtered_list = ProductFilter(self.request.GET, queryset=qs)
        return product_filtered_list.qs
```

It is possible to access the filtered objects in your views, such as with pagination, in `f.qs`. This will paginate the filtered objects list.

Read django-filter online: https://riptutorial.com/django/topic/6101/django-filter

Chapter 20: Extending or Substituting User Model

Examples

Custom user model with email as primary login field.

models.py :

```
from __future__ import unicode_literals
from django.db import models
from django.contrib.auth.models import (
        AbstractBaseUser, BaseUserManager, PermissionsMixin)
from django.utils import timezone
from django.utils.translation import ugettext_lazy as _


class UserManager(BaseUserManager):
    def _create_user(self, email,password, is_staff, is_superuser, **extra_fields):
        now = timezone.now()
        if not email:
            raise ValueError('users must have an email address')
        email = self.normalize_email(email)
        user = self.model(email = email,
                          is_staff = is_staff,
                          is_superuser = is_superuser,
                          last_login = now,
                          date_joined = now,
                          **extra_fields)
        user.set_password(password)
        user.save(using = self._db)
        return user

    def create_user(self, email, password=None, **extra_fields):
        user = self._create_user(email, password, False, False, **extra_fields)
        return user

    def create_superuser(self, email, password, **extra_fields):
        user = self._create_user(email, password, True, True, **extra_fields)
        return user

class User(AbstractBaseUser,PermissionsMixin):
    """My own custom user class"""

    email = models.EmailField(max_length=255, unique=True, db_index=True,
verbose_name=_('email address'))
    date_joined = models.DateTimeField(auto_now_add=True)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    objects = UserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = []
```

```
    class Meta:
        verbose_name = _('user')
        verbose_name_plural = _('users')

    def get_full_name(self):
    """Return the email."""
        return self.email

    def get_short_name(self):
    """Return the email."""
        return self.email
```

forms.py :

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from .models import User


class RegistrationForm(UserCreationForm):
    email = forms.EmailField(widget=forms.TextInput(
        attrs={'class': 'form-control','type':'text','name': 'email'}),
        label="Email")
    password1 = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name':'password1'}),
        label="Password")
    password2 = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name': 'password2'}),
        label="Password (again)")

    '''added attributes so as to customise for styling, like bootstrap'''
    class Meta:
        model = User
        fields = ['email','password1','password2']
        field_order = ['email','password1','password2']

    def clean(self):
    """
    Verifies that the values entered into the password fields match
    NOTE : errors here will appear in 'non_field_errors()'
    """
        cleaned_data = super(RegistrationForm, self).clean()
        if 'password1' in self.cleaned_data and 'password2' in self.cleaned_data:
            if self.cleaned_data['password1'] != self.cleaned_data['password2']:
                raise forms.ValidationError("Passwords don't match. Please try again!")
        return self.cleaned_data

    def save(self, commit=True):
        user = super(RegistrationForm,self).save(commit=False)
        user.set_password(self.cleaned_data['password1'])
        if commit:
            user.save()
        return user

#The save(commit=False) tells Django to save the new record, but dont commit it to the
database yet

class AuthenticationForm(forms.Form): # Note: forms.Form NOT forms.ModelForm
    email = forms.EmailField(widget=forms.TextInput(
        attrs={'class': 'form-control','type':'text','name': 'email','placeholder':'Email'}),
```

```
        label='Email')
    password = forms.CharField(widget=forms.PasswordInput(
        attrs={'class':'form-control','type':'password', 'name':
'password','placeholder':'Password'}),
        label='Password')

    class Meta:
        fields = ['email', 'password']
```

views.py :

```
from django.shortcuts import redirect, render, HttpResponse
from django.contrib.auth import login as django_login, logout as django_logout, authenticate
as django_authenticate
#importing as such so that it doesn't create a confusion with our methods and django's default
methods

from django.contrib.auth.decorators import login_required
from .forms import AuthenticationForm, RegistrationForm


def login(request):
    if request.method == 'POST':
        form = AuthenticationForm(data = request.POST)
        if form.is_valid():
            email = request.POST['email']
            password = request.POST['password']
            user = django_authenticate(email=email, password=password)
            if user is not None:
                if user.is_active:
                    django_login(request,user)
                    return redirect('/dashboard') #user is redirected to dashboard
    else:
        form = AuthenticationForm()

    return render(request,'login.html',{'form':form,})

def register(request):
    if request.method == 'POST':
        form = RegistrationForm(data = request.POST)
        if form.is_valid():
            user = form.save()
            u = django_authenticate(user.email = user, user.password = password)
            django_login(request,u)
            return redirect('/dashboard')
    else:
        form = RegistrationForm()

    return render(request,'register.html',{'form':form,})

def logout(request):
    django_logout(request)
    return redirect('/')

@login_required(login_url ="/")
def dashboard(request):
    return render(request, 'dashboard.html',{})
```

settings.py :

```
AUTH_USER_MODEL = 'myapp.User'
```

admin.py

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.models import Group
from .models import User


class UserAdmin(BaseUserAdmin):
    list_display = ('email','is_staff')
    list_filter = ('is_staff',)
    fieldsets = ((None,
                  {'fields':('email','password')}), ('Permissions',{'fields':('is_staff',)}),)
    add_fieldsets = ((None, {'classes': ('wide',), 'fields': ('email', 'password1',
'password2')}),)
    search_fields =('email',)
    ordering = ('email',)
    filter_horizontal = ()

admin.site.register(User, UserAdmin)
admin.site.unregister(Group)
```

Use the `email` as username and get rid of the `username` field

If you want to get rid of the `username` field and use `email` as unique user identifier, you will have to create a custom `User` model extending `AbstractBaseUser` instead of `AbstractUser`. Indeed, `username` and `email` are defined in `AbstractUser` and you can't override them. This means you will also have to redefine all fields you want that are defined in `AbstractUser`.

```
from django.contrib.auth.models import (
    AbstractBaseUser, PermissionsMixin, BaseUserManager,
)
from django.db import models
from django.utils import timezone
from django.utils.translation import ugettext_lazy as _

class UserManager(BaseUserManager):

    use_in_migrations = True

    def _create_user(self, email, password, **extra_fields):
        if not email:
            raise ValueError('The given email must be set')
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_user(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', False)
        extra_fields.setdefault('is_superuser', False)
        return self._create_user(email, password, **extra_fields)

    def create_superuser(self, email, password, **extra_fields):
```

```
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)

        if extra_fields.get('is_staff') is not True:
            raise ValueError('Superuser must have is_staff=True.')
        if extra_fields.get('is_superuser') is not True:
            raise ValueError('Superuser must have is_superuser=True.')

    return self._create_user(email, password, **extra_fields)


class User(AbstractBaseUser, PermissionsMixin):
    """PermissionsMixin contains the following fields:
        - `is_superuser`
        - `groups`
        - `user_permissions`
     You can omit this mix-in if you don't want to use permissions or
     if you want to implement your own permissions logic.
     """

    class Meta:
        verbose_name = _("user")
        verbose_name_plural = _("users")
        db_table = 'auth_user'
        # `db_table` is only needed if you move from the existing default
        # User model to a custom one. This enables to keep the existing data.

    USERNAME_FIELD = 'email'
    """Use the email as unique username."""

    REQUIRED_FIELDS = ['first_name', 'last_name']

    GENDER_MALE = 'M'
    GENDER_FEMALE = 'F'
    GENDER_CHOICES = [
        (GENDER_MALE, _("Male")),
        (GENDER_FEMALE, _("Female")),
    ]

    email = models.EmailField(
        verbose_name=_("email address"), unique=True,
        error_messages={
            'unique': _(
                "A user is already registered with this email address"),
        },
    )
    gender = models.CharField(
        max_length=1, blank=True, choices=GENDER_CHOICES,
        verbose_name=_("gender"),
    )
    first_name = models.CharField(
        max_length=30, verbose_name=_("first name"),
    )
    last_name = models.CharField(
        max_length=30, verbose_name=_("last name"),
    )
    is_staff = models.BooleanField(
        verbose_name=_("staff status"),
        default=False,
        help_text=_(
            "Designates whether the user can log into this admin site."
```

```
        ),
    )
    is_active = models.BooleanField(
        verbose_name=_("active"),
        default=True,
        help_text=_(
            "Designates whether this user should be treated as active. "
            "Unselect this instead of deleting accounts."
        ),
    )
    date_joined = models.DateTimeField(
        verbose_name=_("date joined"), default=timezone.now,
    )

    objects = UserManager()
```

Extend Django User Model Easily

Our `UserProfile` class

Create a `UserProfile` model class with the relationship of `OneToOne` to the default `User` model:

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save

class UserProfile(models.Model):
    user = models.OneToOneField(User, related_name='user')
    photo = FileField(verbose_name=_("Profile Picture"),
                      upload_to=upload_to("main.UserProfile.photo", "profiles"),
                      format="Image", max_length=255, null=True, blank=True)
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)
    phone = models.CharField(max_length=20, blank=True, default='')
    city = models.CharField(max_length=100, default='', blank=True)
    country = models.CharField(max_length=100, default='', blank=True)
    organization = models.CharField(max_length=100, default='', blank=True)
```

Django Signals at work

Using Django Signals, create a new `UserProfile` immediately a `User` object is created. This function can be tucked beneath the `UserProfile` model class in the same file, or place it wherever you like. I don't care, as along as you reference it properly.

```
def create_profile(sender, **kwargs):
    user = kwargs["instance"]
    if kwargs["created"]:
        user_profile = UserProfile(user=user)
        user_profile.save()
post_save.connect(create_profile, sender=User)
```

`inlineformset_factory` to the rescue

Now for your `views.py`, you might have something like this:

```
from django.shortcuts import render, HttpResponseRedirect
from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import User
from .models import UserProfile
from .forms import UserForm
from django.forms.models import inlineformset_factory
from django.core.exceptions import PermissionDenied
@login_required() # only logged in users should access this
def edit_user(request, pk):
    # querying the User object with pk from url
    user = User.objects.get(pk=pk)

    # prepopulate UserProfileForm with retrieved user values from above.
    user_form = UserForm(instance=user)

    # The sorcery begins from here, see explanation https://blog.khophi.co/extending-django-
user-model-userprofile-like-a-pro/
    ProfileInlineFormset = inlineformset_factory(User, UserProfile, fields=('website', 'bio',
'phone', 'city', 'country', 'organization'))
    formset = ProfileInlineFormset(instance=user)

    if request.user.is_authenticated() and request.user.id == user.id:
        if request.method == "POST":
            user_form = UserForm(request.POST, request.FILES, instance=user)
            formset = ProfileInlineFormset(request.POST, request.FILES, instance=user)

            if user_form.is_valid():
                created_user = user_form.save(commit=False)
                formset = ProfileInlineFormset(request.POST, request.FILES,
instance=created_user)

                if formset.is_valid():
                    created_user.save()
                    formset.save()
                    return HttpResponseRedirect('/accounts/profile/')

        return render(request, "account/account_update.html", {
            "noodle": pk,
            "noodle_form": user_form,
            "formset": formset,
        })
    else:
        raise PermissionDenied
```

Our Template

Then spit everything to your template `account_update.html` as so:

```
{% load material_form %}
<!-- Material form is just a materialize thing for django forms -->
<div class="col s12 m8 offset-m2">
      <div class="card">
        <div class="card-content">
        <h2 class="flow-text">Update your information</h2>
          <form action="." method="POST" class="padding">
            {% csrf_token %} {{ noodle_form.as_p }}
            <div class="divider"></div>
            {{ formset.management_form }}
                {{ formset.as_p }}
            <button type="submit" class="btn-floating btn-large waves-light waves-effect"><i
```

```
class="large material-icons">done</i></button>
            <a href="#" onclick="window.history.back(); return false;" title="Cancel"
class="btn-floating waves-effect waves-light red"><i class="material-icons">history</i></a>

        </form>
        </div>
    </div>
</div>
```

Above snippet taken from Extending Django UserProfile like a Pro

Specifing a custom User model

Django's built-in `User` model is not always appropiate for some kinds of projects. On some sites it might make more sense to use an email address instead of a username for instance.

You can override the default `User` model adding your customized `User` model to the `AUTH_USER_MODEL` setting, in your projects settings file:

```
AUTH_USER_MODEL = 'myapp.MyUser'
```

Note that it's highly adviced to create the `AUTH_USER_MODEL` before creating any migrations or running `manage.py migrate` for the first time. Due to limitations of Django's synamic dependecy feature.

For example on your blog you might want other authors to be able to sign-in with an email address instead of the regular username, so we create a custom `User` model with an email address as `USERNAME_FIELD`:

```
from django.contrib.auth.models import AbstractBaseUser

class CustomUser(AbstractBaseUser):
     email = models.EmailField(unique=True)

     USERNAME_FIELD = 'email'
```

By inherinting the `AbstractBaseUser` we can construct a compliant `User` model. `AbstractBaseUser` provides the core implementation of a `User` model.

In order to let the Django `manage.py createsuperuser` command know which other fields al required we can specify a `REQUIRED_FIELDS`. This value has no effect in other parts of Django!

```
class CustomUser(AbstractBaseUser):
    ...
    first_name = models.CharField(max_length=254)
    last_name = models.CharField(max_length=254)
    ...
    REQUIRED_FIELDS = ['first_name', 'last_name']
```

To be compliant with other part of Django we still have to specify the value `is_active`, the functions `get_full_name()` and `get_short_name()`:

```
class CustomUser(AbstractBaseUser):
    ...
    is_active = models.BooleanField(default=False)
    ...
    def get_full_name(self):
        full_name = "{0} {1}".format(self.first_name, self.last_name)
        return full_name.strip()

    def get_short_name(self):
        return self.first_name
```

You should also create a custom `UserManager` for your `User` model, which allows Django to use the `create_user()` and `create_superuser()` functions:

```
from django.contrib.auth.models import BaseUserManager

class CustomUserManager(BaseUserManager):
    def create_user(self, email, first_name, last_name, password=None):
        if not email:
            raise ValueError('Users must have an email address')

        user = self.model(
            email=self.normalize_email(email),
        )

        user.set_password(password)
        user.first_name = first_name
        user.last_name = last_name
        user.save(using=self._db)
        return user

    def create_superuser(self, email, first_name, last_name, password):
        user = self.create_user(
            email=email,
            first_name=first_name,
            last_name=last_name,
            password=password,
        )

        user.is_admin = True
        user.is_active = True
        user.save(using=self.db)
        return user
```

Referencing the User model

Your code will not work in projects where you reference the `User` model (*and where the `AUTH_USER_MODEL` setting has been changed*) directly.

For example: if you want to create `Post` model for a blog with a customized `User` model, you should specify the custom `User` model like this:

```
from django.conf import settings
from django.db import models

class Post(models.Model):
```

```
    author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
```

Read Extending or Substituting User Model online:
https://riptutorial.com/django/topic/1209/extending-or-substituting-user-model

Chapter 21: F() expressions

Introduction

An F() expression is a way for Django to use a Python object to refer to the value of model field or annotated column in the database without having to pull the value into Python memory. This allows developers to avoid certain race conditions and also filtering results based on model field values.

Syntax

- from django.db.models import F

Examples

Avoiding race conditions

See this Q&A question if you don't know what race conditions are.

The following code may be subject to race conditions :

```
article = Article.objects.get(pk=69)
article.views_count += 1
article.save()
```

If `views_count` is equal to `1337`, this will result in such query:

```
UPDATE app_article SET views_count = 1338 WHERE id=69
```

If two clients access this article at the same time, what *may* happen is that the second HTTP request executes `Article.objects.get(pk=69)` before the first executes `article.save()`. Thus, both requests will have `views_count = 1337`, increment it, and save `views_count = 1338` to the database, while it should actually be `1339`.

To fix this, use an `F()` expression:

```
article = Article.objects.get(pk=69)
article.views_count = F('views_count') + 1
article.save()
```

This, on the other hand, will result in such query:

```
UPDATE app_article SET views_count = views_count + 1 WHERE id=69
```

Updating queryset in bulk

Let's assume that we want to remove 2 upvotes from all the articles of the author with id `51`. Doing this only with Python would execute `N` queries (`N` being the number of articles in the queryset):

```
for article in Article.objects.filter(author_id=51):
    article.upvotes -= 2
    article.save()
    # Note that there is a race condition here but this is not the focus
    # of this example.
```

What if instead of pulling all the articles into Python, looping over them, decreasing the upvotes, and saving each updated one back to the database, there was another way?
Using an `F()` expression, can do it in one query:

```
Article.objects.filter(author_id=51).update(upvotes=F('upvotes') - 2)
```

Which can be translated in the following SQL query:

```
UPDATE app_article SET upvotes = upvotes - 2 WHERE author_id = 51
```

Why is this better?

- Instead of Python doing the work, we pass the load into the database which is fine tuned to make such queries.
- Effectively cuts down on the number of database queries needed to achieve the wanted result.

Execute Arithmetic operations between fields

`F()` expressions can be used to execute arithmetic operations (`+`, `-`, `*` etc.) among model fields, in order to define an algebraic lookup/connection between them.

- Let model be:

  ```
  class MyModel(models.Model):
      int_1 = models.IntegerField()
      int_2 = models.IntegerField()
  ```

- Now lets assume that we want to retrieve all the objects of `MyModel` table who's `int_1` and `int_2` fields satisfy this equation: `int_1 + int_2 >= 5`. Utilizing `annotate()` and `filter()` we get:

  ```
  result = MyModel.objects.annotate(
               diff=F(int_1) + F(int_2)
           ).filter(diff__gte=5)
  ```

  `result` now contains all of the aforementioned objects.

Although the example utilizes `Integer` fields, this method will work on every field on which an arithmetic operation can be applied.

Read F() expressions online: https://riptutorial.com/django/topic/2765/f---expressions

Chapter 22: Form Widgets

Examples

Simple text input widget

The most simple example of widget is custom text input. For instance, to create an `<input type="tel">`, you have to subclass `TextInput` and set `input_type` to `'tel'`.

```
from django.forms.widgets import TextInput

class PhoneInput(TextInput):
    input_type = 'tel'
```

Composite widget

You can create widgets composed of multiple widgets using `MultiWidget`.

```
from datetime import date

from django.forms.widgets import MultiWidget, Select
from django.utils.dates import MONTHS

class SelectMonthDateWidget(MultiWidget):
    """This widget allows the user to fill in a month and a year.

    This represents the first day of this month or, if `last_day=True`, the
    last day of this month.
    """

    default_nb_years = 10

    def __init__(self, attrs=None, years=None, months=None, last_day=False):
        self.last_day = last_day

        if not years:
            this_year = date.today().year
            years = range(this_year, this_year + self.default_nb_years)
        if not months:
            months = MONTHS

        # Here we will use two `Select` widgets, one for months and one for years
        widgets = (Select(attrs=attrs, choices=months.items()),
                   Select(attrs=attrs, choices=((y, y) for y in years)))
        super().__init__(widgets, attrs)

    def format_output(self, rendered_widgets):
        """Concatenates rendered sub-widgets as HTML"""
        return (
            '<div class="row">'
            '<div class="col-xs-6">{}</div>'
            '<div class="col-xs-6">{}</div>'
            '</div>'
        ).format(*rendered_widgets)
```

```
    def decompress(self, value):
        """Split the widget value into subwidgets values.
        We expect value to be a valid date formated as `%Y-%m-%d`.
        We extract month and year parts from this string.
        """
        if value:
            value = date(*map(int, value.split('-')))
            return [value.month, value.year]
        return [None, None]

    def value_from_datadict(self, data, files, name):
        """Get the value according to provided `data` (often from `request.POST`)
        and `files` (often from `request.FILES`, not used here)
        `name` is the name of the form field.

        As this is a composite widget, we will grab multiple keys from `data`.
        Namely: `field_name_0` (the month) and `field_name_1` (the year).
        """
        datalist = [
            widget.value_from_datadict(data, files, '{}_{}'.format(name, i))
            for i, widget in enumerate(self.widgets)]
        try:
            # Try to convert it as the first day of a month.
            d = date(day=1, month=int(datelist[0]), year=int(datelist[1]))
            if self.last_day:
                # Transform it to the last day of the month if needed
                if d.month == 12:
                    d = d.replace(day=31)
                else:
                    d = d.replace(month=d.month+1) - timedelta(days=1)
        except (ValueError, TypeError):
            # If we failed to recognize a valid date
            return ''
        else:
            # Convert it back to a string with format `%Y-%m-%d`
            return str(d)
```

Read Form Widgets online: https://riptutorial.com/django/topic/1230/form-widgets

Chapter 23: Forms

Examples

ModelForm Example

Create a ModelForm from an existing Model class, by subclassing `ModelForm`:

```
from django import forms

class OrderForm(forms.ModelForm):
    class Meta:
        model = Order
        fields = ['item', 'order_date', 'customer', 'status']
```

Defining a Django form from scratch (with widgets)

Forms can be defined, in a similar manner to models, by subclassing `django.forms.Form`. Various field input options are available such as `CharField`, `URLField`, `IntegerField`, etc.

Defining a simple contact form can be seen below:

```
from django import forms

class ContactForm(forms.Form):
    contact_name = forms.CharField(
        label="Your name", required=True,
        widget=forms.TextInput(attrs={'class': 'form-control'}))
    contact_email = forms.EmailField(
        label="Your Email Address", required=True,
        widget=forms.TextInput(attrs={'class': 'form-control'}))
    content = forms.CharField(
        label="Your Message", required=True,
        widget=forms.Textarea(attrs={'class': 'form-control'}))
```

Widget is Django's representation of HTML user-input tags and can be used to render custom html for form fields (eg: as a text box is rendered for the content input here)

`attrs` are attributes that will be copied over as is to the rendered html for the form.

Eg: `content.render("name", "Your Name")` gives

```
<input title="Your name" type="text" name="name" value="Your Name" class="form-control" />
```

Removing a modelForm's field based on condition from views.py

If we have a Model as following,

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

```
from django.contrib.auth.models import User

class UserModuleProfile(models.Model):
    user = models.OneToOneField(User)
    expired = models.DateTimeField()
    admin = models.BooleanField(default=False)
    employee_id = models.CharField(max_length=50)
    organisation_name = models.ForeignKey('Organizations', on_delete=models.PROTECT)
    country = models.CharField(max_length=100)
    position = models.CharField(max_length=100)

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

And a model form which uses this model as following,

```
from .models import UserModuleProfile, from django.contrib.auth.models import User
from django import forms

class UserProfileForm(forms.ModelForm):
    admin = forms.BooleanField(label="Make this User
Admin",widget=forms.CheckboxInput(),required=False)
    employee_id = forms.CharField(label="Employee Id ")
    organisation_name = forms.ModelChoiceField(label='Organisation
Name',required=True,queryset=Organizations.objects.all(),empty_label="Select an Organization")
    country = forms.CharField(label="Country")
    position = forms.CharField(label="Position")

    class Meta:
        model = UserModuleProfile
        fields = ('admin','employee_id','organisation_name','country','position',)

    def __init__(self, *args, **kwargs):
        admin_check = kwargs.pop('admin_check', False)
        super(UserProfileForm, self).__init__(*args, **kwargs)
        if not admin_check:
            del self.fields['admin']
```

Notice that below the Meta class in form I added a **init** function which we can use while initializing the form from views.py to delete a form field (or some other actions). I will explain this later.

So This form can be used by for user registration purposes and we want all the fields defined in the Meta class of the form. But what if we want to use the same form when we edit the user but when we do we don't want to show the admin field of the form?

We can simply send an additional argument when we initialize the form based on some logic and delete the admin field from backend.

```
def edit_profile(request,user_id):
    context = RequestContext(request)
    user = get_object_or_404(User, id=user_id)
    profile = get_object_or_404(UserModuleProfile, user_id=user_id)
    admin_check = False
    if request.user.is_superuser:
        admin_check = True
    # If it's a HTTP POST, we're interested in processing form data.
    if request.method == 'POST':
```

```
        # Attempt to grab information from the raw form information.
        profile_form =
UserProfileForm(data=request.POST,instance=profile,admin_check=admin_check)
        # If the form is valid...
        if profile_form.is_valid():
            form_bool = request.POST.get("admin", "xxx")
            if form_bool == "xxx":
                form_bool_value = False
            else:
                form_bool_value = True
            profile = profile_form.save(commit=False)
            profile.user = user
            profile.admin = form_bool_value
            profile.save()
            edited = True
        else:
            print profile_form.errors

    # Not a HTTP POST, so we render our form using ModelForm instance.
    # These forms will be blank, ready for user input.
    else:
        profile_form = UserProfileForm(instance = profile,admin_check=admin_check)

    return render_to_response(
            'usermodule/edit_user.html',
            {'id':user_id, 'profile_form': profile_form, 'edited': edited, 'user':user},
            context)
```

As you can see I have shown here a simple edit example using the form we created earlier. Notice when I initialized the form i passed an additional `admin_check` variable which contains either `True` or `False`.

```
profile_form = UserProfileForm(instance = profile,admin_check=admin_check)
```

Now If you notice the form we wrote earlier you can see that in the **init** we try to catch the `admin_check` param that we pass from here. If the value is False we simply delete the `admin` Field from the form and use it. And Since this is a model form admin field could not be null in the model we simply check if the form post had admin field in the form post, if not we set it to `False` in the view code in following code of the view.

```
form_bool = request.POST.get("admin", "xxx")
if form_bool == "xxx":
    form_bool_value = False
else:
    form_bool_value = True
```

File Uploads with Django Forms

First of all we need to add `MEDIA_ROOT` and `MEDIA_URL` to our `settings.py` file

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

Also here you will work with `ImageField`, so remember in such cases install Pillow library (`pip`

`install pillow`). Otherwise, you will have such error:

```
ImportError: No module named PIL
```

Pillow is a fork of PIL, the Python Imaging Library, which is no longer maintained. Pillow is backwards compatible with PIL.

Django comes with two form fields to upload files to the server, `FileField` and `ImageField`, the following is an example of using these two fields in our form

forms.py:

```
from django import forms


class UploadDocumentForm(forms.Form):
    file = forms.FileField()
    image = forms.ImageField()
```

views.py:

```
from django.shortcuts import render
from .forms import UploadDocumentForm


def upload_doc(request):
    form = UploadDocumentForm()
    if request.method == 'POST':
        form = UploadDocumentForm(request.POST, request.FILES)  # Do not forget to add:
request.FILES
        if form.is_valid():
            # Do something with our files or simply save them
            # if saved, our files would be located in media/ folder under the project's base
folder
            form.save()
    return render(request, 'upload_doc.html', locals())
```

upload_doc.html:

```
<html>
    <head>File Uploads</head>
    <body>
        <form enctype="multipart/form-data" action="" method="post"> <!-- Do not forget to
add: enctype="multipart/form-data" -->
            {% csrf_token %}
            {{ form }}
            <input type="submit" value="Save">
        </form>
    </body>
</html>
```

Validation of fields and Commit to model (Change user e-mail)

There are already implemented forms within Django to change the user password, one example

being SetPasswordForm.

There aren't, however, forms to modify the user e-mail and I think the following example is important to understand how to use a form correctly.

The following example performs the following checks:

- E-mail have in fact changed - very useful if you need to validate the e-mail or update mail chimp;
- Both e-mail and confirmation e-mail are the same - the form has two fields for e-mail, so the update is less error prone.

And in the end, it saves the new e-mail in the user object (updates the user e-mail). Notice that the `__init__()` method requires a user object.

```
class EmailChangeForm(forms.Form):
    """
    A form that lets a user change set their email while checking for a change in the
    e-mail.
    """
    error_messages = {
        'email_mismatch': _("The two email addresses fields didn't match."),
        'not_changed': _("The email address is the same as the one already defined."),
    }

    new_email1 = forms.EmailField(
        label=_("New email address"),
        widget=forms.EmailInput,
    )

    new_email2 = forms.EmailField(
        label=_("New email address confirmation"),
        widget=forms.EmailInput,
    )

    def __init__(self, user, *args, **kwargs):
        self.user = user
        super(EmailChangeForm, self).__init__(*args, **kwargs)

    def clean_new_email1(self):
        old_email = self.user.email
        new_email1 = self.cleaned_data.get('new_email1')
        if new_email1 and old_email:
            if new_email1 == old_email:
                raise forms.ValidationError(
                    self.error_messages['not_changed'],
                    code='not_changed',
                )
        return new_email1

    def clean_new_email2(self):
        new_email1 = self.cleaned_data.get('new_email1')
        new_email2 = self.cleaned_data.get('new_email2')
        if new_email1 and new_email2:
            if new_email1 != new_email2:
                raise forms.ValidationError(
                    self.error_messages['email_mismatch'],
                    code='email_mismatch',
```

```
                )
        return new_email2

    def save(self, commit=True):
        email = self.cleaned_data["new_email1"]
        self.user.email = email
        if commit:
            self.user.save()
        return self.user



def email_change(request):
    form = EmailChangeForm()
    if request.method=='POST':
        form = Email_Change_Form(user,request.POST)
        if form.is_valid():
            if request.user.is_authenticated:
                if form.cleaned_data['email1']  == form.cleaned_data['email2']:
                    user = request.user
                    u = User.objects.get(username=user)
                    # get the proper user
                    u.email = form.cleaned_data['email1']
                    u.save()
                    return HttpResponseRedirect("/accounts/profile/")
    else:
        return render_to_response("email_change.html", {'form':form},
                                  context_instance=RequestContext(request))
```

Read Forms online: https://riptutorial.com/django/topic/1217/forms

Chapter 24: Formsets

Syntax

- NewFormSet = formset_factory(SomeForm, extra=2)
- formset = NewFormSet(initial = [{'some_field': 'Field Value', 'other_field': 'Other Field Value',}])

Examples

Formsets with Initialized and unitialized data

`Formset` is a way to render multiple forms in one page, like a grid of data. Ex: This `ChoiceForm` might be associated with some question of sort. like, Kids are most Intelligent between which age?.

`appname/forms.py`

```
from django import forms
class ChoiceForm(forms.Form):
    choice = forms.CharField()
    pub_date = forms.DateField()
```

In your views you can use `formset_factory` constructor which takes takes `Form` as a parameter its `ChoiceForm` in this case and `extra` which describes how many extra forms other than initialized form/forms needs to be rendered, and you can loop over the `formset` object just like any other iterable.

If the formset is not initialized with data it prints the number of forms equal to `extra + 1` and if the formset is initialized it prints `initialized + extra` where `extra` number of empty forms other than initialized ones.

`appname/views.py`

```
import datetime
from django.forms import formset_factory
from appname.forms import ChoiceForm
    ChoiceFormSet = formset_factory(ChoiceForm, extra=2)
    formset = ChoiceFormSet(initial=[
      {'choice': 'Between 5-15 ?',
        'pub_date': datetime.date.today(),}
      ])
```

if you loop over `formset object` like this for form in formset: print(form.as_table())

`Output in rendered template`

```
<tr>
<th><label for="id_form-0-choice">Choice:</label></th>
<td><input type="text" name="form-0-choice" value="Between 5-15 ?" id="id_form-0-choice"
```

```
/></td>
</tr>
<tr>
<th><label for="id_form-0-pub_date">Pub date:</label></th>
<td><input type="text" name="form-0-pub_date" value="2008-05-12" id="id_form-0-pub_date"
/></td>
</tr>
<tr>
<th><label for="id_form-1-choice">Choice:</label></th>
<td><input type="text" name="form-1-choice" id="id_form-1-choice" /></td>
</tr>
<tr>
<th><label for="id_form-1-pub_date">Pub date:</label></th>
<td><input type="text" name="form-1-pub_date" id="id_form-1-pub_date" /></td
</tr>
<tr>
<th><label for="id_form-2-choice">Choice:</label></th>
<td><input type="text" name="form-2-choice" id="id_form-2-choice" /></td>
</tr>
<tr>
<th><label for="id_form-2-pub_date">Pub date:</label></th>
<td><input type="text" name="form-2-pub_date" id="id_form-2-pub_date" /></td>
</tr>
```

Read Formsets online: https://riptutorial.com/django/topic/6082/formsets

Chapter 25: Generic Views

Introduction

Generic views are views that perform a certain pre-defined action, like creating, editing or deleting objects, or simply showing a template.

Generic views have to be distinguished from functional views, which are always hand-written to perform the required tasks. In a nutshell, it can be said that generic views need to be configured, while functional views need to be programmed.

Generic views may save a lot of time, especially when you have many standardized tasks to perform.

Remarks

These examples show that generic views generally make standardized tasks much simpler. Instead of programming everything from scratch, you configure what other people have already programmed for you. This makes sense in many situations, as it allows you concentrate more on the design of your projects rather than the processes in the background.

So, should you *always* use them? No. They only make sense as long as your tasks are fairly standardizes (loading, editig, deleting objects) and the more repetitive your tasks are. Using one specific generic view only once and then override all its methods to perform very speficic tasks may not make sense. You may be better off with a functional view here.

However, if you have plenty of views that require this functionality or if your tasks match excatly the defined tasks of a specific generic view, then generic views are exactly what you need in order to make your life simpler.

Examples

Minimum Example: Functional vs. Generic Views

Example for a functional view to create an object. Excluding comments and blank lines, we need 15 lines of code:

```
# imports
from django.shortcuts import render_to_response
from django.http import HttpResponseRedirect

from .models import SampleObject
from .forms import SampleObjectForm

# view functioon
def create_object(request):
```

```
    # when request method is 'GET', show the template
    if request.method == GET:
        # perform actions, such as loading a model form
        form = SampleObjectForm()
        return render_to_response('template.html', locals())

    # if request method is 'POST', create the object and redirect
    if request.method == POST:
        form = SampleObjectForm(request.POST)

        # save object and redirect to success page if form is valid
        if form.is_valid:
            form.save()
            return HttpResponseRedirect('url_to_redirect_to')

        # load template with form and show errors
        else:
            return render_to_response('template.html', locals())
```

Example for a 'Class-Based Generic View' to perform the same task. We only need 7 lines of code to achieve the same task:

```
from django.views.generic import CreateView

from .models import SampleObject
from .forms import SampleObjectForm

class CreateObject(CreateView):
    model = SampleObject
    form_class = SampleObjectForm
    success_url = 'url_to_redirect_to'
```

Customizing Generic Views

The above example only works if your tasks are entirely standard tasks. You do not add extra context here, for example.

Let's make a more realistic example. Assume we want to add a page title to the template. In the functional view, this would work like this - with just one additional line:

```
def create_object(request):
    page_title = 'My Page Title'

    # ...

    return render_to_response('template.html', locals())
```

This is more difficult (or: counter-intutitive) to achieve with generic views. As they are class-based, you need to override one or several of the class's method to achieve the desired outcome. In our example, we need to override the class's *get_context_data* method like so:

```
class CreateObject(CreateView):
    model = SampleObject
    form_class = SampleObjectForm
```

```
    success_url = 'url_to_redirect_to'

    def get_context_data(self, **kwargs):

        # Call class's get_context_data method to retrieve context
        context = super().get_context_data(**kwargs)

        context['page_title'] = 'My page title'
        return context
```

Here, we need four additional lines to code instead of just one - at least for the *first* additional context variable we want to add.

Generic Views with Mixins

The true power of generic views unfolds when you combine them with Mixins. A mixin is a just another class defined by you whose methods can be inherited by your view class.

Assume you want every view to show the additional variable 'page_title' in the template. Instead of overriding the get_context_data method each time you define the view, you create a mixin with this method and let your views inherit from this mixin. Sounds more complicated than it actually is:

```
# Your Mixin
class CustomMixin(object):

    def get_context_data(self, **kwargs):

        # Call class's get_context_data method to retrieve context
        context = super().get_context_data(**kwargs)

        context['page_title'] = 'My page title'
        return context

# Your view function now inherits from the Mixin
class CreateObject(CustomMixin, CreateView):
    model = SampleObject
    form_class = SampleObjectForm
    success_url = 'url_to_redirect_to'

# As all other view functions which need these methods
class EditObject(CustomMixin, EditView):
    model = SampleObject
    # ...
```

The beauty of this is that your code becomes much more structured than it is mostly the case with functional views. Your entire logic behind specific tasks sits in one place and one place only. Also, you will save tremendous amounts of time especially when you have many views that always perform the same tasks, except with different objects

Read Generic Views online: https://riptutorial.com/django/topic/9452/generic-views

Chapter 26: How to reset django migrations

Introduction

As you develop a Django app, there might be situations where you can save a lot of time by just cleaning up and resetting your migrations.

Examples

Resetting Django Migration: Deleting existing database and migrating as fresh

Drop/Delete your database If you are using SQLite for your database, just delete this file. If you are using MySQL/Postgres or any other database system, you will have to drop the database and then recreate a fresh database.

You will now need to delete all the migrations file except "init.py" file located inside the migrations folder under your app folder.

Usually the migrations folder is located at

```
/your_django_project/your_app/migrations
```

Now that you have deleted the database and the migrations file, just run the following commands as you would migrate the first time you setup django project.

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

Read How to reset django migrations online: https://riptutorial.com/django/topic/9513/how-to-reset-django-migrations

Chapter 27: How to use Django with Cookiecutter?

Examples

Installing and setting up django project using Cookiecutter

Following are the Prerequisites for installing Cookiecutter:

- pip
- virtualenv
- PostgreSQL

Create a virtualenv for your project and activate it:

```
$ mkvirtualenv <virtualenv name>
$ workon <virtualenv name>
```

Now install Cookiecutter using:

```
$ pip install cookiecutter
```

Change directories into the folder where you want your project to live. Now execute the following command to generate a django project:

```
$ cookiecutter https://github.com/pydanny/cookiecutter-django.git
```

This command runs cookiecutter with the cookiecutter-django repo, asking us to enter project-specific details. Press “enter” without typing anything to use the default values, which are shown in [brackets] after the question.

```
project_name [project_name]: example_project
repo_name [example_project]:
author_name [Your Name]: Atul Mishra
email [Your email]: abc@gmail.com
description [A short description of the project.]: Demo Project
domain_name [example.com]: example.com
version [0.1.0]: 0.1.0
timezone [UTC]: UTC
now [2016/03/08]: 2016/03/08
year [2016]: 2016
use_whitenoise [y]: y
use_celery [n]: n
use_mailhog [n]: n
use_sentry [n]: n
use_newrelic [n]: n
use_opbeat [n]: n
windows [n]: n
```

```
use_python2 [n]: n
```

More details about the project generation options can be found in the offical documentation. The project is now setup.

Read How to use Django with Cookiecutter? online: https://riptutorial.com/django/topic/5385/how-to-use-django-with-cookiecutter-

Chapter 28: Internationalization

Syntax

- gettext(message)
- ngettext(singular, plural, number)
- ugettext(message)
- ungettext(singular, plural, number)
- pgettext(context, message)
- npgettext(context, singular, plural, number)
- gettext_lazy(message)
- ngettext_lazy(singular, plural, number=None)
- ugettext_lazy(message)
- ungettext_lazy(singular, plural, number=None)
- pgettext_lazy(context, message)
- npgettext_lazy(context, singular, plural, number=None)
- gettext_noop(message)
- ugettext_noop(message)

Examples

Introduction to Internationalization

Setting up

settings.py

```
from django.utils.translation import ugettext_lazy as _

USE_I18N = True  # Enable Internationalization
LANGUAGE_CODE = 'en'  # Language in which original texts are written
LANGUAGES = [  # Available languages
    ('en', _("English")),
    ('de', _("German")),
    ('fr', _("French")),
]

# Make sure the LocaleMiddleware is included, AFTER SessionMiddleware
# and BEFORE middlewares using internationalization (such as CommonMiddleware)
MIDDLEWARE_CLASSES = [
   'django.contrib.sessions.middleware.SessionMiddleware',
   'django.middleware.locale.LocaleMiddleware',
   'django.middleware.common.CommonMiddleware',
]
```

Marking strings as translatable

The first step in translation is is to *mark strings as translatable*. This is passing them through one of the `gettext` functions (See the Syntax section). For instance, here is an example model definition:

```
from django.utils.translation import ugettext_lazy as _
# It is common to import gettext as the shortcut `_` as it is often used
# several times in the same file.

class Child(models.Model):

    class Meta:
        verbose_name = _("child")
        verbose_name_plural = _("children")

    first_name = models.CharField(max_length=30, verbose_name=_("first name"))
    last_name = models.CharField(max_length=30, verbose_name=_("last name"))
    age = models.PositiveSmallIntegerField(verbose_name=_("age"))
```

All strings encapsulated in `_()` are now marked as translatable. When printed, they will always be displayed as the encapsulated string, whatever the chosen language (since no translation is available yet).

Translating strings

This example is sufficient to get started with translation. Most of the time you will only want to mark strings as translatable to **anticipate prospective internationalization** of your project. Thus, this is covered in another example.

Lazy vs Non-Lazy translation

When using non-lazy translation, strings are translated immediately.

```
>>> from django.utils.translation import activate, ugettext as _
>>> month = _("June")
>>> month
'June'
>>> activate('fr')
>>> _("June")
'juin'
>>> activate('de')
>>> _("June")
'Juni'
>>> month
'June'
```

When using laziness, translation only occurs when actually used.

```
>>> from django.utils.translation import activate, ugettext_lazy as _
```

```
>>> month = _("June")
>>> month
<django.utils.functional.lazy.<locals>.__proxy__ object at 0x7f61cb805780>
>>> str(month)
'June'
>>> activate('fr')
>>> month
<django.utils.functional.lazy.<locals>.__proxy__ object at 0x7f61cb805780>
>>> "month: {}".format(month)
'month: juin'
>>> "month: %s" % month
'month: Juni'
```

You have to use lazy translation in cases where:

- Translation may not be activated (language not selected) when `_("some string")` is evaluated
- Some strings may be evaluated only at startup (eg. in class attributes such as model and form fields definitions)

Translation in templates

To enable translation in templates you must load the `i18n` library.

```
{% load i18n %}
```

Basic translation is made with the `trans` template tag.

```
{% trans "Some translatable text" %}
{# equivalent to python `ugettext("Some translatable text")` #}
```

The `trans` template tag supports context:

```
{% trans "May" context "month" %}
{# equivalent to python `pgettext("May", "month")` #}
```

To include placeholders in your translation string, as in:

```
_("My name is {first_name} {last_name}").format(first_name="John", last_name="Doe")
```

You will have to use the `blocktrans` template tag:

```
{% blocktrans with first_name="John" last_name="Doe" %}
  My name is {{ first_name }} {{ last_name }}
{% endblocktrans %}
```

Of course instead of `"John"` and `"Doe"` you can have variables and filters:

```
{% blocktrans with first_name=user.first_name last_name=user.last_name|title %}
  My name is {{ first_name }} {{ last_name }}
{% endblocktrans %}
```

If `first_name` and `last_name` are already in your context, you can even omit the `with` clause:

```
{% blocktrans %}My name is {{ first_name }} {{ last_name }}{% endblocktrans %}
```

However, only "top-level" context variables can be use. **This will NOT work:**

```
{% blocktrans %}
    My name is {{ user.first_name }} {{ user.last_name }}
{% endblocktrans %}
```

This is mainly because the variable name is used as placeholder in translation files.

The `blocktrans` template tag also accepts pluralization.

```
{% blocktrans count nb=users|length }}
    There is {{ nb }} user.
{% plural %}
    There are {{ nb }} users.
{% endblocktrans %}
```

Finally, regardless of the `i18n` library, you can pass translatable strings to template tags using the `_("")` syntax.

```
{{ site_name|default:_("It works!") }}
{% firstof var1 var2 _("translatable fallback") %}
```

This is some magic built-in django template system to mimic a function call syntax but this ain't a function call. `_("It works!")` passed to the `default` template tag as a string `'_("It works!")'` which is then parsed a translatable string, just as `name` would be parsed as a variable and `"name"` would be parsed as a string.

Translating strings

To translate strings, you will have to create translation files. To do so, django ships with the management command `makemessages`.

```
$ django-admin makemessages -l fr
processing locale fr
```

The above command will discover all strings marked as translatable within your installed apps and create one language file for each app for french translation. For instance, if you have only one app `myapp` containing translatable strings, this will create a file `myapp/locale/fr/LC_MESSAGES/django.po`. This file may look like the following:

```
# SOME DESCRIPTIVE TITLE
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR
#
#, fuzzy
```

```
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2016-07-24 14:01+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: myapp/models.py:22
msgid "user"
msgstr ""

#: myapp/models.py:39
msgid "A user already exists with this email address."
msgstr ""

#: myapp/templates/myapp/register.html:155
#, python-format
msgid ""
"By signing up, you accept our <a href=\"%(terms_url)s\" "
"target=_blank>Terms of services</a>."
msgstr ""
```

You will first have to fill in the placeholders (emphasized with uppercases). Then translate the strings. `msgid` is the string marked as translatable in your code. `msgstr` is where you have to write the translation of the string right above.

When a string contains placeholders, you will have to include them in your translation as well. For instance, you will translate the latest message as the following:

```
#: myapp/templates/myapp/register.html:155
#, python-format
msgid ""
"By signing up, you accept our <a href=\"%(terms_url)s\" "
"target=_blank>Terms of services</a>."
msgstr ""
"En vous inscrivant, vous acceptez nos <a href=\"%(terms_url)s\" "
"target=_blank>Conditions d'utilisation</a>"
```

Once your translation file is completed, you will have to compile the `.po` files into `.mo` files. This is done by calling the `compilemessages` management command:

```
$ django-admin compilemessages
```

That's it, now translations are available.

To update your translation files when you make changes to your code, you can rerun `django-admin makemessages -l fr`. This will update `.po` files, keeping your existing translations and adding the new ones. Deleted strings will still be available in comments. To update `.po` files for all languages, run `django-admin makemessages -a`. Once your `.po` files are updated, don't forget to run `django-admin`

`compilemessages` again to generate `.mo` files.

Noop use case

`(u)gettext_noop` allows you to mark a string as translatable without actually translating it.

A typical use case is when you want to log a message for developers (in English) but also want to display it to the client (in the requested language). **You can pass a variable to `gettext`, but its content won't be discovered as a translatable string because it is, per definition, variable.**.

```
# THIS WILL NOT WORK AS EXPECTED
import logging
from django.contrib import messages

logger = logging.getLogger(__name__)

error_message = "Oops, something went wrong!"
logger.error(error_message)
messages.error(request, _(error_message))
```

The error message won't appear in the `.po` file and you will have to remember it exists to add it manually. To fix this, you can use `gettext_noop`.

```
error_message = ugettext_noop("Oops, something went wrong!")
logger.error(error_message)
messages.error(request, _(error_message))
```

Now the string `"Oops, something went wrong!"` will be discovered and available in the `.po` file when generated. And the error will still be logged in English for developers.

Common pitfalls

fuzzy translations

Sometimes `makemessages` may think that the string it found for translation is somewhat similar to already existing translation. It will when mark it in the `.po` file with a special `fuzzy` comment like this:

```
#: templates/randa/map.html:91
#, fuzzy
msgid "Country"
msgstr "Länderinfo"
```

Even if translation is correct or you updated it to correct one it will not be used to translate your project unless you remove `fuzzy` comment line.

Multiline strings

`makemessages` parses files in various formats, from plain text to python code and it is not designed to follow every possible rule for having multi-line strings in those formats. Most of the time it will work

just fine with single line strings but if you have construction like this:

```
translation = _("firstline"
"secondline"
"thirdline")
```

It will only pick up `firstline` for translation. Solution for this is to avoid using multiline strings when possible.

Read Internationalization online: https://riptutorial.com/django/topic/2579/internationalization

Chapter 29: JSONField - a PostgreSQL specific field

Syntax

- JSONField(**options)

Remarks

- Django's `JSONField` actually stores the data in a Postgres `JSONB` column, which is only available in Postgres 9.4 and later.
- `JSONField` is great when you want a more flexible schema. For example if you want to change the keys without having to do any data migrations, or if not all your objects have the same structure.
- If you're storing data with static keys, consider using multiple normal fields instead of `JSONField`s instead, as querying `JSONField` can get quite tedious sometimes.

Chaining queries

You can chain queries together. For example, if a dictionary exists inside a list, add two underscores and your dictionary query.

> Don't forget to separate queries with double underscores.

Examples

Creating a JSONField

Available in Django 1.9+

```
from django.contrib.postgres.fields import JSONField
from django.db import models

class IceCream(models.Model):
    metadata = JSONField()
```

You can add the normal `**options` if you wish.

> ***!*** Note that you must put `'django.contrib.postgres'` in `INSTALLED_APPS` in your `settings.py`

Creating an object with data in a JSONField

Pass data in native Python form, for example `list`, `dict`, `str`, `None`, `bool`, etc.

```
IceCream.objects.create(metadata={
    'date': '1/1/2016',
    'ordered by': 'Jon Skeet',
    'buyer': {
         'favorite flavor': 'vanilla',
         'known for': ['his rep on SO', 'writing a book']
     },
    'special requests': ['hot sauce'],
})
```

See the note in the "Remarks" section about using `JSONField` in practice.

Querying top-level data

```
IceCream.objects.filter(metadata__ordered_by='Guido Van Rossum')
```

Querying data nested in dictionaries

Get all ice cream cones that were ordered by people liking chocolate:

```
IceCream.objects.filter(metadata__buyer__favorite_flavor='chocolate')
```

See the note in the "Remarks" section about chaining queries.

Querying data present in arrays

An integer will be interpreted as an index lookup.

```
IceCream.objects.filter(metadata__buyer__known_for__0='creating stack overflow')
```

See the note in the "Remarks" section about chaining queries.

Ordering by JSONField values

Ordering directly on `JSONField` is not yet supported in Django. But it's possible via RawSQL using PostgreSQL functions for jsonb:

```
from django.db.models.expressions import RawSQL
RatebookDataEntry.objects.all().order_by(RawSQL("data->>%s", ("json_objects_key",)))
```

This example orders by `data['json_objects_key']` inside `JSONField` named `data`:

```
data = JSONField()
```

Read JSONField - a PostgreSQL specific field online:
https://riptutorial.com/django/topic/1759/jsonfield---a-postgresql-specific-field

Chapter 30: Logging

Examples

Logging to Syslog service

It is possible to configure Django to output log to a local or remote syslog service. This configuration uses the python builtin SysLogHandler.

```
from logging.handlers import SysLogHandler
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'standard': {
            'format' : "[YOUR PROJECT NAME] [%(asctime)s] %(levelname)s [%(name)s:%(lineno)s]
%(message)s",
            'datefmt' : "%d/%b/%Y %H:%M:%S"
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
        'syslog': {
            'class': 'logging.handlers.SysLogHandler',
            'formatter': 'standard',
            'facility': 'user',
            # uncomment next line if rsyslog works with unix socket only (UDP reception
disabled)
            #'address': '/dev/log'
        }
    },
    'loggers': {
        'django':{
            'handlers': ['syslog'],
            'level': 'INFO',
            'disabled': False,
            'propagate': True
        }
    }
}

# loggers for my apps, uses INSTALLED_APPS in settings
# each app must have a configured logger
# level can be changed as desired: DEBUG, INFO, WARNING...
MY_LOGGERS = {}
for app in INSTALLED_APPS:
    MY_LOGGERS[app] = {
        'handlers': ['syslog'],
        'level': 'DEBUG',
        'propagate': True,
    }
LOGGING['loggers'].update(MY_LOGGERS)
```

Django basic logging configuration

Internally, Django uses the Python logging system. There is many way to configure the logging of a project. Here is a base:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': "[%(asctime)s] %(levelname)s [%(name)s:%(lineno)s] %(message)s",
            'datefmt': "%Y-%m-%d %H:%M:%S"
        },
    },
    'handlers': {
        'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'default'
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
            'propagate': True,
            'level': 'INFO',
        },
    }
}
```

Formatters

It can be used to configure logs appearence when they are printed to output. You can define many formatters by setting a key string to each different formatter. A formatter is then used when declaring a handler.

Handlers

Can be used to configure where the logs will be printed. In the example above, they are sent to stdout and stderr. There is various handler classes:

```
'rotated_logs': {
    'class': 'logging.handlers.RotatingFileHandler',
    'filename': '/var/log/my_project.log',
    'maxBytes': 1024 * 1024 * 5,  # 5 MB
    'backupCount': 5,
    'formatter': 'default'
    'level': 'DEBUG',
},
```

This will produce logs in file tergeted by `filename`. In this example, a new log file will be created when the current reach the size of 5 MB (the old one is renamed to my_project.log.1) and the latest 5 files will be kept for archive.

```
'mail_admins': {
    'level': 'ERROR',
    'class': 'django.utils.log.AdminEmailHandler'
},
```

This will send each log by eamil to users specified in `ADMINS` setting variable. The level is set to `ERROR`, so only logs with level `ERROR` will be sent by e-mail. This is extremely useful to stay informed on potential errors 50x on a production server.

Other handlers can be used with Django. For a full list, please read the corresponding documentation. Like formatters, you can define many handlers in a same project, setting for each a different key string. Each handler can be used in a specific logger.

Loggers

In `LOGGING`, the last part configure for each module the minimal logging level, the handlers(s) to use, etc.

Read Logging online: https://riptutorial.com/django/topic/1231/logging

Chapter 31: Management Commands

Introduction

Management commands are powerful and flexible scripts that can perform actions on your Django project or the underlying database. In addition to various default commands, it's possible to write your own!

Compared to regular Python scripts, using the management command framework means that some tedious setup work is automatically done for you behind the scenes.

Remarks

Management commands can be called either from:

- `django-admin <command> [options]`
- `python -m django <command> [options]`
- `python manage.py <command> [options]`
- `./manage.py <command> [options]` if manage.py has execution permissions (`chmod +x manage.py`)

To use management commands with Cron:

```
*/10 * * * * pythonuser /var/www/dev/env/bin/python /var/www/dev/manage.py <command> [options]
> /dev/null
```

Examples

Creating and Running a Management Command

To perform actions in Django using commandline or other services (where the user/request is not used), you can use the `management commands`.

Django modules can be imported as needed.

For each command a separate file needs to be created: `myapp/management/commands/my_command.py` (The `management` and `commands` directories must have an empty __init__.py file)

```
from django.core.management.base import BaseCommand, CommandError

# import additional classes/modules as needed
# from myapp.models import Book

class Command(BaseCommand):
    help = 'My custom django management command'

    def add_arguments(self, parser):
       parser.add_argument('book_id', nargs='+', type=int)
       parser.add_argument('author' , nargs='+', type=str)
```

```
    def handle(self, *args, **options):
       bookid = options['book_id']
       author = options['author']
       # Your code goes here

        # For example:
        # books = Book.objects.filter(author="bob")
        # for book in books:
        #    book.name = "Bob"
        #    book.save()
```

Here class name **Command** is mandatory which extends **BaseCommand** or one of its subclasses.

The name of the management command is the name of the file containing it. To run the command in the example above, use the following in your project directory:

```
python manage.py my_command
```

> Note that starting a command can take a few second (because of the import of the modules). So in some cases it is advised to create `daemon` processes instead of `management commands`.

More on management commands

Get list of existing commands

You can get list of available commands by following way:

```
>>> python manage.py help
```

If you don't understand any command or looking for optional arguments then you can use **-h** argument like this

```
>>> python manage.py command_name -h
```

Here command_name will be your desire command name, this will show you help text from the command.

```
>>> python manage.py runserver -h
>>> usage: manage.py runserver [-h] [--version] [-v {0,1,2,3}]
                           [--settings SETTINGS] [--pythonpath PYTHONPATH]
                           [--traceback] [--no-color] [--ipv6] [--nothreading]
                           [--noreload] [--nostatic] [--insecure]
                           [addrport]

Starts a lightweight Web server for development and also serves static files.

positional arguments:
  addrport              Optional port number, or ipaddr:port
```

```
optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -v {0,1,2,3}, --verbosity {0,1,2,3}
                        Verbosity level; 0=minimal output, 1=normal output,
                        2=verbose output, 3=very verbose output
  --settings SETTINGS   The Python path to a settings module, e.g.
                        "myproject.settings.main". If this isn't provided, the
                        DJANGO_SETTINGS_MODULE environment variable will be
                        used.
  --pythonpath PYTHONPATH
                        A directory to add to the Python path, e.g.
                        "/home/djangoprojects/myproject".
  --traceback           Raise on CommandError exceptions
  --no-color            Don't colorize the command output.
  --ipv6, -6            Tells Django to use an IPv6 address.
  --nothreading         Tells Django to NOT use threading.
  --noreload            Tells Django to NOT use the auto-reloader.
  --nostatic            Tells Django to NOT automatically serve static files
                        at STATIC_URL.
  --insecure            Allows serving static files even if DEBUG is False.
```

List of available command

Using django-admin instead of manage.py

You can get rid of `manage.py` and use the `django-admin` command instead. To do so, you will have to manually do what `manage.py` does:

- Add your project path to your PYTHONPATH
- Set the DJANGO_SETTINGS_MODULE

```
export PYTHONPATH="/home/me/path/to/your_project"
export DJANGO_SETTINGS_MODULE="your_project.settings"
```

This is especially useful in a virtualenv where you can set these environment variables in the `postactivate` script.

`django-admin` command has the advantage of being available wherever you are on your filesystem.

Builtin Management Commands

Django comes with a number of builtin management commands, using `python manage.py [command]` or, when manage.py has +x (executable) rights simply `./manage.py [command]` . The following are some of the most frequently used:

Get a list of all available commands

```
./manage.py help
```

Run your Django server on localhost:8000; essential for local testing

```
./manage.py runserver
```

Run a python (or ipython if installed) console with the Django settings of your project preloaded (attempting to access parts of your project in a python terminal without doing this will fail).

```
./manage.py shell
```

Create a new database migration file based on the changes you have made to your models. See Migrations

```
./manage.py makemigrations
```

Apply any unapplied migrations to the current database.

```
./manage.py migrate
```

Run your project's test suite. See Unit Testing

```
./manage.py test
```

Take all of the static files of your project and puts them in the folder specified in `STATIC_ROOT` so they can be served in production.

```
./manage.py collectstatic
```

Allow to create superuser.

```
./manage.py createsuperuser
```

Change the password of a specified user.

```
./manage.py changepassword username
```

Full list of available commands

Read Management Commands online: https://riptutorial.com/django/topic/1661/management-commands

Chapter 32: Many-to-many relationships

Examples

With a through model

```
class Skill(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Developer(models.Model):
    name = models.CharField(max_length=50)
    skills = models.ManyToManyField(Skill, through='DeveloperSkill')

class DeveloperSkill(models.Model):
    """Developer skills with respective ability and experience."""

    class Meta:
        order_with_respect_to = 'developer'
        """Sort skills per developer so that he can choose which
        skills to display on top for instance.
        """
        unique_together = [
            ('developer', 'skill'),
        ]
        """It's recommended that a together unique index be created on
        `(developer,skill)`. This is especially useful if your database is
        being access/modified from outside django. You will find that such an
        index is created by django when an explicit through model is not
        being used.
        """


    ABILITY_CHOICES = [
        (1, "Beginner"),
        (2, "Accustomed"),
        (3, "Intermediate"),
        (4, "Strong knowledge"),
        (5, "Expert"),
    ]

    developer = models.ForeignKey(Developer, models.CASCADE)
    skill = models.ForeignKey(Skill, models.CASCADE)
    """The many-to-many relation between both models is made by the
    above two foreign keys.

    Other fields (below) store information about the relation itself.
    """

    ability = models.PositiveSmallIntegerField(choices=ABILITY_CHOICES)
    experience = models.PositiveSmallIntegerField(help_text="Years of experience.")
```

It's recommended that a together unique index be created on `(developer,skill)`. This is especially useful if your database is being access/modified from outside django. You will find that such an index is created by django when an explicit through model is not being used.

Simple Many To Many Relationship.

```
class Person(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Club(models.Model):
    name = models.CharField(max_length=50)
    members = models.ManyToManyField(Person)
```

Here we define a relationship where a club has many `Person`s and members and a Person can be a member of several different `Club`s.

Though we define only two models, django actually creates three tables in the database for us. These are `myapp_person`, `myapp_club` and myapp_club_members. Django automatically creates a unique index on `myapp_club_members(club_id,person_id)` columns.

Using ManyToMany Fields

We use this model from the first example:

```
class Person(models.Model):
    name = models.CharField(max_length=50)
    description = models.TextField()

class Club(models.Model):
    name = models.CharField(max_length=50)
    members = models.ManyToManyField(Person)
```

Add Tom and Bill to the Nightclub:

```
tom = Person.objects.create(name="Tom", description="A nice guy")
bill = Person.objects.create(name="Bill", description="Good dancer")

nightclub = Club.objects.create(name="The Saturday Night Club")
nightclub.members.add(tom, bill)
```

Who is in the club?

```
for person in nightclub.members.all():
    print(person.name)
```

Will give you

```
Tom
Bill
```

Read Many-to-many relationships online: https://riptutorial.com/django/topic/2379/many-to-many-relationships

Chapter 33: Mapping strings to strings with HStoreField - a PostgreSQL specific field

Syntax

- FooModel.objects.filter(field_name__key_name='value to query')

Remarks

Examples

Setting up HStoreField

First, we'll need to do some setup to get `HStoreField` working.

1. make sure `django.contrib.postgres` is in your `INSTALLED_APPS
2. Add `HStoreExtension` to your migrations. Remember to put `HStoreExtension` before any `CreateModel` or `AddField` migrations.

```
from django.contrib.postgres.operations import HStoreExtension
from django.db import migrations

class FooMigration(migrations.Migration):
    # put your other migration stuff here
    operations = [
        HStoreExtension(),
        ...
    ]
```

Adding HStoreField to your model

> -> Note: make sure you set up `HStoreField` first before going on with this example. (above)

No parameters are required for initializing a `HStoreField`.

```
from django.contrib.postgres.fields import HStoreField
from django.db import models

class Catalog(models.model):
    name = models.CharField(max_length=200)
    titles_to_authors = HStoreField()
```

Creating a new model instance

Pass a native python dictionary mapping strings to strings to `create()`.

```
Catalog.objects.create(name='Library of Congress', titles_to_authors={
    'Using HStoreField with Django': 'CrazyPython and la communidad',
    'Flabbergeists and thingamajigs': 'La Artista Fooista',
    'Pro Git': 'Scott Chacon and Ben Straub',
})
```

Performing key lookups

```
Catalog.objects.filter(titles__Pro_Git='Scott Chacon and Ben Straub')
```

Using contains

Pass a `dict` object to `field_name__contains` as a keyword argument.

```
Catalog.objects.filter(titles__contains={
        'Pro Git': 'Scott Chacon and Ben Straub'})
```

Equivalent to the SQL operator `@>`.

Read Mapping strings to strings with HStoreField - a PostgreSQL specific field online: https://riptutorial.com/django/topic/2670/mapping-strings-to-strings-with-hstorefield---a-postgresql-specific-field

Chapter 34: Meta: Documentation Guidelines

Remarks

This is an extension of Python's "Meta: Documentation Guidelines" for Django.

These are just proposals, not recommendations. Feel free to edit anything here if you disagree or have something else to mention.

Examples

Unsupported versions don't need special mention

It is unlikely that someone uses an unsupported version of Django, and at his own risks. If ever someone does, it must be his concern to know if a feature exists in the given version.

Considering the above, it is useless to mention specificities of an unsupported version.

1.6

This kind of block is useless because no sane person uses Django < 1.6.

1.8

This kind of block is useless because no sane person uses Django < 1.8.

This also goes for topics. At the time of writing this example, Class based views states supported versions are `1.3-1.9`. We can safely assume this is actually equivalent to `All versions`. This also avoids upgrading all topics supported versions every time a new version is released.

> Current supported versions are: `1.8`[1] `1.9`[2] `1.10`[1]

1. Security fixes, data loss bugs, crashing bugs, major functionality bugs in newly-introduced features, and regressions from older versions of Django.
2. Security fixes and data loss bugs.

Read Meta: Documentation Guidelines online: https://riptutorial.com/django/topic/5243/meta--documentation-guidelines

Chapter 35: Middleware

Introduction

Middleware in Django is a framework that allows code to hook into the response / request processing and alter the input or output of Django.

Remarks

Middleware needs to be added to your settings.py `MIDDLEWARE_CLASSES` list before it will be included in execution. The default list that Django provides when creating a new project is as follows:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

These are all functions that will run in **order** on every request (once before it reaches your view code in `views.py` and once in reverse order for `process_response` callback, before version 1.10). They do a variety of things such as injecting the Cross Site Request Forgery (csrf) token.

The order matters because if some middleware does a redirect, then the all the subsequent middleware will never run. Or if a middleware expects the csrf token to be there, it has to run after the `CsrfViewMiddleware`.

Examples

Add data to requests

Django makes it really easy to add additional data onto requests for use within the view. For example, we can parse out the subdomain on the request's META and attach it as a separate property on the request by using middleware.

```
class SubdomainMiddleware:
    def process_request(self, request):
        """
        Parse out the subdomain from the request
        """
        host = request.META.get('HTTP_HOST', '')
        host_s = host.replace('www.', '').split('.')
        request.subdomain = None
        if len(host_s) > 2:
```

```
            request.subdomain = host_s[0]
```

If you add data with middleware to your request, you can access that newly added data further down the line. Here we'll use the parsed subdomain to determine something like what organization is accessing your application. This approach is useful for apps that are deployed with a DNS setup with wildcard subdomains that all point to a single instance and the person accessing the app wants a skinned version dependent on the access point.

```
class OrganizationMiddleware:
    def process_request(self, request):
        """
        Determine the organization based on the subdomain
        """
        try:
            request.org = Organization.objects.get(domain=request.subdomain)
        except Organization.DoesNotExist:
            request.org = None
```

Remember that order matters when having middleware depend on one another. For requests, you'll want the dependent middleware to be placed after the dependency.

```
MIDDLEWARE_CLASSES = [
    ...
    'myapp.middleware.SubdomainMiddleware',
    'myapp.middleware.OrganizationMiddleware',
    ...
]
```

Middleware to filter by IP address

First: The path structure

If you don't have it you need to create the **middleware** folder within your app following the structure:

```
yourproject/yourapp/middleware
```

The folder middleware should be placed in the same folder as settings.py, urls, templates...

Important: Don't forget to create the init.py empty file inside the middleware folder so your app recognizes this folder

Instead of having a separate folder containing your middleware classes, it is also possible to put your functions in a single file, `yourproject/yourapp/middleware.py`.

Second: Create the middleware

Now we should create a file for our custom middleware. In this example let's suppose we want a middleware that filter the users based on their IP address, we create a file called **filter_ip_middleware.py**:

```
#yourproject/yourapp/middleware/filter_ip_middleware.py
from django.core.exceptions import PermissionDenied

class FilterIPMiddleware(object):
    # Check if client IP address is allowed
    def process_request(self, request):
        allowed_ips = ['192.168.1.1', '123.123.123.123', etc...] # Authorized ip's
        ip = request.META.get('REMOTE_ADDR') # Get client IP address
        if ip not in allowed_ips:
            raise PermissionDenied # If user is not allowed raise Error

       # If IP address is allowed we don't do anything
       return None
```

Third: Add the middleware in our 'settings.py'

We need to look for the `MIDDLEWARE_CLASSES` inside the settings.py and there we need to add our middleware (*Add it in the last position*). It should be like:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
     # Above are Django standard middlewares

     # Now we add here our custom middleware
     'yourapp.middleware.filter_ip_middleware.FilterIPMiddleware'
)
```

Done! Now every request from every client will call your custom middleware and process your custom code!

Globally handling exception

Say you have implemented some logic to detect attempts to modify an object in the database while the client that submitted changes didn't have the latest modifications. If such case happens, you raise a custom exception `ConfictError(detailed_message)`.

Now you want to return an HTTP 409 (Confict) status code when this error occurs. You may typically use as middleware for this instead of handling it in each view that might raise this exception.

```
class ConfictErrorHandlingMiddleware:
    def process_exception(self, request, exception):
        if not isinstance(exception, ConflictError):
            return  # Propagate other exceptions, we only handle ConflictError
        context = dict(confict_details=str(exception))
        return TemplateResponse(request, '409.html', context, status=409)
```

Understanding Django 1.10 middleware's new style

Django 1.10 introduced a new middleware style where `process_request` and `process_response` are merged together.

In this new style, *a middleware is a callable that returns another callable*. Well, actually the **former is a middleware factory** and **the latter is the actual middleware**.

The *middleware factory* takes as single argument the next *middleware* in the middlewares stack, or the view itself when the bottom of the stack is reached.

The *middleware* takes the request as single argument and **always returns an `HttpResponse`.**

The best example to illustrate how new-style *middleware* works is probably to show how to make a backward-compatible *middleware*:

```
class MyMiddleware:

    def __init__(self, next_layer=None):
        """We allow next_layer to be None because old-style middlewares
        won't accept any argument.
        """
        self.get_response = next_layer

    def process_request(self, request):
        """Let's handle old-style request processing here, as usual."""
        # Do something with request
        # Probably return None
        # Or return an HttpResponse in some cases

    def process_response(self, request, response):
        """Let's handle old-style response processing here, as usual."""
        # Do something with response, possibly using request.
        return response

    def __call__(self, request):
        """Handle new-style middleware here."""
        response = self.process_request(request)
        if response is None:
            # If process_request returned None, we must call the next middleware or
            # the view. Note that here, we are sure that self.get_response is not
            # None because this method is executed only in new-style middlewares.
            response = self.get_response(request)
        response = self.process_response(request, response)
        return response
```

Read Middleware online: https://riptutorial.com/django/topic/1721/middleware

Chapter 36: Migrations

Parameters

`django-admin` command	Details
`makemigrations <my_app>`	Generate migrations for `my_app`
`makemigrations`	Generate migrations for all apps
`makemigrations --merge`	Resolve migration conflicts for all apps
`makemigrations --merge <my_app>`	Resolve migration conflicts for `my_app`
`makemigrations --name <migration_name> <my_app>`	Generate a migration for `my_app` with the name `migration_name`
`migrate <my_app>`	Apply pending migrations of `my_app` to the database
`migrate`	Apply all pending migrations to the database
`migrate <my_app> <migration_name>`	Apply or unapply up to `migration_name`
`migrate <my_app> zero`	Unapply all migrations in `my_app`
`sqlmigrate <my_app> <migration_name>`	Prints the SQL for the named migration
`showmigrations`	Shows all migrations for all apps
`showmigrations <my_app>`	Shows all migrations in `my_app`

Examples

Working with migrations

Django uses migrations to propagate changes you make to your models to your database. Most of the time django can generate them for you.

To create a migration, run:

```
$ django-admin makemigrations <app_name>
```

This will create a migration file in the `migration` submodule of `app_name`. The first migration will be named `0001_initial.py`, the other will start with `0002_`, then `0003`, ...

If you omit `<app_name>` this will create migrations for all your `INSTALLED_APPS`.

To propagate migrations to your database, run:

```
$ django-admin migrate <app_name>
```

To show all your migrations, run:

```
$ django-admin showmigrations app_name
app_name
  [X] 0001_initial
  [X] 0002_auto_20160115_1027
  [X] 0003_somemodel
  [ ] 0004_auto_20160323_1826
```

- `[X]` means that the migration was propagated to your database
- `[ ]` means that the migration was not propagated to your database. Use `django-admin migrate` to propagate it

You call also revert migrations, this can be done by passing the migration name to the `migrate command`. Given the above list of migrations (shown by `django-admin showmigrations`):

```
$ django-admin migrate app_name 0002  # Roll back to migration 0002
$ django-admin showmigrations app_name
app_name
  [X] 0001_initial
  [X] 0002_auto_20160115_1027
  [ ] 0003_somemodel
  [ ] 0004_auto_20160323_1826
```

Manual migrations

Sometimes, migrations generated by Django are not sufficient. This is especially true when you want to make **data migrations**.

For instance, let's you have such model:

```
class Article(models.Model):
    title = models.CharField(max_length=70)
```

This model already have existing data and now you want to add a `SlugField`:

```
class Article(models.Model):
    title = models.CharField(max_length=70)
    slug = models.SlugField(max_length=70)
```

You created the migrations to add the field, but now you would like to set the slug for all existing article, according to their `title`.

Of course, you could just do something like this in the terminal:

```
$ django-admin shell
>>> from my_app.models import Article
>>> from django.utils.text import slugify
>>> for article in Article.objects.all():
...     article.slug = slugify(article.title)
...     article.save()
...
>>>
```

But you will have to do this in all your environments (ie. your office desktop, your laptop, ...), all your coworkers will have to do so as well, and you will have to think about it on staging and when pushing live.

To make it once and for all, we will do it in a migration. First create an empty migration:

```
$ django-admin makemigrations --empty app_name
```

This will create an empty migration file. Open it, it contains an base skeleton. Let's say your previous migration was named `0023_article_slug` and this one is named `0024_auto_20160719_1734`. Here is what we will write in our migration file:

```
# -*- coding: utf-8 -*-
# Generated by Django 1.9.7 on 2016-07-19 15:34
from __future__ import unicode_literals

from django.db import migrations
from django.utils.text import slugify


def gen_slug(apps, schema_editor):
    # We can't import the Article model directly as it may be a newer
    # version than this migration expects. We use the historical version.
    Article = apps.get_model('app_name', 'Article')
    for row in Article.objects.all():
        row.slug = slugify(row.name)
        row.save()


class Migration(migrations.Migration):

    dependencies = [
        ('hosting', '0023_article_slug'),
    ]

    operations = [
        migrations.RunPython(gen_slug, reverse_code=migrations.RunPython.noop),
        # We set `reverse_code` to `noop` because we cannot revert the migration
        # to get it back in the previous state.
        # If `reverse_code` is not given, the migration will not be reversible,
        # which is not the behaviour we expect here.
    ]
```

Fake migrations

When a migration is run, Django stores the name of the migration in a django_migrations table.

Create and Fake initial migrations for existing schema

If your app already has models and database tables, and doesn't have migrations. First create initial migrations for you app.

```
python manage.py makemigrations your_app_label
```

Now fake initial migrations as applied

```
python manage.py migrate --fake-initial
```

Fake all migrations in all apps

```
python manage.py migrate --fake
```

Fake single app migrations

```
python manage.py migrate --fake core
```

Fake single migration file

```
python manage.py migrate myapp migration_name
```

Custom names for migration files

Use the `makemigrations --name <your_migration_name>` option to allow naming the migrations(s) instead of using a generated name.

```
python manage.py makemigrations --name <your_migration_name> <app_name>
```

Solving migration conflicts

Introduction

Sometimes migrations conflict, resulting in making the migration unsuccesful. This can happen in a lot of scenerio's, however it can occur on a regular basis when developing one app with a team.

Common migration conflicts happen while using source control, especially when the feature-per-branch method is used. For this scenario we will use a model called `Reporter` with the attributes `name` and `address`.

Two developers at this point are going to develop a feature, thus they both get this initial copy of the `Reporter` model. Developer A adds an `age` which results in the file `0002_reporter_age.py` file. Developer B adds a `bank_account` field which resulsts in `0002_reporter_bank_account`. Once these developers merge their code together and attempt to migrate the migrations, a migration conflict occurred.

This conflict occurs because these migrations both alter the same model, `Reporter`. On top of that, the new files both start with 0002.

Merging migrations

There are several ways of doing it. The following is in the recommended order:

1. The most simple fix for this is by running the makemigrations command with a --merge flag.

   ```
   python manage.py makemigrations --merge <my_app>
   ```

   This will create a new migration solving the previous conflict.

2. When this extra file is not welcome in the development environment for personal reasons, an option is to delete the conflicting migrations. Then, a new migration can be made using the regular `makemigrations` command. When custom migrations are written, such as `migrations.RunPython`, need to be accounted for using this method.

Change a CharField to a ForeignKey

First off, let's assume this is your initial model, inside an application called `discography`:

```
from django.db import models

class Album(models.Model):
    name = models.CharField(max_length=255)
    artist = models.CharField(max_length=255)
```

Now, you realize that you want to use a ForeignKey for the artist instead. This is a somewhat complex process, which has to be done in several steps.

Step 1, add a new field for the ForeignKey, making sure to mark it as null (note that the model we are linking to is also now included):

```
from django.db import models

class Album(models.Model):
    name = models.CharField(max_length=255)
    artist = models.CharField(max_length=255)
    artist_link = models.ForeignKey('Artist', null=True)

class Artist(models.Model):
    name = models.CharField(max_length=255)
```

...and create a migration for this change.

```
./manage.py makemigrations discography
```

Step 2, populate your new field. In order to do this, you have to create an empty migration.

```
./manage.py makemigrations --empty --name transfer_artists discography
```

Once you have this empty migration, you want to add a single `RunPython` operation to it in order to link your records. In this case, it could look something like this:

```
def link_artists(apps, schema_editor):
    Album = apps.get_model('discography', 'Album')
    Artist = apps.get_model('discography', 'Artist')
    for album in Album.objects.all():
        artist, created = Artist.objects.get_or_create(name=album.artist)
        album.artist_link = artist
        album.save()
```

Now that your data is transferred to the new field, you could actually be done and leave everything as is, using the new `artist_link` field for everything. Or, if you want to do a bit of cleanup, you want to create two more migrations.

For your first migration, you will want to delete your original field, `artist`. For your second migration, rename the new field `artist_link` to `artist`.

This is done in multiple steps to ensure that Django recognizes the operations properly.

Read Migrations online: https://riptutorial.com/django/topic/1200/migrations

Chapter 37: Model Aggregations

Introduction

Aggregations are methods allowing the execution of operations on (individual and/or groups of) rows of objects derived from a Model.

Examples

Average, Minimum, Maximum, Sum from Queryset

```
class Product(models.Model):
    name = models.CharField(max_length=20)
    price = models.FloatField()
```

To Get average price of all products:

```
>>> from django.db.models import Avg, Max, Min, Sum
>>> Product.objects.all().aggregate(Avg('price'))
# {'price__avg': 124.0}
```

To Get Minimum price of all products:

```
>>> Product.objects.all().aggregate(Min('price'))
# {'price__min': 9}
```

To Get Maximum price of all products:

```
>>> Product.objects.all().aggregate(Max('price'))
# {'price__max':599 }
```

To Get SUM of prices of all products:

```
>>> Product.objects.all().aggregate(Sum('price'))
# {'price__sum':92456 }
```

Count the number of foreign relations

```
class Category(models.Model):
    name = models.CharField(max_length=20)


class Product(models.Model):
    name = models.CharField(max_length=64)
    category = models.ForeignKey(Category, on_delete=models.PROTECT)
```

To get the number products for each category:

```
>>> categories = Category.objects.annotate(Count('product'))
```

This adds the `<field_name>__count` attribute to each instance returned:

```
>>> categories.values_list('name', 'product__count')
[('Clothing', 42), ('Footwear', 12), ...]
```

You can provide a custom name for your attribute by using a keyword argument:

```
>>> categories = Category.objects.annotate(num_products=Count('product'))
```

You can use the annotated field in querysets:

```
>>> categories.order_by('num_products')
[<Category: Footwear>, <Category: Clothing>]

>>> categories.filter(num_products__gt=20)
[<Category: Clothing>]
```

GROUB BY ... COUNT/SUM Django ORM equivalent

We can perform a `GROUP BY ... COUNT` or a `GROUP BY ... SUM` SQL equivalent queries on Django ORM, with the use of `annotate()`, `values()`, `order_by()` and the `django.db.models`'s `Count` and `Sum` methods respectfully:

Let our model be:

```
   class Books(models.Model):
       title  = models.CharField()
       author = models.CharField()
       price = models.FloatField()
```

`GROUP BY ... COUNT`:

- Lets assume that we want to count how many book objects per distinct author exist in our `Books` table:

  ```
  result = Books.objects.values('author')
                        .order_by('author')
                        .annotate(count=Count('author'))
  ```

- Now `result` contains a queryset with two columns: `author` and `count`:

  ```
    author    | count
  ------------|-------
   OneAuthor  |   5
  OtherAuthor |   2
     ...      |  ...
  ```

GROUB BY ... SUM:

- Lets assume that we want to sum the price of all the books per distinct author that exist in our `Books` table:

```
result = Books.objects.values('author')
                      .order_by('author')
                      .annotate(total_price=Sum('price'))
```

- Now `result` contains a queryset with two columns: `author` and `total_price`:

```
  author     | total_price
-------------|-------------
 OneAuthor   |    100.35
OtherAuthor  |     50.00
    ...      |      ...
```

Read Model Aggregations online: https://riptutorial.com/django/topic/3775/model-aggregations

Chapter 38: Model Field Reference

Parameters

Parameter	Details
null	If true, empty values may be stored as `null` in the database
blank	If true, then the field will not be required in forms. If fields are left blank, Django will use the default field value.
choices	An iterable of 2-element iterables to be used as choices for this field. If set, field is rendered as a drop-down in the admin. `[('m', 'Male'),('f','Female'),('z','Prefer Not to Disclose')]`. To group options, simply nest the values: `[('Video Source',((1,'YouTube'),(2,'Facebook')),('Audio Source',((3, 'Soundcloud'),(4, 'Spotify'))]`
db_column	By default, django uses the field name for the database column. Use this to provide a custom name
db_index	If `True`, an index will be created on this field in the database
db_tablespace	The tablespace to use for this field's index. *This field is only used if the database engine supports it, otherwise its ignored*.
default	The default value for this field. Can be a value, or a callable object. For mutable defaults (a list, a set, a dictionary) you **must** use a callable. Due to compatibility with migrations, you cannot use lambdas.
editable	If `False`, the field is not shown in the model admin or any `ModelForm`. Default is `True`.
error_messages	Used to customize the default error messages shown for this field. The value is a dictionary, with the keys representing the error and the value being the message. Default keys (for error messages) are `null`, `blank`, `invalid`, `invalid_choice`, `unique` and `unique_for_date`; additional error messages may be defined by custom fields.
help_text	Text to be displayed with the field, to assist users. HTML is allowed.
on_delete	When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint specified by the on_delete argument. This is the second positional argument for both `ForeignKey` and `OneToOneField` fields. Other fields do not have this argument.
primary_key	If `True`, this field will be the primary key. Django automatically adds a

Parameter	Details
	primary key; so this is only required if you wish to create a custom primary key. You can only have one primary key per model.
unique	If `True`, errors are raised if duplicate values are entered for this field. This is a database-level restriction, and not simply a user-interface block.
unique_for_date	Set the value to a `DateField` or `DateTimeField`, and errors will be raised if there are duplicate values *for the same date or date time*.
unique_for_month	Similar to `unique_for_date`, except checks are limited for the month.
unique_for_year	Similar to `unique_for_date`, except checks are limited to the year.
verbose_name	A friendly name for the field, used by django in various places (such as creating labels in the admin and model forms).
validators	A list of validators for this field.

Remarks

- You can write your own fields if you find it necessary
- You can override functions of the base model class, most commonly the `save()` function

Examples

Number Fields

Examples of numeric fields are given:

AutoField

An auto-incrementing integer generally used for primary keys.

```
from django.db import models

class MyModel(models.Model):
    pk = models.AutoField()
```

> Each model gets a primary key field (called `id`) by default. Therefore, it is not necessary to duplicate an id field in the model for the purposes of a primary key.

BigIntegerField

An integer fitting numbers from `-9223372036854775808` to `9223372036854775807`(`8 Bytes`).

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

```
class MyModel(models.Model):
    number_of_seconds = models.BigIntegerField()
```

IntegerField

The IntegerField is used to store integer values from -2147483648 to 2147483647 (`4 Bytes`).

```
from django.db import models

class Food(models.Model):
    name = models.CharField(max_length=255)
    calorie = models.IntegerField(default=0)
```

`default` parameter is not mandatory. But it's useful to set a default value.

PositiveIntegerField

Like an IntegerField, but must be either positive or zero (0). The PositiveIntegerField is used to store integer values from 0 to 2147483647 (`4 Bytes`). This can be useful at field which should be semantically positive. For example if you are recording foods with its calories, it should not be negative. This field will prevent negative values via its validations.

```
from django.db import models

class Food(models.Model):
    name = models.CharField(max_length=255)
    calorie = models.PositiveIntegerField(default=0)
```

`default` parameter is not mandatory. But it's useful to set a default value.

SmallIntegerField

The SmallIntegerField is used to store integer values from -32768 to 32767 (`2 Bytes`). This field is useful for values not are not extremes.

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=255)
    temperature = models.SmallIntegerField(null=True)
```

PositiveSmallIntegerField

The SmallIntegerField is used to store integer values from 0to 32767 (`2 Bytes`). Just like SmallIntegerField this field is useful for values not going so high and should be semantically positive. For example it can store age which cannot be negative.

```
from django.db import models

class Staff(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    age = models.PositiveSmallIntegerField(null=True)
```

Besides PositiveSmallIntegerField is useful for choices, this is the Djangoic way of implementing Enum:

```
from django.db import models
from django.utils.translation import gettext as _

APPLICATION_NEW = 1
APPLICATION_RECEIVED = 2
APPLICATION_APPROVED = 3
APPLICATION_REJECTED = 4

APLICATION_CHOICES = (
    (APPLICATION_NEW, _('New')),
    (APPLICATION_RECEIVED, _('Received')),
    (APPLICATION_APPROVED, _('Approved')),
    (APPLICATION_REJECTED, _('Rejected')),
)

class JobApplication(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    status = models.PositiveSmallIntegerField(
        choices=APLICATION_CHOICES,
        default=APPLICATION_NEW
    )
    ...
```

> Definition of the choices as class variables or module variables according to the situation is a good way to use them. If choices are passed to field without friendly names than it will create confusion.

DecimalField

A fixed-precision decimal number, represented in Python by a Decimal instance. Unlike IntegerField and its derivatives this field has 2 required arguments:

1. *DecimalField.max_digits*: The maximum number of digits allowed in the number. Note that this number must be greater than or equal to decimal_places.
2. *DecimalField.decimal_places*: The number of decimal places to store with the number.

If you want to store numbers up to 99 with 3 decimal places you need use `max_digits=5` and `decimal_places=3`:

```
class Place(models.Model):
    name = models.CharField(max_length=255)
    atmospheric_pressure = models.DecimalField(max_digits=5, decimal_places=3)
```

BinaryField

This is a specialized field, used to store binary data. It only accepts **bytes**. Data is base64 serialized upon storage.

As this is storing binary data, this field cannot be used in a filter.

```
from django.db import models

class MyModel(models.Model):
    my_binary_data = models.BinaryField()
```

CharField

The CharField is used for storing defined lengths of text. In the example below up to 128 characters of text can be stored in the field. Entering a string longer than this will result in a validation error being raised.

```
from django.db import models

class MyModel(models.Model):
    name = models.CharField(max_length=128, blank=True)
```

DateTimeField

DateTimeField is used to store date time values.

```
class MyModel(models.Model):
    start_time = models.DateFimeField(null=True, blank=True)
    created_on = models.DateTimeField(auto_now_add=True)
    updated_on = models.DateTimeField(auto_now=True)
```

A `DateTimeField` has two optional parameters:

- `auto_now_add` sets the value of the field to current datetime when the object is created.
- `auto_now` sets the value of the field to current datetime every time the field is saved.

These options and the `default` parameter are mutually exclusive.

ForeignKey

ForeignKey field is used to create a `many-to-one` relationship between models. Not like the most of other fields requires positional arguments. The following example demonstrates the car and owner relation:

```
from django.db import models

class Person(models.Model):
    GENDER_FEMALE = 'F'
```

```
    GENDER_MALE = 'M'

    GENDER_CHOICES = (
        (GENDER_FEMALE, 'Female'),
        (GENDER_MALE, 'Male'),
    )

    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
    age = models.SmallIntegerField()


class Car(model.Model)
    owner = models.ForeignKey('Person')
    plate = models.CharField(max_length=15)
    brand = models.CharField(max_length=50)
    model = models.CharField(max_length=50)
    color = models.CharField(max_length=50)
```

The first argument of the field is the class to which the model is related. The second positional argument is `on_delete` argument. In the current versions this argument is not required, but it will be required in Django 2.0. The default functionality of the argument is shown as following:

```
class Car(model.Model)
    owner = models.ForeignKey('Person', on_delete=models.CASCADE)
    ...
```

> This will cause Car objects to be deleted from the model when its owner deleted from Person model. This is the default functionality.

```
class Car(model.Model)
    owner = models.ForeignKey('Person', on_delete=models.PROTECT)
    ...
```

> This will prevents Person objects to be deleted if they are related to at least one Car object. All of the Car objects which reference a Person object should be deleted first. And then the Person Object can be deleted.

Read Model Field Reference online: https://riptutorial.com/django/topic/3686/model-field-reference

Chapter 39: Models

Introduction

In the basic case, a model is Python class that maps to a single database table. The attributes of the class map to columns in the table and an instance of the class represents a row in database table. The models inherit from `django.db.models.Model` which provides a rich API for adding and filtering results from the database.

Create Your First Model

Examples

Creating your first model

Models are typically defined in the `models.py` file under your application subdirectory. The `Model` class of `django.db.models` module is a good starting class to extend your models from. For example:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey('Author', on_delete=models.CASCADE,
related_name='authored_books')
    publish_date = models.DateField(null=True, blank=True)

    def __str__(self): # __unicode__ in python 2.*
        return self.title
```

Each attribute in a model represents a column in the database.

- `title` is a text with a maximum length of 100 characters
- `author` is a `ForeignKey` which represents a relationship to another model/table, in this case `Author` (used only for example purposes). `on_delete` tells the database what to do with the object should the related object (an `Author`) be deleted. (It should be noted that since django 1.9 `on_delete` can be used as the second positional argument. In django 2 it is a **required** argument and it is advisable to treat it as such immediately. In older versions it will default to `CASCADE`.)
- `publish_date` stores a date. Both `null` and `blank` are set to `True` to indicate that it is not a required field (i.e. you may add it at a later date or leave it empty.)

Along with the attributes we define a method `__str__` this returns the title of the book which will be used as its `string` representation where necessary, rather than the default.

Applying the changes to the database (Migrations)

After creating a new model or modifying existing models, you will need to generate migrations for your changes and then apply the migrations to the specified database. This can be done by using the Django's built-in migrations system. Using the `manage.py` utility when in the project root directory:

```
python manage.py makemigrations <appname>
```

The above command will create the migration scripts that are necessary under `migrations` subdirectory of your application. If you omit the `<appname>` parameter, all the applications defined in the `INSTALLED_APPS` argument of `settings.py` will be processed. If you find it necessary, you can edit the migrations.

You can check what migrations are required without actually creating the migration use the --dry-run option, eg:

```
python manage.py makemigrations --dry-run
```

To apply the migrations:

```
python manage.py migrate <appname>
```

The above command will execute the migration scripts generated in the first step and physically update the database.

If the model of existing database is changed then following command is needed for making necessary changes.

```
python manage.py migrate --run-syncdb
```

Django will create the table with name `<appname>_<classname>` by default. Sometime you don't want to use it. If you want to change the default name, you can announce the table name by setting the `db_table` in the class `Meta`:

```
from django.db import models

class YourModel(models.Model):
    parms = models.CharField()
    class Meta:
        db_table = "custom_table_name"
```

If you want to see what SQL code will be executed by a certain migration just run this command:

```
python manage.py sqlmigrate <app_label> <migration_number>
```

Django >1.10
The new `makemigrations --check` option makes the command exit with a non-zero status when model changes without migrations are detected.

See Migrations for more details on migrations.

Creating a model with relationships

Many-to-One Relationship

```
from django.db import models

class Author(models.Model):
   name = models.CharField(max_length=50)

#Book has a foreignkey (many to one) relationship with author
class Book(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    publish_date = models.DateField()
```

Most generic option. Can be used anywhere you would like to represent a relationship

Many-to-Many Relationship

```
class Topping(models.Model):
    name = models.CharField(max_length=50)

# One pizza can have many toppings and same topping can be on many pizzas
class Pizza(models.Model):
    name = models.CharField(max_length=50)
    toppings = models.ManyToManyField(Topping)
```

Internally this is represented via another table. And `ManyToManyField` should be put on models that will be edited on a form. Eg: `Appointment` will have a `ManyToManyField` called `Customer`, `Pizza` has `Toppings` and so on.

Many-to-Many Relationship using Through classes

```
class Service(models.Model):
     name = models.CharField(max_length=35)

class Client(models.Model):
    name = models.CharField(max_length=35)
    age = models.IntegerField()
    services = models.ManyToManyField(Service, through='Subscription')

class Subscription(models.Model):
     client = models.ForeignKey(Client)
     service = models.ForeignKey(Service)
     subscription_type = models.CharField(max_length=1, choices=SUBSCRIPTION_TYPES)
     created_at = models.DateTimeField(default=timezone.now)
```

This way, we can actually keep more metadata about a relationship between two entities. As can be seen, a client can be subscribed to several services via several subscription types. The only difference in this case is that to add new instances to the M2M relation, one cannot use the shortcut method `pizza.toppings.add(topping)`, instead, a new object of the *through* class should be created, `Subscription.objects.create(client=client, service=service, subscription_type='p')`

In other languages `through tables` are also known as a `JoinColumn` , `Intersection table` or `mapping table`

One-to-One Relationship

```
class Employee(models.Model):
   name = models.CharField(max_length=50)
   age = models.IntegerField()
   spouse = models.OneToOneField(Spouse)

class Spouse(models.Model):
   name = models.CharField(max_length=50)
```

Use these fields when you will only ever have a composition relationship between the two models.

Basic Django DB queries

Django ORM is a powerful abstraction that lets you store and retrieve data from the database without writing sql queries yourself.

Let's assume the following models:

```
class Author(models.Model):
   name = models.CharField(max_length=50)

class Book(models.Model):
   name = models.CharField(max_length=50)
   author = models.ForeignKey(Author)
```

Assuming you've added the above code to a django application and run the `migrate` command (so that your database is created). Start the Django shell by

```
python manage.py shell
```

This starts the standard python shell but with relevant Django libraries imported, so that you can directly focus on the important parts.

Start by importing the models we just defined (I am assuming this is done in a file `models.py`)

```
from .models import Book, Author
```

Run your first select query:

```
>>> Author.objects.all()
[]
>>> Book.objects.all()
[]
```

Lets create an author and book object:

```
>>> hawking = Author(name="Stephen hawking")
>>> hawking.save()
>>> history_of_time = Book(name="history of time", author=hawking)
>>> history_of_time.save()
```

or use create function to create model objects and save in one line code

```
>>> wings_of_fire = Book.objects.create(name="Wings of Fire", author="APJ Abdul Kalam")
```

Now lets run the query

```
>>> Book.objects.all()
[<Book: Book object>]
>>> book = Book.objects.first() #getting the first book object
>>> book.name
u'history of time'
```

Let's add a where clause to our select query

```
>>> Book.objects.filter(name='nothing')
[]
>>> Author.objects.filter(name__startswith='Ste')
[<Author: Author object>]
```

To get the details about the author of a given book

```
>>> book = Book.objects.first() #getting the first book object
>>> book.author.name # lookup on related model
u'Stephen hawking'
```

To get all the books published by Stephen Hawking (Lookup book by its author)

```
>>> hawking.book_set.all()
[<Book: Book object>]
```

`_set` is the notation used for "Reverse lookups" i.e. while the lookup field is on the Book model, we can use `book_set` on an author object to get all his/her books.

A basic unmanaged table.

At some point in your use of Django, you may find yourself wanting to interact with tables which have already been created, or with database views. In these cases, you would not want Django to manage the tables through its migrations. To set this up, you need to add only one variable to your model's `Meta` class: `managed = False`.

Here is an example of how you might create an unmanaged model to interact with a database view:

```
class Dummy(models.Model):
    something = models.IntegerField()
```

```
    class Meta:
       managed = False
```

This may be mapped to a view defined in SQL as follows.

```
CREATE VIEW myapp_dummy AS
SELECT id, something FROM complicated_table
WHERE some_complicated_condition = True
```

Once you have this model created, you can use it as you would any other model:

```
>>> Dummy.objects.all()
[<Dummy: Dummy object>, <Dummy: Dummy object>, <Dummy: Dummy object>]
>>> Dummy.objects.filter(something=42)
[<Dummy: Dummy object>]
```

Advanced models

A model can provide a lot more information than just the data about an object. Let's see an example and break it down into what it is useful for:

```
from django.db import models
from django.urls import reverse
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class Book(models.Model):
    slug = models.SlugField()
    title = models.CharField(max_length=128)
    publish_date = models.DateField()

    def get_absolute_url(self):
        return reverse('library:book', kwargs={'pk':self.pk})

    def __str__(self):
        return self.title

    class Meta:
        ordering = ['publish_date', 'title']
```

Automatic primary key

You might notice the use of `self.pk` in the `get_absolute_url` method. The `pk`field is an alias to the primary key of a model. Also, django will automatically add a primary key if it's missing. That's one less thing to worry and let you set foreign key to any models and get them easily.

Absolute url

The first function that is defined is `get_absolute_url`. This way, if you have an book, you can get a link to it without fiddling with the url tag, resolve, attribute and the like. Simply call

`book.get_absolute_url` and you get the right link. As a bonus, your object in the django admin will gain a button "view on site".

String representation

Have a `__str__` method let you use the object when you need to display it. For example, with the previous method, adding a link to the book in a template is as simple as `<a href="{{ book.get_absolute_url }}">{{ book }}</a>`. Straight to the point. This method also control what is displayed in the admin drop-down, for foreign key for example.

The class decorator let you define the method once for both `__str__` and `__unicode__` on python 2 while causing no issue on python 3. If you expect your app to run on both version, that's the way to go.

Slug field

The slug field is similar to a char field but accept less symbols. By default, only letters, numbers, underscores or hyphens. It is useful if you want to identify an object using a nice representation, in url for example.

The Meta class

The `Meta` class let us define a lot more of information on the whole collection of item. Here, only the default ordering is set. It is useful with the ListView object for example. It take an ideally short list of field to use for sorting. Here, book will be sorted first by publication date then by title if the date is the same.

Other frequents attributes are `verbose_name` and `verbose_name_plural`. By default, they are generated from the name of the model and should be fine. But the plural form is naive, simply appending an 's' to the singular so you might want to set it explicitly in some case.

Computed Values

Once a model object has been fetched, it becomes a fully realized instance of the class. As such, any additional methods can be accessed in forms and serializers (like Django Rest Framework).

Using python properties is an elegant way to represent additional values that are not stored in the database due to varying circumstances.

```
def expire():
    return timezone.now() + timezone.timedelta(days=7)

class Coupon(models.Model):
    expiration_date = models.DateField(default=expire)

    @property
    def is_expired(self):
        return timezone.now() > self.expiration_date
```

While most cases you can supplement data with annotations on your querysets, computed values as model properties are ideal for computations that can not be evaluated simply within the scope of a query.

Additionally, properties, since they are declared on the python class and not as part of the schema, are not available for querying against.

Adding a string representation of a model

To create a human-readable presentation of a model object you need to implement `Model.__str__()` method (or `Model.__unicode__()` on python2). This method will be called whenever you call `str()` on a instance of your model (including, for instance, when the model is used in a template). Here's an example:

1. Create a book model.

```
# your_app/models.py

from django.db import models

class Book(models.Model):
    name = models.CharField(max_length=50)
    author = models.CharField(max_length=50)
```

2. Create an instance of the model, and save it in the database:

```
>>> himu_book = Book(name='Himu Mama', author='Humayun Ahmed')
>>> himu_book.save()
```

3. Execute `print()` on the instance:

```
>>> print(himu_book)
<Book: Book object>
```

<Book: Book object>, the default output, is of no help to us. To fix this, let's add a `__str__` method.

```
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class Book(models.Model):
    name = models.CharField(max_length=50)
    author = models.CharField(max_length=50)

    def __str__(self):
        return '{} by {}'.format(self.name, self.author)
```

Note the `python_2_unicode_compatible` decorator is needed only if you want your code to be compatible with python 2. This decorator copies the `__str__` method to create a `__unicode__` method. Import it from `django.utils.encoding`.

Now if we call the print function the book instance again:

```
>>> print(himu_book)
Himu Mama by Humayun Ahmed
```

Much better!

The string representation is also used when the model is used in a `ModelForm` for `ForeignKeyField` and `ManyToManyField` fields.

Model mixins

In same cases different models could have same fields and same procedures in the product life cycle. To handle these similarities without having code repetition inheritance could be used. Instead of inheriting a whole class, **mixin** design pattern offers us to inherit (*or some says include*) some methods and attributes. Let's see an example:

```
class PostableMixin(models.Model):
    class Meta:
        abstract=True

    sender_name = models.CharField(max_length=128)
    sender_address = models.CharField(max_length=255)
    receiver_name = models.CharField(max_length=128)
    receiver_address = models.CharField(max_length=255)
    post_datetime = models.DateTimeField(auto_now_add=True)
    delivery_datetime = models.DateTimeField(null=True)
    notes = models.TextField(max_length=500)

class Envelope(PostableMixin):
    ENVELOPE_COMMERCIAL = 1
    ENVELOPE_BOOKLET = 2
    ENVELOPE_CATALOG = 3

    ENVELOPE_TYPES = (
        (ENVELOPE_COMMERCIAL, 'Commercial'),
        (ENVELOPE_BOOKLET, 'Booklet'),
        (ENVELOPE_CATALOG, 'Catalog'),
    )

    envelope_type = models.PositiveSmallIntegerField(choices=ENVELOPE_TYPES)

class Package(PostableMixin):
    weight = models.DecimalField(max_digits=6, decimal_places=2)
    width = models.DecimalField(max_digits=5, decimal_places=2)
    height = models.DecimalField(max_digits=5, decimal_places=2)
    depth = models.DecimalField(max_digits=5, decimal_places=2)
```

To turn a model into an abstract class, you will need to mention `abstract=True` in its inner `Meta` class. Django does not create any tables for abstract models in the database. However for the models `Envelope` and `Package`, corresponding tables would be created in the database.

Furthermore the fields some model methods will be needed at more than one models. Thus these methods could be added to mixins to prevent code repetition. For example if we create a method

to set delivery date to `PostableMixin` it will be accesible from both of its children:

```
class PostableMixin(models.Model):
    class Meta:
        abstract=True

    ...
    ...

    def set_delivery_datetime(self, dt=None):
        if dt is None:
            from django.utils.timezone import now
            dt = now()

        self.delivery_datetime = dt
        self.save()
```

This method could be used as following on the children:

```
>> envelope = Envelope.objects.get(pk=1)
>> envelope.set_delivery_datetime()

>> pack = Package.objects.get(pk=1)
>> pack.set_delivery_datetime()
```

UUID Primary key

A model by default will use an auto incrementing (integer) primary key. This will give you a sequence of keys 1, 2, 3.

Different primary key types can be set on a model with a small alterations to the model.

A UUID is a universally unique identifier, this is 32 character random identifier which can be used as an ID. This is a good option to use when you do not want sequential ID's assigned to records in your database.When used on PostgreSQL, this stores in a uuid datatype, otherwise in a char(32).

```
import uuid
from django.db import models

class ModelUsingUUID(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
```

The generated key will be in the format `7778c552-73fc-4bc4-8bf9-5a2f6f7b7f47`

Inheritance

Inheritance among models can be done in two ways:

- a common abstract class (see the "Model mixins" example)
- a common model with multiple tables

The multi tables inheritance will create one table for the common fields and one per child model

example:

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

will create 2 tables, one for `Place` and one for `Restaurant` with a hidden `OneToOne` field to `Place` for the common fields.

note that this will need an extra query to the places tables every time you fetch an Restaurant Object.

Read Models online: https://riptutorial.com/django/topic/888/models

Chapter 40: Project Structure

Examples

Repository > Project > Site/Conf

For a Django project with `requirements` and `deployment tools` under source control. This example builds upon concepts from the Two Scoops of Django. They have published a template:

```
repository/
    docs/
    .gitignore
    project/
        apps/
            blog/
                migrations/
                static/ #( optional )
                    blog/
                        some.css
                templates/ #( optional )
                    blog/
                        some.html
                models.py
                tests.py
                admin.py
                apps.py #( django 1.9 and later )
                views.py
            accounts/
                #... ( same as blog )
            search/
                #... ( same as blog )
        conf/
            settings/
                local.py
                development.py
                production.py
            wsgi
            urls.py
        static/
        templates/
    deploy/
        fabfile.py
    requirements/
        base.txt
        local.txt
    README
    AUTHORS
    LICENSE
```

Here `apps` and `conf` folders contain `user created applications` and `core configuration folder` for the project respectively.

`static` and `templates` folders in `project` directory contains static files and `html markup` files respectively that are being used globally throughout the project.

And all app folders `blog`, `accounts` and `search` may also (mostly) contain `static` and `templates` folders.

Namespacing static and templates files in django apps

`static` and `templates` folder in the apps may should also contain a folder with the name of app `ex. blog` this is a convention used to prevent namespace pollution, so we reference the files like `/blog/base.html` rather than `/base.html` which provides more clarity about the file we are referencing and preserves namespace.

Example: `templates` folder inside `blog` and `search` applications contains a file with name `base.html`, and when referencing the file in `views` your application gets confused in which file to render.

```
(Project Structure)
.../project/
    apps/
        blog/
            templates/
                base.html
        search/
            templates/
                base.html

(blog/views.py)
def some_func(request):
    return render(request, "/base.html")

(search/views.py)
def some_func(request):
    return render(request, "/base.html")

## After creating a folder inside /blog/templates/(blog) ##

(Project Structure)
.../project/
    apps/
        blog/
            templates/
                blog/
                    base.html
        search/
            templates/
                search/
                    base.html

(blog/views.py)
def some_func(request):
    return render(request, "/blog/base.html")

(search/views.py)
def some_func(request):
    return render(request, "/search/base.html")
```

Read Project Structure online: https://riptutorial.com/django/topic/4299/project-structure

Chapter 41: Querysets

Introduction

A `Queryset` is fundamentally a list of objects derived from a `Model`, by a compilation of database queries.

Examples

Simple queries on a standalone model

Here is a simple model that we will use to run a few test queries:

```
class MyModel(models.Model):
    name = models.CharField(max_length=10)
    model_num = models.IntegerField()
    flag = models.NullBooleanField(default=False)
```

Get a single model object where the id/pk is 4:
(If there are no items with the id of 4 or there are more than one, this will throw an exception.)

```
MyModel.objects.get(pk=4)
```

All model objects:

```
MyModel.objects.all()
```

Model objects that have `flag` set to `True`:

```
MyModel.objects.filter(flag=True)
```

Model objects with a `model_num` greater than 25:

```
MyModel.objects.filter(model_num__gt=25)
```

Model objects with the `name` of "Cheap Item" and `flag` set to `False`:

```
MyModel.objects.filter(name="Cheap Item", flag=False)
```

Models simple search `name` for specific string(Case-sensitive):

```
MyModel.objects.filter(name__contains="ch")
```

Models simple search `name` for specific string(Case-insensitive):

```
MyModel.objects.filter(name__icontains="ch")
```

Advanced queries with Q objects

Given the model:

```
class MyModel(models.Model):
    name = models.CharField(max_length=10)
    model_num = models.IntegerField()
    flag = models.NullBooleanField(default=False)
```

We can use `Q` objects to create `AND` , `OR` conditions in your lookup query. For example, say we want all objects that have `flag=True` **OR** `model_num>15`.

```
from django.db.models import Q
MyModel.objects.filter(Q(flag=True) | Q(model_num__gt=15))
```

The above translates to `WHERE flag=True OR model_num > 15` similarly for an **AND** you would do.

```
MyModel.objects.filter(Q(flag=True) & Q(model_num__gt=15))
```

`Q` objects also allow us to make **NOT** queries with the use of `~`. Let's say we wanted to get all objects that have `flag=False` **AND** `model_num!=15`, we would do:

```
MyModel.objects.filter(Q(flag=True) & ~Q(model_num=15))
```

If using Q objects and "normal" parameters in `filter()`, then the `Q` objects must come *first*. The following query searches for models with (`flag` set to `True` or a model number greater than `15`) and a name that starts with "H".

```
from django.db.models import Q
MyModel.objects.filter(Q(flag=True) | Q(model_num__gt=15), name__startswith="H")
```

Note: `Q` objects can be used with any lookup function that takes keyword arguments such as `filter`, `exclude`, `get`. Make sure that when you use with `get` that you will only return one object or the `MultipleObjectsReturned` exception will be raised.

Reduce number of queries on ManyToManyField (n+1 issue)

Problem

```
# models.py:
class Library(models.Model):
    name = models.CharField(max_length=100)
    books = models.ManyToManyField(Book)

class Book(models.Model):
```

```
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.all()

    # Query the database on each iteration (len(author) times)
    # if there is 100 librairies, there will have 100 queries plus the initial query
    for library in libraries:
        books = library.books.all()
        books[0].title
        # ...

    # total : 101 queries
```

Solution

Use `prefetch_related` on `ManyToManyField` if you know that you will need to access later a field which is a `ManyToManyField` field.

```
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books').all()

    # Does not query the database again, since `books` is pre-populated
    for library in libraries:
        books = library.books.all()
        books[0].title
        # ...

    # total : 2 queries - 1 for libraries, 1 for books
```

`prefetch_related` can also be used on lookup fields :

```
# models.py:
class User(models.Model):
    name = models.CharField(max_length=100)

class Library(models.Model):
    name = models.CharField(max_length=100)
    books = models.ManyToManyField(Book)

class Book(models.Model):
    title = models.CharField(max_length=100)
    readers = models.ManyToManyField(User)
```

```
 # views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books', 'books__readers').all()

    # Does not query the database again, since `books` and `readers` is pre-populated
```

```
    for library in libraries:
        for book in library.books.all():
            for user in book.readers.all():
                user.name
                # ...

    # total : 3 queries - 1 for libraries, 1 for books, 1 for readers
```

However, once the queryset has been executed, the data fetched can't be altered without hitting again the database. The following would execute extra queries for example:

```
 # views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related('books').all()
    for library in libraries:
        for book in library.books.filter(title__contains="Django"):
            print(book.name)
```

The following can be optimized using a `Prefetch` object, introduced in Django 1.7:

```
from django.db.models import Prefetch
# views.py
def myview(request):
    # Query the database.
    libraries = Library.objects.prefetch_related(
        Prefetch('books', queryset=Book.objects.filter(title__contains="Django")
    ).all()
    for library in libraries:
        for book in library.books.all():
            print(book.name)  # Will print only books containing Django for each library
```

Reduce number of queries on ForeignKey field (n+1 issue)

Problem

Django querysets are evaluated in a lazy fashion. For example:

```
# models.py:
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author, related_name='books')
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
    # Query the database
    books = Book.objects.all()

    for book in books:
        # Query the database on each iteration to get author (len(books) times)
```

```
        # if there is 100 books, there will have 100 queries plus the initial query
        book.author
        # ...

    # total : 101 queries
```

The code above causes django to query the database for the author of each book. This is inefficient, and it is better to only have a single query.

Solution

Use `select_related` on `ForeignKey` if you know that you will need to later access a `ForeignKey` field.

```
# views.py
def myview(request):
    # Query the database.
    books = Books.objects.select_related('author').all()

    for book in books:
        # Does not query the database again, since `author` is pre-populated
        book.author
        # ...

    # total : 1 query
```

`select_related` can also be used on lookup fields:

```
# models.py:
class AuthorProfile(models.Model):
    city = models.CharField(max_length=100)

class Author(models.Model):
    name = models.CharField(max_length=100)
    profile = models.OneToOneField(AuthorProfile)

class Book(models.Model):
    author = models.ForeignKey(Author, related_name='books')
    title = models.CharField(max_length=100)
```

```
# views.py
def myview(request):
    books = Book.objects.select_related('author')\
                        .select_related('author__profile').all()

    for book in books:
        # Does not query database
        book.author.name
        # or
        book.author.profile.city
        # ...

    # total : 1 query
```

Get SQL for Django queryset

The `query` attribute on queryset gives you SQL equivalent syntax for your query.

```
>>> queryset = MyModel.objects.all()
>>> print(queryset.query)
SELECT "myapp_mymodel"."id", ... FROM "myapp_mymodel"
```

> **Warning:**
>
> This output should only be used for debugging purposes. The generated query is not backend-specific. As such, the parameters aren't quoted properly, leaving it vulnerable to SQL injection, and the query may not even be executable on your database backend.

Get first and last record from QuerySet

To get First object:

```
MyModel.objects.first()
```

To get last objects:

```
MyModel.objects.last()
```

Using Filter First object:

```
MyModel.objects.filter(name='simple').first()
```

Using Filter Last object:

```
MyModel.objects.filter(name='simple').last()
```

Advanced queries with F objects

> An F() object represents the value of a model field or annotated column. It makes it possible to refer to model field values and perform database operations using them without actually having to pull them out of the database into Python memory. - F() expressions

It is appropriate to use `F()` objects whenever you need to reference another field's value in your query. By itself, `F()` objects do not mean anything, and they cannot and should not be called outside of a queryset. They are used to reference a field's value on the same queryset.

For example, given a model ...

```
SomeModel(models.Model):
    ...
    some_field = models.IntegerField()
```

... a user can query objects where the `some_field` value is twice its `id` by referencing the `id` field's value while filtering using `F()` like this:

```
SomeModel.objects.filter(some_field=F('id') * 2)
```

`F('id')` simply references the `id` value for that same instance. Django uses it to create corresponding SQL statement. In this case something closely resembling this:

```
SELECT * FROM some_app_some_model
WHERE some_field = ((id * 2))
```

Without `F()` expressions this would be accomplished with either raw SQL or filtering in Python (which reduces the performance especially when there are lots of objects).

References:

- Filters can reference fields on model
- F expressions
- Answer from TinyInstance

From `F()` class definition:

> An object capable of resolving references to existing query objects. - F source

Note: This example posted came from the answer listed above with consent from TinyInstance.

Read Querysets online: https://riptutorial.com/django/topic/1235/querysets

Chapter 42: RangeFields - a group of PostgreSQL specific fields

Syntax

- from django.contrib.postgres.fields import *RangeField
- IntegerRangeField(**options)
- BigIntegerRangeField(**options)
- FloatRangeField(**options)
- DateTimeRangeField(**options)
- DateRangeField(**options)

Examples

Including numeric range fields in your model

There are three kinds of numeric `RangeField`s in Python. `IntegerField`, `BigIntegerField`, and `FloatField`. They convert to `psycopg2` `NumericRange`s, but accept input as native Python tuples. **The lower bound is included and the upper bound is excluded.**

```
class Book(models.Model):
    name = CharField(max_length=200)
    ratings_range = IntegerRange()
```

Setting up for RangeField

1. add `'django.contrib.postgres'` to your `INSTALLED_APPS`
2. install `psycopg2`

Creating models with numeric range fields

It's simpler and easier to input values as a Python tuple instead of a `NumericRange`.

```
Book.objects.create(name='Pro Git', ratings_range=(5, 5))
```

Alternative method with `NumericRange`:

```
Book.objects.create(name='Pro Git', ratings_range=NumericRange(5, 5))
```

Using contains

This query selects all books with any rating less than three.

```
bad_books = Books.objects.filter(ratings_range__contains=(1, 3))
```

Using contained_by

This query gets all books with ratings greater than or equal to zero and less than six.

```
all_books = Book.objects.filter(ratings_range_contained_by=(0, 6))
```

Using overlap

This query gets all overlapping appointments from six to ten.

```
Appointment.objects.filter(time_span__overlap=(6, 10))
```

Using None to signify no upper bound

This query selects all books with any rating greater than or equal to four.

```
maybe_good_books = Books.objects.filter(ratings_range__contains=(4, None))
```

Ranges operations

```
from datetime import timedelta

from django.utils import timezone
from psycopg2.extras import DateTimeTZRange

# To create a "period" object we will use psycopg2's DateTimeTZRange
# which takes the two datetime bounds as arguments
period_start = timezone.now()
period_end = period_start + timedelta(days=1, hours=3)
period = DateTimeTZRange(start, end)

# Say Event.timeslot is a DateTimeRangeField

# Events which cover at least the whole selected period,
Event.objects.filter(timeslot__contains=period)

# Events which start and end within selected period,
Event.objects.filter(timeslot__contained_by=period)

# Events which, at least partially, take place during the selected period.
Event.objects.filter(timeslot__overlap=period)
```

Read RangeFields - a group of PostgreSQL specific fields online:
https://riptutorial.com/django/topic/2630/rangefields---a-group-of-postgresql-specific-fields

Chapter 43: Running Celery with Supervisor

Examples

Celery Configuration

CELERY

1. Installation - `pip install django-celery`
2. Add
3. Basic project structure.

```
- src/
  - bin/celery_worker_start # will be explained later on
  - logs/celery_worker.log
  - stack/__init __.py
  - stack/celery.py
  - stack/settings.py
  - stack/urls.py
  - manage.py
```

4. Add `celery.py` file to your `stack/stack/` folder.

```
from __future__ import absolute_import
import os
from celery import Celery
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'stack.settings')
from django.conf import settings  # noqa
app = Celery('stack')
app.config_from_object('django.conf:settings')
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

5. to your `stack/stack/__init__.py` add following code:

```
from __future__ import absolute_import
from .celery import app as celery_app  # noqa
```

6. Create a task and mark it for example as `@shared_task()`

```
@shared_task()
def add(x, y):
    print("x*y={}".format(x*y))
```

7. Running celery worker "by hand":

`celery -A stack worker -l info` if you also want to add

Running Supervisor

1. Create a script to start celery worker. Insert your script within your app. For example:
   `stack/bin/celery_worker_start`

```
#!/bin/bash

NAME="StackOverflow Project - celery_worker_start"

PROJECT_DIR=/home/stackoverflow/apps/proj/proj/
ENV_DIR=/home/stackoverflow/apps/proj/env/

echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd "${PROJECT_DIR}"

if [ -d "${ENV_DIR}" ]
then
    . "${ENV_DIR}bin/activate"
fi

celery -A stack --loglevel='INFO'
```

2. Add execution rights to your newly created script:

```
chmod u+x bin/celery_worker_start
```

3. Install supervisor (skip this test if supervisor already installed)

```
apt-get install supervisor
```

4. Add config file for your supervisor in order to start you celery. Place it in
   `/etc/supervisor/conf.d/stack_supervisor.conf`

```
[program:stack-celery-worker]
command = /home/stackoverflow/apps/stack/src/bin/celery_worker_start
user = polsha
stdout_logfile = /home/stackoverflow/apps/stack/src/logs/celery_worker.log
redirect_stderr = true
environment = LANG = en_US.UTF-8,LC_ALL = en_US.UTF-8
numprocs = 1
autostart = true
autorestart = true
startsecs = 10
stopwaitsecs = 600
priority = 998
```

5. Reread and update supervisor

```
sudo supervisorctl reread
    stack-celery-worker: available
sudo supervisorctl update
    stack-celery-worker: added process group
```

6. Basic commands

```
sudo supervisorctl status stack-celery-worker
    stack-celery-worker      RUNNING    pid 18020, uptime 0:00:50
sudo supervisorctl stop stack-celery-worker
    stack-celery-worker: stopped
sudo supervisorctl start stack-celery-worker
    stack-celery-worker: started
sudo supervisorctl restart stack-celery-worker
    stack-celery-worker: stopped
    stack-celery-worker: started
```

Celery + RabbitMQ with Supervisor

Celery requires a broker to handle message-passing. We use RabbitMQ because it's easy to setup and it is well supported.

Install rabbitmq using the following command

```
sudo apt-get install rabbitmq-server
```

Once the installation is complete, create user, add a virtual host and set permissions.

```
sudo rabbitmqctl add_user myuser mypassword
sudo rabbitmqctl add_vhost myvhost
sudo rabbitmqctl set_user_tags myuser mytag
sudo rabbitmqctl set_permissions -p myvhost myuser ".*" ".*" ".*"
```

To start the server:

```
sudo rabbitmq-server
```

We can install celery with pip:

```
pip install celery
```

In your Django settings.py file, your broker URL would then look something like

```
BROKER_URL = 'amqp://myuser:mypassword@localhost:5672/myvhost'
```

Now start the celery worker

```
celery -A your_app worker -l info
```

This command start a Celery worker to run any tasks defined in your django app.

Supervisor is a Python program that allows you to control and keep running any unix processes. It can also restart crashed processes. We use it to make sure Celery workers are always running.

First, Install supervisor

```
sudo apt-get install supervisor
```

Create your_proj.conf file in your supervisor conf.d (/etc/supervisor/conf.d/your_proj.conf):

```
[program:your_proj_celery]
command=/home/your_user/your_proj/.venv/bin/celery --app=your_proj.celery:app worker -l info
directory=/home/your_user/your_proj
numprocs=1
stdout_logfile=/home/your_user/your_proj/logs/celery-worker.log
stderr_logfile=/home/your_user/your_proj/logs/low-worker.log
autostart=true
autorestart=true
startsecs=10
```

Once our configuration file is created and saved, we can inform Supervisor of our new program through the supervisorctl command. First we tell Supervisor to look for any new or changed program configurations in the /etc/supervisor/conf.d directory with:

```
sudo supervisorctl reread
```

Followed by telling it to enact any changes with:

```
sudo supervisorctl update
```

Once our programs are running, there will undoubtedly be a time when we want to stop, restart, or see their status.

```
sudo supervisorctl status
```

For restart your celery instance:

```
sudo supervisorctl restart your_proj_celery
```

Read Running Celery with Supervisor online: https://riptutorial.com/django/topic/7091/running-celery-with-supervisor

Chapter 44: Security

Examples

Cross Site Scripting (XSS) protection

XSS attacks consist in injecting HTML (or JS) code in a page. See What is cross site scripting for more information.

To prevent from this attack, by default, Django escapes strings passed through a template variable.

Given the following context:

```
context = {
    'class_name': 'large" style="font-size:4000px',
    'paragraph': (
        "<script type=\"text/javascript\">alert('hello world!');</script>"),
}
```

```
<p class="{{ class_name }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large&quot; style=&quot;font-size: 4000px">&lt;script&gt;alert(&#39;hello
world!&#39;);&lt;/script&gt;</p>
```

If you have variables containing HTML that you trust and actually want to render, you must explicitly say it is safe:

```
<p class="{{ class_name|safe }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large" style="font-size: 4000px">&lt;script&gt;alert(&#39;hello
world!&#39;);&lt;/script&gt;</p>
```

If you have a block containing multiple variables that are all safe, you can locally disable auto escaping:

```
{% autoescape off %}
<p class="{{ class_name }}">{{ paragraph }}</p>
{% endautoescape %}
<!-- Will be rendered as: -->
<p class="large" style="font-size: 4000px"><script>alert('hello world!');</script></p>
```

You can also mark a string as safe outside of the template:

```
from django.utils.safestring import mark_safe

context = {
    'class_name': 'large" style="font-size:4000px',
    'paragraph': mark_safe(
        "<script type=\"text/javascript\">alert('hello world!');</script>"),
```

```
}
```

```
<p class="{{ class_name }}">{{ paragraph }}</p>
<!-- Will be rendered as: -->
<p class="large&amp;quot; style=&amp;quot;font-size: 4000px"><script>alert('hello
world!');</script></p>
```

Some Django utilities such as `format_html` already return strings marked as safe:

```
from django.utils.html import format_html

context = {
    'var': format_html('<b>{}</b> {}', 'hello', '<i>world!</i>'),
}
```

```
<p>{{ var }}</p>
<!-- Will be rendered as -->
<p><b>hello</b> &amp;lt;i&amp;gt;world!&amp;lt;/i&amp;gt;</p>
```

Clickjacking protection

> Clickjacking is a malicious technique of tricking a Web user into clicking on something different from what the user perceives they are clicking on. Learn more

To enable clickjacking protection, add the `XFrameOptionsMiddleware` to your middleware classes. This should already be there if you didn't remove it.

```
# settings.py
MIDDLEWARE_CLASSES = [
    ...
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ...
]
```

This middleware sets the 'X-Frame-Options' header to your all your responses, unless explicitly exempted or already set (not overridden if already set in the response). By default it is set to "SAMEORIGIN". To change this, use the `X_FRAME_OPTIONS` setting:

```
X_FRAME_OPTIONS = 'DENY'
```

You can override the default behaviour on a per-view basis.

```
from django.utils.decorators import method_decorator
from django.views.decorators.clickjacking import (
    xframe_options_exempt, xframe_options_deny, xframe_options_sameorigin,
)

xframe_options_exempt_m = method_decorator(xframe_options_exempt, name='dispatch')

@xframe_options_sameorigin
def my_view(request, *args, **kwargs):
    """Forces 'X-Frame-Options: SAMEORIGIN'."""
```

```
    return HttpResponse(...)

@method_decorator(xframe_options_deny, name='dispatch')
class MyView(View):
    """Forces 'X-Frame-Options: DENY'."""

@xframe_options_exempt_m
class MyView(View):
    """Does not set 'X-Frame-Options' header when passing through the
    XFrameOptionsMiddleware.
    """
```

Cross-site Request Forgery (CSRF) protection

> Cross-site request forgery, also known as one-click attack or session riding and abbreviated as CSRF or XSRF, is a type of malicious exploit of a website where unauthorized commands are transmitted from a user that the website trusts. Learn more

To enable CSRF protection, add the `CsrfViewMiddleware` to your middleware classes. This middleware is enabled by default.

```
# settings.py
MIDDLEWARE_CLASSES = [
    ...
    'django.middleware.csrf.CsrfViewMiddleware',
    ...
]
```

This middleware will set a token in a cookie on the outgoing response. Whenever an incoming request uses an unsafe method (any method except `GET`, `HEAD`, `OPTIONS` and `TRACE`), the cookie must match a token that is send as the `csrfmiddlewaretoken` form data or as the `X-CsrfToken` header. This ensures that the client initiating the request is also the owner of the cookie and, by extension, the (authenticated) session.

If a request is made over `HTTPS`, strict referrer checking is enabled. If the `HTTP_REFERER` header does not match the host of the current request or a host in `CSRF_TRUSTED_ORIGINS` (new in 1.9), the request is denied.

Forms that use the `POST` method should include the CSRF token in the template. The `{% csrf_token %}` template tag will output a hidden field, and will ensure that the cookie is set on the response:

```
<form method='POST'>
{% csrf_token %}
...
</form>
```

Individual views that are not vulnerable to CSRF attacks can be made exempt using the `@csrf_exempt` decorator:

```
from django.views.decorators.csrf import csrf_exempt
```

```
@csrf_exempt
def my_view(request, *args, **kwargs):
    """Allows unsafe methods without CSRF protection"""
    return HttpResponse(...)
```

Although not recommended, you can disable the `CsrfViewMiddleware` if many of your views are not vulnerable to CSRF attacks. In this case you can use the `@csrf_protect` decorator to protect individual views:

```
from django.views.decorators.csrf import csrf_protect

@csrf_protect
def my_view(request, *args, **kwargs):
    """This view is protected against CSRF attacks if the middleware is disabled"""
    return HttpResponse(...)
```

Read Security online: https://riptutorial.com/django/topic/2957/security

Chapter 45: Settings

Examples

Setting the timezone

You can set the timezone that will be used by Django in the `settings.py` file. Examples:

```
TIME_ZONE = 'UTC'  # use this, whenever possible
TIME_ZONE = 'Europe/Berlin'
TIME_ZONE = 'Etc/GMT+1'
```

Here is the list of valid timezones

*When running in a **Windows** environment this must be set to the same as your **system time zone** .*

If you do not want Django to use timezone-aware datetimes:

```
USE_TZ = False
```

Django best practices call for using `UTC` for storing information in the database:

> Even if your website is available in only one time zone, it’s still good practice to store data in UTC in your database. The main reason is Daylight Saving Time (DST). Many countries have a system of DST, where clocks are moved forward in spring and backward in autumn. If you’re working in local time, you’re likely to encounter errors twice a year, when the transitions happen.

https://docs.djangoproject.com/en/stable/topics/i18n/timezones/

Accessing settings

Once you've got all your settings, you'll want to use them in your code. To do so, add the following import to your file:

```
from django.conf import settings
```

You may then access your settings as attributes of the `settings` module, for example:

```
if not settings.DEBUG:
    email_user(user, message)
```

Using BASE_DIR to ensure app portability

It's a bad idea to hard code paths in your application. One should always use relative urls so that

your code can work seamlessly across different machines. The best way to set this up is to define a variable like this

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(__file__))
```

Then use this `BASE_DIR` variable to define all your other settings.

```
TEMPLATE_PATH = os.path.join(BASE_DIR, "templates")
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),

]
```

And so on. This ensures that you can port your code across different machines without any worries.

However, `os.path` is a bit verbose. For instance if your settings module is `project.settings.dev`, you will have to write:

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.dirname(__file__)))
```

An alternative is to use the `unipath` module (which you can install with `pip install unipath`).

```
from unipath import Path

BASE_DIR = Path(__file__).ancestor(2)  # or ancestor(3) if using a submodule

TEMPLATE_PATH = BASE_DIR.child('templates')
STATICFILES_DIRS = [
    BASE_DIR.child('static'),
]
```

Using Environment variables to manage Settings across servers

Using environment variables is a widely used way to setting an app's config depending on it environment, as stated in The Twelve-Factor App.

As configurations are likely to change between deployment environments, this is a very interesting way to modify the configuration without having to dig in the app's source code, as well as keeping secrets outside the application files and source code repository.

In Django, the main settings are located as `settings.py` in your project's folder. As it is a simple Python file, you can use Python's `os` module from the standard library to access the environment (and even have appropriate defaults).

settings.py

```
import os
```

```
SECRET_KEY = os.environ.get('APP_SECRET_KEY', 'unsafe-secret-key')

DEBUG = bool(os.environ.get('DJANGO_DEBUG', True) == 'False')

ALLOWED_HOSTS = os.environ.get('DJANGO_ALLOWED_HOSTS', '').split()

DATABASES = {
    'default': {
        'ENGINE': os.environ.get('APP_DB_ENGINE', 'django.db.backends.sqlite3'),
        'NAME': os.environ.get('DB_NAME', 'db.sqlite'),
        'USER': os.environ.get('DB_USER', ''),
        'PASSWORD': os.environ.get('DB_PASSWORD', ''),
        'HOST': os.environ.get('DB_HOST', None),
        'PORT': os.environ.get('DB_PORT', None),
        'CONN_MAX_AGE': 600,
    }
}
```

With Django you can change your database technology, so that you can use sqlite3 on your development machine (and that should be a sane default for committing to a source control system). Although this is possible it is not advisable:

> Backing services, such as the app's database, queueing system, or cache, is one area where dev/prod parity is important. (The Twelve-Factor App - Dev/prod parity)

For using a DATABASE_URL parameter for database connection, please take a look at the related example.

Using multiple settings

Django default project layout creates a single `settings.py`. This is often useful to split it like this:

```
myprojectroot/
    myproject/
        __init__.py
        settings/
            __init__.py
            base.py
            dev.py
            prod.py
            tests.py
```

This enables you to work with different settings according to whether you are in development, production, tests or whatever.

When moving from the default layout to this layout, the original `settings.py` becomes `settings/base.py`. When every other submodule will "subclass" `settings/base.py` by starting with `from .base import *`. For instance, here is what `settings/dev.py` may look like:

```
# -*- coding: utf-8 -*-
from .base import *  # noqa

DEBUG = True
```

```
INSTALLED_APPS.extend([
    'debug_toolbar',
])
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
INTERNAL_IPS = ['192.168.0.51', '192.168.0.69']
```

Alternative #1

For `django-admin` commands to work properly, you will have to set `DJANGO_SETTINGS_MODULE` environment variable (which defaults to `myproject.settings`). In development, you will set it to `myproject.settings.dev`. In production, you will set it to `myproject.settings.prod`. If you use a virtualenv, best is to set it in your `postactivate` script:

```
#!/bin/sh
export PYTHONPATH="/home/me/django_projects/myproject:$VIRTUAL_ENV/lib/python3.4"
export DJANGO_SETTINGS_MODULE="myproject.settings.dev"
```

If you want to use a settings module that is not pointed by `DJANGO_SETTINGS_MODULE` for one time, you can use the `--settings` option of `django-admin`:

```
django-admin test --settings=myproject.settings.tests
```

Alternative #2

If you want to leave `DJANGO_SETTINGS_MODULE` at its default configuration (`myproject.settings`), you can simply tell the `settings` module which configuration to load by placing the import in your `__init__.py` file.

In the above example, the same result could be achieved by having an `__init__.py` set to:

```
from .dev import *
```

Using multiple requirements files

Each requirements files should match the name of a settings files. Read Using multiple settings for more information.

Structure

```
djangoproject
├── config
│   ├── __init__.py
│   ├── requirements
│   │   ├── base.txt
│   │   ├── dev.txt
```

```
|   |   ├── test.txt
|   |   └── prod.txt
|   └── settings
└── manage.py
```

In `base.txt` file, place dependencies used in all environments.

```
# base.txt
Django==1.8.0
psycopg2==2.6.1
jinja2==2.8
```

And in all other files, include base dependencies with `-r base.txt`, and add specific dependencies needed for the current environment.

```
# dev.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in dev env
django-queryinspect==0.1.0
```

```
# test.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in test env
nose==1.3.7
django-nose==1.4
```

```
# prod.txt
-r base.txt # includes all dependencies in `base.txt`

# specific dependencies only used in production env
django-queryinspect==0.1.0
gunicorn==19.3.0
django-storages-redux==1.3
boto==2.38.0
```

Finally, to install dependencies. Example, on dev env : `pip install -r config/requirements/dev.txt`

Hiding secret data using a JSON file

When using a VCS such as Git or SVN, there are some secret data that must never be versioned (whether the repository is public or private).

Among those data, you find the `SECRET_KEY` setting and the database password.

A common practice to hide these settings from version control is to create a file `secrets.json` at the root of your project (thanks "*Two Scoops of Django*" for the idea):

```
{
    "SECRET_KEY": "N4HE:AMk:.Ader5354DR453TH8SHTQr",
    "DB_PASSWORD": "v3ry53cr3t"
}
```

And add it to your ignore list (`.gitignore` for git):

```
*.py[co]
*.sw[po]
*~
/secrets.json
```

Then add the following function to your `settings` module:

```
import json
import os
from django.core.exceptions import ImproperlyConfigured

with open(os.path.join(BASE_DIR, 'secrets.json')) as secrets_file:
    secrets = json.load(secrets_file)

def get_secret(setting, secrets=secrets):
    """Get secret setting or fail with ImproperlyConfigured"""
    try:
        return secrets[setting]
    except KeyError:
        raise ImproperlyConfigured("Set the {} setting".format(setting))
```

Then fill the settings this way:

```
SECRET_KEY = get_secret('SECRET_KEY')
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgres',
        'NAME': 'db_name',
        'USER': 'username',
        'PASSWORD': get_secret('DB_PASSWORD'),
    },
}
```

Credits: Two Scoops of Django: Best Practices for Django 1.8, by Daniel Roy Greenfeld and Audrey RoyGreenfeld. Copyright 2015 Two Scoops Press (ISBN 978-0981467344)

Using a DATABASE_URL from the environment

In PaaS sites such as Heroku, it is usual to receive the database information as a single URL environment variable, instead of several parameters (host, port, user, password...).

There is a module, `dj_database_url` which automatically extracts the DATABASE_URL environment variable to a Python dictionary appropriate for injecting the database settings in Django.

Usage:

```
import dj_database_url

if os.environ.get('DATABASE_URL'):
    DATABASES['default'] =
```

```
        dj_database_url.config(default=os.environ['DATABASE_URL'])
```

Read Settings online: https://riptutorial.com/django/topic/942/settings

Chapter 46: Signals

Parameters

Class/Method	The Why
UserProfile() Class	The UserProfile class extends the Django default User Model.
create_profile() method	The create_profile() method is executed, whenever a Django User model `post_save` signal is released.

Remarks

Now, the details.

Django signals is a way to inform your app of certain tasks (such as a model pre- or post-save or delete) when it takes place.

These signals allow you to perform actions of your choice immediately that signal is released.

For instance, **anytime** a new Django User is created, the User Model releases a signal, with associating params such as `sender=User` allowing you to specifically target your listening of signals to a specific activity that happens, in this case, a new user creation.

In the above example, the intention is to have a UserProfile object created, *immediately* after a User object is created. Therefore, by listening to a `post_save` signal from the `User` model (the default Django User Model) specifically, we create a `UserProfile` object just after a new `User` is created.

The Django Documentation provides extensive documentation on all the possible signals available .

However, the above example is to explain in practical terms a typical use case when using Signals can be a useful addition.

"With great power, comes great responsibility". It can be tempting to having signals scattered across your entire app or project just because they're awesome. Well, Don't. Because they're cool doesn't make them the go-to solution for every simple situation that comes to mind.

Signals are great for, as usual, not everything. Login/Logouts, signals are great. Key models releasing signs, like the User Model, if fine.

Creating signals for each and every model in your app can get overwhelming at a point, and defeat the whole idea of the sparring use of Django Signals.

Do not use signals when (based on Two Scoops of Django book):

- The signal relates to one particular model and can be moved into one of that model's methods, possibly called by `save()`.
- The signal can be replaced with a custom model manager method.
- The signal relates to a particular view and can be moved into that view

It might be okay to use signals when:

- Your signal receiver needs to make changes to more than one model.
- You want to dispatch the same signal from multiple apps and have them handled the same way by a common receiver.
- You want to invalidate a cache after a model save.
- You have an unusual scenario that needs a callback, and there's no other way to handle it besides using a signal. For example, you want to trigger something based on the `save()` or `init()` of a third-party app's model. You can't modify the third-party code and extending it might be impossible, so a signal provides a trigger for a callback.

Examples

Extending User Profile Example

This example is a snippet taken from the Extending Django User Profile like a Pro

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save

class UserProfile(models.Model):
    user = models.OneToOneField(User, related_name='user')
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)

def create_profile(sender, **kwargs):
    user = kwargs["instance"]
    if kwargs["created"]:
        user_profile = UserProfile(user=user)
        user_profile.save()
post_save.connect(create_profile, sender=User)
```

Different syntax to post/pre a signal

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver

class UserProfile(models.Model):
    user = models.OneToOneField(User, related_name='user')
    website = models.URLField(default='', blank=True)
    bio = models.TextField(default='', blank=True)

@receiver(post_save, sender=UserProfile)
def post_save_user(sender, **kwargs):
```

```
    user = kwargs.get('instance')
    if kwargs.get('created'):
        ...
```

How to find if it's an insert or update in the pre_save signal

By utilizing the `pre_save` we can determine if a `save` action on our database was about updating an existing object or creating a new one.

In order to achieve this you can check the state of the model object:

```
    @receiver(pre_save, sender=User)
    def pre_save_user(sender, instance, **kwargs):
        if not instance._state.adding:
            print ('this is an update')
        else:
            print ('this is an insert')
```

Now every time a `save` action takes place, the `pre_save` signal will run and will print:

- `this is an update` if the action derived from an update action.
- `this is an insert` if the action derived from an insert action.

Note that this method does not require any additional database queries.

Inheriting Signals on Extended Models

Django's signals are restricted to precise class signatures upon registration, and thus subclassed models are not immediately registered onto the same signal.

Take this model and signal for example

```
class Event(models.Model):
    user = models.ForeignKey(User)


class StatusChange(Event):
    ...


class Comment(Event):
    ...


def send_activity_notification(sender, instance: Event, raw: bool, **kwargs):
    """
    Fire a notification upon saving an event
    """

    if not raw:
        msg_factory = MessageFactory(instance.id)
        msg_factory.on_activity(str(instance))
post_save.connect(send_activity_notification, Event)
```

With extended models, you must manually attach the signal onto each subclass else they won't be effected.

```
post_save.connect(send_activity_notification, StatusChange)
post_save.connect(send_activity_notification, Comment)
```

With Python 3.6, you can leverage some additional class methods build into classes to automate this binding.

```
class Event(models.Model):

    @classmethod
    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        post_save.connect(send_activity_notification, cls)
```

Read Signals online: https://riptutorial.com/django/topic/2555/signals

Chapter 47: Template Tags and Filters

Examples

Custom Filters

Filters allows you to apply a function to a variable. This function may take **0** or **1** argument. Here is the syntax:

```
{{ variable|filter_name }}
{{ variable|filter_name:argument }}
```

Filters can be chained so this is perfectly valid:

```
{{ variable|filter_name:argument|another_filter }}
```

If translated to python the above line would give something like this:

```
print(another_filter(filter_name(variable, argument)))
```

In this example, we will write a custom filter `verbose_name` that applies to a Model (instance or class) or a QuerySet. It will return the verbose name of a model, or its verbose name plural if the argument is set to `True`.

```
@register.filter
def verbose_name(model, plural=False):
    """Return the verbose name of a model.
    `model` can be either:
      - a Model class
      - a Model instance
      - a QuerySet
      - any object refering to a model through a `model` attribute.

    Usage:
      - Get the verbose name of an object
          {{ object|verbose_name }}
      - Get the plural verbose name of an object from a QuerySet
          {{ objects_list|verbose_name:True }}
    """
    if not hasattr(model, '_meta'):
        # handle the case of a QuerySet (among others)
        model = model.model
    opts = model._meta
    if plural:
        return opts.verbose_name_plural
    else:
        return opts.verbose_name
```

Simple tags

The simplest way of defining a custom template tag is to use a `simple_tag`. These are very straightforward to setup. The function name will be the tag name (though you can override it), and arguments will be tokens ("words" separated by spaces, except spaces enclosed between quotes). It even supports keyword arguments.

Here is a useless tag that will illustrate our example:

```
{% useless 3 foo 'hello world' foo=True bar=baz.hello|capfirst %}
```

Let `foo` and `baz` be context variables like the following:

```
{'foo': "HELLO", 'baz': {'hello': "world"}}
```

Say we want this very useless tag to render like this:

```
HELLO;hello world;bar:World;foo:True<br/>
HELLO;hello world;bar:World;foo:True<br/>
HELLO;hello world;bar:World;foo:True<br/>
```

Kind of arguments concatenation repeated 3 times (3 being the first argument).

Here is what the tag implementation may look like:

```
from django.utils.html import format_html_join

@register.simple_tag
def useless(repeat, *args, **kwargs):
    output = ';'.join(args + ['{}:{}'.format(*item) for item in kwargs.items()])
    outputs = [output] * repeat
    return format_html_join('\n', '{}<br/>', ((e,) for e in outputs))
```

`format_html_join` allows to mark `<br/>` as safe HTML, but not the content of `outputs`.

Advanced custom tags using Node

Sometimes what you want to do is just too complex for a `filter` or a `simple_tag`. Fow this you will need to create a compilation function and a renderer.

In this example we will create a template tag `verbose_name` with the following syntax:

Example	Description
`{% verbose_name obj %}`	Verbose name of a model
`{% verbose_name obj 'status' %}`	Verbose name of the field "status"
`{% verbose_name obj plural %}`	Verbose name plural of a model
`{% verbose_name obj plural capfirst %}`	Capitalized verbose name plural of a model

Example	Description
`{% verbose_name obj 'foo' capfirst %}`	Capitalized verbose name of a field
`{% verbose_name obj field_name %}`	Verbose name of a field from a variable
`{% verbose_name obj 'foo'|add:'_bar' %}`	Verbose name of a field "foo_bar"

The reason why we can't do this with a simple tag is that `plural` and `capfirst` are neither variables nor strings, they are "keywords". We could obviously decide to pass them as strings `'plural'` and `'capfirst'`, but it may conflict with fields with these names. Would `{% verbose_name obj 'plural' %}` mean "verbose name plural of `obj`" or "verbose name of `obj.plural`"?

First let's create the compilation function:

```
@register.tag(name='verbose_name')
def do_verbose_name(parser, token):
    """
    - parser: the Parser object. We will use it to parse tokens into
              nodes such as variables, strings, ...
    - token: the Token object. We will use it to iterate each token
             of the template tag.
    """
    # Split tokens within spaces (except spaces inside quotes)
    tokens = token.split_contents()
    tag_name = tokens[0]
    try:
        # each token is a string so we need to parse it to get the actual
        # variable instead of the variable name as a string.
        model = parser.compile_filter(tokens[1])
    except IndexError:
        raise TemplateSyntaxError(
            "'{}' tag requires at least 1 argument.".format(tag_name))

    field_name = None
    flags = {
        'plural': False,
        'capfirst': False,
    }

    bits = tokens[2:]
    for bit in bits:
        if bit in flags.keys():
            # here we don't need `parser.compile_filter` because we expect
            # 'plural' and 'capfirst' flags to be actual strings.
            if flags[bit]:
                raise TemplateSyntaxError(
                    "'{}' tag only accept one occurrence of '{}' flag".format(
                        tag_name, bit)
                )
            flags[bit] = True
            continue
        if field_name:
            raise TemplateSyntaxError((
                "'{}' tag only accept one field name at most. {} is the second "
                "field name encountered."
            ).format(tag_name, bit)
        field_name = parser.compile_filter(bit)
```

```
    # VerboseNameNode is our renderer which code is given right below
    return VerboseNameNode(model, field_name, **flags)
```

And now the renderer:

```
class VerboseNameNode(Node):

    def __init__(self, model, field_name=None, **flags):
        self.model = model
        self.field_name = field_name
        self.plural = flags.get('plural', False)
        self.capfirst = flags.get('capfirst', False)

    def get_field_verbose_name(self):
        if self.plural:
            raise ValueError("Plural is not supported for fields verbose name.")
        return self.model._meta.get_field(self.field_name).verbose_name

    def get_model_verbose_name(self):
       if self.plural:
           return self.model._meta.verbose_name_plural
       else:
           return self.model._meta.verbose_name

    def render(self, context):
        """This is the main function, it will be called to render the tag.
        As you can see it takes context, but we don't need it here.
        For instance, an advanced version of this template tag could look for an
        `object` or `object_list` in the context if `self.model` is not provided.
        """
        if self.field_name:
            verbose_name = self.get_field_verbose_name()
        else:
            verbose_name = self.get_model_verbose_name()
        if self.capfirst:
            verbose_name = verbose_name.capitalize()
        return verbose_name
```

Read Template Tags and Filters online: https://riptutorial.com/django/topic/1305/template-tags-and-filters

Chapter 48: Templating

Examples

Variables

Variables you have provided in your view context can be accessed using double-brace notation:

In your `views.py`:

```
class UserView(TemplateView):
  """ Supply the request user object to the template """

  template_name = "user.html"

  def get_context_data(self, **kwargs):
    context = super(UserView, self).get_context_data(**kwargs)
    context.update(user=self.request.user)
    return context
```

In `user.html`:

```
<h1>{{ user.username }}</h1>

<div class="email">{{ user.email }}</div>
```

The dot notation will access:

- properties of the object, e.g. `user.username` will be `{{ user.username }}`
- dictionary lookups, e.g. `request.GET["search"]` will be `{{ request.GET.search }}`
- methods with no arguments, e.g. `users.count()` will be `{{ user.count }}`

Template variables cannot access methods that take arguments.

Variables can also be tested and looped over:

```
{% if user.is_authenticated %}
  {% for item in menu %}
    <li><a href="{{ item.url }}">{{ item.name }}</a></li>
  {% endfor %}
{% else %}
  <li><a href="{% url 'login' %}">Login</a>
{% endif %}
```

URLs are accessed using `{% url 'name' %}` format, where the names correspond to names in your `urls.py`.

`{% url 'login' %}` - Will probably render as `/accounts/login/`
`{% url 'user_profile' user.id %}` - Arguments for URLs are supplied in order
`{% url next %}` - URLs can be variables

Templating in Class Based Views

You can pass data to a template in a custom variable.

In your `views.py`:

```
from django.views.generic import TemplateView
from MyProject.myapp.models import Item

class ItemView(TemplateView):
    template_name = "item.html"

    def items(self):
        """ Get all Items """
        return Item.objects.all()

    def certain_items(self):
        """ Get certain Items """
        return Item.objects.filter(model_field="certain")

    def categories(self):
        """ Get categories related to this Item """
        return Item.objects.get(slug=self.kwargs['slug']).categories.all()
```

A simple list in your `item.html`:

```
{% for item in view.items %}
<ul>
    <li>{{ item }}</li>
</ul>
{% endfor %}
```

You can also retrieve additional properties of the data.

Assuming your model `Item` has a `name` field:

```
{% for item in view.certain_items %}
<ul>
    <li>{{ item.name }}</li>
</ul>
{% endfor %}
```

Templating in Function Based Views

You can use a template in a function based view as follows:

```
from django.shortcuts import render

def view(request):
    return render(request, "template.html")
```

If you want to use template variables, you can do so as follows:

```
from django.shortcuts import render

def view(request):
    context = {"var1": True, "var2": "foo"}
    return render(request, "template.html", context=context)
```

Then, in `template.html`, you can refer to your variables like so:

```
<html>
{% if var1 %}
    <h1>{{ var2 }}</h1>
{% endif %}
</html>
```

Template filters

The Django template system has built-in *tags* and *filters*, which are functions inside template to render content in a specific way. Multiple filters can be specified with pipes and filters can have arguments, just as in variable syntax.

```
{{ "MAINROAD 3222"|lower }}    # mainroad 3222
{{ 10|add:15}}                 # 25
{{ "super"|add:"glue" }}       # superglue
{{ "A7"|add:"00" }}            # A700
{{ myDate | date:"D d M Y"}}   # Wed 20 Jul 2016
```

A list of available **built-in filters** can be found at https://docs.djangoproject.com/en/dev/ref/templates/builtins/#ref-templates-builtins-filters .

Creating custom filters

To add your own template filters, create a folder named `templatetags` inside your app folder. Then add a `__init__.py`, and the file your file that will contain the filters:

```
#/myapp/templatetags/filters.py
from django import template

register = template.Library()

@register.filter(name='tostring')
def to_string(value):
    return str(value)
```

To actually use the filter you need to load it in your template:

```
#templates/mytemplate.html
{% load filters %}
{% if customer_id|tostring = customer %} Welcome back {% endif%}
```

Tricks

Even though the filters seem simple at first, it allows to do some nifty things:

```
{% for x in ""|ljust:"20" %}Hello World!{% endfor %}    # Hello World!Hello World!Hel...
{{ user.name.split|join:"_" }} ## replaces whitespace with '_'
```

See also template tags for more information.

Prevent sensitive methods from being called in templates

When an object is exposed to the template context, its arguments-less methods are available. This is useful when these functions are "getters". But it can be hazardeous if these methods alter some data or have some side effects. Eventhough you likely trust the template writer, he may not be aware of a function's side effects or think call the wrong attribute by mistake.

Given the following model:

```
class Foobar(models.Model):
    points_credit = models.IntegerField()

    def credit_points(self, nb_points=1):
        """Credit points and return the new points credit value."""
        self.points_credit = F('points_credit') + nb_points
        self.save(update_fields=['points_credit'])
        return self.points_credit
```

If you write this, by mistake, in a template:

```
 You have {{ foobar.credit_points }} points!
```

This will increment the number of points each time the template is called. And you may not even notice it.

To prevent this, you must set the `alters_data` attribute to `True` to methods that have side effects. This will make it impossible to call them from a template.

```
def credit_points(self, nb_points=1):
    """Credit points and return the new points credit value."""
    self.points_credit = F('points_credit') + nb_points
    self.save(update_fields=['points_credit'])
    return self.points_credit
credit_points.alters_data = True
```

Use of {% extends %} , {% include %} and {% blocks %}

summary

- **{% extends %}**: this declares the template given as an argument as the current template's parent. Usage: `{% extends 'parent_template.html' %}`.
- **{% block %}{% endblock %}**: This is used to define sections in your templates, so that if another template extends this one, it'll be able to replace whatever html code has been

written inside of it. Blocks are identified by their name. Usage: `{% block content %} <html_code> {% endblock %}`.

- **{% include %}**: this will insert a template within the current one. Be aware that the included template will receive the request's context, and you can give it custom variables too. Basic usage: `{% include 'template_name.html' %}`, usage with variables: `{% include 'template_name.html' with variable='value' variable2=8 %}`

Guide

Suppose you are building up your front end side code with having common layouts for all code and you do not want to repeat the code for every template. Django gives you in built tags for doing so.
Suppose we have one blog website having 3 templates which share the same layout:

```
project_directory
    ..
    templates
      front-page.html
      blogs.html
      blog-detail.html
```

1) Define `base.html` file,

```
<html>
  <head>
  </head>

  <body>
        {% block content %}
        {% endblock %}
   </body>
</html>
```

2) Extend it in `blog.html` like,

```
{% extends 'base.html' %}

{% block content %}
    # write your blog related code here
{% endblock %}

# None of the code written here will be added to the template
```

Here we extended the base layout so its HTML layout is now available in the `blog.html` file.The concept of `{ % block %}` is template inheritance which allows you to build a base “skeleton” template that contains all the common elements of your site and defines blocks that child templates can override.

3) Now suppose all of your 3 templates also having same HTML div which defines some popular posts.Instead of being written the 3 times create one new template `posts.html`.

blog.html

```
{% extends 'base.html' %}

{% block content %}
    # write your blog related code here
    {% include 'posts.html' %} # includes posts.html in blog.html file without passing any
data
    <!-- or -->
    {% include 'posts.html' with posts=postdata %} # includes posts.html in blog.html file
with passing posts data which is context of view function returns.
{% endblock %}
```

Read Templating online: https://riptutorial.com/django/topic/588/templating

Chapter 49: Timezones

Introduction

Timezones are often a hassle for developers. Django offers some great utilities at your disposal to make timezones easy to work with.

Even if your project is operating in a single time zone, it is still good practice to store data as UTC in your database to handle cases of daylight savings. If you are operating on multiple timezones then storing time data as UTC is a must.

Examples

Enable Time Zone Support

First is first, ensure that `USE_TZ = True` in your `settings.py` file. Also set a default time zone value to `TIME_ZONE` such as `TIME_ZONE='UTC'`. View a complete list of timezones here.

If `USE_TZ` is False, `TIME_ZONE` will be the time zone that Django will use to store all datetimes. When `USE_TZ` is enabled, `TIME_ZONE` is the default time zone that Django will use to display datetimes in templates and to interpret datetimes entered in forms.

With time zone support enabled, django will store `datetime` data in the database as the time zone `UTC`

Setting Session Timezones

Python's `datetime.datetime` objects have a `tzinfo` attribute that is used to store time zone information. When the attribute is set the object is considered Aware, when the attribute is not set it is considered a Naive.

To ensure that a timezone is naive or aware, you can use `.is_naive()` and `.is_aware()`

If you have `USE_TZ` enabled in your `settings.py` file, a `datetime` will have time zone information attached to it as long as your default `TIME_ZONE` is set in `settings.py`

While this default time zone may be good in some cases it is likely not enough especially if you are handling users in multiple time zones. In order to accomplish this, middleware must be used.

```
import pytz

from django.utils import timezone

# make sure you add `TimezoneMiddleware` appropriately in settings.py
class TimezoneMiddleware(object):
    """
    Middleware to properly handle the users timezone
    """
```

```
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # make sure they are authenticated so we know we have their tz info.
        if request.user.is_authenticated():
            # we are getting the users timezone that in this case is stored in
            # a user's profile
            tz_str = request.user.profile.timezone
            timezone.activate(pytz.timezone(tz_str))
        # otherwise deactivate and the default time zone will be used anyway
        else:
            timezone.deactivate()

        response = self.get_response(request)
        return response
```

There are a few new things going on. To learn more about middleware and what it does check out that part of the documentation. In `__call__` we are handling the setting of the timezone data. At first we make sure the user is authenticated, to make sure that we have timezone data for this user. Once we know we do, we active the timezone for the users session using `timezone.activate()`. In order to convert the time zone string we have to something usable by datetime, we use `pytz.timezone(str)`.

Now, when datetime objects are accessed in templates they will automatically be converted from the 'UTC' format of the database to whatever time zone the user is in. Just access the datetime object and its timezone will be set assuming the previous middleware is set up properly.

```
{{ my_datetime_value }}
```

If you desire a fine grained control of whether the user's timezone is used take a look at the following:

```
{% load tz %}
{% localtime on %}
    {# this time will be respect the users time zone #}
    {{ your_date_time }}
{% endlocaltime %}

{% localtime off %}
    {# this will not respect the users time zone #}
    {{ your_date_time }}
{% endlocaltime %}
```

Note, this method described only works in Django 1.10 and on. To support django from prior to 1.10 look into MiddlewareMixin

Read Timezones online: https://riptutorial.com/django/topic/10566/timezones

Chapter 50: Unit Testing

Examples

Testing - a complete example

This assumes that you have read the documentation about starting a new Django project. Let us assume that the main app in your project is named td (short for test driven). To create your first test, create a file named test_view.py and copy paste the following content into it.

```
from django.test import Client, TestCase

class ViewTest(TestCase):

    def test_hello(self):
        c = Client()
        resp = c.get('/hello/')
        self.assertEqual(resp.status_code, 200)
```

You can run this test by

```
 ./manage.py test
```

and it will most naturally fail! You will see an error similar to the following.

```
Traceback (most recent call last):
  File "/home/me/workspace/td/tests_view.py", line 9, in test_hello
    self.assertEqual(resp.status_code, 200)
AssertionError: 200 != 404
```

Why does that happen? Because we haven't defined a view for that! So let's do it. Create a file called views.py and place in it the following code

```
from django.http import HttpResponse
def hello(request):
    return HttpResponse('hello')
```

Next map it to the /hello/ by editing urls py as follows:

```
from td import views

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/', views.hello),
    ....
]
```

Now run the test again `./manage.py test` again and viola!!

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```

Testing Django Models Effectively

Assuming a class

```
from django.db import models

class Author(models.Model):
   name = models.CharField(max_length=50)

    def __str__(self):
        return self.name

    def get_absolute_url(self):
        return reverse('view_author', args=[str(self.id)])


class Book(models.Model):
    author = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    private = models.BooleanField(default=false)
    publish_date = models.DateField()

    def get_absolute_url(self):
        return reverse('view_book', args=[str(self.id)])

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

Testing examples

```
from django.test import TestCase
from .models import Book, Author

class BaseModelTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super(BaseModelTestCase, cls).setUpClass()
        cls.author = Author(name='hawking')
        cls.author.save()
        cls.first_book = Book(author=cls.author, name="short_history_of_time")
        cls.first_book.save()
        cls.second_book = Book(author=cls.author, name="long_history_of_time")
        cls.second_book.save()


class AuthorModelTestCase(BaseModelTestCase):
    def test_created_properly(self):
         self.assertEqual(self.author.name, 'hawking')
         self.assertEqual(True, self.first_book in self.author.book_set.all())

    def test_absolute_url(self):
```

```
        self.assertEqual(self.author.get_absolute_url(), reverse('view_author',
args=[str(self.author.id)]))

class BookModelTestCase(BaseModelTestCase):

    def test_created_properly(self:
        ...
        self.assertEqual(1, len(Book.objects.filter(name__startswith='long'))

    def test_absolute_url(self):
        ...
```

Some points

- `created_properly` tests are used to verify the state properties of django models. They help catch sitautions where we've changed default values, file_upload_paths etc.
- `absolute_url` might seem trivial but I've found that it's helped me prevent some bugs when changing url paths
- I similarly write test cases for all the methods implemented inside a model (using `mock` objects etc)
- By defining a common `BaseModelTestCase` we can setup the necessary relationships between models to ensure proper testing.

Finally, when in doubt, write a test. Trivial behavior changes are caught by paying attention to detail and long forgotten pieces of code don't end up causing unnecessary trouble.

Testing Access Control in Django Views

tl;dr : Create a base class that defines two user objects (say `user` and `another_user`). Create your other models and define three `Client` instances.

- `self.client` : Representing `user` logged in browser
- `self.another_client` : Representing `another_user` 's client
- `self.unlogged_client` : Representing unlogged person

Now access all your public and private urls from these three client objects and dictact the response you expect. Below I've showcased the strategy for a `Book` object that can either be `private` (owned by a few privileged users) or `public` (visible to everyone).

```
from django.test import TestCase, RequestFactory, Client
from django.core.urlresolvers import reverse

class BaseViewTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super(BaseViewTestCase, cls).setUpClass()
        cls.client = Client()
        cls.another_client = Client()
        cls.unlogged_client = Client()
        cls.user = User.objects.create_user(
                'dummy',password='dummy'
                )
```

```
        cls.user.save()
        cls.another_user = User.objects.create_user(
                'dummy2', password='dummy2'
                )
        cls.another_user.save()
        cls.first_book = Book.objects.create(
                name='first',
                private = true
        )
        cls.first_book.readers.add(cls.user)
        cls.first_book.save()
        cls.public_book = Template.objects.create(
                name='public',
                private=False
        )
        cls.public_book.save()


    def setUp(self):
        self.client.login(username=self.user.username, password=self.user.username)
        self.another_client.login(username=self.another_user.username,
password=self.another_user.username)


"""
   Only cls.user owns the first_book and thus only he should be able to see it.
   Others get 403(Forbidden) error
"""
class PrivateBookAccessTestCase(BaseViewTestCase):

    def setUp(self):
        super(PrivateBookAccessTestCase, self).setUp()
        self.url = reverse('view_book',kwargs={'book_id':str(self.first_book.id)})

    def test_user_sees_own_book(self):
        response = self.client.get(self.url)
        self.assertEqual(200, response.status_code)
        self.assertEqual(self.first_book.name,response.context['book'].name)
        self.assertTemplateUsed('myapp/book/view_template.html')

    def test_user_cant_see_others_books(self):
        response = self.another_client.get(self.url)
        self.assertEqual(403, response.status_code)

    def test_unlogged_user_cant_see_private_books(self):
        response = self.unlogged_client.get(self.url)
        self.assertEqual(403, response.status_code)

"""
    Since book is public all three clients should be able to see the book
"""
 class PublicBookAccessTestCase(BaseViewTestCase):

    def setUp(self):
        super(PublicBookAccessTestCase, self).setUp()
        self.url = reverse('view_book',kwargs={'book_id':str(self.public_book.id)})

    def test_user_sees_book(self):
        response = self.client.get(self.url)
        self.assertEqual(200, response.status_code)
        self.assertEqual(self.public_book.name,response.context['book'].name)
```

```
        self.assertTemplateUsed('myapp/book/view_template.html')

    def test_another_user_sees_public_books(self):
        response = self.another_client.get(self.url)
        self.assertEqual(200, response.status_code)

    def test_unlogged_user_sees_public_books(self):
        response = self.unlogged_client.get(self.url)
        self.assertEqual(200, response.status_code)
```

The Database and Testing

Django uses special database settings when testing so that tests can use the database normally but by default run on an empty database. Database changes in one test will not be seen by another. For example, both of the following tests will pass:

```
from django.test import TestCase
from myapp.models import Thing

class MyTest(TestCase):

    def test_1(self):
        self.assertEqual(Thing.objects.count(), 0)
        Thing.objects.create()
        self.assertEqual(Thing.objects.count(), 1)

    def test_2(self):
        self.assertEqual(Thing.objects.count(), 0)
        Thing.objects.create(attr1="value")
        self.assertEqual(Thing.objects.count(), 1)
```

Fixtures

If you want to have database objects used by multiple tests, either create them in the `setUp` method of the test case. Additionally, if you have defined fixtures in your django project, they can be included like so:

```
class MyTest(TestCase):
    fixtures = ["fixture1.json", "fixture2.json"]
```

By default, django is looking for fixtures in the `fixtures` directory in each app. Further directories can be set using the `FIXTURE_DIRS` setting:

```
# myapp/settings.py
FIXTURE_DIRS = [
    os.path.join(BASE_DIR, 'path', 'to', 'directory'),
]
```

Let's assume you have created a model as follows:

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

```
class Person(models.Model):
    """A person defined by his/her first- and lastname."""
    firstname = models.CharField(max_length=255)
    lastname = models.CharField(max_length=255)
```

Then your .json fixtures could look like that:

```
# fixture1.json
[
    { "model": "myapp.person",
        "pk": 1,
        "fields": {
            "firstname": "Peter",
            "lastname": "Griffin"
        }
    },
    { "model": "myapp.person",
        "pk": 2,
        "fields": {
            "firstname": "Louis",
            "lastname": "Griffin"
        }
    },
]
```

Reuse the test-database

To speed up your test-runs you can tell the management-command to reuse the test-database (and to prevent it from being created before and deleted after every test-run). This can be done using the keepdb (or shorthand `-k`) flag like so:

```
# Reuse the test-database (since django version 1.8)
$ python manage.py test --keepdb
```

Limit the number of tests executed

It is possible to limit the tests executed by `manage.py test` by specifying which modules should be discovered by the test runner:

```
# Run only tests for the app names "app1"
$ python manage.py test app1

# If you split the tests file into a module with several tests files for an app
$ python manage.py test app1.tests.test_models

# it's possible to dig down to individual test methods.
$ python manage.py test app1.tests.test_models.MyTestCase.test_something
```

If you want to run a bunch of tests you can pass a pattern of filenames. For example, you may want to run only tests that involving of your models:

```
$ python manage.py test -p test_models*
```

```
Creating test database for alias 'default'...
.............................................
----------------------------------------------------------------------
Ran 115 tests in 3.869s

OK
```

Finally, it is possible to stop the test suite at the first fail, using `--failfast`. This argument allows to get quickly the potential error encountered in the suite:

```
$ python manage.py test app1
...F..
----------------------------------------------------------------------
Ran 6 tests in 0.977s

FAILED (failures=1)


$ python manage.py test app1 --failfast
...F
======================================================================
[Traceback of the failing test]
----------------------------------------------------------------------
Ran 4 tests in 0.372s

FAILED (failures=1)
```

Read Unit Testing online: https://riptutorial.com/django/topic/1232/unit-testing

Chapter 51: URL routing

Examples

How Django handles a request

Django handles a request by routing the incoming URL path to a view function. The view function is responsible for returning a response back to the client making the request. Different URLs are usually handled by different view functions. To route the request to a specific view function, Django looks at your URL configuration (or URLconf for short). The default project template defines the URLconf in `<myproject>/urls.py`.

Your URLconf should be a python module that defines an attribute named `urlpatterns`, which is a list of `django.conf.urls.url()` instances. Each `url()` instance must at minimum define a regular expression (a regex) to match against the URL, and a target, which is either a view function or a different URLconf. If a URL pattern targets a view function, it is a good idea to give it a name to easily reference the pattern later on.

Let's take a look at a basic example:

```
# In <myproject>/urls.py

from django.conf.urls import url

from myapp.views import home, about, blog_detail

urlpatterns = [
    url(r'^$', home, name='home'),
    url(r'^about/$', about, name='about'),
    url(r'^blog/(?P<id>\d+)/$', blog_detail, name='blog-detail'),
]
```

This URLconf defines three URL patterns, all targeting a view: `home`, `about` and `blog-detail`.

- `url(r'^$', home, name='home'),`

The regex contains a start anchor '^', immediately followed by an end anchor '$'. This pattern will match requests where the URL path is an empty string, and route them to the `home` view defined in `myapp.views`.

- `url(r'^about/$', about, name='about'),`

This regex contains a start anchor, followed by the literal string `about/`, and the end anchor. This will match the URL `/about/` and route it to the `about` view. Since every non-empty URL start with a `/`, Django conveniently cuts of the first slash for you.

- `url(r'^blog/(?P<id>\d+)/$', blog_detail, name='blog-detail'),`

This regex is a bit more complex. It defines the start anchor and the literal string `blog/`, like the

previous pattern. The next part, `(?P<id>\d+)`, is called a capturing group. A capturing group, like its name suggest, captures a part of the string, and Django passes the captured string as an argument to the view function.

The syntax of a capturing group is `(?P<name>pattern)`. `name` defines the name of the group, which is also the name that Django uses to pass the argument to the view. The pattern defines which characters are matched by the group.

In this case, the name is `id`, so the function `blog_detail` must accept a parameter named `id`. The pattern is `\d+`. `\d` signifies that the pattern only matches number characters. `+` signifies that the pattern must match one or more characters.

Some common patterns:

Pattern	Used for	Matches
`\d+`	id	One or more numerical characters
`[\w-]+`	slug	One or more alphanumerical characters, underscores or dashes
`[0-9]{4}`	year (long)	Four numbers, zero through nine
`[0-9]{2}`	year (short) month day of month	Two numbers, zero through nine
`[^/]+`	path segment	Anything except a slash

The capturing group in the `blog-detail` pattern is followed by a literal `/`, and the end anchor.

Valid URLs include:

- `/blog/1/ # passes id='1'`
- `/blog/42/ # passes id='42'`

Invalid URLs are for example:

- `/blog/a/ # 'a' does not match '\d'`
- `/blog// # no characters in the capturing group does not match '+'`

Django processes each URL pattern in the same order they are defined in `urlpatterns`. This is important if multiple patterns can match the same URL. For example:

```
urlpatterns = [
    url(r'blog/(?P<slug>[\w-]+)/$', blog_detail, name='blog-detail'),
    url(r'blog/overview/$', blog_overview, name='blog-overview'),
]
```

In the above URLconf, the second pattern is not reachable. The pattern would match the URL

`/blog/overview/`, but instead of calling the `blog_overview` view, the URL will first match the `blog-detail` pattern and call the `blog_detail` view with an argument `slug='overview'`.

To make sure that the URL `/blog/overview/` is routed to the `blog_overview` view, the pattern should be put above the `blog-detail` pattern:

```
urlpatterns = [
    url(r'blog/overview/$', blog_overview, name='blog-overview'),
    url(r'blog/(?P<slug>[\w-]+)/$', blog_detail, name='blog-detail'),
]
```

Set the URL namespace for a reusable app (Django 1.9+)

Configure your app's URLconf to automatically use a URL namespace by setting the `app_name` attribute:

```
# In <myapp>/urls.py
from django.conf.urls import url

from .views import overview

app_name = 'myapp'
urlpatterns = [
    url(r'^$', overview, name='overview'),
]
```

This will set the application namespace to `'myapp'` when it is included in the root URLconf>. The user of your reusable app does not need to do any configuration other than including your URLs:

```
# In <myproject>/urls.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^myapp/', include('myapp.urls')),
]
```

Your reusable app can now reverse URLs using the application namespace:

```
>>> from django.urls import reverse
>>> reverse('myapp:overview')
'/myapp/overview/'
```

The root URLconf can still set an instance namespace with the `namespace` parameter:

```
# In <myproject>/urls.py
urlpatterns = [
    url(r'^myapp/', include('myapp.urls', namespace='mynamespace')),
]
```

Both the application namespace and instance namespace can be used to reverse the URLs:

```
>>> from django.urls import reverse
>>> reverse('myapp:overview')
'/myapp/overview/'
>>> reverse('mynamespace:overview')
'/myapp/overview/'
```

The instance namespace defaults to the application namespace if it is not explicitly set.

Read URL routing online: https://riptutorial.com/django/topic/3299/url-routing

Chapter 52: Using Redis with Django - Caching Backend

Remarks

Using django-redis-cache or django-redis are both effective solutions for storing all cached items. While it is certainly possible for Redis to be setup directly as a `SESSION_ENGINE`, one effective strategy is to setup the caching (as above) and declare your default cache as a `SESSION_ENGINE`. While this is really the topic for another documentaiton article, its relevance leads to inclusion.

Simply add the following to `settings.py`:

```
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
```

Examples

Using django-redis-cache

One potential implementation of Redis as a backend caching utility is the django-redis-cache package.

This example assumes you already have a Redis server operating.

```
$ pip install django-redis-cache
```

Edit your `settings.py` to include a `CACHES` object (see Django documentation on caching).

```
CACHES = {
    'default': {
        'BACKEND': 'redis_cache.RedisCache',
        'LOCATION': 'localhost:6379',
        'OPTIONS': {
            'DB': 0,
        }
    }
}
```

Using django-redis

One potential implementation of Redis as a backend caching utility is the django-redis package.

This example assumes you already have a Redis server operating.

```
$ pip install django-redis
```

Edit your `settings.py` to include a `CACHES` object (see Django documentation on caching).

```
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
        }
    }
}
```

Read Using Redis with Django - Caching Backend online:
https://riptutorial.com/django/topic/4085/using-redis-with-django---caching-backend

Chapter 53: Views

Introduction

A view function, or view for short, is simply a Python function that takes a Web request and returns a Web response. -Django Documentation-

Examples

[Introductory] Simple View (Hello World Equivalent)

Let's create a very simple view to respond a "Hello World" template in html format.

1. To do that go to `my_project/my_app/views.py` (Here we are housing our view functions) and add the following view:

```
from django.http import HttpResponse

def hello_world(request):
    html = "<html><title>Hello World!</title><body>Hello World!</body></html>"
    return HttpResponse(html)
```

2. To call this view, we need to configure a url pattern in `my_project/my_app/urls.py`:

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^hello_world/$', views.hello_world, name='hello_world'),
]
```

3. Start the server: `python manage.py runserver`

   Now if we hit `http://localhost:8000/hello_world/`, our template (the html string) will be rendered in our browser.

Read Views online: https://riptutorial.com/django/topic/7490/views

Credits

S. No	Chapters	Contributors
1	Getting started with Django	A. Raza, Abhishek Jain, Aidas Bendoraitis, Alexander Tyapkov, Ankur Gupta, Anthony Pham, Antoine Pinsard, arifin4web, Community, e4c5, elbear, ericdwang, ettanany, Franck Dernoncourt, greatwolf, ilse2005, Ivan Semochkin, J F, Jared Hooper, John, John Moutafis, JRodDynamite, Kid Binary, knbk, Louis, Luis Alberto Santana, lxer, maciek, McAbra, MiniGunnR, mnoronha, Nathan Osman, naveen.panwar, nhydock, Nikita Davidenko, noqɬʎdʎzɐɹƆ, Rahul Gupta, rajarshig, Ron, ruddra, sarvajeetsuman, shacker, ssice, Stryker, techydesigner, The Brewmaster, Thereissoupinmyfly, Tom, WesleyJohnson, Zags
2	Administration	Antoine Pinsard, coffee-grinder, George H., Ivan Semochkin, noqɬʎdʎzɐɹƆ, ssice
3	ArrayField - a PostgreSQL-specific field	Antoine Pinsard, e4c5, noqɬʎdʎzɐɹƆ
4	Async Tasks (Celery)	iankit, Mevin Babu
5	Authentication Backends	knbk, Rahul Gupta
6	Class based views	Antoine Pinsard, Antwane, coffee-grinder, e4c5, gkr, knbk, maciek, masnun, Maxime Lorant, nicorellius, pleasedontbelong, Pureferret
7	Context Processors	Antoine Pinsard, Brian Artschwager, Dan Russell, Daniil Ryzhkov, fredley
8	Continuous Integration With Jenkins	pnovotnak
9	CRUD in Django	aisflat439, George H.
10	Custom Managers and Querysets	abidibo, knbk, sudshekhar, Trivial
11	Database Routers	fredley, knbk

12	Database Setup	Ahmad Anwar, Antoine Pinsard, Evans Murithi, Kid Binary, knbk , lxer, Majid, Peter Mortensen
13	Database transactions	Ian Clark
14	Debugging	Antoine Pinsard, Ashutosh, e4c5, Kid Binary, knbk, Sayse, Udi
15	Deployment	Antoine Pinsard, Arpit Solanki, CodeFanatic23, I Am Batman, Ivan Semochkin, knbk, lxer, Maxime S., MaxLunar, Meska, no ɥʎdʎzɐɹƆ, rajarshig, Rishabh Agrahari, Roald Nefs, Rohini Choudhary, sebb
16	Django and Social Networks	Aidas Bendoraitis, aisflat439, Carlos Rojas, Ivan Semochkin, Rexford, Simplans
17	Django from the command line.	e4c5, OliPro007
18	Django Rest Framework	The Brewmaster
19	django-filter	4444, Ahmed Atalla
20	Extending or Substituting User Model	Antoine Pinsard, Jon Clements, mnoronha, Raito, Rexford, rigdonmr, Rishabh Agrahari, Roald Nefs, techydesigner, The_Cthulhu_Kid
21	F() expressions	Antoine Pinsard, John Moutafis, Linville, Omar Shehata, RamenChef, Roald Nefs
22	Form Widgets	Antoine Pinsard, ettanany
23	Forms	Aidas Bendoraitis, Antoine Pinsard, Daniel Rucci, ettanany, George H., knbk, NBajanca, nicorellius, RamenChef, rumman0786, sudshekhar, trpt4him
24	Formsets	naveen.panwar
25	Generic Views	nikolas-berlin
26	How to reset django migrations	Cristus Cleetus
27	How to use Django with Cookiecutter?	Atul Mishra, noɥʎdʎzɐɹƆ, OliPro007, RamenChef
28	Internationalization	Antoine Pinsard, dmvrtx
29	JSONField - a	Antoine Pinsard, Daniil Ryzhkov, Matthew Schinckel, noɥʎd

	PostgreSQL specific field	ʎzɐɹƆ, Omar Shehata, techydesigner
30	Logging	Antwane, Brian Artschwager, RamenChef
31	Management Commands	Antoine Pinsard, aquasan, Brian Artschwager, HorsePunchKid, Ivan Semochkin, John Moutafis, knbk, lxer, MarZab, Nikolay Fominyh, pbaranay, ptim, Rana Ahmed, techydesigner, Zags
32	Many-to-many relationships	Antoine Pinsard, e4c5, knbk, Kostronor
33	Mapping strings to strings with HStoreField - a PostgreSQL specific field	noʮʎdʎzɐɹƆ
34	Meta: Documentation Guidelines	Antoine Pinsard
35	Middleware	AlvaroAV, Antoine Pinsard, George H., knbk, lxer, nhydock, Omar Shehata, Peter Mortensen, Trivial, William Reed
36	Migrations	Antoine Pinsard, engineercoding, Joey Wilhelm, knbk, MicroPyramid, ravigadila, Roald Nefs
37	Model Aggregations	Ian Clark, John Moutafis, ravigadila
38	Model Field Reference	Burhan Khalid, Husain Basrawala, knbk, Matt Seymour, Rod Xavier, scriptmonster, techydesigner, The_Cthulhu_Kid
39	Models	Aidas Bendoraitis, Alireza Aghamohammadi, alonisser, Antoine Pinsard, aquasan, Arpit Solanki, atomh33ls, coffee-grinder, DataSwede, ettanany, Gahan, George H., gkr, Ivan Semochkin, Jamie Cockburn, Joey Wilhelm, kcrk, knbk, Linville, lxer, maazza, Matt Seymour, MuYi, Navid777, nhydock, noʮʎdʎzɐɹƆ, pbaranay, PhoebeB, Rana Ahmed, Saksow, Sanyam Khurana, scriptmonster, Selcuk, SpiXel, sudshekhar, techydesigner, The_Cthulhu_Kid, Utsav T, waterproof, zurfyx
40	Project Structure	Antoine Pinsard, naveen.panwar, nicorellius
41	Querysets	Antoine Pinsard, Brian Artschwager, Chalist, coffee-grinder, DataSwede, e4c5, Evans Murithi, George H., John Moutafis, Justin, knbk, Louis Barranqueiro, Maxime Lorant, MicroPyramid, nima, ravigadila, Sanyam Khurana, The Brewmaster

42	RangeFields - a group of PostgreSQL specific fields	Antoine Pinsard, noɥʎdʎzɐɔ
43	Running Celery with Supervisor	RéÑjïth, sebb
44	Security	Antoine Pinsard, knbk
45	Settings	allo, Antoine Pinsard, Brian Artschwager, fredley, J F, knbk, Louis, Louis Barranqueiro, lxer, Maxime Lorant, NBajanca, Nils Werner, ProfSmiles, RamenChef, Sanyam Khurana, Sayse, Selcuk, SpiXel, ssice, sudshekhar, Tema, The Brewmaster
46	Signals	Antoine Pinsard, e4c5, Hetdev, John Moutafis, Majid, nhydock, Rexford
47	Template Tags and Filters	Antoine Pinsard, irakli khitarishvili, knbk, Medorator, naveen.panwar, The_Cthulhu_Kid
48	Templating	Adam Starrh, Alasdair, Aniket, Antoine Pinsard, Brian H., coffee-grinder, doctorsherlock, fredley, George H., gkr, lxer, Stephen Leppik, Zags
49	Timezones	William Reed
50	Unit Testing	Adrian17, Antoine Pinsard, e4c5, Kim, Matthew Schinckel, Maxime Lorant, Patrik Stenmark, SandroM, sudshekhar, Zags
51	URL routing	knbk
52	Using Redis with Django - Caching Backend	Majid, The Brewmaster
53	Views	ettanany, HorsePunchKid, John Moutafis