Django Tutorial

# The Django web framework

**model-template-view** (**MTV**) pattern, which is a variant of **model-view-controller** (**MVC**)

# Setting up Django

$ pip install Django

>>> import django

>>> django.VERSION

(1, 8, 4, 'final', 0)

### Creating an isolated Python environment

$ pip install virtualenv

$ **virtualenv my\_env**

**$ source my\_env/bin/activate**

**(my\_env)**laptop:~ zenx $

### Installing Django with pip

pip install Django==1.8.6

### Starting the project

$ django-admin startproject regex

# Part1

### Creating a project

**$** django-admin startproject mysite

Let’s look at what [**startproject**](https://docs.djangoproject.com/en/1.9/ref/django-admin/#django-admin-startproject) created:

mysite/

manage.py

mysite/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

### The development server

**$** python manage.py runserver

### Creating the Polls app

**$** python manage.py startapp polls

That’ll create a directory **polls**, which is laid out like this:

polls/

\_\_init\_\_.py

admin.py

apps.py

migrations/

\_\_init\_\_.py

models.py

tests.py

views.py

### Write your first view

polls/views.py

from django.http import HttpResponse

def index(request):

return HttpResponse("Hello, world. You're at the polls index.")

In the **polls/urls.py** file include the following code:

polls/urls.py

from django.conf.urls import url

from . import views

urlpatterns = [

url(r'^$', views.index, name='index'),

]

mysite/urls.py

from django.conf.urls import include, url

from django.contrib import admin

urlpatterns = [

url(r'^polls/', include('polls.urls')),

url(r'^admin/', admin.site.urls),

]

**$** python manage.py runserver

# Part2

### Setup Database : PostgresSQL

#### Install psycopg packagr on Mac

Download & install : http://initd.org/psycopg/

or Install via PIP

$ pip install psycopg2

Update: ./settings.py

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': 'mydatabase',

'USER': 'mydatabaseuser',

'PASSWORD': 'mypassword',

'HOST': '127.0.0.1',

'PORT': '5432',

}

}

Create tables

$ python manage.py migrate

### Creating models

polls/models.py

from django.db import models

class Question(models.Model):

question\_text = models.CharField(max\_length=200)

pub\_date = models.DateTimeField('date published')

class Choice(models.Model):

question = models.ForeignKey(Question, on\_delete=models.CASCADE)

choice\_text = models.CharField(max\_length=200)

votes = models.IntegerField(default=0)

### Activating models

First we need to tell our project that the **polls** app is installed.

mysite/settings.py

INSTALLED\_APPS = [

**'polls.apps.PollsConfig',**

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

**$** python manage.py makemigrations polls

**polls/migrations/0001\_initial.py**

**$** python manage.py sqlmigrate polls 0001

$ python manage.py migrate

Operations to perform:

Apply all migrations: admin, contenttypes, polls, auth, sessions

Running migrations:

Rendering model states... DONE

Applying polls.0001\_initial... OK

three-step guide to making model changes:

* Change your models (in **models.py**).
* Run [**python manage.py makemigrations**](https://docs.djangoproject.com/en/1.9/ref/django-admin/#django-admin-makemigrations) to create migrations for those changes
* Run [**python manage.py migrate**](https://docs.djangoproject.com/en/1.9/ref/django-admin/#django-admin-migrate) to apply those changes to the database.

### Playing with the API

**$** python manage.py shell

Bypassing manage.py

If you’d rather not use manage.py, no problem. Just set the [DJANGO\_SETTINGS\_MODULE](https://docs.djangoproject.com/en/1.9/topics/settings/#envvar-DJANGO_SETTINGS_MODULE) environment variable to mysite.settings, start a plain Python shell, and set up Django:

>>> import django

>>> django.setup()

If this raises an [AttributeError](https://docs.python.org/3/library/exceptions.html#AttributeError), you’re probably using a version of Django that doesn’t match this tutorial version. You’ll want to either switch to the older tutorial or the newer Django version.

You must run python from the same directory manage.py is in, or ensure that directory is on the Python path, so that import mysite works.

For more information on all of this, see the [django-admin documentation](https://docs.djangoproject.com/en/1.9/ref/django-admin/).

Once you’re in the shell, explore the [database API](https://docs.djangoproject.com/en/1.9/topics/db/queries/):

**>>> from** **polls.models** **import** Question, Choice # Import the model classes we just wrote.

# No questions are in the system yet.

**>>>** Question.objects.all()

[]

# Create a new Question.

# Support for time zones is enabled in the default settings file, so

# Django expects a datetime with tzinfo for pub\_date. Use timezone.now()

# instead of datetime.datetime.now() and it will do the right thing.

**>>> from** **django.utils** **import** timezone

**>>>** q = Question(question\_text="What's new?", pub\_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.

**>>>** q.save()

# Now it has an ID. Note that this might say "1L" instead of "1", depending

# on which database you're using. That's no biggie; it just means your

# database backend prefers to return integers as Python long integer

# objects.

**>>>** q.id

1

# Access model field values via Python attributes.

**>>>** q.question\_text

"What's new?"

**>>>** q.pub\_date

datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Change values by changing the attributes, then calling save().

**>>>** q.question\_text = "What's up?"

**>>>** q.save()

# objects.all() displays all the questions in the database.

**>>>** Question.objects.all()

[<Question: Question object>]

polls/models.py

from django.db import models

from django.utils.encoding import python\_2\_unicode\_compatible

@python\_2\_unicode\_compatible # only if you need to support Python 2

class Question(models.Model):

# ...

def \_\_str\_\_(self):

return self.question\_text

@python\_2\_unicode\_compatible # only if you need to support Python 2

class Choice(models.Model):

# ...

def \_\_str\_\_(self):

return self.choice\_text

polls/models.py

import datetime

from django.db import models

from django.utils import timezone

class Question(models.Model):

# ...

def was\_published\_recently(self):

return self.pub\_date >= timezone.now() - datetime.timedelta(days=1)

**python manage.py shell** again:

>>> from polls.models import Question, Choice

# Make sure our \_\_str\_\_() addition worked.

>>> Question.objects.all()

[<Question: What's up?>]

# Django provides a rich database lookup API that's entirely driven by

# keyword arguments.

>>> Question.objects.filter(id=1)

[<Question: What's up?>]

>>> Question.objects.filter(question\_text\_\_startswith='What')

[<Question: What's up?>]

# Get the question that was published this year.

>>> from django.utils import timezone

>>> current\_year = timezone.now().year

>>> Question.objects.get(pub\_date\_\_year=current\_year)

<Question: What's up?>

# Request an ID that doesn't exist, this will raise an exception.

>>> Question.objects.get(id=2)

Traceback (most recent call last):

...

DoesNotExist: Question matching query does not exist.

# Lookup by a primary key is the most common case, so Django provides a

# shortcut for primary-key exact lookups.

# The following is identical to Question.objects.get(id=1).

>>> Question.objects.get(pk=1)

<Question: What's up?>

# Make sure our custom method worked.

>>> q = Question.objects.get(pk=1)

>>> q.was\_published\_recently()

True

# Give the Question a couple of Choices. The create call constructs a new

# Choice object, does the INSERT statement, adds the choice to the set

# of available choices and returns the new Choice object. Django creates

# a set to hold the "other side" of a ForeignKey relation

# (e.g. a question's choice) which can be accessed via the API.

>>> q = Question.objects.get(pk=1)

# Display any choices from the related object set -- none so far.

>>> q.choice\_set.all()

[]

# Create three choices.

>>> q.choice\_set.create(choice\_text='Not much', votes=0)

<Choice: Not much>

>>> q.choice\_set.create(choice\_text='The sky', votes=0)

<Choice: The sky>

>>> c = q.choice\_set.create(choice\_text='Just hacking again', votes=0)

# Choice objects have API access to their related Question objects.

>>> c.question

<Question: What's up?>

# And vice versa: Question objects get access to Choice objects.

>>> q.choice\_set.all()

[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]

>>> q.choice\_set.count()

3

# The API automatically follows relationships as far as you need.

# Use double underscores to separate relationships.

# This works as many levels deep as you want; there's no limit.

# Find all Choices for any question whose pub\_date is in this year

# (reusing the 'current\_year' variable we created above).

>>> Choice.objects.filter(question\_\_pub\_date\_\_year=current\_year)

[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]

# Let's delete one of the choices. Use delete() for that.

>>> c = q.choice\_set.filter(choice\_text\_\_startswith='Just hacking')

>>> c.delete()

## Introducing the Django Admin

### Creating an admin user

First we’ll need to create a user who can login to the admin site. Run the following command:

**$** python manage.py createsuperuser

Enter your desired username and press enter.

Username: admin

You will then be prompted for your desired email address:

Email address: admin@example.com

The final step is to enter your password. You will be asked to enter your password twice, the second time as a confirmation of the first.

Password: \*\*\*\*\*\*\*\*\*\*

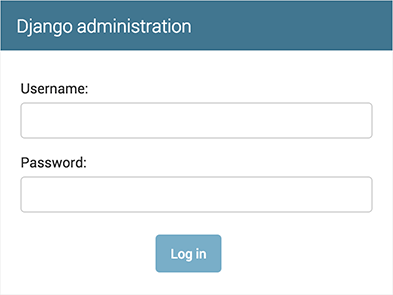
Password (again): \*\*\*\*\*\*\*\*\*

Superuser created successfully.

### Start the development server

**$** python manage.py runserver

<http://127.0.0.1:8000/admin/>.



### Make the poll app modifiable in the admin

polls/admin.py

from django.contrib import admin

from .models import Question

admin.site.register(Question)

# part 3

### Overview

**in a blog application, you might have the following views:**

* Blog homepage – displays the latest few entries.
* Entry “detail” page – permalink page for a single entry.
* Year-based archive page – displays all months with entries in the given year.
* Month-based archive page – displays all days with entries in the given month.
* Day-based archive page – displays all entries in the given day.
* Comment action – handles posting comments to a given entry.

**In our poll application, we’ll have the following four views:**

* Question “index” page – displays the latest few questions.
* Question “detail” page – displays a question text, with no results but with a form to vote.
* Question “results” page – displays results for a particular question.
* Vote action – handles voting for a particular choice in a particular question.

### Writing more views

polls/views.py

**def** detail(request, question\_id):

**return** HttpResponse("You're looking at question **%s**." % question\_id)

**def** results(request, question\_id):

response = "You're looking at the results of question **%s**."

**return** HttpResponse(response % question\_id)

**def** vote(request, question\_id):

**return** HttpResponse("You're voting on question **%s**." % question\_id)

Wire these new views into the **polls.urls** module by adding the following [**url()**](https://docs.djangoproject.com/en/1.9/ref/urls/#django.conf.urls.url) calls:

polls/urls.py

from django.conf.urls import url

from . import views

urlpatterns = [

# ex: /polls/

url(r'^$', views.index, name='index'),

# ex: /polls/5/

url(r'^(?P<question\_id>[0-9]+)/$', views.detail, name='detail'),

# ex: /polls/5/results/

url(r'^(?P<question\_id>[0-9]+)/results/$', views.results, name='results'),

# ex: /polls/5/vote/

url(r'^(?P<question\_id>[0-9]+)/vote/$', views.vote, name='vote'),

]

### Write views that actually do something

All Django wants is that [**HttpResponse**](https://docs.djangoproject.com/en/1.9/ref/request-response/#django.http.HttpResponse). Or an exception.

polls/views.py

from django.http import HttpResponse

from .models import Question

def index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

output = ', '.join([q.question\_text for q in latest\_question\_list])

return HttpResponse(output)

# Leave the rest of the views (detail, results, vote) unchanged

First, create a directory called **templates** in your **polls** directory. Django will look for templates in there.

Your project’s [**TEMPLATES**](https://docs.djangoproject.com/en/1.9/ref/settings/#std:setting-TEMPLATES) setting describes how Django will load and render templates. The default settings file configures a **DjangoTemplates** backend whose [**APP\_DIRS**](https://docs.djangoproject.com/en/1.9/ref/settings/#std:setting-TEMPLATES-APP_DIRS) option is set to **True**. By convention **DjangoTemplates** looks for a “templates” subdirectory in each of the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/1.9/ref/settings/#std:setting-INSTALLED_APPS).

Within the **templates** directory you have just created, create another directory called **polls**, and within that create a file called **index.html**. In other words, your template should be at **polls/templates/polls/index.html**. Because of how the **app\_directories** template loader works as described above, you can refer to this template within Django simply as **polls/index.html**.

**Template namespacing**

Now we *might* be able to get away with putting our templates directly **in polls/templates** (rather than creating another **polls** subdirectory), but it would actually be a bad idea. Django will choose the first template it finds whose name matches, and if you had a template with the same name in a *different* application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the easiest way to ensure this is by *namespacing* them. That is, by putting those templates inside *another* directory named for the application itself.

polls/templates/polls/index.html

{% **if** latest\_question\_list %}

<**ul**>

{% **for** question **in** latest\_question\_list %}

<**li**><**a** href="/polls/{{ question.id }}/">{{ question.question\_text }}</**a**></**li**>

{% **endfor** %}

</**ul**>

{% **else** %}

<**p**>No polls are available.</**p**>

{% **endif** %}

Now let’s update our **index** view in **polls/views.py** to use the template:

polls/views.py

from django.http import HttpResponse

from django.template import loader

from .models import Question

def index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

template = loader.get\_template('polls/index.html')

context = {

'latest\_question\_list': latest\_question\_list,

}

return HttpResponse(template.render(context, request))

#### A shortcut: [render()](https://docs.djangoproject.com/en/1.9/topics/http/shortcuts/#django.shortcuts.render)

polls/views.py

from django.shortcuts import render

from .models import Question

def index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

context = {'latest\_question\_list': latest\_question\_list}

return render(request, 'polls/index.html', context)

#### Raising a 404 error

polls/views.py

from django.http import Http404

from django.shortcuts import render

from .models import Question

# ...

def detail(request, question\_id):

try:

question = Question.objects.get(pk=question\_id)

except Question.DoesNotExist:

raise Http404("Question does not exist")

return render(request, 'polls/detail.html', {'question': question})

polls/templates/polls/detail.html

{{ question }}

#### A shortcut: [get\_object\_or\_404()](https://docs.djangoproject.com/en/1.9/topics/http/shortcuts/#django.shortcuts.get_object_or_404)

polls/views.py

**from** **django.shortcuts** **import** get\_object\_or\_404, render

**from** **.models** **import** Question

# ...

**def** detail(request, question\_id):

question = get\_object\_or\_404(Question, pk=question\_id)

**return** render(request, 'polls/detail.html', {'question': question})

There’s also a [**get\_list\_or\_404()**](https://docs.djangoproject.com/en/1.9/topics/http/shortcuts/#django.shortcuts.get_list_or_404) function, which works just as [**get\_object\_or\_404()**](https://docs.djangoproject.com/en/1.9/topics/http/shortcuts/#django.shortcuts.get_object_or_404) – except using [**filter()**](https://docs.djangoproject.com/en/1.9/ref/models/querysets/#django.db.models.query.QuerySet.filter) instead of [**get()**](https://docs.djangoproject.com/en/1.9/ref/models/querysets/#django.db.models.query.QuerySet.get). It raises [**Http404**](https://docs.djangoproject.com/en/1.9/topics/http/views/#django.http.Http404) if the list is empty.

### Use the template system

polls/templates/polls/detail.html

<**h1**>{{ question.question\_text }}</**h1**>

<**ul**>

{% **for** choice **in** question.choice\_set.all %}

<**li**>{{ choice.choice\_text }}</**li**>

{% **endfor** %}

</**ul**>

See the [template guide](https://docs.djangoproject.com/en/1.9/topics/templates/) for more about templates.

### Removing hardcoded URLs in templates

<**li**><**a** href="/polls/{{ question.id }}/">{{ question.question\_text }}</**a**></**li**>

The problem with this hardcoded, tightly-coupled approach is that it becomes challenging to change URLs on projects with a lot of templates. However, since you defined the name argument in the [**url()**](https://docs.djangoproject.com/en/1.9/ref/urls/#django.conf.urls.url) functions in the **polls.urls** module, you can remove a reliance on specific URL paths defined in your url configurations by using the **{% url %}** template tag:

<**li**><**a** href="{% **url** 'detail' question.id %}">{{ question.question\_text }}</**a**></**li**>

change it in **polls/urls.py**:

...

# added the word 'specifics'

url(r'^specifics/(?P<question\_id>[0-9]+)/$', views.detail, name='detail'),

...

### Namespacing URL names

The answer is to add namespaces to your URLconf. In the **polls/urls.py** file, go ahead and add an **app\_name** to set the application namespace:

polls/urls.py

from django.conf.urls import url

from . import views

**app\_name = 'polls'**

urlpatterns = [

url(r'^$', views.index, name='index'),

url(r'^(?P<question\_id>[0-9]+)/$', views.detail, name='detail'),

url(r'^(?P<question\_id>[0-9]+)/results/$', views.results, name='results'),

url(r'^(?P<question\_id>[0-9]+)/vote/$', views.vote, name='vote'),

]

Now change your **polls/index.html** template from:

polls/templates/polls/index.html

<**li**><**a** href="{% **url** 'detail' question.id %}">{{ question.question\_text }}</**a**></**li**>

to point at the namespaced detail view:

polls/templates/polls/index.html

<**li**><**a** href="{% **url** **'polls:**detail' question.id %}">{{ question.question\_text }}</**a**></**li**>

# part 4

### Write a simple form

polls/templates/polls/detail.html

<**h1**>{{ question.question\_text }}</**h1**>

{% **if** error\_message %}<**p**><**strong**>{{ error\_message }}</**strong**></**p**>{% **endif** %}

<**form** action="{% **url** 'polls:vote' question.id %}" method="post">

{% **csrf\_token** %}

{% **for** choice **in** question.choice\_set.all %}

<**input** type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />

<**label** for="choice{{ forloop.counter }}">{{ choice.choice\_text }}</**label**><**br** />

{% **endfor** %}

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

</**form**>

polls/urls.py

url(r'^(?P<question\_id>[0-9]+)/vote/$', views.vote, name='vote'),

polls/views.py

**from** **django.shortcuts** **import** get\_object\_or\_404, render

**from** **django.http** **import** HttpResponseRedirect, HttpResponse

**from** **django.core.urlresolvers** **import** reverse

**from** **.models** **import** Choice, Question

# ...

**def** vote(request, question\_id):

question = get\_object\_or\_404(Question, pk=question\_id)

**try**:

selected\_choice = question.choice\_set.get(pk=request.POST['choice'])

**except** (**KeyError**, Choice.DoesNotExist):

# Redisplay the question voting form.

**return** render(request, 'polls/detail.html', {

'question': question,

'error\_message': "You didn't select a choice.",

})

**else**:

selected\_choice.votes += 1

selected\_choice.save()

# Always return an HttpResponseRedirect after successfully dealing

# with POST data. This prevents data from being posted twice if a

# user hits the Back button.

**return** HttpResponseRedirect(reverse('polls:results', args=(question.id,)))

After somebody votes in a question, the **vote()** view redirects to the results page for the question. Let’s write that view:

polls/views.py

**from** **django.shortcuts** **import** get\_object\_or\_404, render

**def** results(request, question\_id):

question = get\_object\_or\_404(Question, pk=question\_id)

**return** render(request, 'polls/results.html', {'question': question})

Now, create a **polls/results.html** template:

polls/templates/polls/results.html

<**h1**>{{ question.question\_text }}</**h1**>

<**ul**>

{% **for** choice **in** question.choice\_set.all %}

<**li**>{{ choice.choice\_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</**li**>

{% **endfor** %}

</**ul**>

<**a** href="{% **url** 'polls:detail' question.id %}">Vote again?</**a**>

**Note**

The code for our vote() view does have a small problem. It first gets the selected\_choice object from the database, then computes the new value of votes, and then saves it back to the database. If two users of your website try to vote at *exactly the same time*, this might go wrong: The same value, let’s say 42, will be retrieved for votes. Then, for both users the new value of 43 is computed and saved, but 44 would be the expected value.

This is called a *race condition*. If you are interested, you can read [Avoiding race conditions using F()](https://docs.djangoproject.com/en/1.9/ref/models/expressions/#avoiding-race-conditions-using-f) to learn how you can solve this issue.

#### Avoid Race Conditions

# Tintin filed a news story!

reporter = Reporters.objects.get(name='Tintin')

reporter.stories\_filed += 1

reporter.save()

But instead we could also have done:

from django.db.models import F

reporter = Reporters.objects.get(name='Tintin')

reporter.stories\_filed = F('stories\_filed') + 1

reporter.save()

reporter = Reporters.objects.get(pk=reporter.pk)

# Or, more succinctly:

reporter.refresh\_from\_db()

### Use generic views: Less code is better

#### Amend URLconf

First, open the **polls/urls.py** URLconf and change it like so:

polls/urls.py

from django.conf.urls import url

from . import views

app\_name = 'polls'

urlpatterns = [

url(r'^$', views.IndexView.as\_view(), name='index'),

url(r'^(?P<pk>[0-9]+)/$', views.DetailView.as\_view(), name='detail'),

url(r'^(?P<pk>[0-9]+)/results/$', views.ResultsView.as\_view(), name='results'),

url(r'^(?P<question\_id>[0-9]+)/vote/$', views.vote, name='vote'),

]

patterns has changed from **<question\_id>** to **<pk>**.

#### Amend views

polls/views.py

**from** **django.shortcuts** **import** get\_object\_or\_404, render

**from** **django.http** **import** HttpResponseRedirect

**from** **django.core.urlresolvers** **import** reverse

**from** **django.views** **import** generic

**from** **.models** **import** Choice, Question

**class** **IndexView**(generic.ListView):

template\_name = 'polls/index.html'

context\_object\_name = 'latest\_question\_list'

**def** get\_queryset(self):

"""Return the last five published questions."""

**return** Question.objects.order\_by('-pub\_date')[:5]

**class** **DetailView**(generic.DetailView):

model = Question

template\_name = 'polls/detail.html'

**class** **ResultsView**(generic.DetailView):

model = Question

template\_name = 'polls/results.html'

**def** vote(request, question\_id):

... # same as above, no changes needed.

For full details on generic views, see the [generic views documentation](https://docs.djangoproject.com/en/1.9/topics/class-based-views/).

# Part5

## Introducing automated testing

### Writing our first test

#### We identify a bug

check the method using the [**shell**](https://docs.djangoproject.com/en/1.9/ref/django-admin/#django-admin-shell):

>>> import datetime

>>> from django.utils import timezone

>>> from polls.models import Question

>>> # create a Question instance with pub\_date 30 days in the future

>>> future\_question = Question(pub\_date=timezone.now() + datetime.timedelta(days=30))

>>> # was it published recently?

>>> future\_question.was\_published\_recently()

True

#### Create a test to expose the bug

polls/tests.py

import datetime

from django.utils import timezone

from django.test import TestCase

from .models import Question

class QuestionMethodTests(TestCase):

def test\_was\_published\_recently\_with\_future\_question(self):

"""

was\_published\_recently() should return False for questions whose

pub\_date is in the future.

"""

time = timezone.now() + datetime.timedelta(days=30)

future\_question = Question(pub\_date=time)

self.assertEqual(future\_question.was\_published\_recently(), False)

#### Running tests

$ python manage.py test polls

and you’ll see something like:

Creating test database for alias 'default'...

F

======================================================================

FAIL: test\_was\_published\_recently\_with\_future\_question (polls.tests.QuestionMethodTests)

----------------------------------------------------------------------

Traceback (most recent call last):

File "/path/to/mysite/polls/tests.py", line 16, in test\_was\_published\_recently\_with\_future\_question

self.assertEqual(future\_question.was\_published\_recently(), False)

AssertionError: True != False

----------------------------------------------------------------------

Ran 1 test in 0.001s

FAILED (failures=1)

Destroying test database for alias 'default'...

#### Fixing the bug

polls/models.py

**def** was\_published\_recently(self):

now = timezone.now()

**return** now - datetime.timedelta(days=1) <= self.pub\_date <= now

and run the test again:

Creating test database for alias 'default'...

.

----------------------------------------------------------------------

Ran 1 test in 0.001s

OK

Destroying test database for alias 'default'...

### More comprehensive tests

polls/tests.py

**def** test\_was\_published\_recently\_with\_old\_question(self):

"""

was\_published\_recently() should return False for questions whose

pub\_date is older than 1 day.

"""

time = timezone.now() - datetime.timedelta(days=30)

old\_question = Question(pub\_date=time)

self.assertEqual(old\_question.was\_published\_recently(), False)

**def** test\_was\_published\_recently\_with\_recent\_question(self):

"""

was\_published\_recently() should return True for questions whose

pub\_date is within the last day.

"""

time = timezone.now() - datetime.timedelta(hours=1)

recent\_question = Question(pub\_date=time)

self.assertEqual(recent\_question.was\_published\_recently(), True)

### The Django test client

The first is to set up the test environment in the [**shell**](https://docs.djangoproject.com/en/1.9/ref/django-admin/#django-admin-shell):

>>> from django.test.utils import setup\_test\_environment

>>> setup\_test\_environment()

>>> from django.test import Client

>>> # create an instance of the client for our use

>>> client = Client()

With that ready, we can ask the client to do some work for us:

**>>>** # get a response from '/'

**>>>** response = client.get('/')

**>>>** # we should expect a 404 from that address

**>>>** response.status\_code

404

**>>>** # on the other hand we should expect to find something at '/polls/'

**>>>** # we'll use 'reverse()' rather than a hardcoded URL

**>>> from** **django.core.urlresolvers** **import** reverse

**>>>** response = client.get(reverse('polls:index'))

**>>>** response.status\_code

200

**>>>** response.content

b'\n\n\n <p>No polls are available.</p>\n\n'

**>>>** # note - you might get unexpected results if your ``TIME\_ZONE``

**>>>** # in ``settings.py`` is not correct. If you need to change it,

**>>>** # you will also need to restart your shell session

**>>> from** **polls.models** **import** Question

**>>> from** **django.utils** **import** timezone

**>>>** # create a Question and save it

**>>>** q = Question(question\_text="Who is your favorite Beatle?", pub\_date=timezone.now())

**>>>** q.save()

**>>>** # check the response once again

**>>>** response = client.get('/polls/')

**>>>** response.content

b'\n\n\n <ul>\n \n <li><a href="/polls/1/">Who is your favorite Beatle?</a></li>\n \n </ul>\n\n'

**>>>** # If the following doesn't work, you probably omitted the call to

**>>>** # setup\_test\_environment() described above

**>>>** response.context['latest\_question\_list']

[<Question: Who is your favorite Beatle?>]

### Improving our view

We need to amend the **get\_queryset()** method and change it so that it also checks the date by comparing it with **timezone.now()**. First we need to add an import:

polls/views.py

from django.utils import timezone

and then we must amend the **get\_queryset** method like so:

polls/views.py

**def** get\_queryset(self):

"""

Return the last five published questions (not including those set to be

published in the future).

"""

**return** Question.objects.filter(

pub\_date\_\_lte=timezone.now()

).order\_by('-pub\_date')[:5]

### Testing our new view

polls/tests.py

from django.core.urlresolvers import reverse

and we’ll create a shortcut function to create questions as well as a new test class:

polls/tests.py

**def** create\_question(question\_text, days):

"""

Creates a question with the given `question\_text` and published the

given number of `days` offset to now (negative for questions published

in the past, positive for questions that have yet to be published).

"""

time = timezone.now() + datetime.timedelta(days=days)

**return** Question.objects.create(question\_text=question\_text,

pub\_date=time)

**class** **QuestionViewTests**(TestCase):

**def** test\_index\_view\_with\_no\_questions(self):

"""

If no questions exist, an appropriate message should be displayed.

"""

response = self.client.get(reverse('polls:index'))

self.assertEqual(response.status\_code, 200)

self.assertContains(response, "No polls are available.")

self.assertQuerysetEqual(response.context['latest\_question\_list'], [])

**def** test\_index\_view\_with\_a\_past\_question(self):

"""

Questions with a pub\_date in the past should be displayed on the

index page.

"""

create\_question(question\_text="Past question.", days=-30)

response = self.client.get(reverse('polls:index'))

self.assertQuerysetEqual(

response.context['latest\_question\_list'],

['<Question: Past question.>']

)

**def** test\_index\_view\_with\_a\_future\_question(self):

"""

Questions with a pub\_date in the future should not be displayed on

the index page.

"""

create\_question(question\_text="Future question.", days=30)

response = self.client.get(reverse('polls:index'))

self.assertContains(response, "No polls are available.",

status\_code=200)

self.assertQuerysetEqual(response.context['latest\_question\_list'], [])

**def** test\_index\_view\_with\_future\_question\_and\_past\_question(self):

"""

Even if both past and future questions exist, only past questions

should be displayed.

"""

create\_question(question\_text="Past question.", days=-30)

create\_question(question\_text="Future question.", days=30)

response = self.client.get(reverse('polls:index'))

self.assertQuerysetEqual(

response.context['latest\_question\_list'],

['<Question: Past question.>']

)

**def** test\_index\_view\_with\_two\_past\_questions(self):

"""

The questions index page may display multiple questions.

"""

create\_question(question\_text="Past question 1.", days=-30)

create\_question(question\_text="Past question 2.", days=-5)

response = self.client.get(reverse('polls:index'))

self.assertQuerysetEqual(

response.context['latest\_question\_list'],

['<Question: Past question 2.>', '<Question: Past question 1.>']

)

### Testing the DetailView

polls/views.py

**class** **DetailView**(generic.DetailView):

...

**def** get\_queryset(self):

"""

Excludes any questions that aren't published yet.

"""

**return** Question.objects.filter(pub\_date\_\_lte=timezone.now())

polls/tests.py

class QuestionIndexDetailTests(TestCase):

def test\_detail\_view\_with\_a\_future\_question(self):

"""

The detail view of a question with a pub\_date in the future should

return a 404 not found.

"""

future\_question = create\_question(question\_text='Future question.',

days=5)

response = self.client.get(reverse('polls:detail',

args=(future\_question.id,)))

self.assertEqual(response.status\_code, 404)

def test\_detail\_view\_with\_a\_past\_question(self):

"""

The detail view of a question with a pub\_date in the past should

display the question's text.

"""

past\_question = create\_question(question\_text='Past Question.',

days=-5)

response = self.client.get(reverse('polls:detail',

args=(past\_question.id,)))

self.assertContains(response, past\_question.question\_text,

status\_code=200)

### When testing, more is better

* a separate **TestClass** for each model or view
* a separate test method for each set of conditions you want to test
* test method names that describe their function

### Further testing

* you can use an “in-browser” framework such as [Selenium](http://seleniumhq.org/) to test the way your HTML actually renders in a browser.
* Django includes [LiveServerTestCase](https://docs.djangoproject.com/en/1.9/topics/testing/tools/#django.test.LiveServerTestCase) to facilitate integration with tools like Selenium.
* [Testing in Django](https://docs.djangoproject.com/en/1.9/topics/testing/) has comprehensive information about testing.

# Part6

add a stylesheet and an image.

### Customize your *app’s* look and feel

First, create a directory called **static** in your **polls** directory.

One of the defaults is **AppDirectoriesFinder** which looks for a “static” subdirectory in each of the [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/1.9/ref/settings/#std:setting-INSTALLED_APPS),

**polls/static/polls/style.css**. Because of how the **AppDirectoriesFinder** staticfile finder works, you can refer to this static file in Django simply as **polls/style.css**

polls/static/polls/style.css

**li** **a** {

color: green;

}

Next, add the following at the top of **polls/templates/polls/index.html**:

polls/templates/polls/index.html

{% **load** staticfiles %}

<**link** rel="stylesheet" type="text/css" href="{% **static** 'polls/style.css' %}" />

### Adding a background-image

polls/static/polls/style.css

body {

background: white url("images/background.gif") no-repeat right bottom;

}

These are the **basics**. For more details on settings and other bits included with the framework see [the static files howto](https://docs.djangoproject.com/en/1.9/howto/static-files/) and [the staticfiles reference](https://docs.djangoproject.com/en/1.9/ref/contrib/staticfiles/). [Deploying static files](https://docs.djangoproject.com/en/1.9/howto/static-files/deployment/) discusses how to use static files on a real server.

# Part7

customizing the Django’s automatically-generated admin

### Customize the admin form

. Replace the **admin.site.register(Question)** line with:

polls/admin.py

from django.contrib import admin

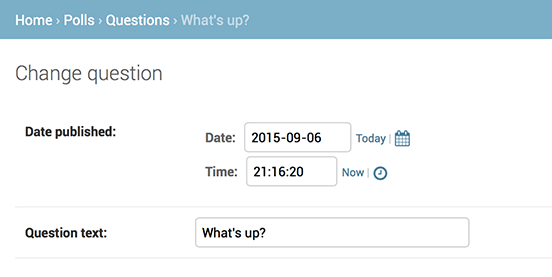
from .models import Question

class QuestionAdmin(admin.ModelAdmin):

fields = ['pub\_date', 'question\_text']

admin.site.register(Question, QuestionAdmin)

the “Publication date” come before the “Question” field:



you might want to split the form up into fieldsets:

polls/admin.py

from django.contrib import admin

from .models import Question

class QuestionAdmin(admin.ModelAdmin):

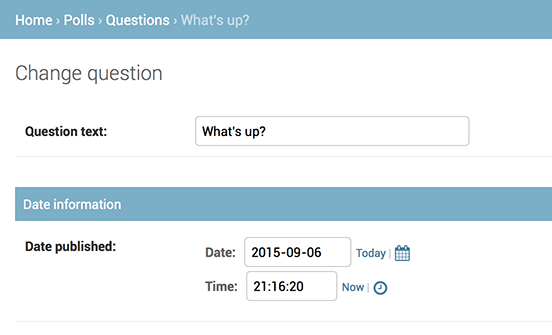
fieldsets = [

(None, {'fields': ['question\_text']}),

('Date information', {'fields': ['pub\_date']}),

]

admin.site.register(Question, QuestionAdmin)



### Adding related objects

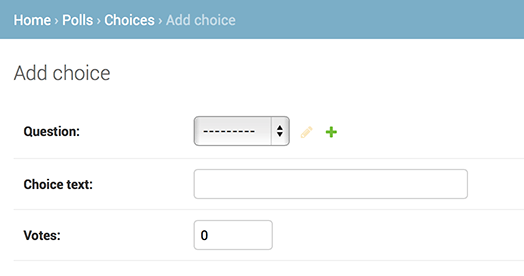
polls/admin.py

from django.contrib import admin

from .models import Choice, Question

# ...

admin.site.register(Choice)



It’d be better if you could add a bunch of Choices directly when you create the **Question** object.

Remove the **register()** call for the **Choice** model. Then, edit the **Question** registration code to read:

polls/admin.py

from django.contrib import admin

from .models import Choice, Question

class ChoiceInline(admin.StackedInline):

model = Choice

extra = 3

class QuestionAdmin(admin.ModelAdmin):

fieldsets = [

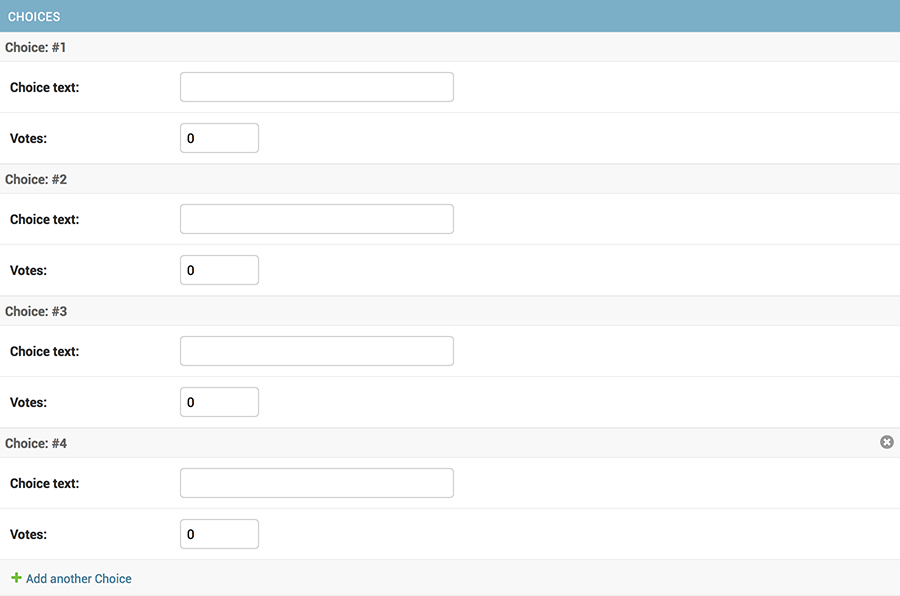
(None, {'fields': ['question\_text']}),

('Date information', {'fields': ['pub\_date'], 'classes': ['collapse']}),

]

inlines = [ChoiceInline]

admin.site.register(Question, QuestionAdmin)



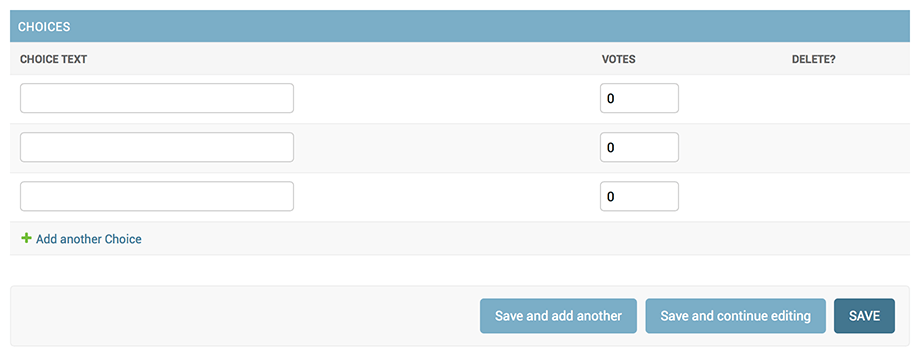
One small problem, though. It takes a lot of screen space to display all the fields for entering related **Choice** objects.

polls/admin.py

class ChoiceInline(admin.TabularInline):

#...

With that **TabularInline** (instead of **StackedInline**), the related objects are displayed in a more compact, table-based format:



### Customize the admin change list

polls/admin.py

**class** **QuestionAdmin**(admin.ModelAdmin):

# ...

list\_display = ('question\_text', 'pub\_date')

Just for good measure, let’s also include the **was\_published\_recently()** method from [Tutorial 2](https://docs.djangoproject.com/en/1.9/intro/tutorial02/):

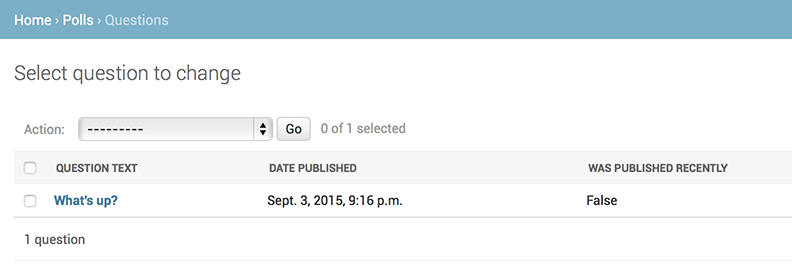
polls/admin.py

**class** **QuestionAdmin**(admin.ModelAdmin):

# ...

list\_display = ('question\_text', 'pub\_date', 'was\_published\_recently')

Now the question change list page looks like this:



Sort Model method attributes

polls/models.py

**class** **Question**(models.Model):

# ...

**def** was\_published\_recently(self):

now = timezone.now()

**return** now - datetime.timedelta(days=1) <= self.pub\_date <= now

was\_published\_recently.admin\_order\_field = 'pub\_date'

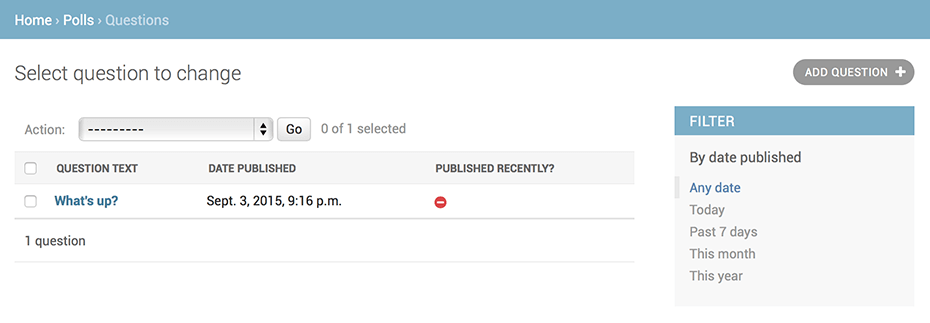
was\_published\_recently.boolean = True

was\_published\_recently.short\_description = 'Published recently?'

For more information on these method properties, see [**list\_display**](https://docs.djangoproject.com/en/1.9/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_display).

Edit your **polls/admin.py** file again and add an improvement to the **Question** change list page: filters using the [**list\_filter**](https://docs.djangoproject.com/en/1.9/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_filter). Add the following line to **QuestionAdmin**:

list\_filter = ['pub\_date']



This is shaping up well. Let’s add some search capability:

search\_fields = ['question\_text']

Now’s also a good time to note that change lists give you free pagination. The default is to display 100 items per page. [**Change list pagination**](https://docs.djangoproject.com/en/1.9/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_per_page), [**search boxes**](https://docs.djangoproject.com/en/1.9/ref/contrib/admin/#django.contrib.admin.ModelAdmin.search_fields), [**filters**](https://docs.djangoproject.com/en/1.9/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_filter), [**date-hierarchies**](https://docs.djangoproject.com/en/1.9/ref/contrib/admin/#django.contrib.admin.ModelAdmin.date_hierarchy), and [**column-header-ordering**](https://docs.djangoproject.com/en/1.9/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_display) all work together like you think they should.

### Customize the admin look and feel

#### Customizing your *project’s* templates

Create a **templates** directory in your project directory (the one that contains **manage.py**).

Open your settings file (**mysite/settings.py**, remember) and add a [**DIRS**](https://docs.djangoproject.com/en/1.9/ref/settings/#std:setting-TEMPLATES-DIRS) option in the [**TEMPLATES**](https://docs.djangoproject.com/en/1.9/ref/settings/#std:setting-TEMPLATES) setting:

mysite/settings.py

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',

],

},

},

]

Now create a directory called **admin** inside **templates**, and copy the template **admin/base\_site.html** from within the default Django admin template directory in the source code of Django itself (**django/contrib/admin/templates**) into that directory.

**Where are the Django source files?**

If you have difficulty finding where the Django source files are located on your system, run the following command:

**$** python -c "import django; print(django.\_\_path\_\_)"

Then, just edit the file and replace **{{ site\_header|default:\_('Django administration') }}** (including the curly braces) with your own site’s name as you see fit. You should end up with a section of code like:

{% **block** branding %}

<**h1** id="site-name"><**a** href="{% **url** 'admin:index' %}">Polls Administration</**a**></**h1**>

{% **endblock** %}

We use this approach to teach you how to override templates. In an actual project, you would probably use the [**django.contrib.admin.AdminSite.site\_header**](https://docs.djangoproject.com/en/1.9/ref/contrib/admin/#django.contrib.admin.AdminSite.site_header) attribute to more easily make this particular customization.

This template file contains lots of text like **{% block branding %}** and **{{ title }}**. The **{%** and **{{** tags are part of Django’s template language. When Django renders **admin/base\_site.html**, this template language will be evaluated to produce the final HTML page, just like we saw in [Tutorial 3](https://docs.djangoproject.com/en/1.9/intro/tutorial03/).

Note that any of Django’s default admin templates can be overridden. To override a template, just do the same thing you did with **base\_site.html** – copy it from the default directory into your custom directory, and make changes.

See the [template loading documentation](https://docs.djangoproject.com/en/1.9/topics/templates/#template-loading) for more information about how Django finds its templates.

### Customize the admin index page

The template to customize is **admin/index.html**. (Do the same as with **admin/base\_site.html** in the previous section – copy it from the default directory to your custom template directory). Edit the file, and you’ll see it uses a template variable called **app\_list**.

### What’s next?

If you are familiar with Python packaging and interested in learning how to turn polls into a “reusable app”, check out [Advanced tutorial: How to write reusable apps](https://docs.djangoproject.com/en/1.9/intro/reusable-apps/).