To know your django version

```
$ python -m django --version
```

From the command line, `cd` into a directory where you'd like to store your code, then run the following command:

```
$ django-admin startproject mysite
```

Let's look at what *startproject* created:

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

Let's verify your Django project works. Change into the outer `mysite` directory, if you haven't already, and run the
following commands:

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

To create your app, make sure you're in the same directory as `manage.py` and type this command:

```
$ 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
```

Let's write the first view. Open the file `polls/views.py` and put the following Python code in it:

polls/views.py

```
from django.http import HttpResponse


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

To create a URLconf in the polls directory, create a file called `urls.py`. Your app directory should now look like:

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    urls.py
    views.py
```

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

polls/urls.py

```
from django.urls import path

from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

mysite/urls.py

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

urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
```

You have now wired an `index` view into the URLconf. Lets verify it's working, run the following comman

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

By default, *INSTALLED_APPS* contains the following apps, all of which come with Django:

- *django.contrib.admin* – The admin site. You'll use it shortly.
- *django.contrib.auth* – An authentication system.
- *django.contrib.contenttypes* – A framework for content types.
- *django.contrib.sessions* – A session framework.
- *django.contrib.messages* – A messaging framework.
- *django.contrib.staticfiles* – A framework for managing static files.

Some of these applications make use of at least one database table, though, so we need to create the tables in the database before we can use them. To do that, run the following command:

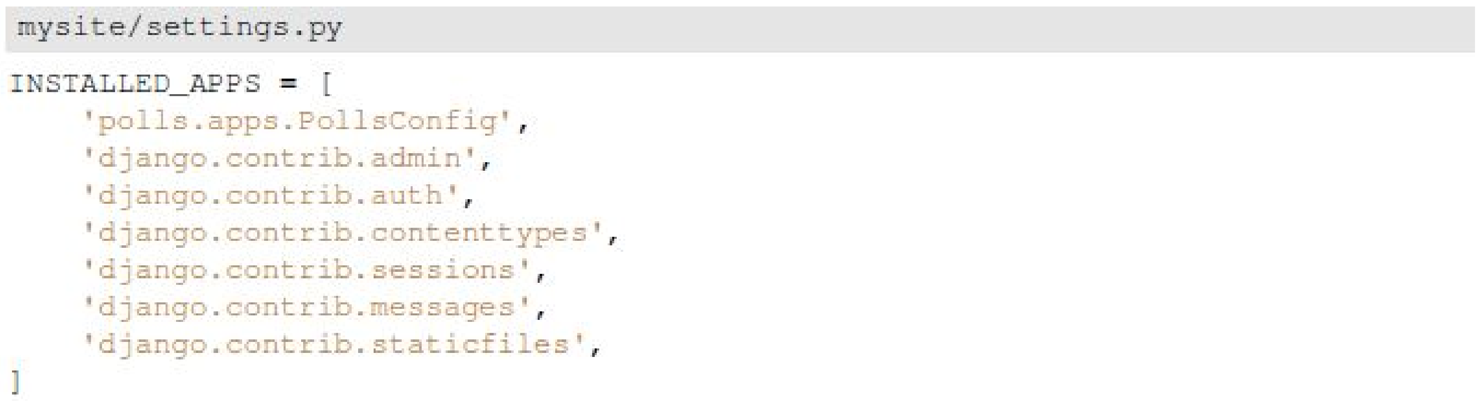
```
$ python manage.py migrate
```

The *migrate* command looks at the *INSTALLED_APPS* setting and creates any necessary database tables according to the database settings in your `mysite/settings.py` file and the database migrations shipped with the app (we'll

polls/models.py

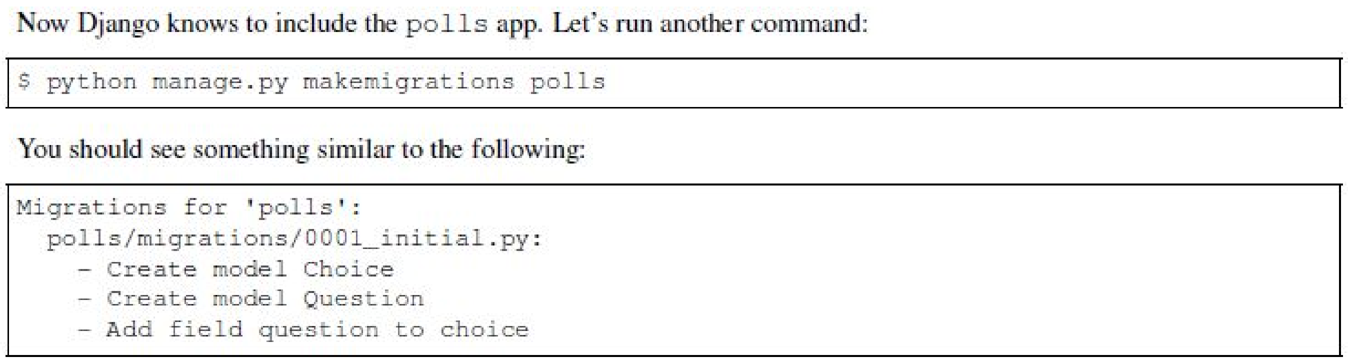
```
from django.db import models


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)
```

To include the app in our project, we need to add a reference to its configuration class in the *INSTALLED_APP* setting. The `PollsConfig` class is in the `polls/apps.py` file, so its dotted path is `'polls.apps PollsConfig'`. Edit the `mysite/settings.py` file and add that dotted path to the *INSTALLED_APPS* settin It'll look like this:



```
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',
]
```

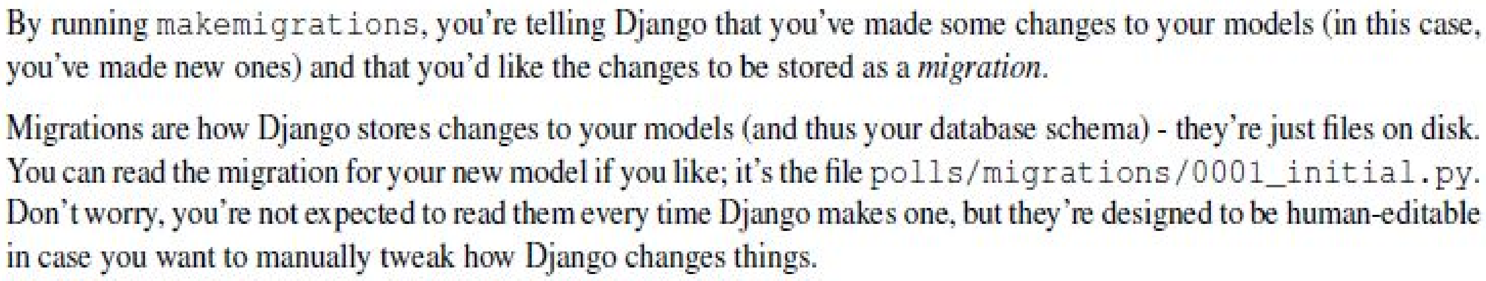


Now Django knows to include the `polls` app. Let's run another command:

```
$ python manage.py makemigrations polls
```

You should see something similar to the following:

```
Migrations for 'polls':
  polls/migrations/0001_initial.py:
    - Create model Choice
    - Create model Question
    - Add field question to choice
```

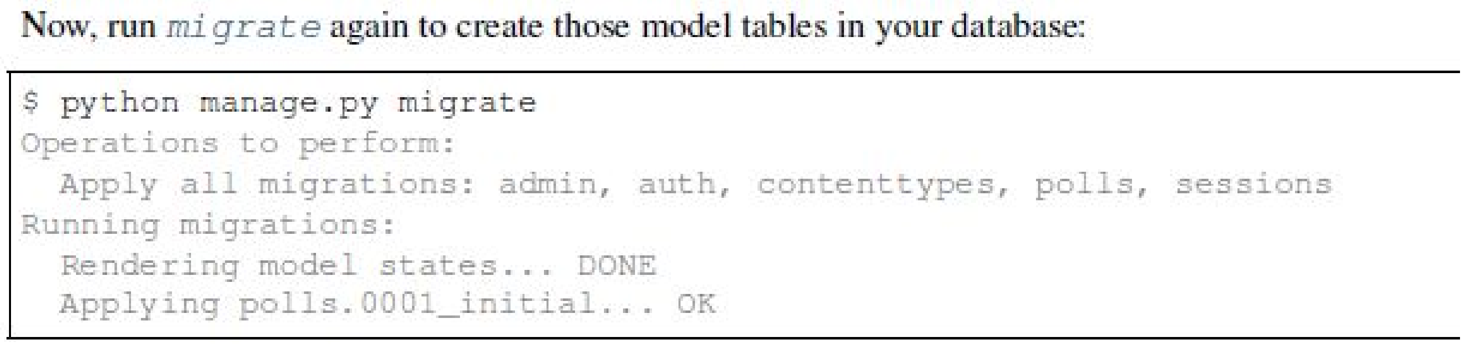


By running `makemigrations`, you're telling Django that you've made some changes to your models (in this case, you've made new ones) and that you'd like the changes to be stored as a *migration*.

Migrations are how Django stores changes to your models (and thus your database schema) - they're just files on disk. You can read the migration for your new model if you like; it's the file `polls/migrations/0001_initial.py`. Don't worry, you're not expected to read them every time Django makes one, but they're designed to be human-editable in case you want to manually tweak how Django changes things.

///////makemigrations command create migrations

////////migrate command create databases



Now, run *`migrate`* again to create those model tables in your database:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, polls, sessions
Running migrations:
  Rendering model states... DONE
  Applying polls.0001_initial... OK
```

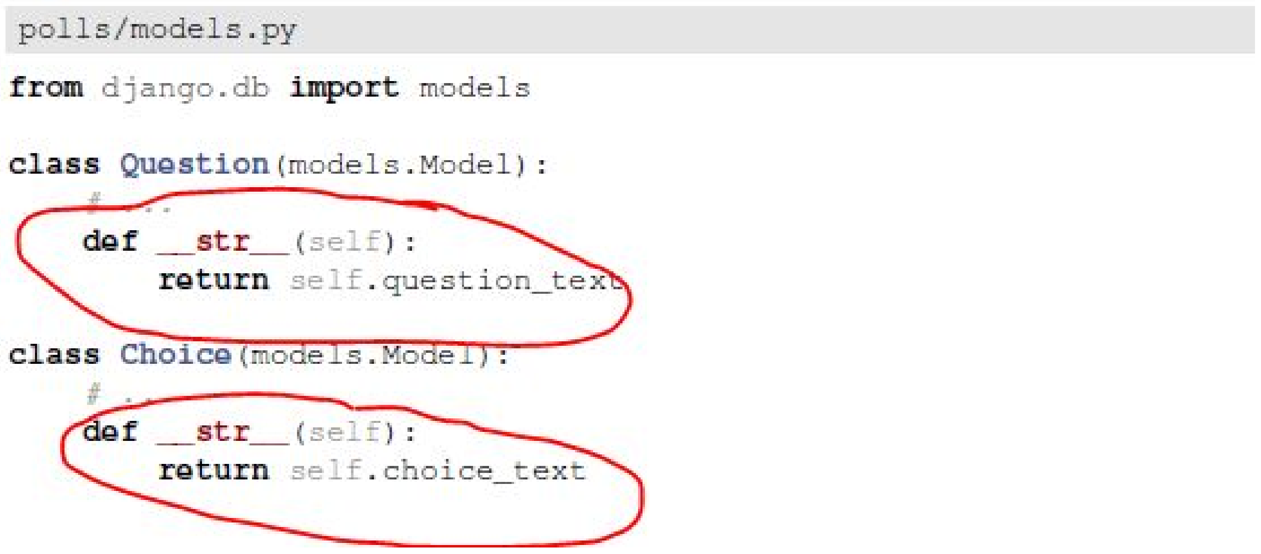
- Change your models (in `models.py`).
- Run *`python manage.py makemigrations`* to create migrations for those changes
- Run *`python manage.py migrate`* to apply those changes to the database.

```
$ python manage.py shell
```

Once you're in the shell, explore the *database API*:

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

# No questions are in the system yet.
```

```
>>> Question.objects.all()
<QuerySet []>

# 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.
>>> 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()
<QuerySet [<Question: Question object (1)>]>
```



```
polls/models.py
```

```python
from django.db import models

class Question(models.Model):
    # ...
    def __str__(self):
        return self.question_text

class Choice(models.Model):
    # ...
    def __str__(self):
        return self.choice_text
```

/////////////////////////////////// just demonstration....

```
polls/models.py
```

```python
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)
```

//

```
>>> from polls.models import Question, Choice

# Make sure our __str__() addition worked.
>>> Question.objects.all()
<QuerySet [<Question: What's up?>]>

# Django provides a rich database lookup API that's entirely driven by
# keyword arguments.
>>> Question.objects.filter(id=1)
<QuerySet [<Question: What's up?>]>
>>> Question.objects.filter(question_text__startswith='What')
<QuerySet [<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()
<QuerySet []>

# 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()
<QuerySet [<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)
<QuerySet [<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()
```

Creating an admin user

```
$ python manage.py createsuperuser
```

Enter your desired username and press enter.

```
Username: admin
```

You will then be prompted for your desired email address:

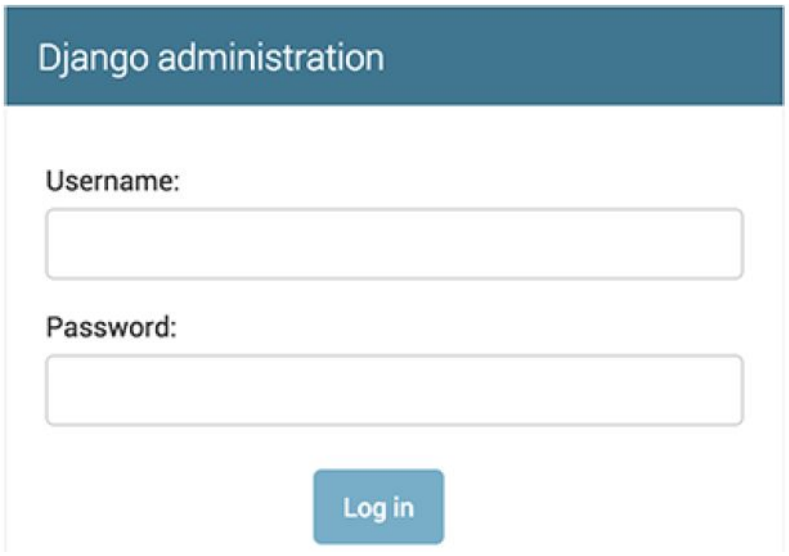
```
Email address: admin@example.com
```

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

```
Password: **********
Password (again): *********
Superuser created successfully.
```

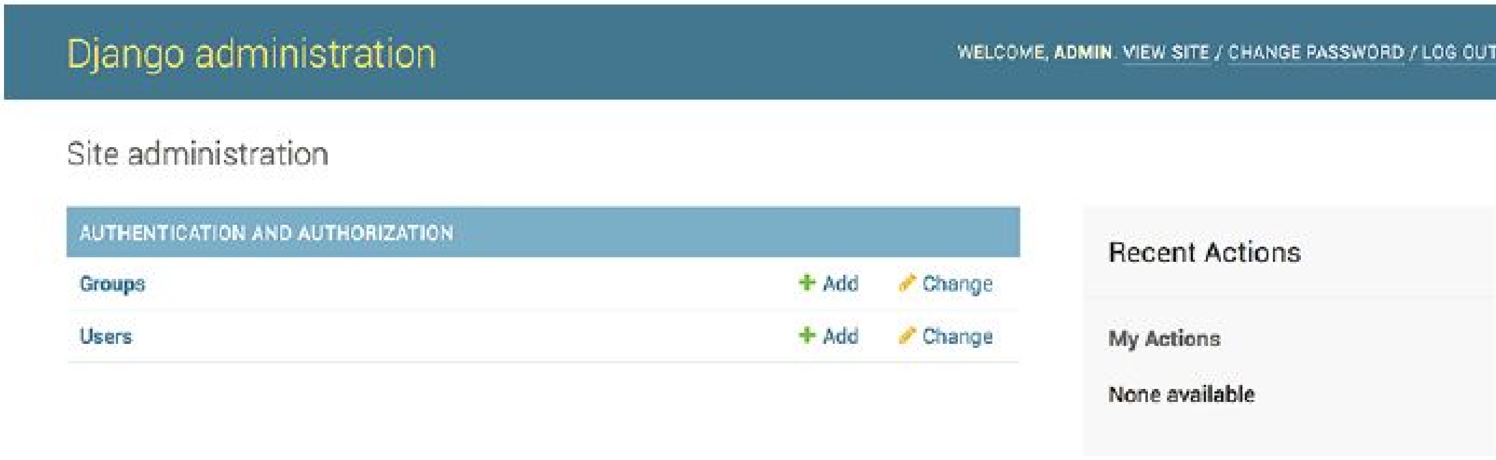
Start the development server

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



Enter the admin site

Now, try logging in with the superuser account you created in the previous step. You should see the Django a
index page:

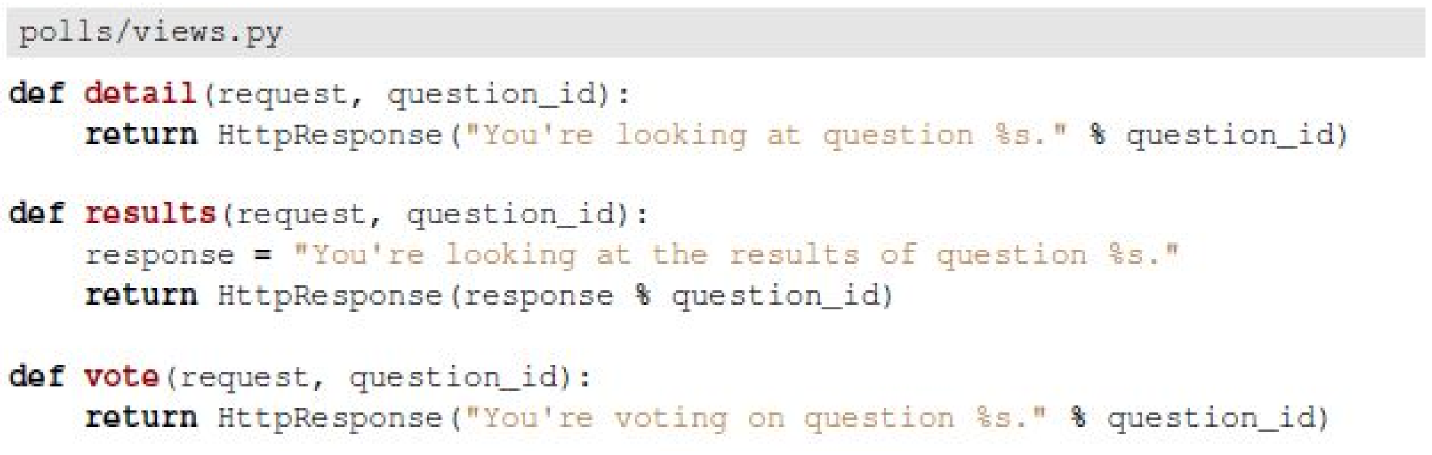


Make the poll app modifiable in the admin

```
polls/admin.py
```

```
from django.contrib import admin

from .models import Question

admin.site.register(Question)
```

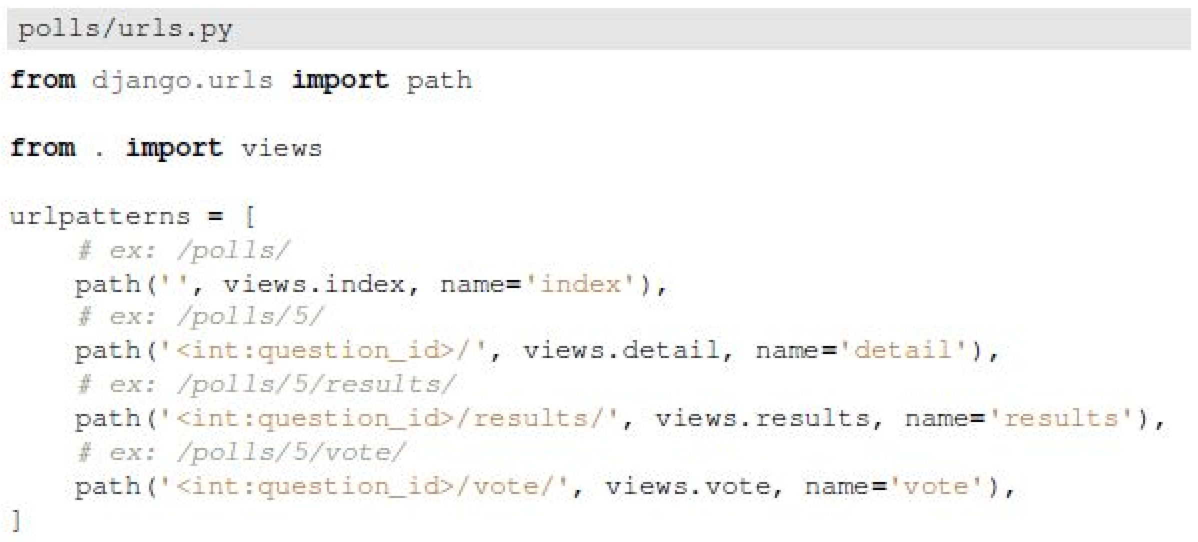
///////////////////Just writing more views



```
polls/views.py
```

```python
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)
```

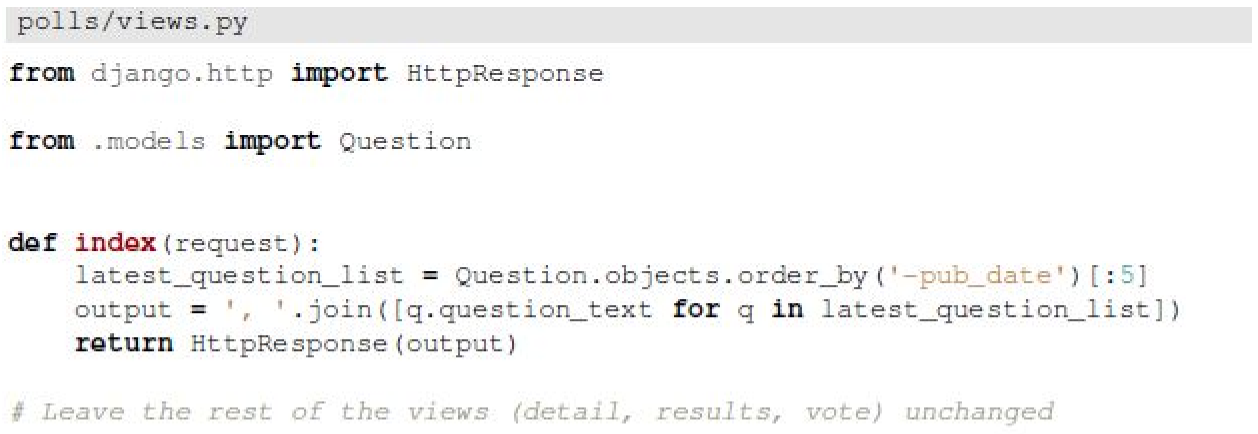
///////adding new views to url.py



```
polls/urls.py
```

```python
from django.urls import path

from . import views

urlpatterns = [
    # ex: /polls/
    path('', views.index, name='index'),
    # ex: /polls/5/
    path('<int:question_id>/', views.detail, name='detail'),
    # ex: /polls/5/results/
    path('<int:question_id>/results/', views.results, name='results'),
    # ex: /polls/5/vote/
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

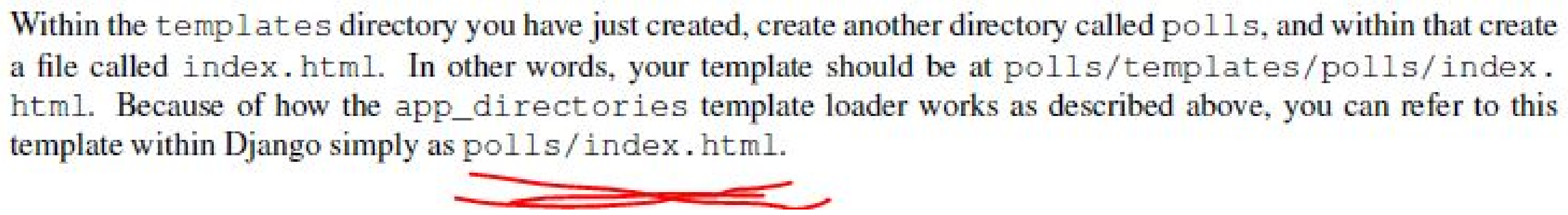
////////////////redefined index view to display last 5 questions



```
polls/views.py
```

```python
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
```

/////////////note about templates



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`.

Put the following code in that template:

```
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 %}
```

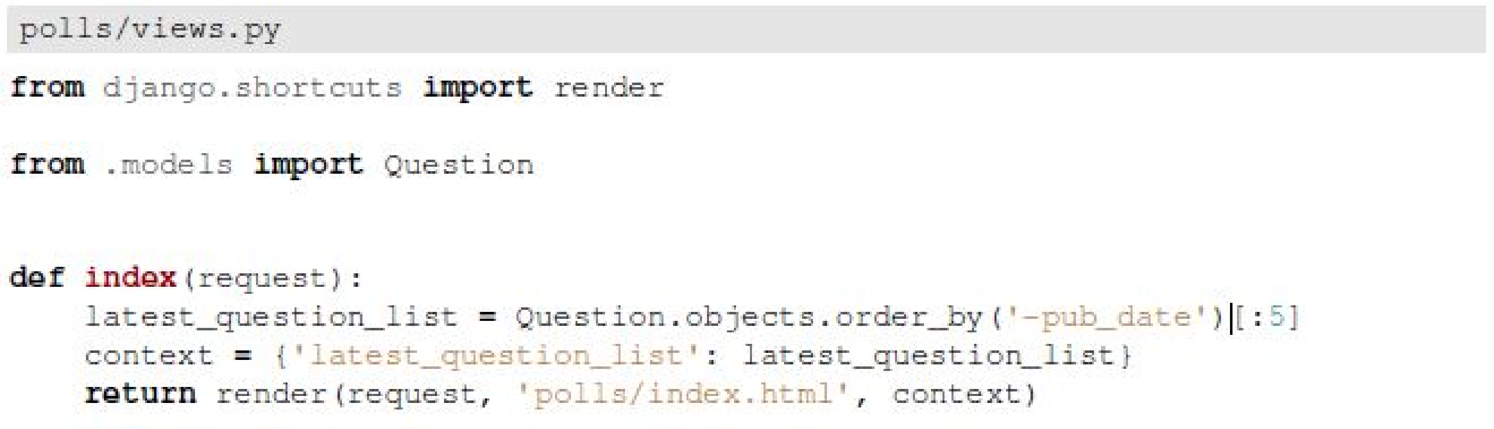
Now let's update our `index` view in `polls/views.py` to use the template:

```
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))
```

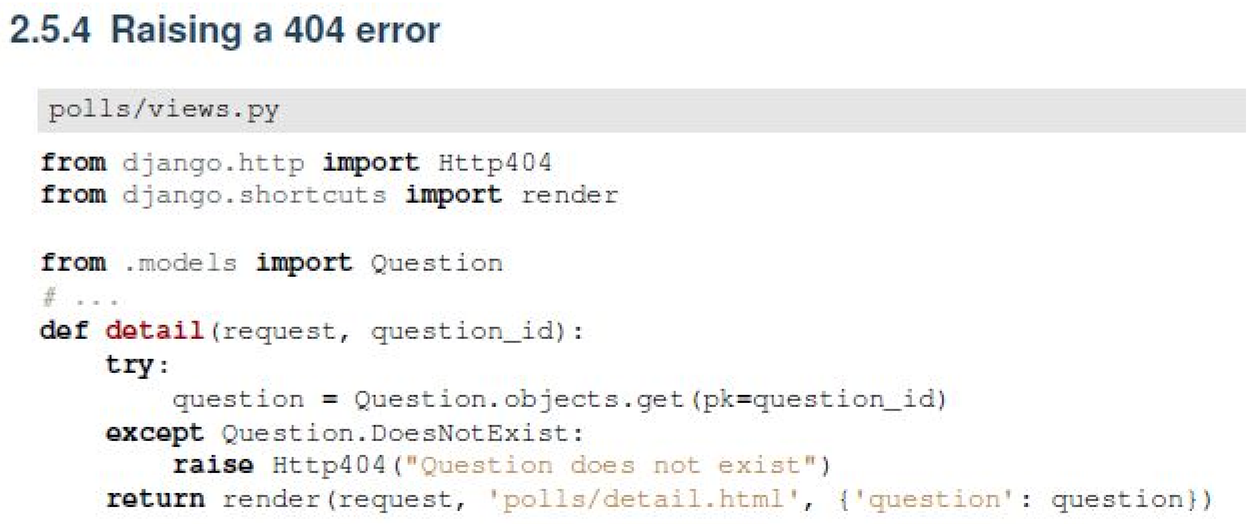
A shortcut: render()

It’s a very common idiom to load a template



```
polls/views.py
```

```python
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)
```

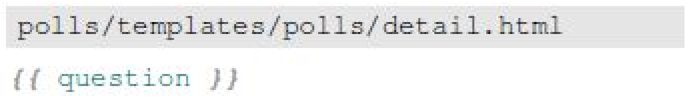


2.5.4 Raising a 404 error

```
polls/views.py
```

```python
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})
```

///////////small detail template



```
polls/templates/polls/detail.html
```

```
{{ question }}
```

/////////////It’s a very common idiom to use get() and raise Http404 if the object doesn’t exist so

A shortcut: 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})
```

//////////////////a better detail view

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>
```

2.5.6 Removing hardcoded URLs in templates

```
<li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
```

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

2.5.7 Namespacing URL names

In real Django projects, there might be five, ten, twenty apps or more.

How does Django differentiate the URL names between them? For example, the polls app has a detail view, and so might an app on the same project that is for a blog. How does one make it so that Django knows which app view to create for a url when using the {% url %} template tag?

/////////////go ahead and add an app_name to set the application namespace://////////////

```
polls/urls.py
```

```python
from django.urls import path

from . import views

app_name = 'polls'
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:question_id>/', views.detail, name='detail'),
    path('<int:question_id>/results/', views.results, name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

Now change your polls/index.html template from:

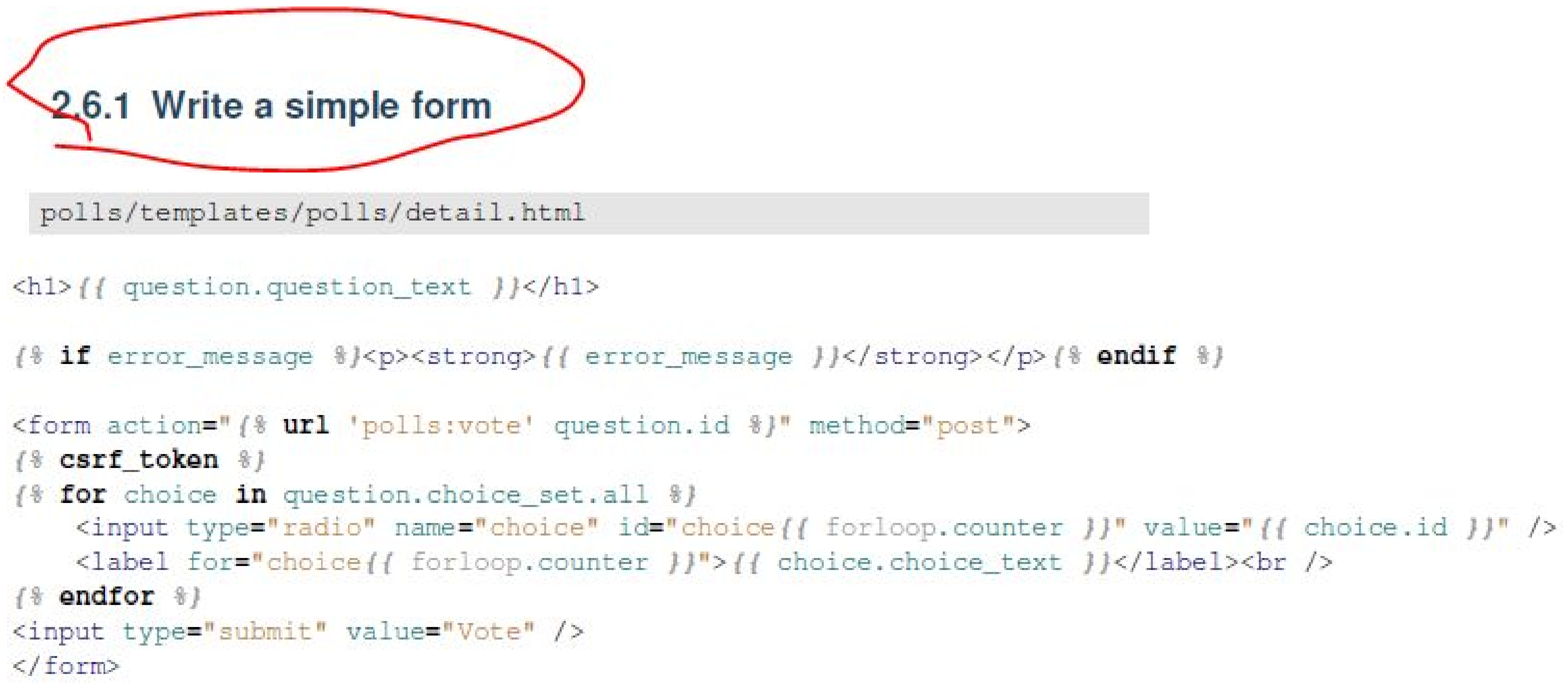
```
polls/templates/polls/index.html
```

```html
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

to this :

```
polls/templates/polls/index.html
```

```html
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
```



2.6.1 Write a simple form

polls/templates/polls/detail.html

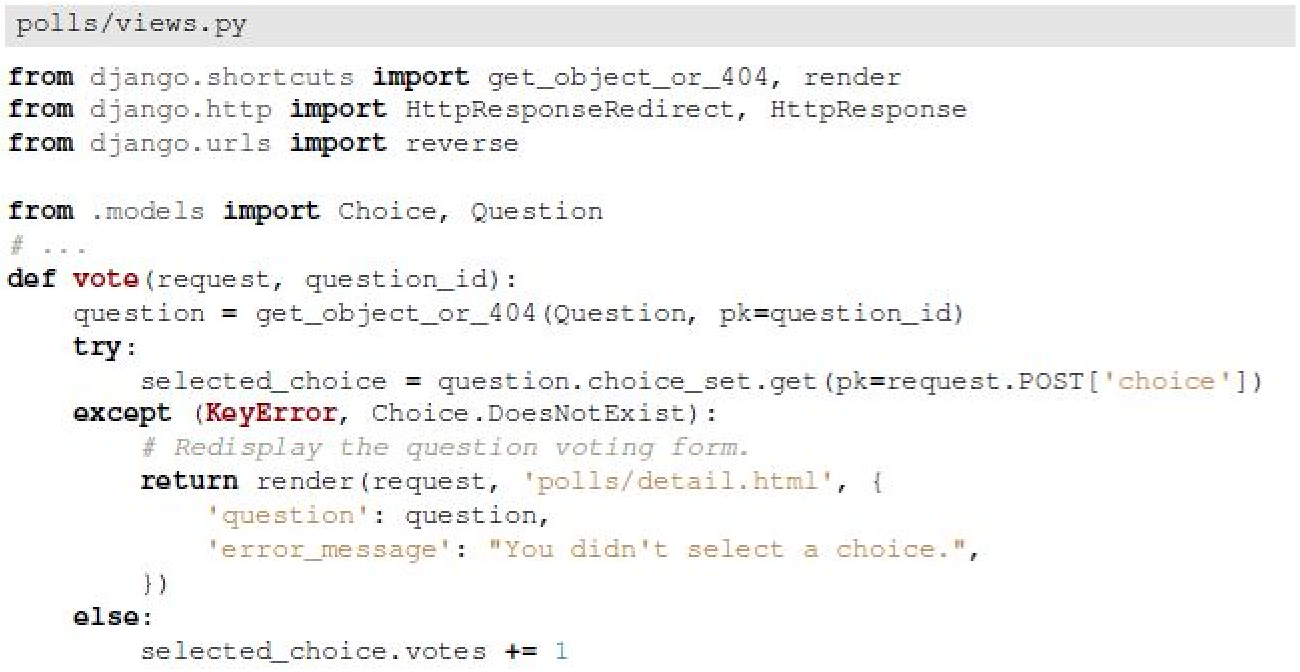
```
<h1>{{ question.question_text }}</h1>

{% 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>
```

this detail form take choice input and “posts” to “votes view”

that “error message “ thing will come when

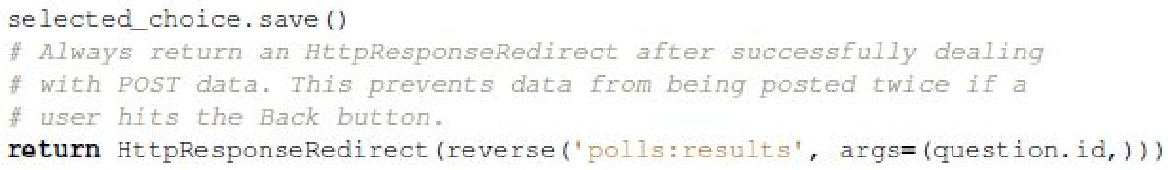
1. u submited “detail form” without option then
2. ur request passed to “votes view” and “vote view” renders a “detail view” with - “error message”

///////////Vote View////////////



```
polls/views.py
```

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect, HttpResponse
from django.urls 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,)))
```

///always return an HttpResponseRedirect after successfully dealing with POST data.

/////////the reverse() function in the HttpResponseRedirect constructor helps avoid having to hardcode a URL in the view function.

/////Results view and template////////////

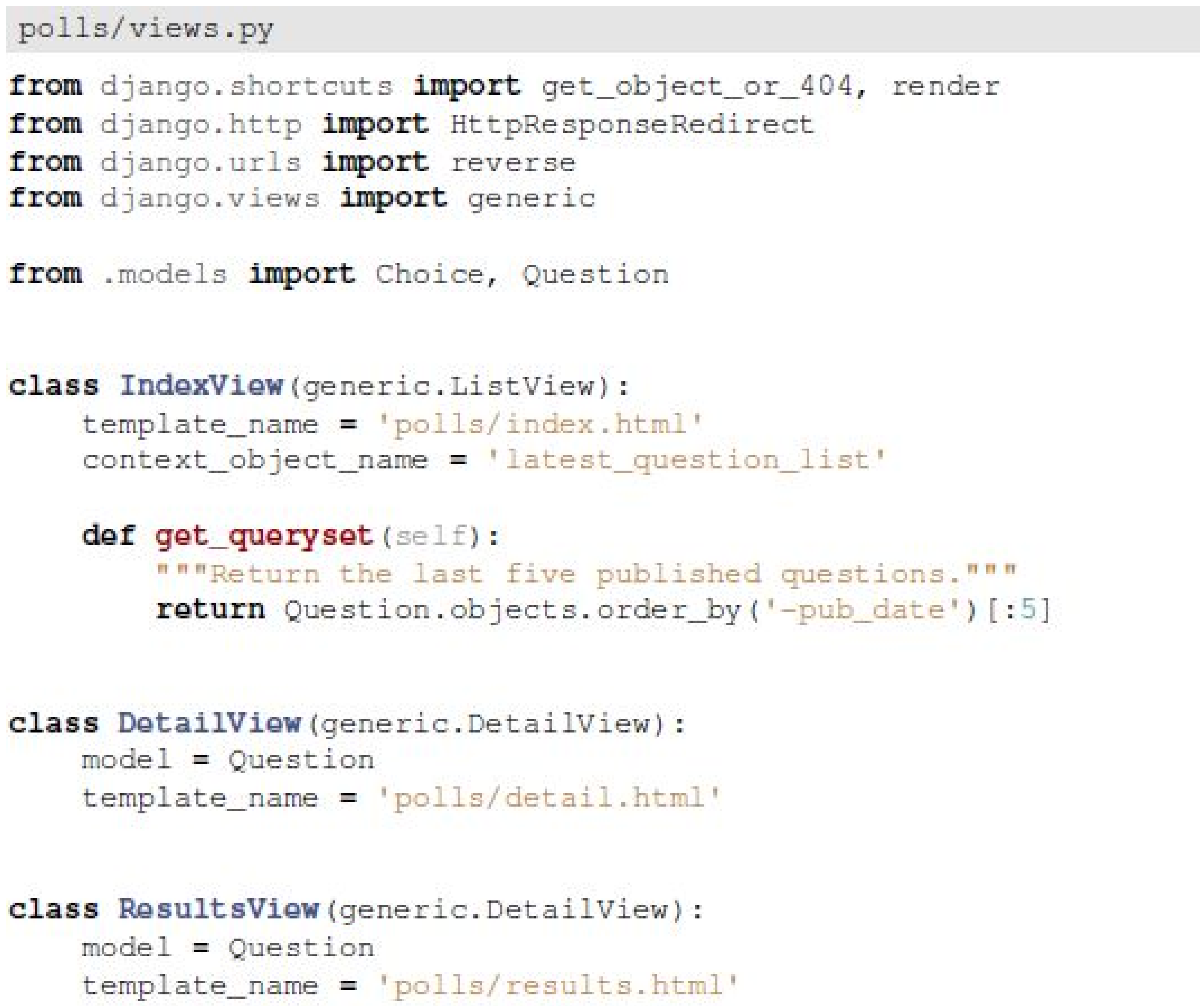
```
polls/views.py
from django.shortcuts import get_object_or_404, render


def results(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/results.html', {'question': question})
```

```
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 %}
```

2.6.2 Use generic views: Less code is better

```
polls/urls.py
from django.urls import path

from . import views

app_name = 'polls'

urlpatterns = [
    path('', views.IndexView.as_view(), name='index'),
    path('<int:pk>/', views.DetailView.as_view(), name='detail'),
    path('<int:pk>/results/', views.ResultsView.as_view(), name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

//////now update views to generic views



polls/views.py

```python
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect
from django.urls 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'
```

“vote view” no changes required