**DJANGO POLLS TUTORIAL, v5.0**

Notes by Michael Brothers

This is a companion to the Django Project Polls Tutorial (https://docs.djangoproject.com/en/5.0/intro/tutorial01/)
As we build out the project, each new variable name will be assigned a number (shown in blue), which will show how different parts of the project relate to one another. Common-use variable names are shown in green.
Section 1 describes the sequence of steps in creating a project.
Section 2 shows the file structure (folders and files).
Section 3 lists all of the named variables.
Section 4 shows the code content of each file.
For the development environment I use Windows 10, Anaconda3, Python 3.10 and Django 5.0.
I use Visual Studio Code as my Python editor.

# Contents

## SECTION I – THE PROJECT

Choosing a development environment is a matter of personal preference. I've found that **Visual Studio Code** from Microsoft has some useful built-in features.

### CREATING A VIRTUAL ENVIRONMENT USING CONDA

To begin a project, you first create a **virtual environment** – this dedicated container stores the particular versions of Python and Django to be used by your project, along with any libraries you may need. When you host your project in production, you'll likely put your virtual environment there as well.

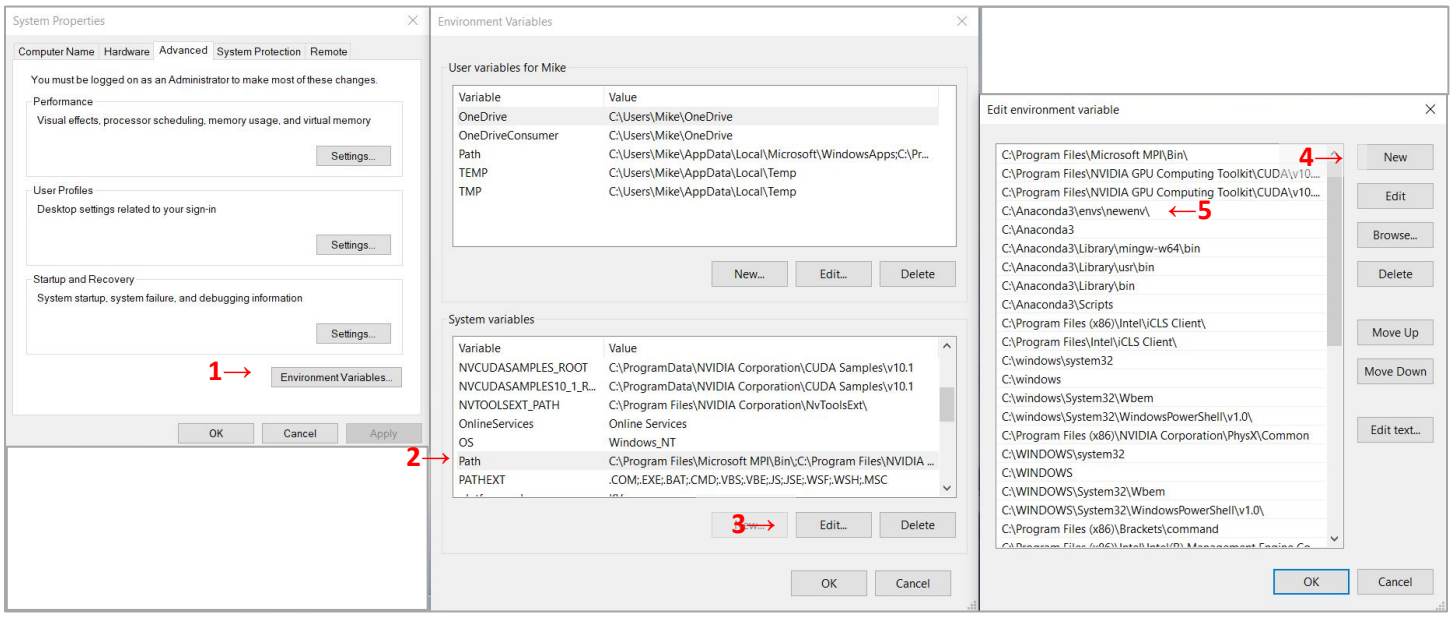
In this section we'll use the Anaconda distribution to build the environment, then use it inside Visual Studio Code.

1. Open a command prompt to your Anaconda distribution and create a new venv:

```
C:\Anaconda3>conda create -n newenv python=3.10 django=5
...
Proceed ([y]/n)? y
...
C:\Anaconda3>conda activate newenv
(newenv) C:\Anaconda3>conda list
# packages in environment at C:\Anaconda3\envs\newenv:
#
# Name                    Version                   Build  Channel
asgiref                   3.7.2              pyhd8ed1ab_0    conda-forge
bzip2                     1.0.8                hcfcfb64_5    conda-forge
ca-certificates           2024.2.2             h56e8100_0    conda-forge
django                    5.0.2              pyhd8ed1ab_0    conda-forge
libffi                    3.4.2                h8ffe710_5    conda-forge
libsqlite                 3.45.1               hcfcfb64_0    conda-forge
libzlib                   1.2.13               hcfcfb64_5    conda-forge
openssl                   3.2.1                hcfcfb64_0    conda-forge
pip                       24.0               pyhd8ed1ab_0    conda-forge
python                    3.10.13      h4de0772_1_cpython    conda-forge
setuptools                69.1.1             pyhd8ed1ab_0    conda-forge
sqlparse                  0.4.4              pyhd8ed1ab_0    conda-forge
tk                        8.6.13               h5226925_1    conda-forge
typing_extensions         4.10.0             pyha770c72_0    conda-forge
tzdata                    2024a                h0c530f3_0    conda-forge
ucrt                      10.0.22621.0         h57928b3_0    conda-forge
vc                        14.3                hcf57466_18    conda-forge
vc14_runtime              14.38.33130         h82b7239_18    conda-forge
vs2015_runtime            14.38.33130         hcb4865c_18    conda-forge
wheel                     0.42.0             pyhd8ed1ab_0    conda-forge
xz                        5.2.6                h8d14728_0    conda-forge
```

NOTE: These versions appeared in early 2024. Yours will likely be different. Also, while "newenv" is our first arbitrary variable name, it won't be seen anywhere inside the project, so I haven't numbered it.

2. Add `C:\Anaconda3\envs\newenv\` to your Windows System Path environment variable.
   This gives Visual Studio Code access to a Conda virtual environment.




**STARTING A DJANGO PROJECT**

3. Decide where to store your project. For this tutorial I've used the Windows desktop.

4. At a command prompt, navigate to your chosen project location, activate the virtual environment, then start a Django project called "mysite01":

```
C:\Users\Mike\Desktop>conda activate newenv
(newenv) C:\Users\Mike\Desktop>django-admin startproject mysite01
```

This creates the following file structure:

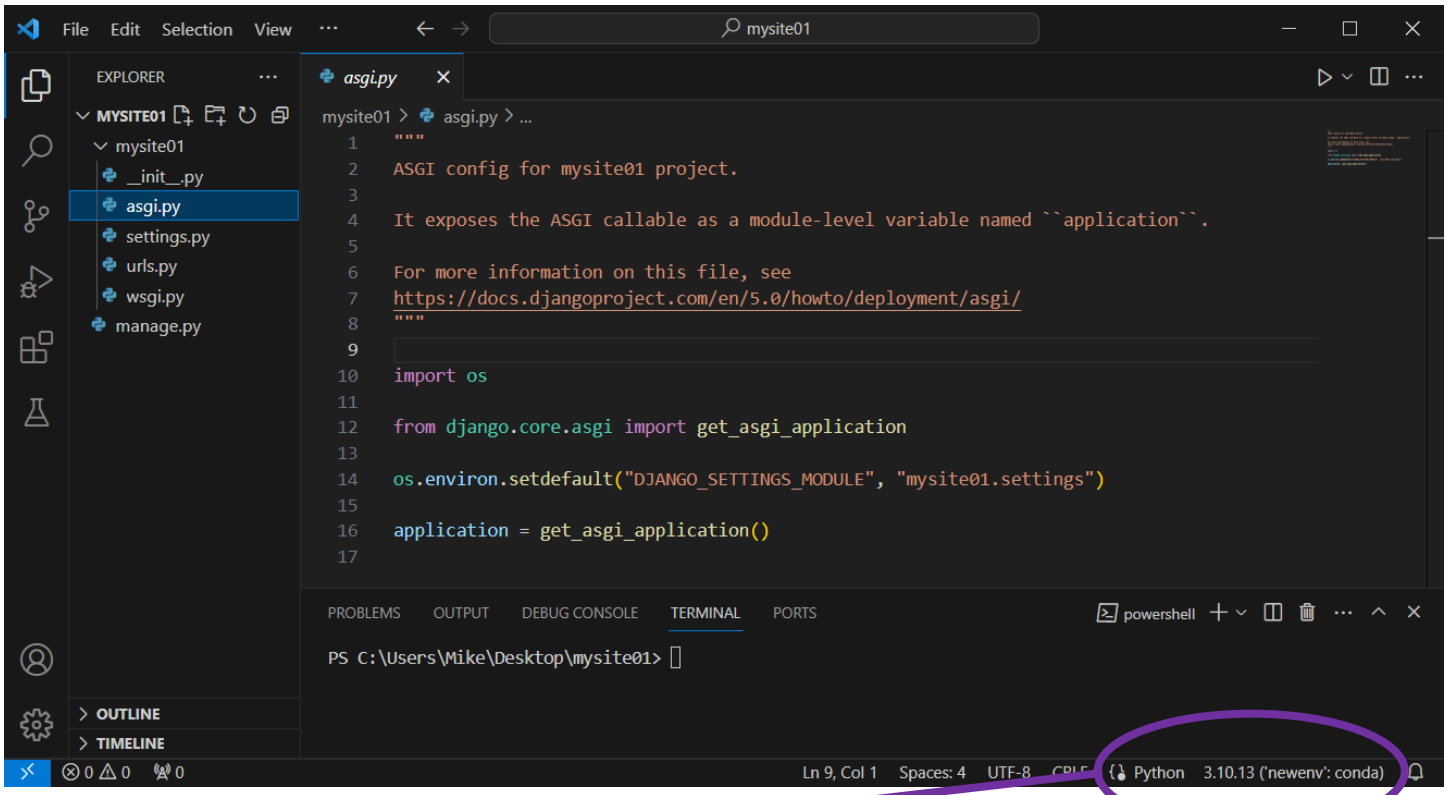
| | |
|---|---|
| 🗀 mysite01 | the outer mysite01 directory is just a container – it can be renamed if you want |
| 🗀 mysite01 | the inner mysite01 directory is the Python package for the project |
| 🗎 __init__.py | this blank script lets python know to treat this directory as a package |
| 🗎 asgi.py | Asynchronous Server Gateway Interface – an entry-point for newer ASGI web servers |
| 🗎 settings.py | where project settings are stored |
| 🗎 urls.py | URLconf file – the URL declarations for the project |
| 🗎 wsgi.py | Web Server Gateway Interface – an entry-point for WSGI-compatible web servers |
| 🗎 manage.py | will be associated with many commands as we build our web app |

5. Now launch Visual Studio Code. If this is your first Python project in VSCode you may need to install the Python extension.

6. Open the `mysite01` folder (Open Folder is found under the File menu).

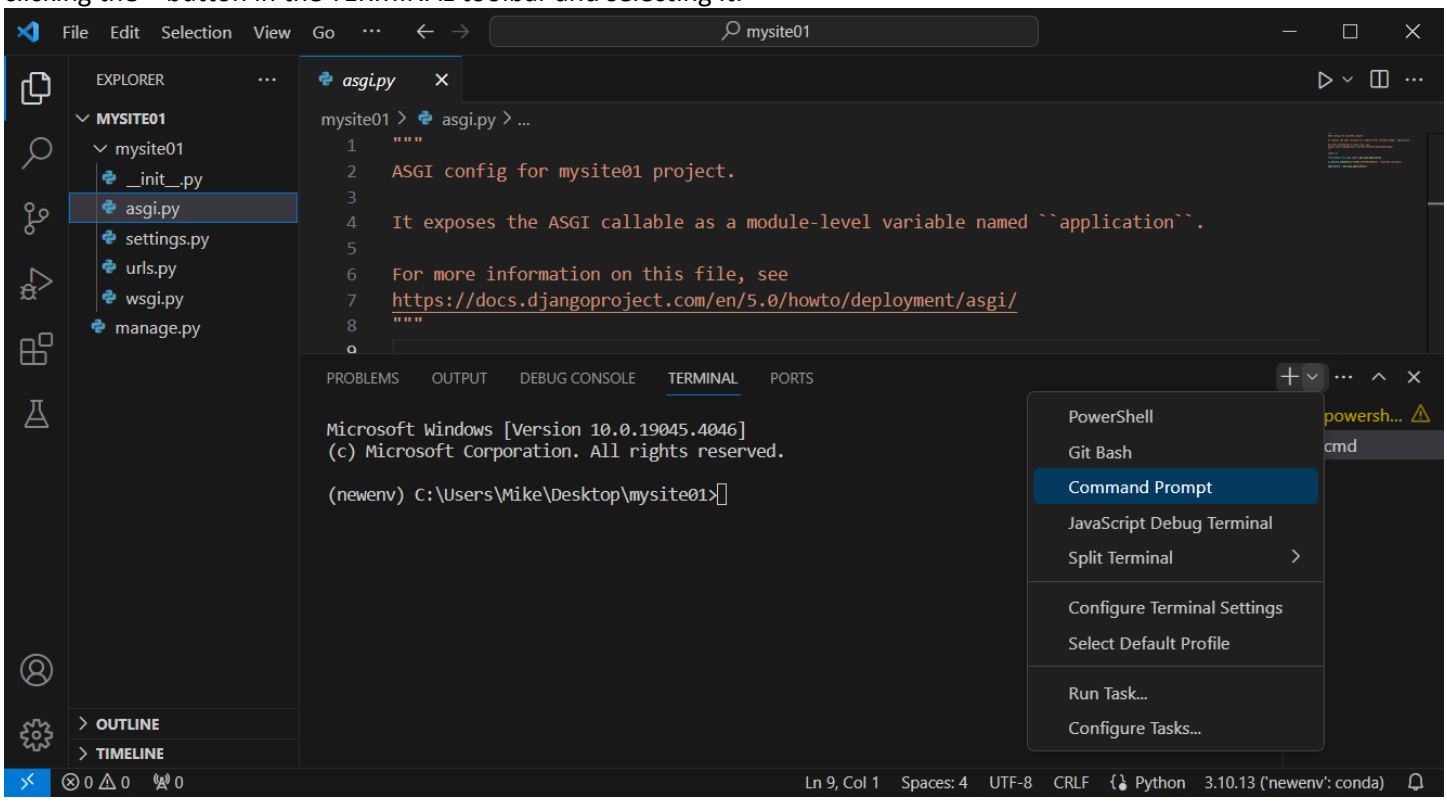
7. Open the Command Palette (found under the View menu or by hitting Ctrl+Shift+P). Search or scroll down to **Python: Select Interpreter** and click on it. Select the `('newenv')` interpreter from the Conda section of interpreters.

8. Expand the inner `mysite01` folder and click on the asgi.py file to reveal its contents:




NOTE: The currently running Python version and virtual environment are shown in the lower right.

9. By default the TERMINAL opens with PowerShell. This is fine, it works, but you can switch to a Command Prompt by clicking the + button in the TERMINAL toolbar and selecting it:



10. To confirm that Visual Studio Code has access to the Conda virtual environment, run

```
(newenv) C:\Users\Mike\Desktop\mysite01>python -m django --version
```

It should return `5.0.x` where "x" is the current Django point release.

11. Continue setting up the project by **running migrations** – Django automatically installs and configures a SQLite database for handling admin functions. This same database engine will be used later for the models we create:

```
(newenv) C:\Users\Mike\Desktop\mysite01>python manage.py migrate
```

This adds a __pycache__ directory with basic cpython scripts, and a top-level file called "db.sqlite3"

NOTE: the online tutorial saves this step for later. Performing it now won't hurt anything *unless* you plan to use something other than SQLite. If so, refer to the Django tutorial on database setup and don't run migrations here.

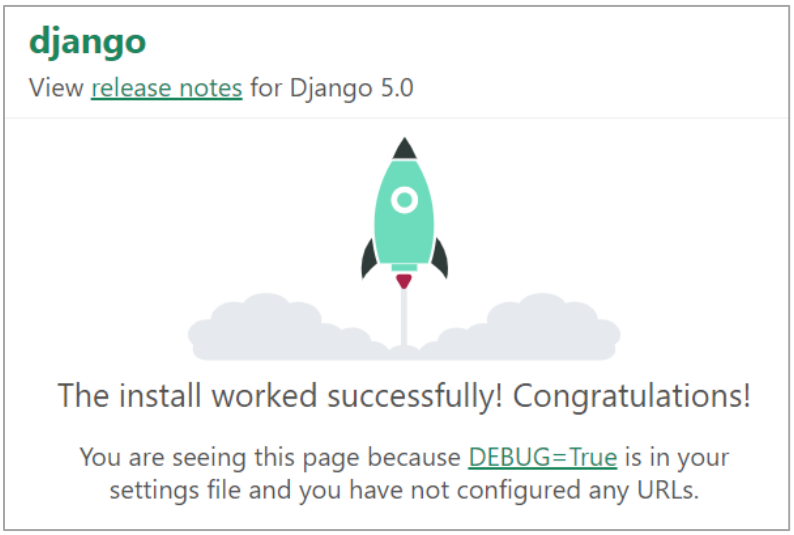
**LAUNCHING THE DEVELOPMENT SERVER**

12. As part of its "batteries included" philosophy, Django comes with a lightweight Web server written in Python:

```
(newenv) C:\Users\Mike\Desktop\mysite01>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
February 27, 2024 - 12:30:53
Django version 5.0.2, using settings 'mysite01.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

*NOTE: if you skipped Step 11, you may see an "unapplied migrations" message – you can ignore it for now.*

13. Now open a browser to http://127.0.0.1:8000 You should see something like:




Quick note: we see that **DEBUG = True** in Django settings. This is fine in development, but we'll want to change this to **False** when we push our files to production, to prevent users from accessing our debugging tools with Django!

**FIRST APPLICATION WITH DJANGO**

A quick note on terminology:

A **Django Project** is a collection of applications and configurations that, when combined together, make up a full web application (your complete website running with Django).

A **Django Application** is created to provide a particular functionality for your entire web application. For example, you could have a registration app, a polling app, a comments app, etc.

14. Back in the terminal, quit the development server with Ctrl+C and create an app called "polls02":

```
(newenv) C:\Users\Mike\Desktop\mysite01>python manage.py startapp polls02
```

This adds the following to the top level container directory:

- mysite01
  - mysite01
  - polls02
    - migrations — stores database-specific information as relates to the models
    - __init__.py — lets Python treat this as a package
    - admin.py — here you register models to be used by Django's admin interface
    - apps.py — stores application-specific configurations
    - models.py — stores the application's data models
    - tests.py — stores functions to test your code
    - views.py — stores functions to handle requests and return responses

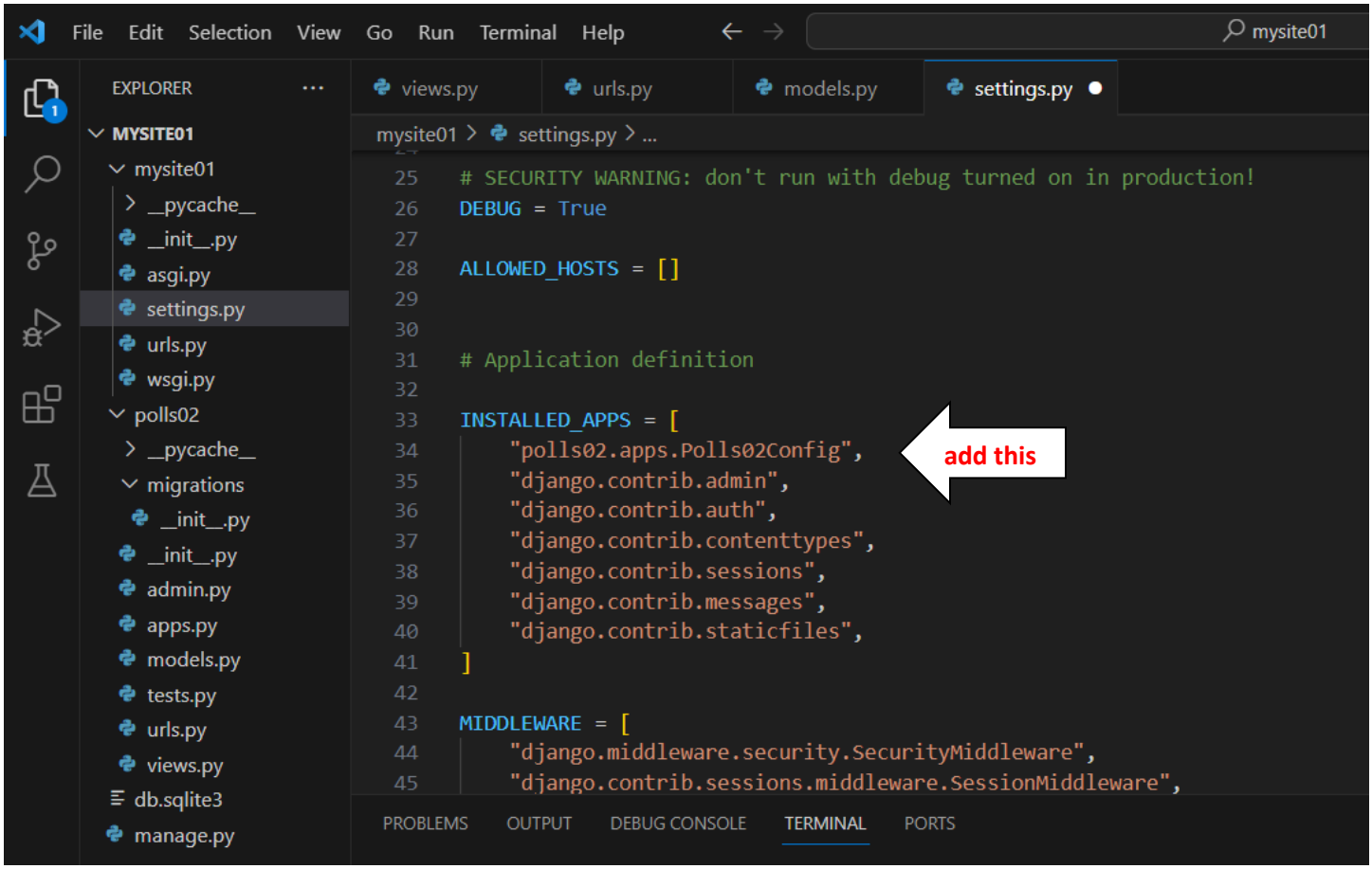
**REGISTERING THE APPLICATION**

When the app was created, Django automatically populated an **apps.py** file with a related configuration class:

```
from django.apps import AppConfig


class Polls02Config(AppConfig):
    default_auto_field = "django.db.models.BigAutoField"
    name = "polls02"
```

Now we need to tell our project that this app is installed.

15. Edit the **settings.py** file in the `mysite01` project directory. Register the polls02 app by adding the dotted subclass **`"polls02.apps.Polls02Config",`** (*include the comma!*) to INSTALLED_APPS:




This step is shown later in the online tutorial. Performing it now won't hurt anything, and it's good practice to do it once you've created an app.

**WRITING YOUR FIRST VIEW**

16. Open \polls02**views.py** and add the following code (in bold):

```
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.
def index04(request):
    return HttpResponse("Hello, world. You're at the polls index.")
```

Here, "index04" is the name of a *function-based view* (more on this later), and "request" is a common argument.

As soon as changes are made, a white dot appears in the views.py file tab. Save the file (either hit Save under the File menu or use Ctrl+S) and this should disappear.

**MAPPING A VIEW TO A URL**

17. Create a new file in the \polls02 directory called **urls.py**. Here we'll enter url patterns specific to the polls02 application, and then link this file to the project-level file. Add the following code:

```
from django.urls import path

from . import views

urlpatterns = [
    path("", views.index04, name="index05"),
]
```

Here, "views.index04" refers back to the function we wrote in views.py, and "index05" is a name we've assigned to this particular URL pattern.

18. Now open the **urls.py** file that lives in the \mysite01 project directory. Add the following code (in bold).
This will allow the project urls.py file to utilize anything we've included in our application urls.py file.
If we want to reuse our application in another project down the road, we can now just copy its entire directory.

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

urlpatterns = [
    path("admin/", admin.site.urls),
    path("polls03/", include("polls02.urls")),
]
```

Here, "polls03" is the url being passed in (ie, http://mysite/polls03/), while "polls02.urls" refers to the urls.py file that lives in our polls02 application directory.

19. Be sure to save any open files.

20. Finally, run the development server and open a browser to http://127.0.0.1:8000/polls03/
You should see "Hello, world. You're at the polls index."

**CREATING MODELS**

The online tutorial provides an excellent description of models and the backend database options available:

> In our poll app, we'll create two models: *Question* and *Choice*. A *Question* has a question and a publication date. A *Choice* has two fields: the text of the choice and a vote tally. Each *Choice* is associated with a *Question*.

21. Open \polls02**models.py** and add the following:

```
from django.db import models

# Create your models here.
class Question06(models.Model):
    question_text061 = models.CharField(max_length=200)
    pub_date062 = models.DateTimeField("date published")

class Choice07(models.Model):
    question071 = models.ForeignKey(Question06, on_delete=models.CASCADE)
    choice_text072 = models.CharField(max_length=200)
    votes073 = models.IntegerField(default=0)
```

Here, "Question06" and "Choice07" are model *classes*, which will become SQL tables.
"question_text061", etc. are field names in machine-readable format, which will become column names.
"date published" is a human-readable name that will appear instead of "pub_date062" in forms.

**MIGRATING MODELS**

Translating Python model classes into backend database tables involves two steps. First we stage the changes in a *migration*, then push the migration to the database.

22. In the terminal, enter the following:

```
(newenv) C:\Users\Mike\Desktop\mysite01>python manage.py makemigrations polls02
```

You should see something like:

```
Migrations for 'polls02':
  polls02\migrations\0001_initial.py
    - Create model Question06
    - Create model Choice07
```

This essentially stages all the changes we want to make to our SQL tables.

23. Now run migrations as we've done before:

```
(newenv) C:\Users\Mike\Desktop\mysite01>python manage.py migrate
```

Django does all the heavy lifting to set up our SQLite tables with our chosen field settings and validations!

**POPULATING MODELS with the ADMIN INTERFACE**

The next section of the online tutorial describes how to add records to models using the Python shell.
I'll skip ahead to Django's more powerful tool, the **admin interface**.

24. Update \polls02**models.py** with additional fields and methods – these will make entering data easier:

```
import datetime
from django.db import models
from django.utils import timezone

# Create your models here.
class Question06(models.Model):
    question_text061 = models.CharField(max_length=200)
    pub_date062 = models.DateTimeField('date published')

    def __str__(self):
        return self.question_text061

    def was_published_recently063(self):
        return self.pub_date062 >= timezone.now() - datetime.timedelta(days=1)

class Choice07(models.Model):
    question071 = models.ForeignKey(Question06, on_delete=models.CASCADE)
    choice_text072 = models.CharField(max_length=200)
    votes073 = models.IntegerField(default=0)

    def __str__(self):
        return self.choice_text072
```

Here, __str__(self) is a *method*. It passes something recognizable (in this case the contents of the "question_text" field) whenever a record is called, rather than the unhelpful object name Python assigns behind the scenes. "was_published_recently063" is another method. This one returns True/False depending on whether the pub_date is more than a day old.

Because we haven't added any new fields or classes to our model, there's no need to run migrations again. However, be sure to save the file so that changes take effect.

25. Spend some time reviewing the Django docs on the admin interface. The first step is to create an admin **superuser**:

```
(newenv) C:\Users\Mike\Desktop\mysite01>python manage.py createsuperuser
```

Enter a desired username and hit Enter:

```
Username (leave blank to use 'mike'): admin
```

Enter an email address:

```
Email address: admin@mysite01.com
```

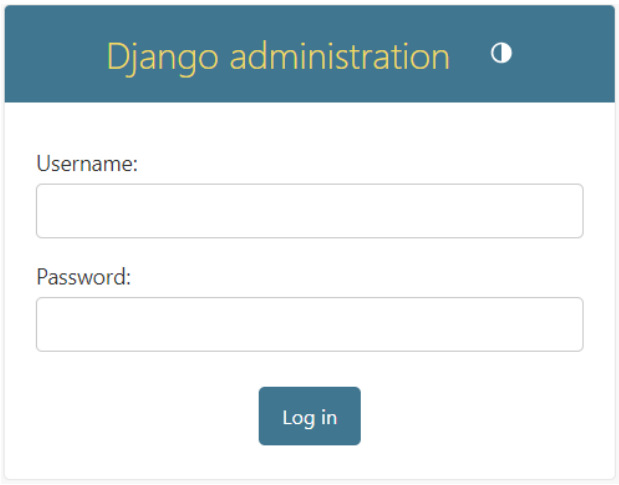
Enter an 8+ character uncommon password. You'll be prompted to enter it twice.
Note that "password" won't be accepted:

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

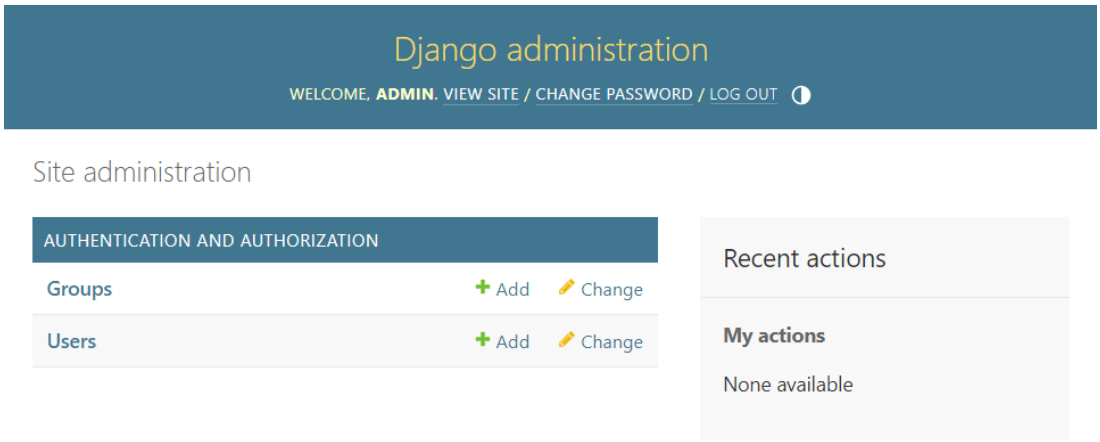
26. Next, run the development server:

```
(newenv) C:\Users\Mike\Desktop\mysite01>python manage.py runserver
```

Open a browser to http://127.0.0.1:8000/admin/ You should see:



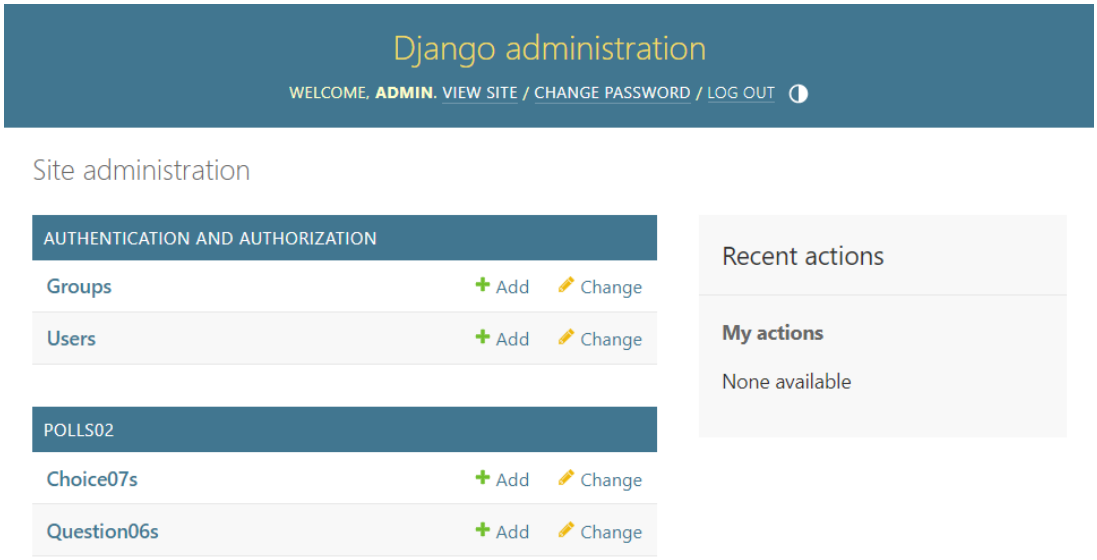

27. Login with your credentials to reach the Django admin page:




28. In order access our Polls02 app from the admin page, we need to register at least one of its models. Add the following (in bold) to \polls02**admin.py**:

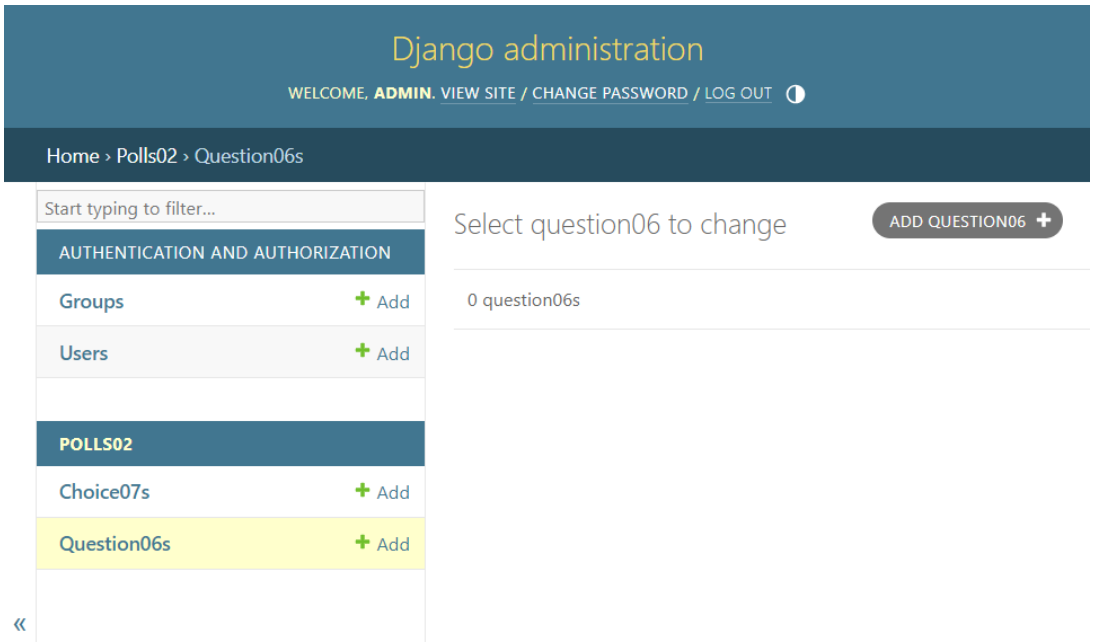
```
from django.contrib import admin
from .models import Question06, Choice07

# Register your models here.
admin.site.register(Question06)
admin.site.register(Choice07)
```

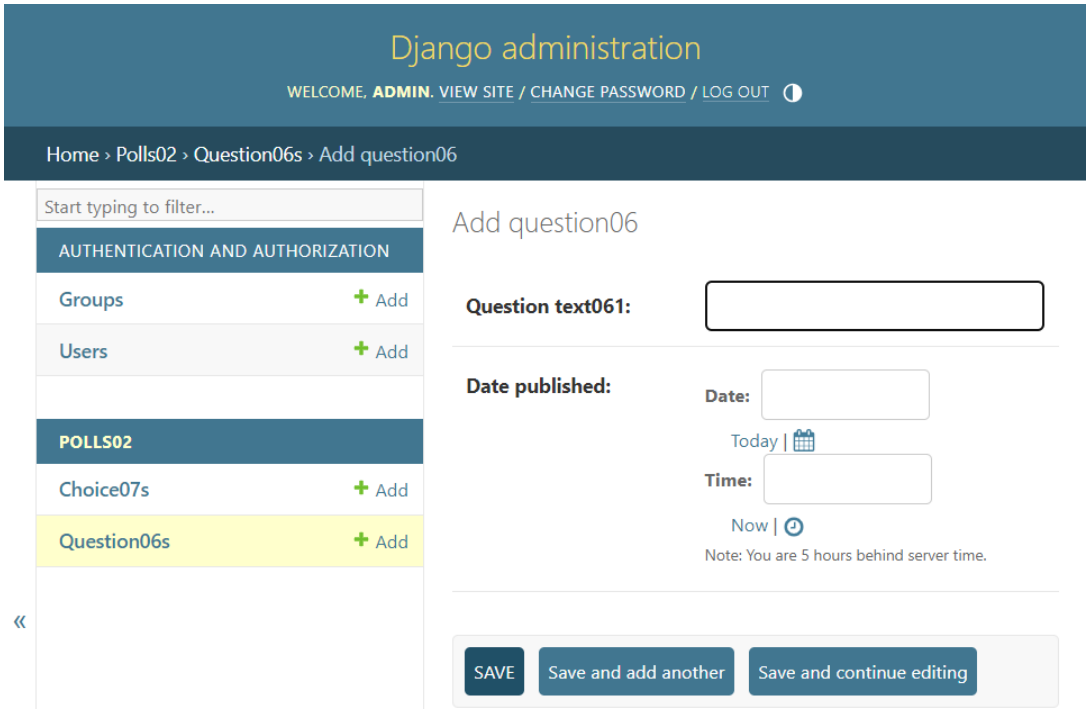
29. Refresh your browser window – you should now see the POLLS02 app:



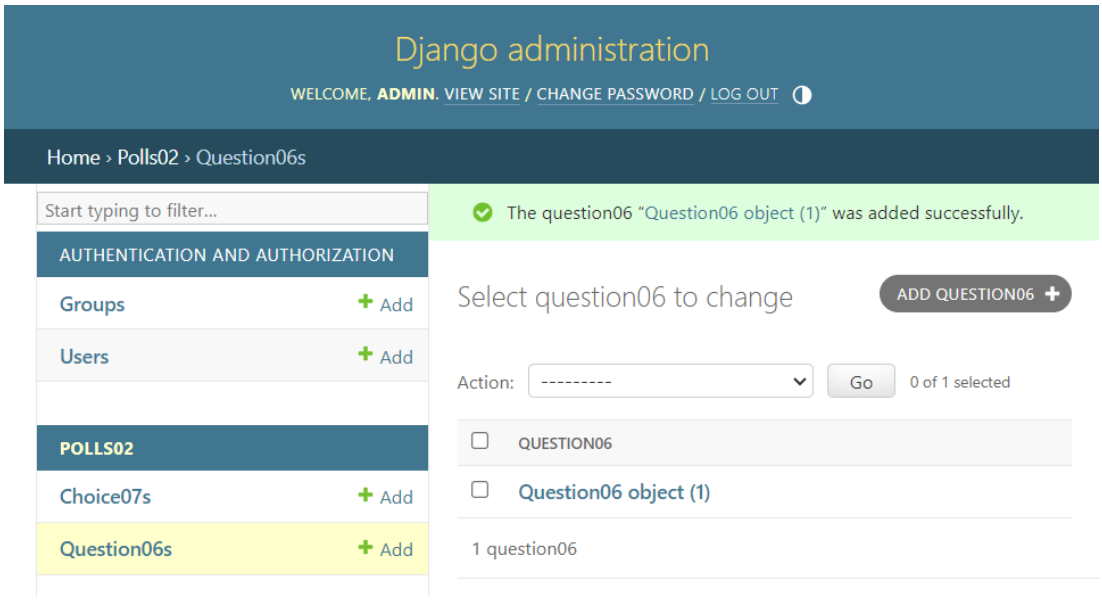

The Choice07s and Question06s fields look weird because Django automatically pluralizes field names. Also, fields sort in alphabetical order.

30. Let's add a Question record. Click on **Question06s**, and then click ADD QUESTION06 (in the upper right):

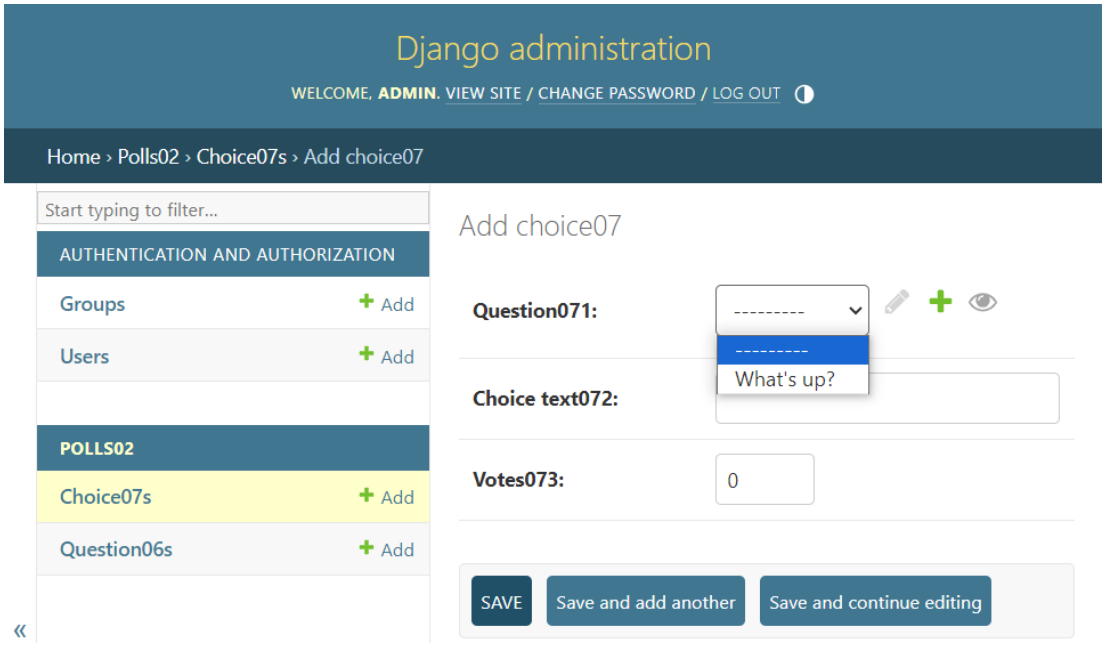





Type some question text like "What's up?" Note that if you try to save at this point you'll get an error, because the DateTimeField is required. Fortunately, Django provides built-in tools for this. Simply click on "Today" and "Now" to enter the current UTC time. Then hit SAVE.



31. Now go back to the Polls02 administration page (click **Polls02** in the header), and open **Choice07s**.
Here we need to link records to existing Question records – this is provided by a handy dropdown list:




NOTE: If you see something other than "What's up?" like "Question06 object (1)", then you likely haven't saved your models.py file. Do that now and refresh this page without closing it. Changes like this take effect immediately!

Select "What's up?", then enter some text like "Not much". Ignore the Votes field for now – we address this in a later section. Hit Save and add another. Add a few more choices like "The sky" and "Just hacking again".

That's it – we've just populated our project!

The admin interface lets you jump right into production without first crafting admin pages and data entry screens. Django allows for a lot of customization here, as well (more on this later).

**MORE WITH VIEWS & URLS**

32. Add the following functions to \polls02**views.py**:

```
def detail08(request, question_id):
    return HttpResponse(f"You're looking at question {question_id}.")

def results09(request, question_id):
    response = f"You're looking at the results of question {question_id}."
    return HttpResponse(response)

def vote10(request, question_id):
    return HttpResponse(f"You're voting on question {question_id}.")
```

Here, "detail08" is another function-based view. It differs from index04 in that it accepts an additional argument. For now, "question_id" is *not* a variable name, rather, it's a placeholder for whatever URL value gets passed to the function.

NOTE: The online tutorial does string formatting with modulo (%) placeholders. I prefer formatted string literals (f-strings) as they're cleaner.

33. Map these new views in the *application* copy of the URLconf file \polls02**urls.py**:

```
urlpatterns = [
    path("", views.index04, name="index05"),
    path("<int:question_id>/", views.detail08, name='detail11'),
    path("<int:question_id>/results/", views.results09, name='results12'),
    path("<int:question_id>/vote/", views.vote10, name='vote13'),
]
```

Now, when we call http://127.0.0.1:8000/polls03/34 in the browser, the response is:

You're looking at question 34.

At this point, nothing has tied back to the database; it simply parrots whatever URL you feed it.

**DJANGO TEMPLATES**

Before diving into database access through views.py, the online tutorial discusses templates.

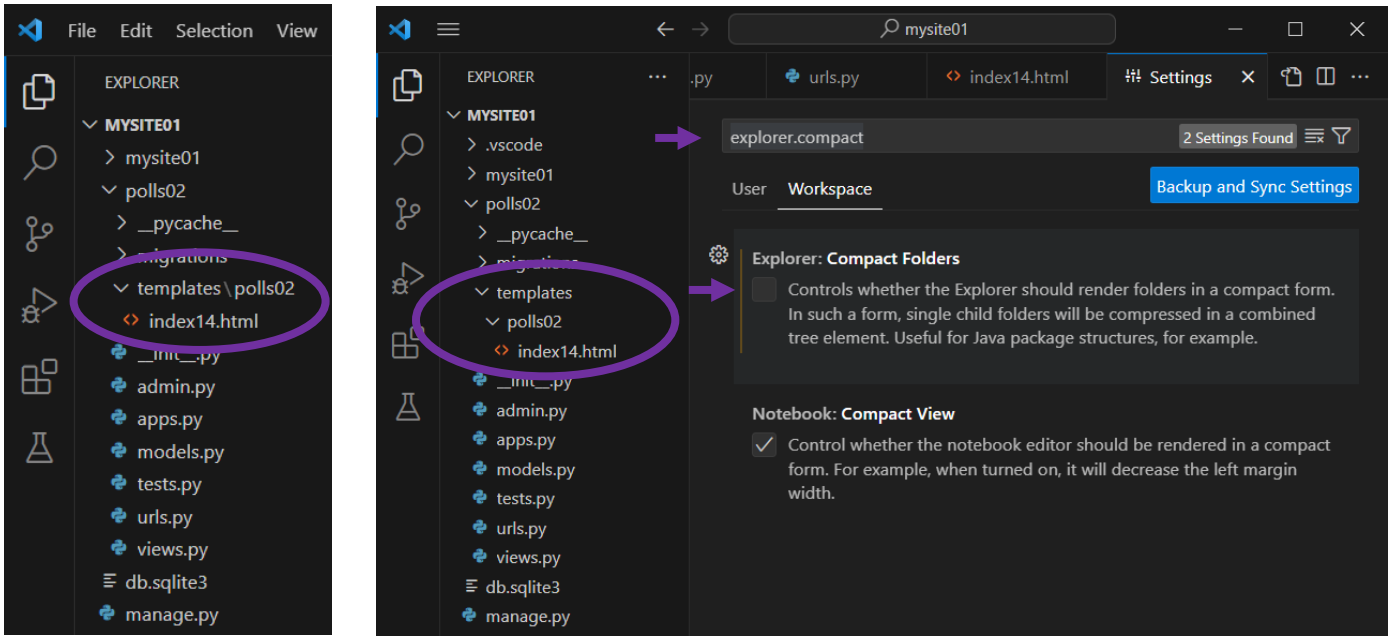
This is a good time to mention Django's take on the Model-View-Controller (MVC) paradigm for serving dynamic web content. In Django, Models are models, but Views are served by Django Templates, and the Controllers are Django Views. For this reason, we refer to Models-Templates-Views (MTV) instead.

34. To get started, add a folder called **templates** inside the application directory, and another folder called **polls02** inside of that. Next, create a file inside \polls02\templates\polls02\ called **index14.html**:

    mysite01
        polls02
            migrations
            **templates**
                **polls02**  this folder is part of the polls02 *namespace* – it's important to reuse the name
                **index14.html**
            __init__.py
            admin.py
            apps.py
            models.py
            tests.py
            views.py

    The easiest way to add files & folders in Visual Studio Code is to right-click on the destination and choose the appropriate task. Adding folders from the File menu brings you out to File Explorer.

    TIP: If your EXPLORER pane puts the new polls02 folder together with templates, it means that single child folders have been compressed into a combined tree element.To change this behavior, open File / Settings and search for "explorer.compact". Uncheck "Explorer: Compact Folders" and you should see polls02 as its own folder.




NOTE: Performing this adjustment adds a **.vscode** folder to MYSITE01. This dot folder houses settings specific to Visual Studio Code and is not really part of our project.

35. Add the following code to **index14.html**:

```
{% if latest_question_list15 %}
    <ul>
    {% for question in latest_question_list15 %}
    <li><a href="/polls03/{{ question.id }}/">
        {{ question.question_text061 }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

Here, "latest_question_list15" is an array that will be passed in by the view in Step 36. "question" is a placeholder used only within this template. "question.id" refers to the primary key field that Django automatically added to our model. It's named "id" by default, and contains an ascending set of integers starting at 1.

36. Next we'll create a view that can pass context to the index template. Add a function to **views.py** called index04a:

```
from django.shortcuts import render
from django.http import HttpResponse
from .models import Question06

def index04a(request):
    latest_question_list16 = Question06.objects.order_by('-pub_date062')[:5]
    context = {'latest_question_list15': latest_question_list16,}
    return render(request, 'polls02/index14.html', context)
```

Here, "latest_question_list16" holds an array of the first five objects in the Question06 model, sorted in reverse pub_date order. "latest_question_list15" carries the array to our template. The vehicle is a context dictionary, where the key is the string "latest_question_list15", and the value is the object "latest_question_list16".
Note that key and value don't have to share the same name.
NOTE: in the last line, render looks to the polls02 folder containing index14.html, *not* the polls03 url pattern.

**GET OBJECT OR 404()**

Django provides an exceptional error handler engineered to respect the loose coupling between models and views. See the online tutorial for more.

37. In **views.py**, add "get_object_or_404" to the list of imports, and add a detail08a function as follows:

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponse
from .models import Question06

def detail08a(request, question_id):
    question19 = get_object_or_404(Question06, pk=question_id)
    return render(request, 'polls02/detail17.html', {'question18': question19})
```

Here the get_object_or_404 method is looking for the name of our model (Question06) and the primary key value being passed. Django uses "pk" as a shortcut. Note that unlike the index04 function, here we didn't define a context dictionary; we simply passed a dictionary key/value pair as its own argument.

38. Create a **detail17.html** file in the \polls02\templates\polls02 folder, with just one line of code:

```
{{ question18 }}
```

This is how Django pulls variable names into a template! Here "question18" is the key in our context dictionary.

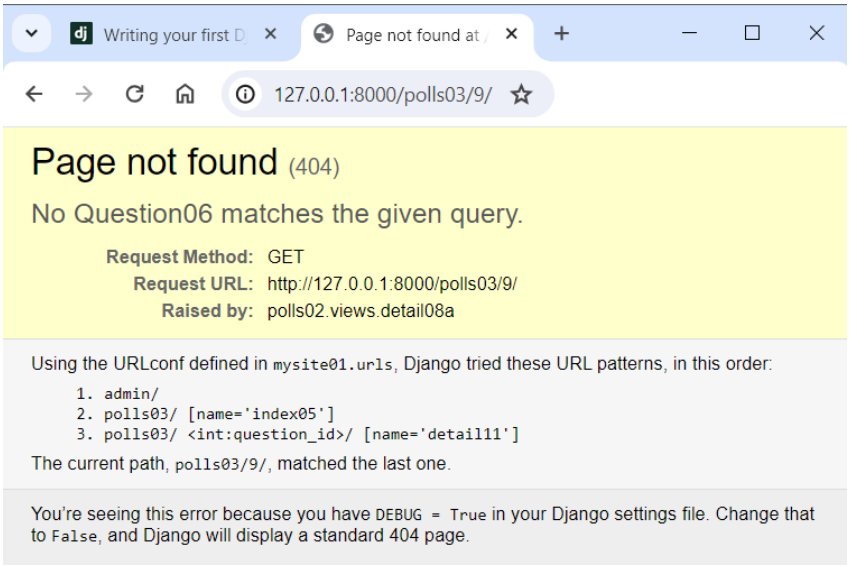
39. Lastly, update \polls02**urls.py** with our new views (leave everything else the same):

```
urlpatterns = [
    path("", views.index04a, name="index05"),
    path("<int:question_id>/", views.detail08a, name='detail11'),
    path("<int:question_id>/results/", views.results09, name='results12'),
    path("<int:question_id>/vote/", views.vote10, name='vote13'),
]
```

40. Enter http://127.0.0.1:8000/polls03/ into a browser. You should see (assuming the development server is still running) as many as five bullets, with questions you've added to your model.

    Click on one of the underlined entries to go to its Detail page. Notice how the URL changes to include an ID number after polls03/. At this point you should see just one entry question. Note that while the primary ID key was passed behind the scenes, what's shown on the page is determined by the model's __str__ method as defined in Step 24 – in this case, the question_text061 field. Without the __str__ method, all we would see here is "Question06 object"

    To see Django's error handler at work, change the ID portion of the URL to one that doesn't exist. I used /9/:



**URL NAMES and NAMESPACES**

Back in Step 35, we hardcoded a URL into the index14.html template file when we wrote:

```
<li><a href="/polls03/{{question.id}}/">{{question.question_text061}}</a></li>
```

This breaks some of the "loose coupling" we're trying to achieve between templates and views – if the URL schema were to change somewhere down the road, we'd have to come back and change this code as well.

To avoid this, Django makes use of *URL names*. Recall that in Step 33, when we mapped the path to call the detail08 view, we assigned the name "detail11" to the url pattern. We can use that here.

41. Change **index14.html** from:

```
<li><a href="/polls03/{{ question.id }}/">
```

to:

```
<li><a href="{% url 'detail11' question.id %}">
```

A quick note on template notation: **{%** *something* **%}** is a *template tag* – it allows us to perform logical operations in our template. In this case, the template tag calls the url.py file, finds the url pattern named "detail11", passes question.id to it, and provides this url as a hyperlink on top of the question_text in our list item. Not bad!

I'll also note that throughout this tutorial we're using Django's built-in Django Template Language (DTL). Other options are available, like Smarty or Jinja2 – to learn more visit the Django docs.

We're not quite done. The above solution works when we have only one application (polls), but what if we have ten applications, and each one has a detail view? The answer is with *namespaces*.

42. Open the \polls02**urls.py** file, and add an application name:

```
from django.urls import path

from . import views

app_name = 'polls02'
urlpatterns = [
    path("", views.index04a, name="index05"),
    path("<int:question_id>/", views.detail08a, name='detail11'),
    path("<int:question_id>/results/", views.results09, name='results12'),
    path("<int:question_id>/vote/", views.vote10, name='vote13'),
]
```

43. Then modify **index14.html** to include the namespace:

```
<li><a href"{% url 'polls02:detail11' question.id %}">
    {{ question.question_text061 }}</a></li>
```

Now the template tag knows *which* urls.py file to go to to find detail11!

NOTE: If you add an app_name to urls.py *without* attaching it to any related url pattern names, Django will raise a NoReverseMatch error when that url is called from the browser.

**FORMS**

Part 4 of the Django Project online tutorial introduces HTML *form* elements. We can greatly expand the functionality of our detail template if we prompt the user to vote on question choices, and retain the answers in our database.

44. Create a new template file in \polls02\templates\polls02\ called **detail17a.html** with the following:

```
<form action="{% url 'polls02:vote13' question18.id %}" method="post">
{% csrf_token %}
<fieldset>
    <legend><h1>{{ question18.question_text061 }}</h1></legend>
    {% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}
    {% for choice in question18.choice07_set.all %}
        <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{
choice.id }}">
        <label for="choice{{ forloop.counter }}">{{ choice.choice_text072
}}</label><br>
    {% endfor %}
</fieldset>
<input type="submit" value="Vote">
</form>
```

Let's break this down:

- The form will post the user's selection through the url namespace once the user clicks "vote".
- The *csrf_token* (shown in red) is important – it's a site security measure to prevent data from being stolen.
- The fieldset contains a header, an error message (if any), and a list of choices related to the selected question.
- The header displays the text of whatever question object has been passed to the template.
- "error_message" is a common name we'll use again in Step 45 when defining a vote view.

45. Next, in **views.py**, add the following imports: HttpResponseRedirect, F, reverse, and the Choice07 model. Change the template for the detail08a view from detail17 to detail17a, then add a function called vote10a:

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponse, HttpResponseRedirect
from django.db.models import F
from django.urls import reverse
from .models import Question06, Choice07

def detail08a(request, question_id):
    question19 = get_object_or_404(Question06, pk=question_id)
    return render(request, 'polls02/detail17a.html', {'question18': question19})

def vote10a(request, question_id):
    question19 = get_object_or_404(Question06, pk=question_id)
    try:
        selected_choice = question19.choice07_set.get(pk=request.POST['choice'])
    except (KeyError, Choice07.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls02/detail17a.html', {
            'question18': question19,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes073 = F("votes073") + 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('polls02:results12',
args=(question19.id,)))
```

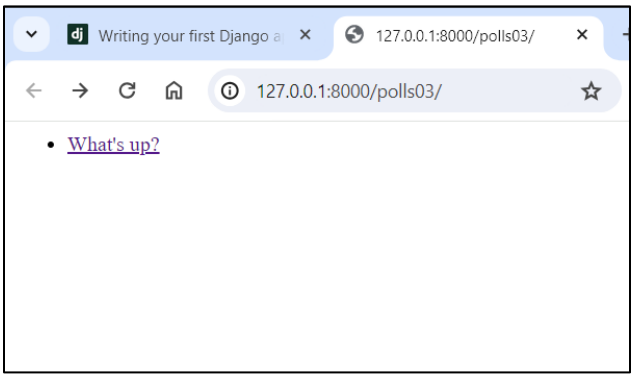
Breakdown:

- I'm reusing the names "question18" and "question19" as the dictionary key/value being passed by vote10a.
- "selected_choice" is a placeholder used only within this view as a way of assigning a vote.
- "error_message" is the common name we passed to the detail17a template.
- F("votes073") is a handy Django built-in expression that lets us modify database field values without pulling the original value into Python memory. For more on F() expressions visit the Django docs.

46. Update \polls02**urls.py** with the new vote view:

```
from django.urls import path

from . import views

app_name = 'polls02'
urlpatterns = [
#     path("", views.index04a, name="index05"),
#     path("<int:question_id>/", views.detail08a, name='detail11'),
#     path("<int:question_id>/results/", views.results09a, name='results12'),
#     path("<int:question_id>/vote/", views.vote10a, name='vote13'),
# ]
```

47. Run the development web server and open a browser to http://127.0.0.1:8000/polls03
Click a question to see its new detail page:

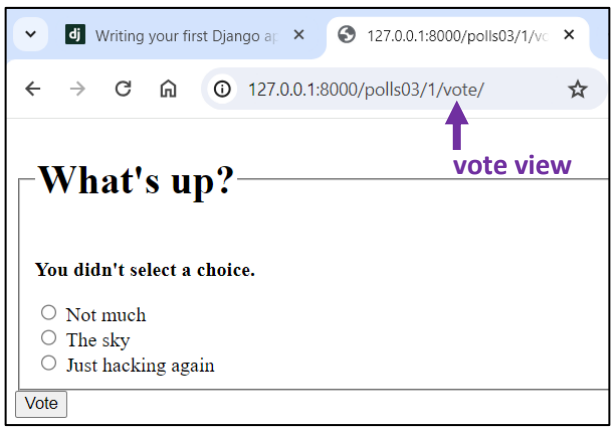




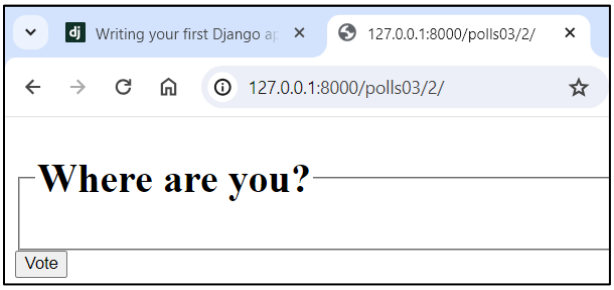
Some interesting things happened in Step 45. First, we try to retrieve any choices that are *children* of the question object passed to the view. Remember that we had to add choices separately through the admin interface, and that a question can be saved without any choices associated with it.

Second, we obtain choice records *through* the question record – that is, **question.choice_set.get** retrieves any choice records that link to the given question.

And finally, we took advantage of loose coupling with the KeyError return render request. If the user hits Vote without a selection, it re-displays the detail template with the message "You didn't select a choice." even though we're technically in the "vote" view.



It's worth noting that the online tutorial does *not* raise a KeyError if no choices exist. If we were to add a second question without linking choices to it, our detail view would look something like this:




48. To round things off, create a results09a function in \polls02**views.py**:

```
def results09a(request, question_id):
    question19 = get_object_or_404(Question06, pk=question_id)
    return render(request, 'polls02/results20.html', {'question18': question19})
```

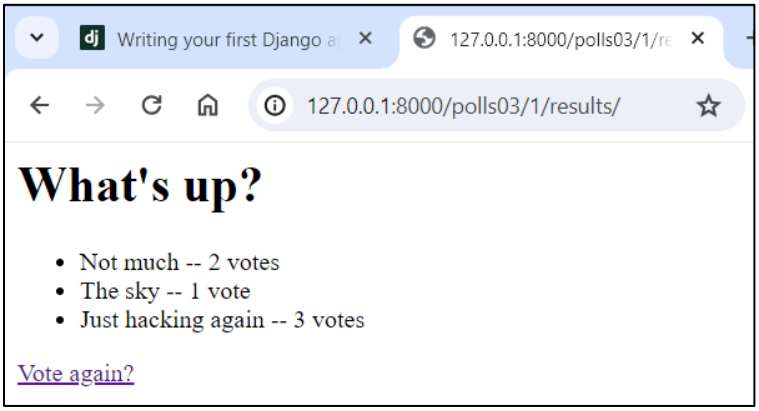
49. Update \polls02**urls.py** with the new view:

```
    path("<int:question_id>/results/", views.results09a, name='results12'),
```

50. Create a new template file called **results20.html**:

```
<h1>{{ question18.question_text061 }}</h1>

<ul>
{% for choice in question18.choice07_set.all %}
    <li>{{ choice.choice_text072 }} -- {{ choice.votes073 }}
        vote{{ choice.votes073|pluralize }}</li>
{% endfor %}
</ul>

<a href="{% url 'polls02:detail11' question18.id %}">Vote again?</a>
```

51. Remember to save all changes. Back in the browser, refresh the detail screen and cast a vote. You should now see the results page!




Something cool here: if the number of votes is greater than one, the "pluralize" filter adds an 's' to 'vote'.

**GENERIC VIEWS**

Up to this point, we've written our own functions to tell Django how to render a view. Part 4 of the online tutorial introduces Django's built-in, generic class-based views, which are designed to handle these common tasks for us.

This section removes a lot of our old code in favor of more streamlined generic views. I've kept the code in views.py for comparison, and I've introduced new variable numbers where appropriate.

52. First, change \polls02**urls.py** to the following:

```
from django.urls import path

from . import views

app_name = 'polls02'
urlpatterns = [
    path("", views.IndexView21.as_view(), name="index05"),
    path("<int:pk>/", views.DetailView22.as_view(), name="detail11"),
    path("<int:pk>/results/", views.ResultsView23.as_view(), name="results12"),
    path("<int:question_id>/vote/", views.vote10a, name="vote13"),
]
```

Note that the name assigned to the match patterns of the second and third views has changed from <question_id> to <pk> (for primary key). This is another one of Django's shortcuts.

53. Next, update \polls02**views.py** with the following new generic views:

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponse, HttpResponseRedirect
from django.db.models import F
from django.urls import reverse
from django.views.generic import ListView, DetailView
from .models import Question06, Choice07

# Create your views here.
class IndexView21(ListView):
    template_name = 'polls02/index14.html'
    context_object_name = 'latest_question_list15'

    def get_queryset(self):
        """Return the last five published questions."""
        return Question06.objects.order_by('-pub_date062')[:5]

class DetailView22(DetailView):
    model = Question06
    template_name = 'polls02/detail17a.html'
    context_object_name = 'question18'

class ResultsView23(DetailView):
    model = Question06
    template_name = 'polls02/results20.html'
    context_object_name = 'question18'


def vote10a(request, question_id):
   ...# leave everything else the same
```

Here we've imported two of Django's generic display views. Note that I changed the import line from the online tutorial – I changed **from django.views import generic** to **from django.views.generic import ListView, DetailView**. This way I don't need to include the module name in the class definition with **class IndexView(generic.ListView)**. For a list of Django's other views visit https://docs.djangoproject.com/en/5.0/ref/class-based-views/.

A major benefit to using class-based views is not having to define values for the context dictionary; the variables "latest_question_list16" and "question19" have disappeared. Instead, Django derives them from the classes themselves.

In the case of IndexView, Django would have passed the name "question_list" automatically. Since our index template is looking to receive "latest_question_list" we have to set that context_object_name ourselves.

Similarly, the DetailView and ResultsView could have gotten by without declaring context_object_names. We're forced to use them here because we've numbered our variables. In the absence of a specific name, Django uses the model name by default, so it would have passed "question" implicitly.

Part 5 of the online tutorial introduces testing. I'm not going to cover it here, but I strongly recommend checking out https://docs.djangoproject.com/en/5.0/topics/testing/

Part 6 and Part 7 cover HTML static files and customizing the admin form, respectively.

Part 8 discusses 3rd party packages like the Django Debug Toolbar.

That's it! I hope this was helpful. And please DON'T add numbers to your variable names – that was only done here for illustration purposes, to show how different parts of a Django project relate. As shown above, if you keep names simple and consistent, Django handles a lot of code for you. Good luck!

## SECTION II – FILE STRUCTURE

| | |
|---|---|
| mysite01 | the outer mysite01 directory is just a container – it can be renamed if you want |
| mysite01 | the inner mysite01 directory is the Python package for the project |
| __pycache__ | houses basic cpython scripts |
| __init__.py | this blank script lets python know to treat this directory as a package |
| asgi.py | Asynchronous Server Gateway Interface – an entry-point for ASGI web servers |
| settings.py | where project settings are stored |
| urls.py | URLconf file – the URL declarations for the project |
| wsgi.py | Web Server Gateway Interface – an entry-point for WSGI-compatible web servers |
| polls02 | this is our first application |
| __pycache__ | |
| migrations | stores database-specific information as relates to the models |
| templates | |
| polls02 | |
| detail17.html | html template files |
| detail17a.html | |
| index14.html | |
| results20.html | |
| __init__.py | lets Python treat this as a package |
| admin.py | here you register models to be used by Django's admin interface |
| apps.py | stores application-specific configurations |
| models.py | stores the application's data models |
| tests.py | stores functions to test your code |
| urls.py | URLconf file – application-specific URL declarations |
| views.py | stores functions and classes to handle requests and return responses |
| db.sqlite3 | |
| manage.py | this will be associated with many commands as we build our web app |

## SECTION III – VARIABLES

| | |
|---|---|
| mysite01 | a Project name. Django assigns this to both the working directory and the project directory. Technically these two folders could have different names – only the project directory name is called on for imports. |
| polls02 | an Application name |
| polls03 | a path in our URLconf file, translates (in development) to http://127.0.0.1:8000/polls03/ |
| index04 | a function-based view |
| index05 | a name assigned to the URLconf that calls on the index04 view |
| Question06<br>Choice07 | model classes |
| question_text061<br>pub_date062<br>question071<br>choice_text072<br>votes073 | fields belonging to the above model classes |
| was_published_recently063 | a class method that returns a value when called |
| detail08<br>results09<br>vote10<br>index04a<br>detail08a<br>results09a<br>vote10a | function-based views |
| detail11<br>results12<br>vote13 | URL pattern names |
| index14<br>detail17<br>detail17a<br>results20 | html Templates |
| latest_question_list15<br>latest_question_list16<br>question18<br>question19 | context dictionary key/value pairs passed from views.py to an html template |
| IndexView21<br>DetailView22<br>ResultsView23 | class-based views (aka generic views) |

**mysite01\urls.py**

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

urlpatterns = [
    path("admin/", admin.site.urls),
    path("polls03/", include("polls02.urls"))
]
```

**polls02\models.py**

```
import datetime
from django.db import models
from django.utils import timezone

# Create your models here.
class Question06(models.Model):
    question_text061 = models.CharField(max_length=200)
    pub_date062 = models.DateTimeField("date published")

    def __str__(self):
        return self.question_text061

    def was_published_recently063(self):
        return self.pub_date062 >= timezone.now() - datetime.timedelta(days=1)

class Choice07(models.Model):
    question071 = models.ForeignKey(Question06, on_delete=models.CASCADE)
    choice_text072 = models.CharField(max_length=200)
    votes073 = models.IntegerField(default=0)

    def __str__(self):
        return self.choice_text072
```

**polls02\urls.py**

```
from django.conf.urls import url

from . import views

app_name = 'polls02'
urlpatterns = [
    path("", views.IndexView21.as_view(), name="index05"),
    path("<int:pk>/", views.DetailView22.as_view(), name="detail11"),
    path("<int:pk>/results/", views.ResultsView23.as_view(), name="results12"),
    path("<int:question_id>/vote/", views.vote10a, name="vote13"),
]

# The following url patterns are no longer part of the project:
#     path("", views.index04a, name="index05"),
#     path("<int:question_id>/", views.detail08a, name='detail11'),
#     path("<int:question_id>/results/", views.results09a, name='results12'),
```

**polls02\views.py**

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponse, HttpResponseRedirect
from django.db.models import F
from django.urls import reverse
from django.views.generic import ListView, DetailView
from .models import Question06, Choice07

# Create your views here.
class IndexView21(ListView):
    template_name = 'polls02/index14.html'
    context_object_name = 'latest_question_list15'

    def get_queryset(self):
        """Return the last five published questions."""
        return Question06.objects.order_by('-pub_date062')[:5]

class DetailView22(DetailView):
    model = Question06
    template_name = 'polls02/detail17a.html'
    context_object_name = 'question18'

class ResultsView23(DetailView):
    model = Question06
    template_name = 'polls02/results20.html'
    context_object_name = 'question18'


def vote10a(request, question_id):
    question19 = get_object_or_404(Question06, pk=question_id)
    try:
        selected_choice = question19.choice07_set.get(pk=request.POST['choice'])
    except (KeyError, Choice07.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls02/detail17a.html', {
            'question18': question19,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes073 += 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('polls02:results12',
args=(question19.id,)))
```

**polls02\views.py, cont'd**

```
# These function-based views are no longer part of the project:
def index04(request):
    return HttpResponse("Hello, world. You're at the polls index.")

def index04a(request):
    latest_question_list16 = Question06.objects.order_by('-pub_date062')[:5]
    context = {'latest_question_list15': latest_question_list16,}
    return render(request, 'polls02/index14.html', context)

def detail08(request, question_id):
    return HttpResponse("You're looking at question %s." % question_id)

def detail08a(request, question_id):
    question19 = get_object_or_404(Question06, pk=question_id)
    return render(request, 'polls02/detail17a.html', {'question18': question19})

def results09(request, question_id):
    response = f"You're looking at the results of question {question_id}."
    return HttpResponse(response)

def results09a(request, question_id):
    question19 = get_object_or_404(Question06, pk=question_id)
    return render(request, 'polls02/results20.html', {'question18': question19})

def vote10(request, question_id):
    return HttpResponse(f"You're voting on question {question_id}.")
```

**polls02\templates\polls02\detail17.html**

```
{{ question18 }}
```

**polls02\templates\polls02\detail17a.html**

```
<form action="{% url 'polls02:vote13' question18.id %}" method="post">
{% csrf_token %}
<fieldset>
    <legend><h1>{{ question18.question_text061 }}</h1></legend>
    {% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}
    {% for choice in question18.choice07_set.all %}
        <input type="radio" name="choice" id="choice{{ forloop.counter }}"
value="{{ choice.id }}">
        <label for="choice{{ forloop.counter }}">{{ choice.choice_text072
}}</label><br>
    {% endfor %}
</fieldset>
<input type="submit" value="Vote">
</form>
```

**polls02\templates\polls02\index14.html**

```
{% if latest_question_list15 %}
    <ul>
    {% for question in latest_question_list15 %}
    <li><a href="{% url 'polls02:detail11' question.id %}">
        {{ question.question_text061 }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

**polls02\templates\polls02\results20.html**

```
<h1>{{ question18.question_text061 }}</h1>

<ul>
{% for choice in question18.choice07_set.all %}
    <li>{{ choice.choice_text072 }} -- {{ choice.votes073 }}
        vote{{ choice.votes073|pluralize }}</li>
{% endfor %}
</ul>

<a href="{% url 'polls02:detail11' question18.id %}">Vote again?</a>
```