Hyperiondev

TASK

Django -
Poll App and Templates

Visit our website

Introduction

WELCOME TO THE DJANGO - POLL APP AND TEMPLATES TASK!

In this task, we will continue building our poll application. In the process we will cover some important topics such as working with regular expressions, basic error handling, and creating forms and templates.

Disclaimer: We've curated the most relevant bits of the official documentation for you and added some additional explanation to make Django as concise and accessible as possible.

EXPANDING OUR POLL APP FUNCTIONALITY

Let's add a few more views to polls/views.py. These views are slightly different because they take an additional argument (question_id):

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

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

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

Wire these new views into the **polls/urls.py** file by adding the following path() calls:

```
from django.urls import path, include
from . import views

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')
]
```

Start your server and then take a look in your browser, at "/polls/34/". It'll run the `detail()` method and display whatever ID you provide in the URL. Try "/polls/34/results/" and "/polls/34/vote/" too – these will display the placeholder

results and voting pages.

When somebody requests a page from your website – say, "/polls/34/" - Django will load the `<mysite>.urls` Python module because it is pointed to by the `ROOT_URLCONF` setting. It finds the variable named `urlpatterns` and traverses the URL patterns in order. After finding the match at `polls/`, it strips off the matching text ("polls/") and sends the remaining text ("34/") to the `polls.urls` URLconf for further processing. There it matches `<int:question_id>/`, resulting in a call to the `detail()` view as shown below:

```
detail(request=<HttpRequest object>, question_id='34')
```

The `question_id='34'` part comes from `<int:question_id>` in `polls.urls`. Angle brackets are used to capture a value from the URL and send it as an argument to the view function. The angle brackets can also contain a converter type (in this case `int`) which specifies what type the value should be converted to. See a list of other default path converters **here**. The name specified in the angle brackets (e.g. `question_id`) defines the name that will be used to identify the value.

USING REGULAR EXPRESSIONS

Validating data manually can be quite a nightmare. For example, in HTML form validation, you need to sometimes check for a valid form for an email (for example, to prevent users from typing in 'hello' as their email address). The general rule for email addresses goes something like:

1. A set of letters, symbols, and numbers (excluding the '@' sign),
2. Followed by the '@' sign,
3. Followed by at least one letter, then a '.', followed by at least one letter.

This is a simple form of email validation, but even then this could be difficult to do manually in Python. A **regular expression** (regex or RE) provides an easy way to automatically define rules for this. This allows you to ensure that data matches a certain pattern.

At times you may want to ensure that the data in a URL matches certain patterns before you process it. For example, if you are expecting the URL to contain a value that describes a year, you would expect that that value would be made up of four whole numbers (e.g. 2018). You can use a regex to match patterns.

A regex specifies a set of strings that matches it. The functions in the `Regex` module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string).

We will look at a few regex basics by considering an example of a URL pattern that contains regular expressions. Let's start with a simple one. Let's say we want to set up a URL in the form of */blog/<year>*, so that we can pass in the year that a specific blog post happened.

To define the pattern for a year, we simply expect 4 digits. To do that in Django, you would set this up in **polls/urls.py**:

```
re_path(r'^blog/(?P<year>[0-9]{4})/$',views.blog_detail)
```

Breaking down what this means, let's start with the `r` that appears before the string. This is simply a way to tell Python that there are no escape characters in the string (such as `\n`).

The first character in the regex is `^`. This simply translates to 'starts with'. This character applies to the string '`blog/`'. This therefore translates to 'starts with blog/'.

The next part has quite a few weird-looking characters. Let's start with `?P<year>`. This is just something to say that this part of the URL gets passed into the `year` parameter in `blog_detail`.

Next is the actual validation part. `[0-9]` specifies the characters that are allowed to match the pattern. This just means that only digits are allowed. `{4}` specifies the number of characters to match the pattern.

Now, let's move on to something a bit more in-depth. Consider the example below:

```
re_path(
r'^blog/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<slug>[\w-]+)/$'
,views.blog_detail),
```

This is similar to the example we started with, but goes further. See if you can figure this out from the string itself. You'll notice some different symbols for the `slug` pattern. The `\w` pattern specifies 'words' (or just any letter, basically). This also includes underscores. The `-` at the end of it means letters plus the `-` character. Finally, the `+` symbol at the end of the `slug` pattern means 'one or more'.

The dollar '$' symbol means '*end of string*'.

In the example above, the URL "**http://127.0.0.1:8000/blog/20181/02/hello**" would not match the path specified because the value "20181" does not consist of exactly 4 digits.

On the other hand, the URL "**http://127.0.0.1:8000/blog/2018/02/hello**" would match the URL pattern.

For a full reference on regex, have a look at **these Python docs**. You may also find this **cheat sheet** for regex useful.

To see what the expression `(?P<slug>[\w-]+)` does, copy and paste the expression (including the parenthesis) **here**.

Notice that if we want to use regular expressions in `urlpatterns`, we use `re_path` instead of `path`. To use `re_path`, you will need to import `re_path` as shown below:

```
from django.urls import re_path
```

For more information about using regular expressions in **urls.py**, see **here**.

CREATE A TEMPLATE

At the moment, our *poll* view is hard-coded and it returns a simple `HttpResponse`. Let's create a template. Inside your **polls/** directory, create a new folder called **templates/** and inside **templates/** create a new folder called **polls/**. Inside **templates/polls/**, create an HTML file called **poll.html**. Place the following Django templating code inside the HTML file:

```
{% 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 %}
```

Okay, let's update our index view in **polls/views.py** to render the template:

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

# Create your views here.

def index(request):
```

```
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {'latest_question_list': latest_question_list}
    return render(request, "polls/poll.html", context)
```

The code loads the template called **polls/poll.html** and passes it a context. The context is a dictionary that maps template variable names to Python objects.

RAISING A 404 ERROR

Now, let's tackle the question detail view – the page that displays the question text for a given poll. Here's the view:

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

New concept here: the view renders an **HTTP 404** error if a question with the requested ID doesn't exist. To use get_object_or_404, you will need to add the following import statement:

```
from django.shortcuts import get_object_or_404
```

Note that we've rendered **detail.html** in the detail view, but **detail.html** doesn't exist. Create detail.html and place the following code in it:

```
<h1> {{question.question_text}}</h1>
<ul>
    {% for choice in question.choice_set.all %}
    <li> {{choice.choice_text}}</li>
    {% endfor %}
</ul>
```

The template system uses **dot-lookup syntax** to access variable attributes. In the example of `{{ question.question_text }}`, first Django does a dictionary lookup on the object `question`. Failing that, it tries an attribute lookup – which works, in this case. If attribute lookup has failed, it would've tried a list-index lookup.

Method-calling happens in the `{% for %}` loop: `question.choice_set.all` is interpreted as the Python code `question.choice_set.all()`, which returns an iterable of `Choice` objects and is suitable for use in the `{% for %}` tag.

REMOVING HARDCODED URLS IN TEMPLATES

Remember, when we wrote the link to a question in the **polls/poll.html** template, the link was partially hardcoded like this:

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

The problem with this hardcoded, tightly-coupled approach is that it becomes challenging to change URLs on projects with many templates. However, since you defined the name argument in the `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:

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

The way this works is by looking up the URL definition as specified in the `polls.urls` module. You can see exactly where the URL name of `detail` is defined below:

```
path('<int:question_id>', views.detail, name='detail')
```

NAMESPACING URL NAMES

Our project has two apps - `polls` and `blog`. How does Django differentiate the URL names between them? For example, the `polls` app has a `detail` view and so does our `blog` (assuming you've started the `polls` app in the same project). How does one ensure that Django knows which app view to create for a URL when using the `{% url %}` template tag? By adding namespaces to your `urlconf`.

In the **polls/urls.py** file, add an `app_name` to set the application namespace:

```
from django.urls import path, include, re_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/poll.html** template from:

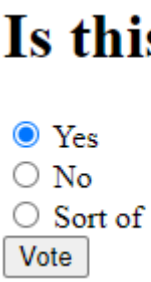
```
<a href = "{% url 'detail' question.id %}/">
```

to:

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

WRITE A SIMPLE FORM

Let's update our poll detail template (**polls/detail.html**) so that the template contains an HTML `<form>` element. We want the template to result in a page that looks similar to the one shown below:



Here the question is displayed in a `<h1>` element. The template also contains a form that contains:

- Radio buttons. Each choice that is associated with that particular question is then displayed using a radio button in a form.
- A button that allows the user to submit their choice.

Notice how this template is created:

```
<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>
```

Let's analyse some lines of the template language used to create this template:

`<form action="{% url 'polls:vote' question.id %}" method="post">`	The form's method is set to "**post**". The HTTP post method is used to send data to the server. The URL that will be used to send data to the server is specified in the action attribute of the form.
`{% csrf_token %}`	The **csrf_token** should be added to all POST forms that are targeted at internal URLs. This provides protection against Cross Site Request Forgery. **See here** for more information about this security risk and built-in middleware to protect against it.

`{% for choice in question.choice_set.all %}`
...
`{% endfor %}`
	Here we use a **for** loop to add a radio button for each choice associated with a specific question in our database. The name of each radio button is **choice** and the **id** attribute for each radio button is the **Choice**'s ID (from the database).

With the template above, when somebody selects one of the radio buttons and submits the form, it'll POST the data "**choice=id**" where **id** is the ID of the selected choice.

Now, let's create a Django view that handles the submitted data and does something with it. Add the following to **polls/views.py**:

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponse, HttpResponseRedirect
from django.urls import reverse
from .models import Question, Choice

# . . .

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,))
        )
```

Let's consider some of the code above that is new to this task.

- `request.POST`: Consider `request.POST`; this is a dictionary-like object that lets you access submitted data by key name. We use the key name `choice` (`request.POST['choice']`) which returns the ID of the selected choice. `request.POST` values are always strings. If the ID of the choice isn't found, a `KeyError` (which is an error which occurs when a mapping (dictionary) key is not found in the set of existing keys) is thrown.
- `HttpResponseRedirect`: here, `HttpResponseRedirect` is used rather than `HttpResponse`. `HttpResponseRedirect` takes the URL to which the user will be redirected as an argument.
- `reverse`: We are using the `reverse` function. This function takes the name of the view and returns a string value that represents the actual URL. For example, the `reverse` function above could return: `/polls/3/results/` where the "3" is the value of `question.id`.

This redirected URL will then call the 'results' view to display the final page. Let's write this results view:

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

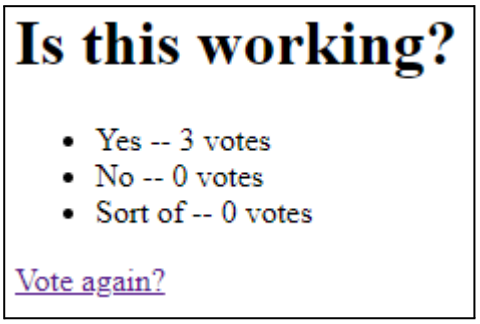
Now, create a **polls/results.html** template and your app should be complete:

```
<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>
```

After you vote, you should be directed to the results view and see a page similar to the one shown below:



Instructions

Feel free at any point to refer back to the material if you get stuck. Remember that if you require more assistance, we are always here to help you!

Compulsory Task

- Make sure you've set up your poll app correctly.
- Login to the admin panel and add a few questions for users to vote on. (Note: if you have created the polls app in a separate project from your blog app, you will have to reconfigure the admin site).
- Work through this task to complete your polls application.
- Customise your templates using **bootstrap**.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.

REFERENCE(S)

Django documentation. (n.d.). Django Software Foundation. Retrieved October 18, 2022, from **https://docs.djangoproject.com/en/4.1/**