


Написание вашего первого приложения Django, часть 4

Это учебное пособие начинается с того места, где остановился [Учебное пособие 3](#). Мы продолжаем приложение для веб-опросов и сосредоточимся на обработке форм и сокращении нашего кода.

 **Где получить помощь:**

Если у вас возникли проблемы с прохождением этого учебника, перейдите в раздел «[Получение справки](#)» в разделе часто задаваемых вопросов.

Напишите минимальную форму

Давайте обновим наш шаблон подробностей опроса (`<polls/detail.html>`) из последнего учебника, чтобы шаблон содержал элемент HTML `<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>
```

Краткое описание:

- The above template displays a radio button for each question choice. The **value** of each radio button is the associated question choice's ID. The **name** of each radio button is **"choice"**. That means, when somebody selects one of the radio buttons and submits the form, it'll send the POST data **choice=#** where # is the ID of the selected choice. This is the basic concept of HTML forms.
- We set the form's **action** to `{% url 'polls:vote' question.id %}`, and we set **method="post"**. Using **method="post"** (as opposed to **method="get"**) is very important, because the act of submitting this form will alter data server-side. Whenever you create a form that alters data server-side, use **method="post"**. This tip isn't specific to Django; it's good Web development practice in general.
- forloop.counter** indicates how many times the **for** tag has gone through its loop
- Since we're creating a POST form (which can have the effect of modifying data), we need to worry about Cross Site Request Forgeries. Thankfully, you don't have to worry too hard, because Django comes with a helpful system for protecting against it. In short, all POST forms that are targeted at internal URLs should use the `{% csrf_token %}` template tag.

Теперь давайте создадим представление Django, которое обрабатывает отправленные данные и что-то с ними делает. Помните, что в [Учебнике 3](#) мы создали URLconf для приложения опросов, которое включает в себя следующую строку:

```
polls/urls.py

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

Мы также создали фиктивную реализацию функции **vote()**. Давайте создадим реальную версию. Добавьте следующее в **polls/views.py**:

```
polls/views.py

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

Этот код включает в себя несколько вещей, которые мы еще не рассмотрели в этом учебном пособии:

- request.POST** is a dictionary-like object that lets you access submitted data by key name. In this case, **request.POST['choice']** returns the ID of the selected choice, as a string. **request.POST** values are always strings.
- Note that Django also provides **request.GET** for accessing GET data in the same way – but we're explicitly using **request.POST** in our code, to ensure that data is only altered via a POST call.
- request.POST['choice']** will raise **KeyError** if **choice** wasn't provided in POST data. The above code checks for **KeyError** and redisplay the question form with an error message if **choice** isn't given.
- After incrementing the choice count, the code returns an **HttpResponseRedirect** rather than a normal **HttpResponse**. **HttpResponseRedirect** takes a single argument: the URL to which the user will be redirected (see the following point for how we construct the URL in this case).

As the Python comment above points out, you should always return an **HttpResponseRedirect** after successfully dealing with POST data. This tip isn't specific to Django; it's good Web development practice in general.

- We are using the **reverse()** function in the **HttpResponseRedirect** constructor in this example. This function helps avoid having to hardcode a URL in the view function. It is given the name of the view that we want to pass control to and the variable portion of the URL pattern that points to that view. In this case, using the URLconf we set up in [Tutorial 3](#), this **reverse()** call will return a string like

```
'/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.

As mentioned in [Tutorial 3](#), **request** is an **HttpRequest** object. For more on **HttpRequest** objects, see the [request and response documentation](#).

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

```
polls/views.py

from django.shortcuts import get_object_or_404, render

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

This is almost exactly the same as the **detail()** view from [Tutorial 3](#). The only difference is the template name. We'll fix this redundancy later.


Теперь создайте шаблон **polls/results.html**:

```
polls/templates/polls/results.html<h1>{{ question.question_text }}</h1>

<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }}
vote{{ choice.votes|pluralize }}</li>
{% endfor %}
</ul>

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

Now, go to **/polls/1/** in your browser and vote in the question. You should see a results page that gets updated each time you vote. If you submit the form without having chosen a choice, you should see the error message.

 **Примечание**

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

Это называется **состоянием гонки**. Если вы заинтересованы, вы можете прочитать «[Избегание условий гонки с помощью F\(\)](#)», чтобы узнать, как вы можете решить эту проблему.

Используйте общие представления: Меньше кода лучше

The **detail()** (from [Tutorial 3](#)) and **results()** views are very short – and, as mentioned above, redundant. The **index()** view, which displays a list of polls, is similar.

Эти представления представляют собой распространенный случай базовой веб-разработки: получение данных из базы данных по параметру, переданному в URL, загрузка шаблона и возврат визуализированного шаблона. Поскольку это так распространено, Django предоставляет ярлык, называемый системой «общие представления».

Общие представления абстрактные общие шаблоны до такой степени, что вам даже не нужно писать код Python для написания приложения.

Давайте конвертируем наше приложение для голосования в использование общей системы представлений, чтобы мы могли удалить кучу нашего собственного кода. Нам придется предпринять несколько шагов, чтобы сделать преобразование. Мы будем:

- Преобразуйте URLconf.
- Удалите некоторые из старых, ненужных видов.
- Введите новые представления на основе общих представлений Django.

Читайте дальше для получения подробной информации.

 **Почему перетасовка кода?**

Как правило, при написании приложения Django вы оцениваете, подходят ли общие представления для вашей проблемы, и будете использовать их с самого начала, а не рефакторинг кода на полпути. Но это руководство намеренно было сосредоточено на написании мнений «трудным путем» до сих пор, чтобы сосредоточиться на основных концепциях.

Вы должны знать базовую математику, прежде чем начать использовать калькулятор.

Изменить URLconf

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

```
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'),
]
```

Note that the name of the matched pattern in the path strings of the second and third patterns has changed from **<question_id>** to **<pk>**.

Изменить представления

Next, we're going to remove our old **index**, **detail**, and **results** views and use Django's generic views instead. To do so, open the **polls/views.py** file and change it like so:

```
polls/views.py

from django.http import HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
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'

def vote(request, question_id):
    ... # same as above, no changes needed.
```

We're using two generic views here: **ListView** and **DetailView**. Respectively, those two views abstract the concepts of "display a list of objects" and "display a detail page for a particular type of object."

- Каждое общее представление должно знать, по какой модели оно будет действовать. Это предоставляется с использованием атрибута **model**.
- The **DetailView** generic view expects the primary key value captured from the URL to be called **"pk"**, so we've changed **question_id** to **pk** for the generic views.

By default, the **DetailView** generic view uses a template called **<app name>/<model name>_detail.html**. In our case, it would use the template **"polls/question_detail.html"**. The **template_name** attribute is used to tell Django to use a specific template name instead of the autogenerated default template name. We also specify the **template_name** for the **results** list view – this ensures that the results view and the detail view have a different appearance when rendered, even though they're both a **DetailView** behind the scenes.

Similarly, the **ListView** generic view uses a default template called **<app name>/<model name>_list.html**; we use **template_name** to tell **ListView** to use our existing **"polls/index.html"** template.


In previous parts of the tutorial, the templates have been provided with a context that contains the **question** and **latest_question_list** context variables. For **DetailView** the **question** variable is provided automatically – since we're using a Django model (**Question**), Django is able to determine an appropriate name for the context variable. To override this we provide the **context_object_name** attribute, specifying that we want to use **latest_question_list** instead. As an alternative approach, you could change your templates to match the new default context variables – but it's a lot easier to tell Django to use the variable you want.

Запустите сервер и используйте новое приложение для опроса на основе общих представлений.

Для получения полной информации об общих видах см. [документацию по общим видам](#).

Когда вам удобно с формами и общими представлениями, прочитайте [часть 5 этого учебника](#), чтобы узнать о тестировании нашего приложения для опросов.

Поддержите Django!

 LawnStret Sydney пожертвовала Django Software Foundation для поддержки разработки Django. Пожертвуйте сегодня!

Содержание

- Написание вашего первого приложения Django, часть 4
 - Написать минимальную форму
 - Использовать общие представления: Лучшее меньше кода
 - Изменить URLconf
 - Изменить виды

Просмотр

- Предыдущий: Написание вашего первого приложения Django, часть 3
- Далее: Написание вашего первого приложения Django, часть 5
- Содержание
- Общий индекс
- Индекс модуля Python

Вы находитесь здесь:

- Документация Django 3.2
 - Введение
 - Написание вашего первого приложения Django, часть 4

Получение помощи

Часто задаваемые вопросы
Попробуйте FAQ – в нем есть ответы на многие распространенные вопросы.

Индекс, индекс модуля или оглавление
Удобно при поиске конкретной информации.

список рассылки django-users
Найдите информацию в архивах списка рассылки django-users или оставьте вопрос.

#django IRC-канал
Задайте вопрос в IRC-канале #django или выполните поиск в журналах IRC, чтобы узнать, задавался ли он раньше.

Трекер билетов
Сообщите об ошибках с документацией Django или Django в нашем трекере билетов.

Скачать:

Автономный (Django 3.2): [HTML](#) | [PDF](#) | [ePub](#)
Предоставлено [Read the Docs](#).