BIT2008A/ITEC2000A: Multimedia Data Management
Fall 2022

Lab 6

Django

Checkpoint 1. In this session, we build a sample project using Django and show how it evolves. The instructions file has several checkpoints and what you implement by the end of each checkpoint should work. First, make your python environment ready. If you are using the docker container, extract docker.zip into a folder. Specify the Django version in requirements.txt using $Django >= 4.0$. Then run *docker compose up* to launch the server. Now, open *localhost:8000* in your browser to make sure that the Django server works fine. Open a terminal with Conda available. If you are using docker, go to containers and open a terminal from *web* as shown in Fig. 1.

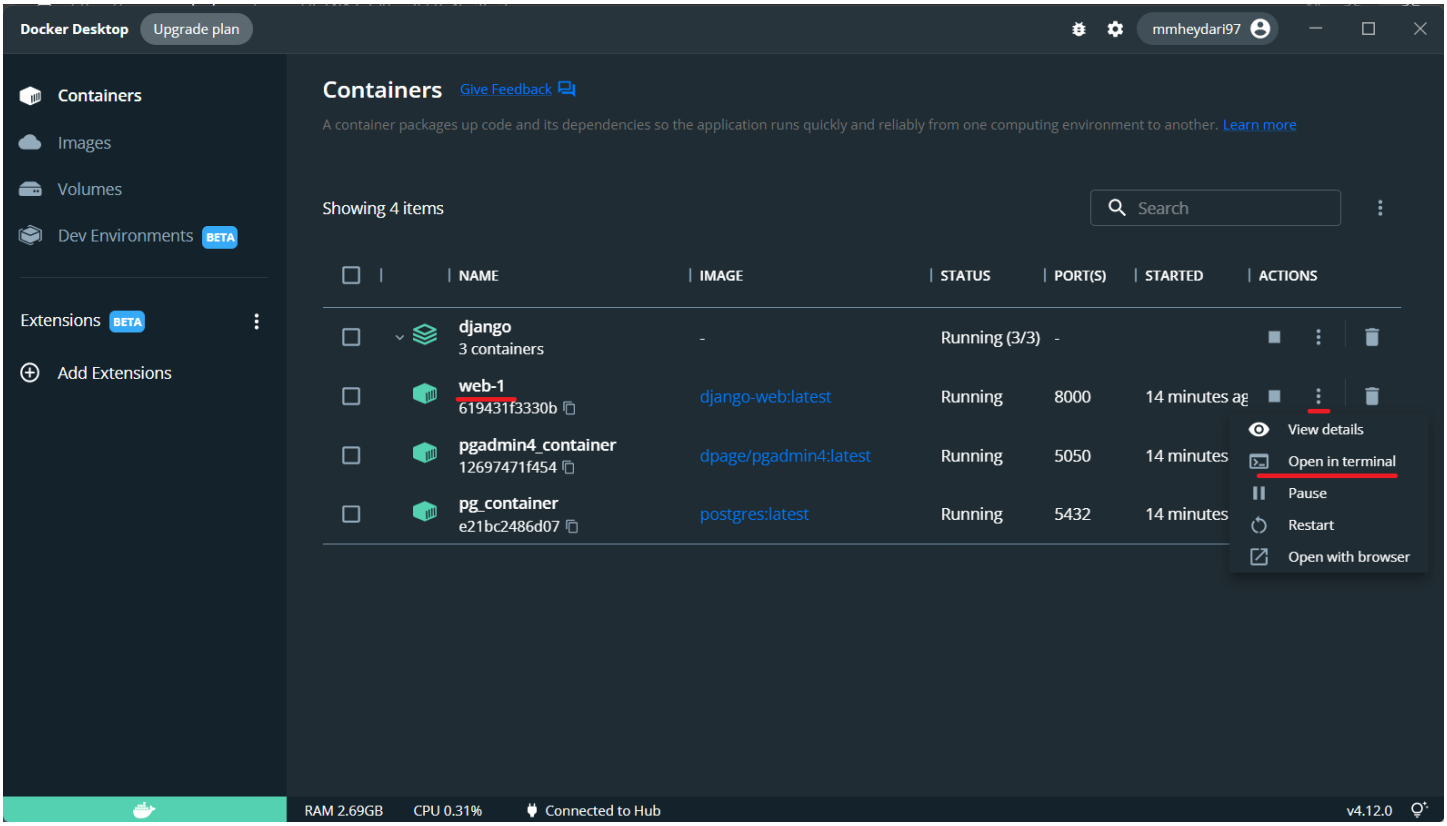

Figure 1. open terminal to run Django commands.

Then run the command below in the terminal that you opened.

```
python manage.py startapp polls
```

After making sure that the new app is created correctly, open the file *polls/views.py*. We implement the functionalities of our application with views. Let's put the following Python code to create a simple function-based view.

```
from django.http import HttpResponse


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

Now, we need to define the routing of our project to use our views. We assign different views to the paths that the users may enter in their address bar. It makes it more modular and scalable to handle the routing of each application with a separate urls.py in each application directory. To do so, create *urls.py* for the polls application and configure it as follows.

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

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

This code maps empty path strings to our index view. We need to determine when to use the views of polls application after all. Consequently, we should link the paths of the polls application to the main urls.py of the **project**. Change your project urls.py in a way that includes polls/urls.py routing as follows.

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

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

By using *include*, we tell Django admin to resolve paths starting with *polls* with the rules that we defined in our polls app urls.py. Go to *http://localhost:8000/polls/* in your browser, and you should see the text "Hello, world. You're at the polls index.", which you defined in the index view.

Checkpoint 2. First off, run the command below inside your Django terminal to make sure that the database connection is working fine.

```
python manage.py migrate
```

To implement more complex functionalities, we need to fetch data from our database. As the MTV design pattern suggests, we code in models.py to interact with our database through Django. Create models for polls application by modifying *models.py* as below.

```
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 make Django detect our applications, we should add them to **INSTALLED_APPS** in project *settings.py*. You can simply include the name *polls*, or use another syntax which is more professional. Add another item to ISTALLED_APPS to make it look like this:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'polls.apps.PollsConfig',
]
```

Now Django knows to include the polls app. Let's run another command to map our models to the database:

```
python manage.py makemigrations polls
python manage.py migrate
```

One of the advantages of abstraction available in Python and other object-oriented languages is that we do not have to worry about every little detail unless we need to change it. There are many behaviors handled for our models just by inheriting from *models.Model*, however, we want to change the textual representation of our models now. It's important to add __str__() methods to your models, not only for your own convenience but also because objects' representations are used throughout Django's automatically-generated admin panel. Add following code to change those representations:

```
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
```

Let's also add a custom functionality to our Question model:

```
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)
```

Now, create a admin for your website to run polls. To do so, run the following commands in the terminal:

```
python manage.py createsuperuser
Username: admin
Email address: admin@admin.com
Password: admin
Password (again): admin
Bypass password validation and create user anyway? [y/N]: y
```

We also need to register our models to be included in the admin panel. To do this, open the *polls/admin.py* file, and edit it to look like this:

```
from django.contrib import admin
from .models import Question, Choice

admin.site.register(Question)
admin.site.register(Choice)
```

Now, try logging in admin panel available in *localhost:8000/admin*, with the superuser account you created in the previous step. You can add some instances using the Django admin page as shown in Fig 2.

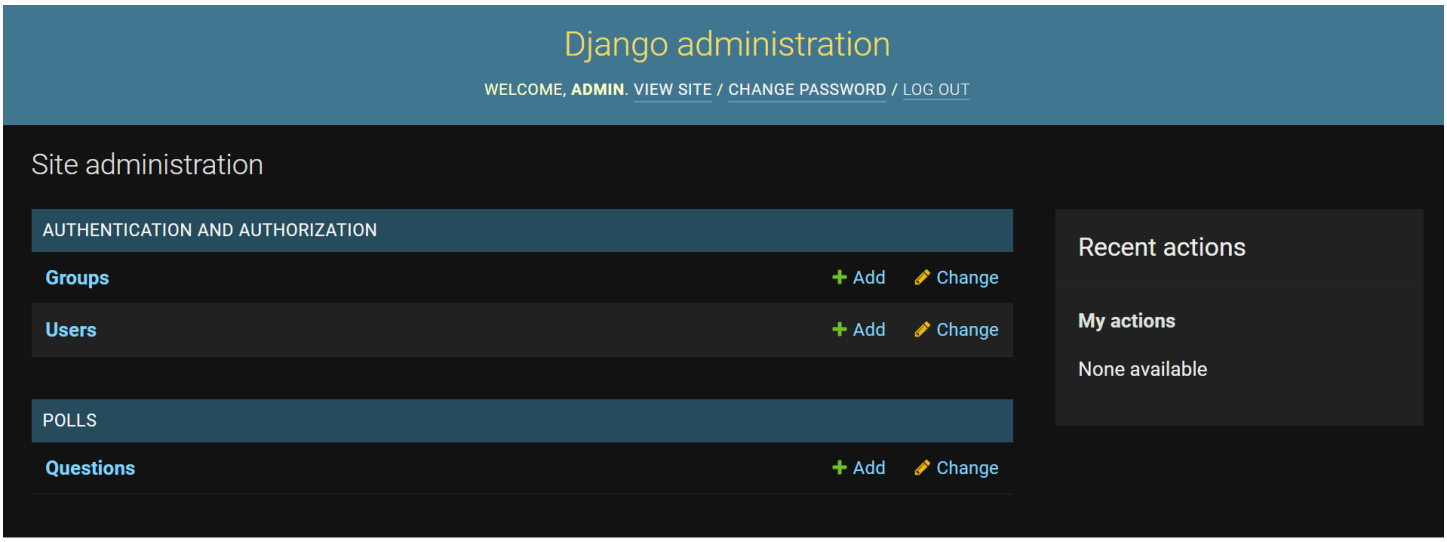

FIGURE 2. this is how the admin panel looks.

Checkpoint 3. First, let's add a few more views to *polls/views.py*. These views are slightly different because they take an argument:

```
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)
```

Again, we have to create the routing to use those views. Your *polls/urls.py* should look like this:

```
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'),
]
```

Assume that we want to show the latest polls of our website on the index page. Following code enables us to do that easily:

```
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
```

The problem with this implementation is that we mixed the representation side of the project with the functionality. To avoid that you should create *polls/templates/polls/index.html* file to develop a template. Write following lines to 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 %}
```

Then, update your app's *view.py* to use the template that you created. We use *render* to save some lines of code:

```
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)
```

Now, let's create a view to address question details – the page that displays the question text for a given poll. We use another shortcut called *get_object_or_404()* to create the view below:

```
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})
```

Then, you can handle the template by creating *./polls/templates/polls/detail.html* and fill it with:

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

The current problem with our index.html is that we included hard-coded URLs in the file:

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

Since we name the URL mappings, we can use them to make our templates loosely coupled. In other words, if want to change a file, we do not like other files to be affected and make the maintenance of our website costy. As a result change index.html like this:

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

Then assign the name that you used in *polls/urls.py*:

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

This way, if we want to change the routing of our application in urls.py, we do not have to worry about the URLs affected in our templates. We also want to be more explicit about the namespacing of our project. We already did that by creating *polls/templates/polls* that makes us specify the application name every time we want to name a template. Now change the urls.py to look like this:

```
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 to the one below to point at the namespaced detail view:

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

Checkpoint 4. We need to create a minimal form to be able to perform voting in the polls. Let's update our poll detail template like below:

```
<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
<fieldset>
    <legend><h1>{{ question.question_text }}</h1></legend>
    {% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}
    {% 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 %}
</fieldset>
<input type="submit" value="Vote">
</form>
```

Now, let's implement a real voting functionality by adding its view:

```
from django.http import HttpResponse, 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,)))
```

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

```
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})
```

There is some redundancy if you compare the code for the results view and the detail view. We will fix that soon. For now, let's create the *resutls.html* template:

```
<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>
```

To avoid redundancy, there are some generic views, available as classes to cover the typical functionalities that we need most. Let's update our views as shown below:

```
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.
```

When we use class based views, we need to update our URL matchings, too:

```
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'),
]
```

Checkpoint 5. Finally, let's do some minimal modifications to the looks of our project. Let's create *polls/static/polls/style.css* file to change the visual attributes of our templates:

```
li a {
    color: green;
}
```

To load this file in our *index.html* template, we should add following lines:

```
{% load static %}

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

References. This tutorial was made using Django official tutorial. You can find more details about each checkpoint or explore other parts of the tutorial if you wish.

- checkpoint 1
- checkpoint 2
- checkpoint 3
- checkpoint 4
- checkpoint 5