Learning Django

Source: https://docs.djangoproject.com/en/3.2/intro/tutorial01/

POLLING APP
DJANGO

—

May 27

Setting up : Virtual Environment

Using python command ***venv*** to create a virtual environment

- Navigate to working directory in terminal
- Run command ***python venv -m ~~environment name~~***
    - A directory named ~~environment name~~ will be created
    - Containing all the required python files

Activate the new environment

- Navigate to directory ~~environment name~~ in terminal
- Run command ***source bin/activate***

Deactivate the environment

- Run command ***deactivate***

Install django : Using pip

- On active environment
    - Run command ***python -m pip install Django***
- Check version
    - Run command ***python -m django --version*** (mine is 3.2.3)

—

May 28

Create a project: mysite

This is name of my complete project

- On active environment and at active environment directory
    - Run Command ***<u>django-admin startproject mysite</u>***
    - A directory "mysite" is created, it has
        - - manage.py : for administering our project
        - - mysite : directory to contain our project
            - - __init__.py : consider this as python package
            - - settings.py : configuration for this project
            - - urls.py : table of content for our project
            - - asgi.py : end point for asgi server
            - - wsgi.py : end point for wsgi server

Running server:

- At outer mysite directory
    - Run command ***<u>python manage.py runserver</u>***
    - Open localhost:8000 (default port) on web browser
    - You should see a rocket :)

Creating an app : polls

A project contains many apps and configuration files for a particular website, an app can be in multiple projects.

- At outer mysite directory
    - Run command ***<u>python manage.py startapp polls</u>***
    - "Polls" directory is created, it has
        - - __init__.py
        - - admin.py

        - apps.py
        - migrations
            - __init__.py
        - models.py
        - tests.py
        - views.py

Writing our first view

Consider writing a webpage

- In poll/views.py write

```
polls > views.py > ...
from django.shortcuts import render
from django.http import HttpResponse


# Create your views here.

def index(request):
    return HttpResponse("Namaste Doston")

```

Link this function to a url by writing a local URLconf file

- Create a file "polls/urls.py" and write

```
views.py 2 ●    urls.py 1 ●
polls > urls.py > ...
from django.urls import path
from . import views

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

```

Link this app's URLconf to ROOT URLconf using "*include()*"

- In mysite/urls.py write

```
The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/3.2/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import include, path

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

- Open localhost:8000/polls/

***index()*(polls/views.py) → urls.py(polls/) → urls.py(mysite/)**

Database setup

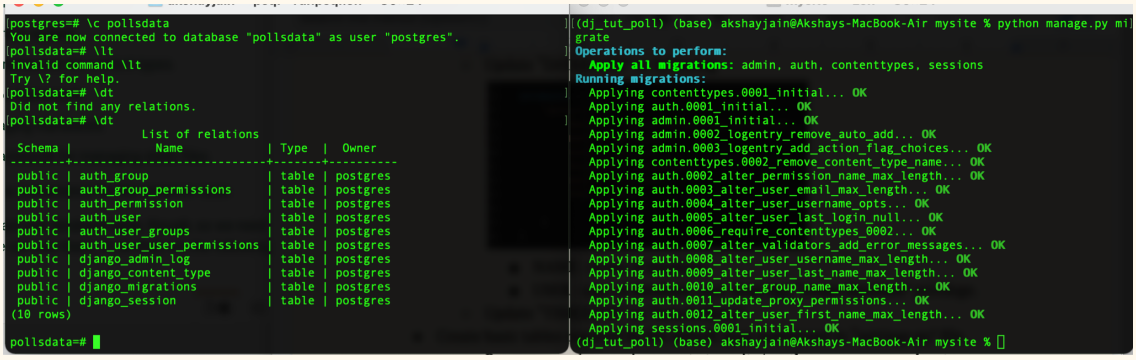
Using PostgreSQL.

- Installing PSQL binding
  - Upgrade pip using command ***pip install -U pip***
  - Install psycopg2 using command ***pip install psycopg2***
- Create a PSQL Database
  - Using PSQL shell command ***CREATE DATABASE pollsdata;***
  - Check using ***\l***

- Configure settings.py file
    - Update “DATABASES” to following

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'pollsdata',
        'USER': 'postgres',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '5433',

    }
}
```

        - NAME : name of database
        - USER : user of database, user must have “create database” privilege.
    - Update “TIMEZONE” to “Asia/Kolkata”
- Create basic tables required by the default apps mentioned in “settings.py” file.
    - Run command on environment at mysite/ ***python manage.py migrate***

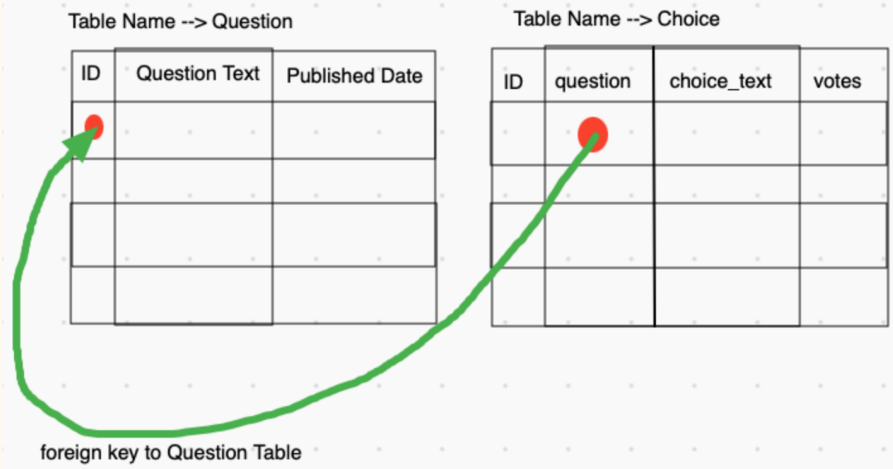


—

May 29

Creating Models: Our DATABASE Layout

Designing our tables

- Visual representation
  - 

- In “polls/models.py” write

```
polls > models.py > ...
from django.db import models

# Create your models here.
class Question(models.Model):
    question_text = models.CharField('Question text', max_length = 200)
    pub_date = models.DateTimeField('Published Date')
```

```
class Choice(models.Model):

    question = models.ForeignKey(Question, on_delete=models.CASCADE)

    choice_text = models.CharField(max_length=200)

    votes = models.IntegerField(default = 0)
```

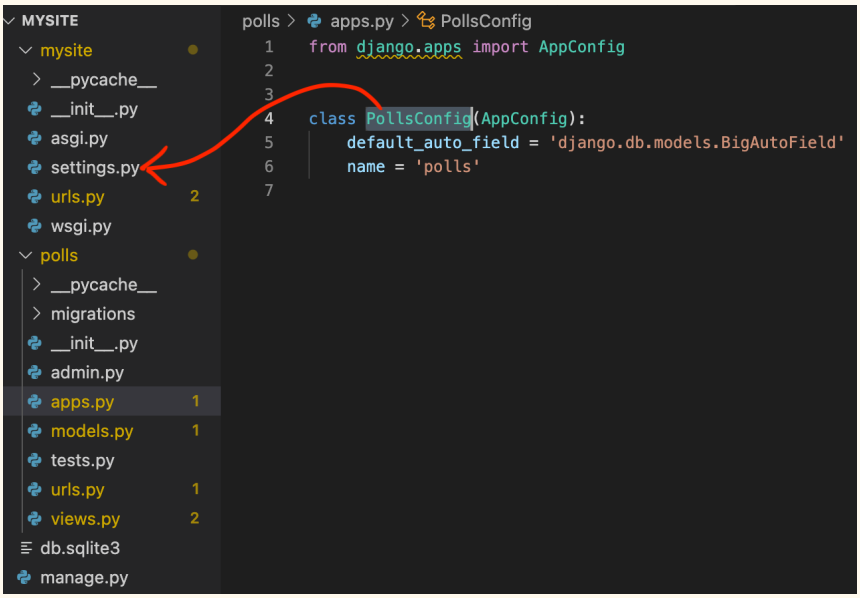
  - models.{name of Field type}({Optional human readable name of table column})
    - IntegerField()
    - CharField(max_length = {integer value})
    - ForeignKey({other model class name})

Installing app : Connecting our “polls” app to our project “mysite”

We need our settings file (project) to point to our app config file (app)

- In setting.py append
  - INSTALLED_APPS = [polls.app.PollsConfig]

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



App migrations : pushing changes to database

models.py file has our table structure but only django knows it.

- Process overview
  - models.py (our tables skeleton) → (convert structures to local files) → (view the sql django generated to create tables) → (run that code to apply changes)

- Process
  - Run command ***python manage.py makemigration polls***

```
  File "/Users/akshayjain/Desktop/AJ work/python and apps/django/polling_app/dj_
tut_poll/mysite/polls/models.py", line 10, in Choice
    choice_text = models.models.CharField(max_length=200)
AttributeError: module 'django.db.models' has no attribute 'models'
(dj_tut_poll) (base) akshayjain@Akshays-MacBook-Air mysite % python manage.py ma
kemigrations polls
Migrations for 'polls':
  polls/migrations/0001_initial.py
    - Create model Question
    - Create model Choice
(dj_tut_poll) (base) akshayjain@Akshays-MacBook-Air mysite %
```

```
polls
  __pycache__
  migrations
    __pycache__
    __init__.py
    0001_initial.py
  __init__.py
```

  - Run command ***python manage.py sqlmigrate polls 0001***

```
(dj_tut_poll) (base) akshayjain@Akshays-MacBook-Air mysite % python manage.py sq
lmigrate polls 0001
BEGIN;
--
-- Create model Question
--
CREATE TABLE "polls_question" ("id" bigserial NOT NULL PRIMARY KEY, "question_te
xt" varchar(200) NOT NULL, "pub_date" timestamp with time zone NOT NULL);
--
-- Create model Choice
--
CREATE TABLE "polls_choice" ("id" bigserial NOT NULL PRIMARY KEY, "choice_text"
varchar(200) NOT NULL, "votes" integer NOT NULL, "question_id" bigint NOT NULL);
ALTER TABLE "polls_choice" ADD CONSTRAINT "polls_choice_question_id_c5b4b260_fk_
polls_question_id" FOREIGN KEY ("question_id") REFERENCES "polls_question" ("id"
) DEFERRABLE INITIALLY DEFERRED;
CREATE INDEX "polls_choice_question_id_c5b4b260" ON "polls_choice" ("question_id
");
COMMIT;
(dj_tut_poll) (base) akshayjain@Akshays-MacBook-Air mysite %
```

  - Run command ***python manage.py migrate***

```
(dj_tut_poll) (base) akshayjain@Akshays-MacBook-Air mysite % python manage.py mi
grate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, polls, sessions
Running migrations:
  Applying polls.0001_initial... OK
```

—

May 31

Playing with API

Using django APIs to control our Database

- Start python shell
  - Run ***python manage.py shell***
- Steps to add an entry in table
  - ***from polls.models import Question, Choice***
  - ***from django.utils import timezone***

    - ***q = Question(question_text = "What's up?", pub_date= timezone.now())***
    - ***q.save()***
- Steps to overwrite
    - ***q.question_text = "What is up?"***
    - ***q.save()***
- View all objects (record as an object)
    - ***Question.objects.all()*** : define ***__str__*** method in class to get readable text
- Adding custom methods to class
    -

```
polls > models.py > Choice > __str__
import datetime
from django.db import models
from django.utils import timezone
# Create your models here.
class Question(models.Model):
    question_text = models.CharField('Question text', max_length = 200)
    pub_date = models.DateTimeField('Published Date')

    def __str__(self):
        return self.question_text

    def was_published_recently(self):
        #retrun false if pub_date is more than 1 day old
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default = 0)

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

- Some handy methods
    - Question.objects.filter(question_text__startswith='What')
    - q = Question.objects.get(pk=1) ; q stores the object
    - q.was_published_recently ; custom method of class Question
- Double underscore technique
    - Current_year = timezone.now().year
    - Question.objects.get(pub_date__year = current_year); using __ to access year
        - __ to separate relationships and go as many levels deep.

- Adding choices to corresponding questions
    - View all choices; run
        - q = Question.objects.get(pk=1) ; q stores the object
        - q.choice_set.all()
    - Adding a choice
        - q.choice_set.create(choice_text= ‘Not much’, votes= 0)
        - q.choice_set.create(choice_text= ‘The sky’, votes= 0)
    - Adding and holding a choice
        - c = q.choice_set.create(choice_text= ‘just hacking’ , votes= 0)
        - View question for this choice
            - c.question
    - Using filter and delete
        - c = q.choice_set.filter(choice_text__startwith=”just”)
        - c.delete()

Create an admin user

Admin user to control admin dashboard

- Run command ***python manage.py createsuperuser***
- Run server and login to localhost:8000/admin

Access our app tables from admin panel

We need to register our app with admin

- In **admin.py** add
    - Import our models
    - admin.site.register(“model/class name”)
    -

```
polls > admin.py
1  from django.contrib import admin
2
3  # Register your models here.
4  from .models import Question
5  admin.site.register(Question)
6
```

—

Jun 1

Our app Views

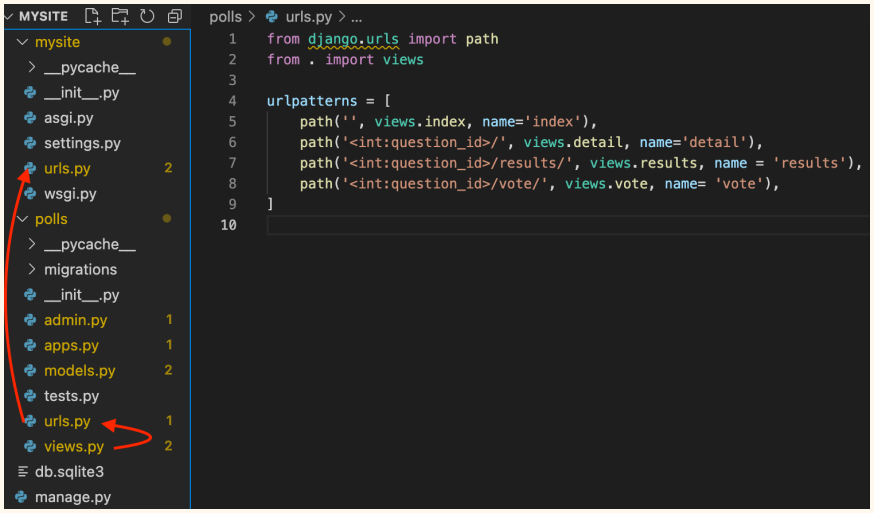
- Question “index” page - displays the latest few question
- Question “detail” page - displays a question text, with no result but with a form to vote.
- Question “result” page - displays results for a particular question.
- Vote action - handles voting for a particular choice in a particular question.

Testing view

- Write following views in **polls/views.py**

```
polls > views.py > vote
from polls.models import Question
from django.shortcuts import render
from django.http import HttpResponse


# Create your views here.

def index(request):
    return HttpResponse("Namaste Doston")

def detail(request, question_id):
    return HttpResponse(f'You are looking at question number {question_id}')

def results(request, question_id):
    return HttpResponse(f'You are looking at result of question number {question_id}')

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

- Connect these views to local **urls.py** file by writing



Real view

Writing a real view for app that query data from our database

- In **polls/views.py** write

```python
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)
```

Real view using templates

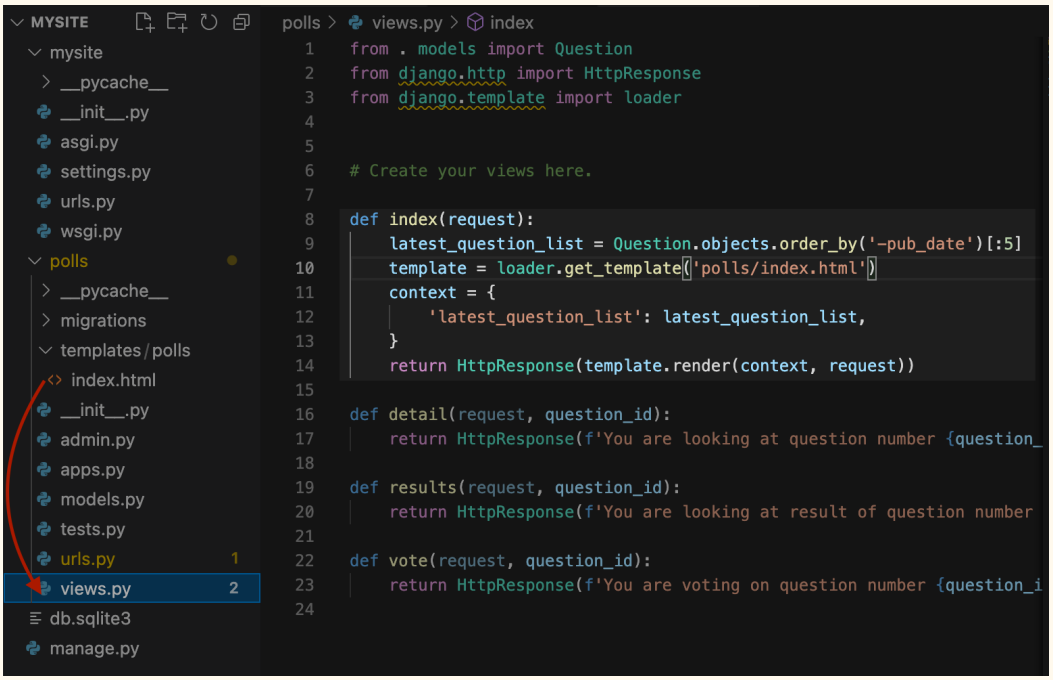
- Create a directory **templates/polls** in **mysite/polls/.**
    - By doing this django is able to tell the difference between 2 same name templates in different apps, and templates can be accessed using **polls/index.html.**
- Create an **index.html** at **mysite/polls/templates/polls/** and write *html+jinja2*

polls > templates > polls > index.html

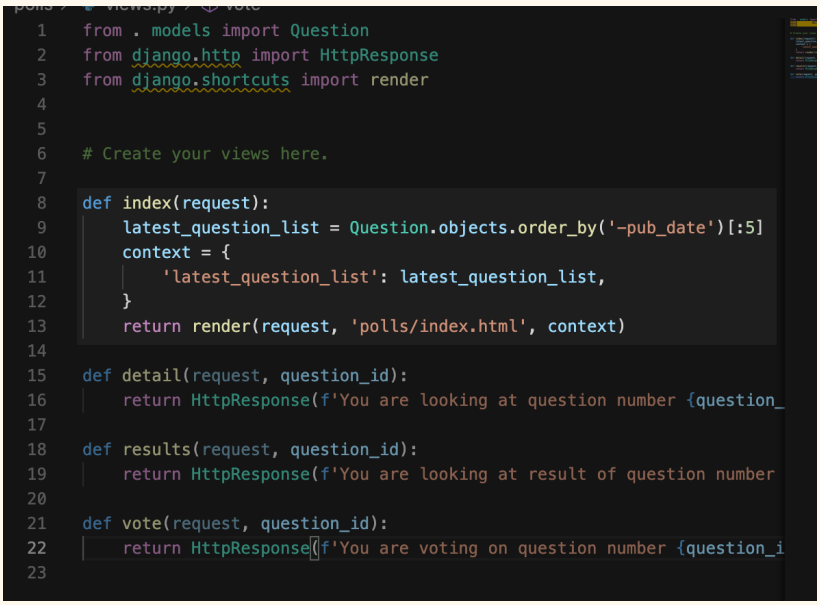
```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>template page</title>
    </head>
    <body>
        <p>This is base template page</p>
        {% if latest_question_list %}
            <ul>
            {% for question in latest_question_list %}
                <li>
                    <a href= "/polls/{{question.id}}"> {{question.question_text}}
                    </a>
                </li>
            {% endfor %}
            </ul>
        {% else %}
            <p> No polls are available. </p>
        {% endif %}
    </body>
</html>
```

    - *Latest_question_list* variable will be provided by django

- Link this **index.html** to our **views.py.**
    - ***from django.template import loader***



- Alternate way to link using only **render** shortcut
    - ***from django.shortcuts import render***



—

Jun 2

Raising 404 Error

Raise a 404 error if object is not found in DB

- Modify detail views to query question table, raise 404 if not found

```
from django.http import HttpResponse, Http404


def detail(request, question_id):
    try:
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("Question in not in DB")
    return render(request, 'polls/details.html', {'question':question})
```

- Create a **polls/details.html**

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>
            Detail page
        </title>
    </head>
    <body>
    {{ question }}
    </body>
</html>
```

Shortcut for 404

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

Displaying **question** and **choices** in **detail.html** page by using "." notation.

```
<!DOCTYPE html>
<html>
   <head>
       <meta charset="utf-8">
       <title>
           Detail page
       </title>
   </head>
   <body>
   <h1> {{question.question_text}} </h1>
   <ul>
       {%for choice in question.choice_set.all%}
       <li> {{choice}} </li>
       {% endfor %}
   </ul>
   </body>
</html>
```

Removing hardcoding url in **index.html** using **url** keyword to reference **urls.py** file.

- BEFORE

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

- AFTER (utilising name = 'detail' in urls.py)

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

What if more than 1 **detail** view exists, like in another app, how will django decide which urlCONF file to read?

- Using **Namespace**
  - In local **urls.py** file insert
    - ```
      app_name = 'polls'
      ```
  - Modify template to include app name
    - ```
      <a href= "{% url 'polls:detail' question.id %}">
      {{question.question_text}}
      ```
    -

—

Jun 3

Creating our detail page form

Its must show

- Question text
- Choices associated to that question in radio button form

Implementation

```
<body>
   <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" value = {{ choice.id }} id =
"choice{{ forloop.counter }}" >
       <label > {{ choice.choice_text }} </label>
       <br>
```

```
    {% endfor %}
    <input type = "submit" value = "Vote">
    </form>
    </body>
```

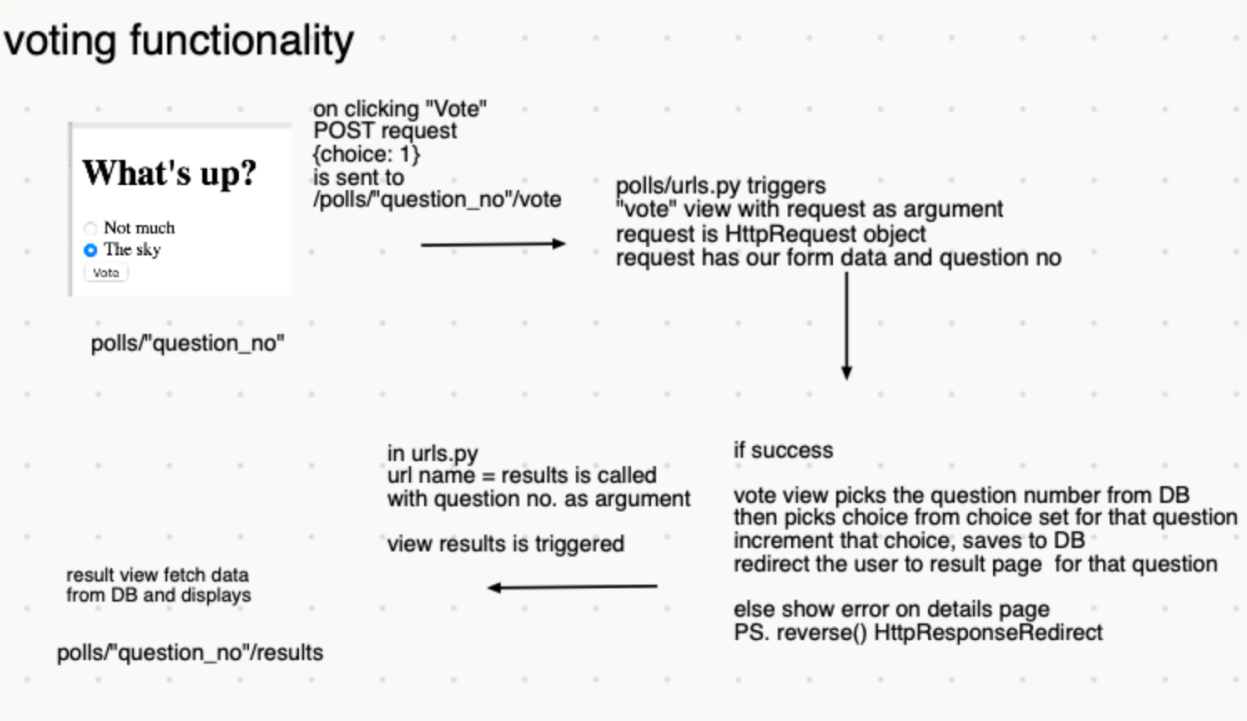
- On submit, a post request is sent to **/polls/"choice_no"/vote** url
- To prevent Cross site request forgery, we include csrf token
- For current question populate all choices
  - Name of all radio buttons is same as only 1 will be sent to backend
  - Values = ID of that choice in DB. eg {'choice': '2'} | ({ name: value})
  - ID : "choice + [1,2,3....]" for DOM based addressing (css, js)
  - For each choice display its text using <label>

—

Jun 4

Implementing voting functionality

Steps



- In **views.py** import

```
from . models import Question, Choice

from django.http import HttpResponse, Http404, HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls import reverse
```

- In view **vote** write

```
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'])
    # first -> dictionary error, second = django error for data not in model
Choice
    except (KeyError, Choice.DoesNotExist):
        # redisplay the question's voting form with error
        return render(request, 'polls/details.html', {'question': question,
'error_message' : "You didn't select a choice"})

    else:
        selected_choice.votes += 1
        selected_choice.save()
        # always return HttpResponseRedirect after successfully
        # dealing with POST data. This prevents data from
        #being posted twice, if user hits back button.
    return HttpResponseRedirect(reverse('polls:results', args = (question.id,)))
```

- PS : **reverse** takes urlpattern name and execute that view (results view) with arguments. args must be iterable. “,” is important at the end.
- Writing **results** view

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

- Creating **results.html** template

```
<!DOCTYPE html>


<html>
   <head>
       <meta charset = "utf-8">
       <title>Result</title>
   </head>
   <body>
       <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>
   </body>
</html>
```

Issue with our implementation:

- Our vote view picks a voting number from DB, increments then saves. If there were more than 1 voting person at a time, "**race condition**" occurs and we don't get the correct voting number.
    - Solution :
        - Using **F()** its lets database increment the value rather than django

```
from django.db.models import F
```

        - BEFORE

```
selected_choice.votes += 1
```

        - AFTER

```
selected_choice.votes = F('votes') + 1
```

—

Jun 13

Using Generic views

Exploring **ListView, DetailView**

Modifying polls/urls.py to use class based views

```
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'),
]
```

- These views expect the variable of name **pk** to query the database

Modify our view.py file to replace **index, detail, and result function**

```
class IndexView(generic.ListView):
   template_name = 'polls/index.html'
   context_object_name = 'latest_question_list'

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

class DetailView(generic.DetailView):
   model = Question
   template_name = 'polls/details.html'

class ResultsView(generic.DetailView):
```

```
    model = Question
    template_name = 'polls/results.html'
```

- In IndexView telling django the our templates expects this name *latest_question_list,* which can be automatically fetched by method ***get_queryset***.
- All generic views expect a model name to display results from and a template_name to display.
- Luckily we have used ***question*** as a variable for **detail, results** templates because django's default variable corresponding to ***model = Question*** is **question.**

—

Jun 14

Testing MODELS

TDD: test driven development i.e. writing test before we jump on development.

Bug: Question model method ***was_published_recently()*** return ***True*** if the question is 1 day old, but also ***True*** if it's from anytime in the future.

Steps to validate bug

- In python shell : ***python manage.py shell***
- ***From django.utils import timezone***
- ***From django.models import Question***
- ***Future_question = Question(question_text= "future question?", pub_date = timezone.now() + datetime.deltatime(days=30))***
- ***future_question.was_published_recently()*** : Return ***True***

Question is created 30 days in the future and it should return False.

Writing an automated test for this bug

In polls/test.py write

```
import datetime

from django.test import TestCase
from django.utils import timezone
```

```
from .models import Question
# Create your tests here.

class QuestionModelTests(TestCase):

    def test_was_published_recently_with_future_question(self):
        """was_published_recently() returns False for question whose pub_date is in future"""

        time = timezone.now() + datetime.timedelta(days = 30)
        future_question = Question(pub_date = time)
        self.assertIs(future_question.was_published_recently(), False)
```

- Here we created a class which is a subclass/childclass of TestCase.
- Our class has a method ***test_was_published_recently_with_future_question()***
    - Name of method must start with word **test**
    - It creates a future question object and checks using ***self.assertIs(method, result)***

Running test case

- In terminal run ***python manage.py test polls***
    - It creates a temporary DB for test cases and destroys it later.
    - It points where our code breaks.

Fixing the bug

In models.py file modify the method ***was_published_recently()*** to check if **pub_date** is in the past.

Adding more Edge Cases

In polls/test.py add

```
    def test_was_published_recently_with_past_question(self):
        """was_published_recently() returns Flase for question whose pub_date is older than 1 day"""

        time = timezone.now() - datetime.timedelta(days = 1, seconds=1)
        old_question = Question(pub_date = time)
        self.assertIs(old_question.was_published_recently(), False)

    def test_was_published_recently_with_recent_question(self):
```

```
        """was_published_recently returns True for question whose pub_date is within the last day"""
        time = timezone.now() - datetime.timedelta(hours = 23 , minutes=59, seconds=59)
        recent_question = Question(pub_date = time)
        self.assertIs(recent_question.was_published_recently(), True)
```

Testing VIEWS

Problem: Our view's don't validate what questions we are sending to templates to display to users. Eg. User is able to see following:

- Future questions, as the Index view returns the latest 5 questions without considering if that question is supposed to be seen by the user, as future questions should be invisible.
- Questions which have no choices associated with them.

Solution

Using ***python manage.py shell*** to understand and demonstrate **Client** Object

- *from django.test.utils import setup_test_environment*
    - This lets us see extra content of webpage, eg **response.context (**template variables**)**
    - No separate DB is created.
- *setup_test_environment()*
- *from django.test import Client*
    - Importing client class to create a test client to act like a user
- *from django.urls import reverse*
    - Convert url name to its URL
- *client = Client()*
- *response = client.get('/')*
    - Requesting root page; thus Not found
- *response.status_code*
    - 404
- *response = client.get(reverse("polls:index"))*
- *response.content*
    - Complete html of webpage
- *response.context['latest_question_list']*
    - response.context is a dictionary; here we try to see what view has passed to our templates.

- <QuerySet[<Question: What’s up?>]>

Implementation for IndexView

Modify **polls/view.py IndexView get_queryset** method and ***from django.utils import timezone***

```
def get_queryset(self):

        """Return the last 5 published questions(not including those set to be
published in future)"""

        return Question.objects.filter(pub_date__lte =
timezone.now()).order_by('-pub_date')[:5]
```

Testing implementation

In **polls/test.py** write

```
def createQuestion(question_text, days):
    """Creates a question with given 'question_text' and with offset with current
time. (positve days = future, negatice days = past)"""
    time = timezone.now() + datetime.timedelta(days = days)
    return Question.objects.create(question_text= question_text, pub_date = time)

class QuestionIndexViewTests(TestCase):

    def test_no_question(self):
        """if No question is published, an appropriate message is displayed on
index page"""
        response = self.client.get(reverse("polls:index"))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'],[])

    def test_past_question(self):
        """Question with pub_date in the past are displayed on index page"""
        question = createQuestion("Past Question", -30)
        response = self.client.get(reverse("polls:index"))
```

```
        self.assertQuerysetEqual(response.context['latest_question_list'], [question])

    def test_future_question(self):
        """Question with pub_date in the future is not displayed on the index page and an appropriate message is displayed"""
        question = createQuestion("Future Question", 30)
        response = self.client.get(reverse("polls:index"))
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_future_question_and_past_question(self):
        """Even both future and past question exist, only past question is diasplayed"""
        question = createQuestion("Past Question", -5)
        createQuestion("Future Question", 30)
        response = self.client.get(reverse("polls:index"))
        self.assertQuerysetEqual(response.context['latest_question_list'], [question])

    def test_two_past_question(self):
        """The index page will display multiple question"""
        question_one = createQuestion("Question One?", -5)
        question_two = createQuestion("Question Two?", -10)
        response = self.client.get(reverse("polls:index"))
        self.assertQuerysetEqual(response.context['latest_question_list'], [question_one, question_two])
```

- *createQuestion* function to create a question with provided argument to save code repetition
- *response = self.client.get(reverse("polls:index"))*
    - Using client method to get response
- assert functions on *response*
    - *self.assertEqual(response.status_code, 200)* -> checks if both are equal
    - *self.assertContains(response, "No polls are available.")* -> checks if it contains
    - *self.assertQuerysetEqual(response.context['latest_question_list'], [~~arugments~~])*
        - *response.context* is a dictionary

        - *response.context['latest_question_list']* shows the list of objects our template has received.
- Each test case creates its own question, there is no sharing of questions.

—

Jun 15

Implementation for DetailView

If someone can guess a url for our future question, then they can access it on detail view, as detail view does not filter questions based on the pub_date.

We need to add a **get_queryset()** method to filter our question

```
class DetailView(generic.DetailView):
    model = Question
    template_name = 'polls/details.html'

    def get_queryset(self):
        """Return the Question only its not in future"""
        return Question.objects.filter(pub_date__lte = timezone.now())
```

—

Jun 16

Testing Implementation for DetailView

In **polls/test.py** write

```
class QuestionDetailView(TestCase):
    def test_future_question(self):
        """The detail view of a question with pub_date in future return 404 not
found"""
        future_question = createQuestion("Future Question", 5)
        url = reverse("polls:detail", args=(future_question.id,))
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)
```

```
    def test_past_question(self):
        """The detail view of question with pub_date in past is displays the question text"""
        past_question = createQuestion("Past Question?", -5)
        url = reverse("polls:detail", args=(past_question.id,))
        response = self.client.get(url)
        self.assertContains(response, past_question.question_text)
```

Implementation for Result View

If someone can guess a url for our future question, then they can access it on the result view, as the result view does not filter questions based on the pub_date.

We need to add **get_queryset()** set method to filter our question in **ResultView**

```
    def get_queryset(self):
        """Return the Question only its not in future"""
        return Question.objects.filter(pub_date__lte = timezone.now())
```

Testing implementation in Result view

In **polls/test.py** writing following tests

```
class QuestionResultView(TestCase):
    def test_future_question(self):
        """Result View of question with pub_date in future return 404 not found"""
        future_question = createQuestion("Future Question", 5)
        url = reverse("polls:results", args=(future_question.id,))
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)

    def test_past_question(self):
```

```
        """Result view of question with pub_date in past displays the question text"""
        past_question = createQuestion("Past Question?", -5)
        url = reverse("polls:results", args=(past_question.id,))
        response = self.client.get(url)
        self.assertContains(response, past_question.question_text)
```

Adding static files

All files like (CSS, JS, images)

- Follow the same process for directories as we did for templates
  - Create css at **polls/static/polls/style.ccs**
    - And access using **polls/style.css**
  - Add images at **polls/static/polls/images**
    - And access using polls/images/background.gif
- At top of **index.html** add

```
{% load static %}
```

  - This will load our static files
- In **<head>** add

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

  - *static* keyword will put an absolute url
- In **style.css** add

```
li a {
   color : green;
}

body {
   background: white url("images/background.gif") no-repeat;
   background-size: 500px;
}
```

—

Jun 17

Modifying Admin form

We can access the admin form for a model at **localhost/admin -> model**

- Changing sequence of **fields** in Question form.

```
class QuestionAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question_text']

admin.site.register(Question, QuestionAdmin)
```

  - Passing a class as a parameter to ***admin.site.register(Question, QuestionAdmin)***
- Adding **FieldsSet** to group some fields into sets.

```
class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,                  {'fields' : ['question_text']}),
        ('Date Information',    {'fields' : ['pub_date']})
        ]

admin.site.register(Question, QuestionAdmin)
```

- Adding choices to our form

```
from .models import Question, Choice
admin.site.register(Choice)
```

- Adding choices to be visible in same page as question

```
class ChoiceInLine(admin.StackedInline):
    model = Choice
    extra = 3
class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,                  {'fields' : ['question_text']}),
        ('Date Information',    {'fields' : ['pub_date'], 'classes' :
['collapse']})
        ]
    inlines = [ChoiceInLine]
```

    - Making date time field hide/show
    - Making 3 extra choices placeholder in form of stack
- Improve visibility by using table type view

```
class ChoiceInLine(admin.TabularInline):
```

Modifying change list page

The page where we see a list of all questions, by default **str()** of each object is visible.

- In class **QuestionAdmin(admin.ModelAdmin)** add

```
list_display = ('question_text', 'pub_date', 'was_published_recently')
```

    - Name of fields.
    - We can add a method also, column name will be the method name without "_"
- Adding a custom name and look to the column using decorator over that method. In **poll/models.py** add

```
from django.contrib import admin
@admin.display(
        boolean = True,
        ordering = 'pub_date',
        description = 'Published recently?')
    def was_published_recently(self):
```

    - This renames the column name and shows "X" instead of False
- Adding Filter functionality wrt to date
    - In **Polls/admin.py QuestionAdmin** add
    - `list_filter = ['pub_date']`
- Adding search functionality
    - In **Polls/admin.py QuestionAdmin** add
    - `search_fields = ['question_text']`

Customizing admin look and feel

Create a directory **templates** in project directory, same directory which contains **manage.py.**

- In **setting.py -> templates** add

```
        'DIRS': [BASE_DIR / 'templates'],
```

    - To look for templates in base directory
- Copy source **base_site.html** from `django/contrib/admin/templates` to **templates/admin/**
- Modify the template to use your custom name

```
<h1 id="site-name"><a href="{% url 'admin:index' %}">My Polling app
Administration</a></h1>
```