CHAPTER 2

THE POLLS APP

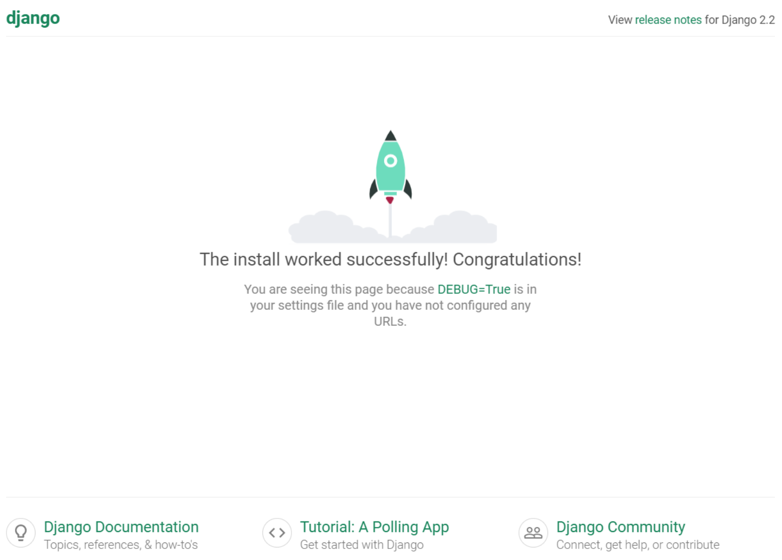
2.1 Creating a Django project

We will create a Django project called polls on PythonAnywhere. PythonAnywhere allows us to host web applications. It has an integrated bash console.

First, we create the virtual environment `polls`. This can be done using `conda create -n polls`. Then, we create the Django project. This can be done using `django-admin startproject mysite`. In the `mysite` directory, this creates the files:

- `manage.py`,
- `mysite/__init__.py`,
- `mysite/settings.py`,
- `mysite/urls.py` and
- `mysite/wsgi.py`.

We can view the created site using the command `py manage.py runserver` in the `mysite` directory. The site looks like this right now:


django
View release notes for Django 2.2
Django Documentation
Topics, references, & how-to's
Tutorial: A Polling App
Get started with Django
Django Community
Connect, get help, or contribute

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Next, we make the app `polls`. We can do this using `django-admin startapp polls`. This creates many files in the `mysite/polls` directory, which are:

- `admin.py`,

- `apps.py`,
- `models.py`,
- `views.py` and
- `tests.py`

2.2 Creating a view

In the file `views.py`, we can write views. For example, we can make this our views file to render the index page:

```python
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello World")
```

A view receives `HttpRequest` and returns `HttpResponse` objects. There are many ways in returning the Http response object- the one above is the simplest way.

At this point, there is no url mapping to this view. So, we now visit the `urls.py` in the `mysite` folder to make use of the `polls` app:

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

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

Now, we can make use of `polls` app in the project. Next, we create another `urls.py` files in the `polls` app and register the `index` view:

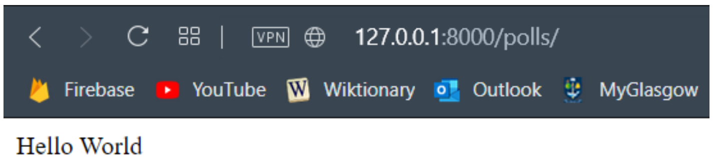
```python
from django.urls import path
from . import views

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

Note that the `index` view now maps to `polls/`. When the url gets checked at `mysite`, we get redirected to `polls` with the empty string- this maps to the `index` view in the `polls` app. It is also possible to omit the final /.

The `path` function takes 3 parameters:

- the route, which is a URL string. We go through the urlpatterns list until we find the first matching route;
- the view, which is the function that django calls (when the route matches) that takes in an `HttpRequest` as an argument;
- the name, which allows the URL to be referenced from elsewhere unambiguously (reverse lookup). This allows for easy alternation of the URLs.

After the files are created and updated, we get the following webpage when we navigate to `polls`:



Hello World

2.3 Creating models

To initalise database setup, we run the command `py manage.py migrate`. In the file `polls/models.py`, we create the models `Question` and `Choice`:

```
from django.db import models

class Question(models.Model):
    question_text=models.CharField(max_length=200)
    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)
```

A model we create has to extend the model `models.Model`. Here, we have a 1-to-many relation between `Question` and `Choice`. In that case, we have the field `question` (one) in the class `Choice` (many). Moreover, we set the `on_delete` parameter to be `models.CASCADE`- this means that when a `Question` object gets deleted, all the `Option` objects associated with that `Question` also get deleted.

For us to make use of models in the `polls` app, we need to update the variable `INSTALLED_APPS` in `settings.py` to include also `'polls'`. This includes the `polls` app in future database migrations. Now, we use the command `py manage.py makemigrations polls` to create these migrations. Then, we use the command `py manage.py migrate` to migrate the models.

Next, we can manually create some data using the command `py manage.py shell`. Then, we can type the following in the command line and see their output:

```
>>> from polls.models import Question, Choice
>>> from django.utils import timezone
>>> ## print all the questions
>>> Question.objects.all()
<QuerySet []>
>>> ## add a question
>>> q = Question(question_text="What's new?", date=timezone.now())
>>> q.save()
>>> ## print id (the automatic primary key) and question text
>>> q.id
1
>>> q.question_text
"What's new"
>>> ## update the question text
>>> q.question_text = "What's up?"
>>> q.save()
>>> ## print all the questions
>>> Question.objects.all()
<QuerySet [<Question: Question object (1)>]>
```

To provide a better string representation of `Question` and `Answer`, we can update our `models.py` file with `__str__` function:

```
from django.db import models

class Question(models.Model):
    question_text=models.CharField(max_length=200)
    date=models.DateTimeField("date published")

    def __str__(self):
        return self.question_text

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
```

Now, we run `py manage.py shell` with the following in the command line:

```
>>> from polls.models import Question, Choice
>>> from django.utils import timezone
>>> ## print all the questions
>>> Question.objects.all()
<QuerySet [<Question: "What's up?">]>
```

Now, we define another function `was_published_recently` to the `Question` model:

```
from datetime import timedelta

def was_published_recently(self):
    return self.date >= timezone.now() - timedelta(days=1)
```

It returns `true` if the object was added in the last day.

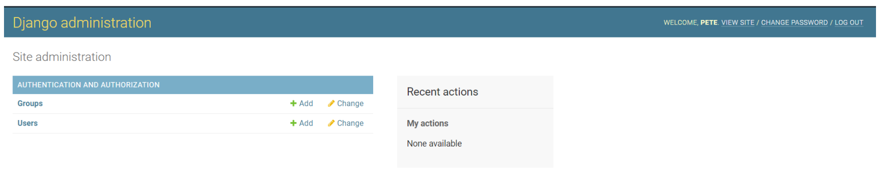
We continue manipulating the model from the shell:

```
>>> from polls.models import Question, Choice
>>> from django.utils import timezone
>>> ## print all the questions
>>> Question.objects.all()
<QuerySet [<Question: "What's up?">]>
>>> ## print all the questions with id 1
>>> Question.objects.filter(id=1)
<QuerySet [<Question: "What's up?">]>
>>> ## print all the questions with id 2
>>> Question.objects.filter(id=2)
<QuerySet []>
>>> ## print all the questions starting with "What"
>>> Question.objects.filter(question_text__startswith='What')
<QuerySet [<Question: "What's up?">]>
>>> ## print the single question published this year
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
<Question: "What's up?">
>>> ## print the single question published in 2020
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=2020)
polls.models.Question.DoesNotExist
>>> ## print the single object with id 1
>>> q = Question.objects.get(pk=1)
>>> q
<Question: "What's up?">
>>> ## check whether the question was published recently
>>> q.was_published_recently()
True
>>> ## print all the choices with question q
>>> q.choice_set.all()
<QuerySet: []>
```

```
>>> ## create choices associated with question q
>>> q.choice_set.create(choice_text="Not much", votes=0)
<Choice: "Not much">
>>> q.choice_set.create(choice_text="The sky")
<Choice: "The sky">
>>> c = Choice(question=q, choice_text="Just hacking again")
>>> c.save()
>>> c
<Choice: "Just hacking again">
>>> c.question
<Question: "What's up?">
>>> ## print all the choices with question q
>>> q.choice_set.all()
<QuerySet: [<Choice: "Not much">, <Choice: "The sky">,
    <Choice: "Just hacking again">]>
>>> q.choice_set.count()
3
>>> ## print all the choices whose question.pub_date.year is current_year
>>> Choice.objects.filter(question__pub_date__year=current_year)
<QuerySet: [<Choice: "Not much">, <Choice: "The sky">,
    <Choice: "Just hacking again">]>
>>> ## remove the "Just hacking again" choice
>>> c = q.choice_set.filter(choice_text__startswith="Just hacking")
>>> c
<QuerySet [<Choice: "Just hacking again">]>
>>> c.delete()
>>> ## print all the choices whose question.pub_date.year is current_year
>>> Choice.objects.filter(question__pub_date__year=current_year)
<QuerySet: [<Choice: "Not much">, <Choice: "The sky">]>
```

Note that the function `Question.objects.filter` returns a `QuerySet` object with multiple (or none) values matching the query, while the function `Question.objects.get` returns a single `Question` that satisfies the query. If there are more than one (or none) that satisfy this property, then it raises an error.

We can create a population script to populate the database to create database content much faster. Another way of creating the content is using the django admin site. We can create a superuser using `py manage.py createsuperuser` and providing username/passsword. Then, we can navigate to the admin page. We can log in with the credentials we used previously and add/edit the models present.

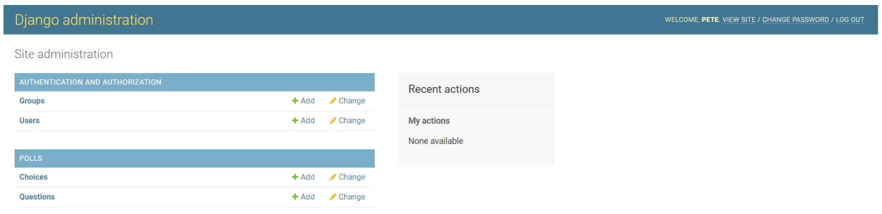


The models `Question` and `Choice` aren't present. To add them, we change the file `admin.py`:

```
from django.contrib import admin
```

```python
from .models import Question, Choice

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

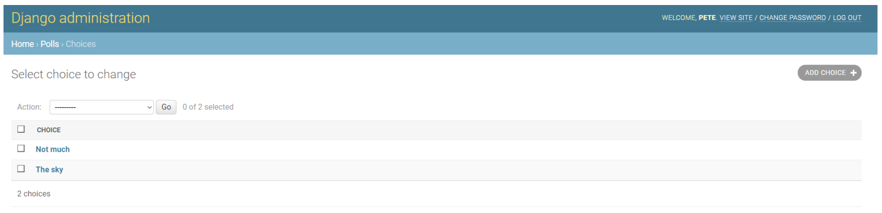
Then, after we reload the webpage, we get the two models:



AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

POLLS		
Choices	+ Add	Change
Questions	+ Add	Change

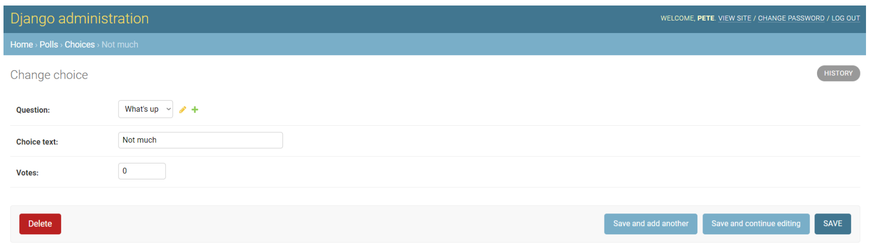
We can browse the objects we have created by clicking on the relevant model. For example, the Choice objects we had are:



☐	CHOICE
☐	Not much
☐	The sky

2 choices

We can also edit/create another Choice object. For example, if we wanted to edit "Not much", we would get:


Django administration
WELCOME, PETE. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Polls › Choices › Not much
Change choice
HISTORY
Delete
Save and add another
Save and continue editing
SAVE

Question:	What's up
Choice text:	Not much
Votes:	0

2.4 Creating more views

We now add more views to `views.py`

```python
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello World")

def detail(request, question_id):
    response = "You're looking at question %s"
    return HttpResponse(response % 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):
    response = "You're voting on question %s"
    return HttpResponse(response % question_id)
```

This is not the final look of the page- we are just creating the pages.

Then, we update the file `urls.py` in the `polls` app:

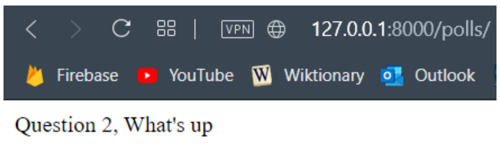
```python
from django.urls import path
from . import views

urlpatterns = [
    ## /polls/
    path('',views.index,name='index'),
    ## e.g. /polls/3
    path('<int:question_id>/',views.detail,name='detail'),
    ## e.g. /polls/3/results
    path('<int:question_id>/results/',views.results,name='results'
    )
    ## e.g. /polls/3/vote
    path('<int:question_id>/vote/',views.vote,name='vote')
]
```

We make use of `<int:question_id>` to allow any integer to pass in. This integer also becomes the second parameter of the view function.

Next, we are now going to change the `index` view so that it returns the latest 3 questions:

```python
from .models import Question

def index(request):
    ## find the latest 3 questions
    latest_questions = Question.objects.order_by('-date')[:3]
    ## present it in a clear string and return it
    output = ", ".join([q.question_text for q in latest_question])
    return HttpResponse(output)
```

We will now add a second question `Question 2`. Then, the index page looks like:


127.0.0.1:8000/polls/

Question 2, What's up

The issue with rendering views like this is that they are all hard-coded within the view functions. They should only deal with logic; a template should handle any views. So, we create the template `mysite/templates/polls/index.html`:

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

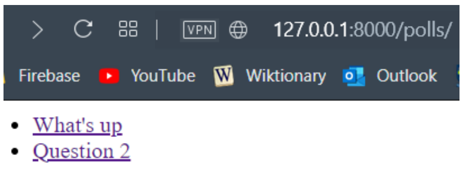
A Django template file is composed of html tags, as well as variables (e.g. `{{ variable }}`) and logic (e.g. `{% logic %}`). To register templates, we need to add to the `settings.py` file the statement

```
TEMPLATE_DIR = os.path.join(BASE_DIR, 'templates')
```

and update the variable `TEMPLATES` accordingly. Finally, we update the function `index` in `views.py`:

```
from django.shortcuts import render

def index(request):
    ## find the latest 3 questions
    latest_questions = Question.objects.order_by('date')[:3]
    context_dict = {'latest_questions_list': latest_questions}
    return render(request, 'polls/index.html', context_dict)
```

The render function returns an HttpResponse object that uses the template provided along with the context dictionary. Now, the index page looks like:



- What's up
- Question 2

At this point, clicking on any of the question shows a very basic details page. Now, we will use the `question_id` to render content specific to that question. However, it is possible that there is no question with the id provided. In that case, we need to redirect to 404 page. So, we update the details view as follows:

```
from django.http import Http404

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

There is a shortcut to doing this:

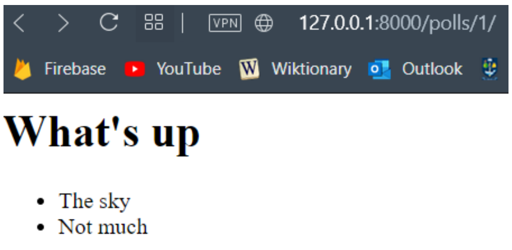
```
from django.shortcuts import get_object_or_404

```

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

Next, we create the template `detail.html`:

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

Then, the details page looks like:



What's up

- The sky
- Not much

When we wrote the link to a question in `index` template, the link was partially hardcoded. This becomes challenging to change URLs on projects with many templates. We resolve this using the name argument in `urls.py` together with the `{% url %}` tag. So, the `index.html` template becomes:

```
1  {% if latest_questions_list %}
2      <ul>
3          {% for question in latest_questions_list %}
4              <li><a href="{% url 'detail' question.id %}">
5                  {{ question.question_text }}
6              </a></li>
7          {% endfor %}
8      </ul>
9  {% else %}
10     <p>No polls are available</p>
11 {% endif %}
```

We have only changed line 4. To namespcace the URL names to differentiate between the apps, we need to add `app_name = 'polls'` to `polls/urls.py`. Also, we need to replace `'detail'` in line 4 of the code above to `'polls:detail'`.

2.5 Creating a form

Next, we update the details page to allow users to vote in a poll. First, we update the template `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 %}
    <div>
        {% for choice in question.choice_set.all %}
            <div>
                <input type="radio" name="choice"
                    id="choice {{ forloop.counter }}"
                    value="{{ choice.id }}">
                <label for="choice {{ forloop.counter }}">
                    {{ choice.choice_text }}
                </label>
            </div>
        {% endfor %}
    </div>
    <input type="submit" value="Vote">
</form>
```

The post action is to `vote` function. Next, we update the functions `vote` and `results`:

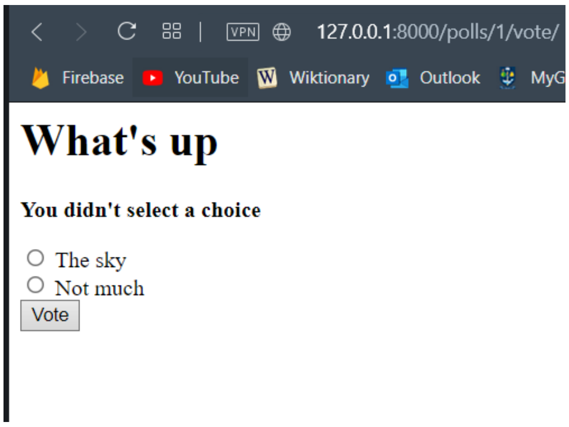
```
from django.urls import reverse
from .models import Choice
from django.http import HttpResponseRedirect

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):
        context_dict = {
            'question': question,
            'error_message': "You didn't select a choice"
        }
        return render(request, 'polls/detail.html', context_dict)
    else:
        selected_choice.votes += 1
        selected_choice.save()

        return HttpResponseRedirect(reverse('polls:results',
            args=(question.id,)))

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

The `reverse` function stops us from hard-coding urls in python code. Then, we create the template `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>
```

The `pluralize` adds an `s` if appropriate. Now, if we vote in the details page without selecting an option, we get the following response:

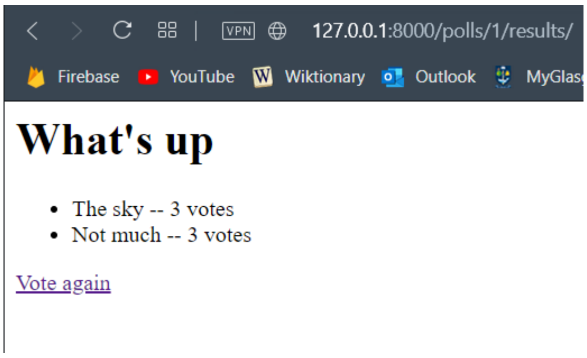

127.0.0.1:8000/polls/1/vote/

What's up

You didn't select a choice

○ The sky
○ Not much

Instead, if we vote for one of the choices, we get the following response:


127.0.0.1:8000/polls/1/results/

What's up

- The sky -- 3 votes
- Not much -- 3 votes

Vote again

2.6 Creating the population script

We will now create a population script that sets up the initial database. To do so, we create the file `mysite/populate_polls.py`. The content within the file is:

```python
import os
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'mysite.settings')

import django
django.setup()

from polls.models import Question, Choice
from datetime import datetime
from pytz import utc

def populate():
    ## delete pre-existing data
    Question.objects.all.delete()
    Choice.objects.all.delete()

    ## question 1
    question1_choices = [
        {
            'choice_text': "The sky",
            'votes': 5
        },
        {
            'choice_text': "Just hacking",
            'votes': 8
        },
        {
            'choice_text': "Not much",
            'votes': 2
        }
    ]

    ## similarly for questions 2-5  ...

    question = {
        "What's up?": {
            "choices": question1_choices,
            "date": datetime(2020, 10, 17, 15, 30, tzinfo=utc)
        },
        ## similarly for questions 2-5 ...
    }

    for question, question_data in question.items():
        q = add_question(question, question_data["date"])
        for c in question_data["choices"]:
            add_choice(q, c["choice_text"], c["votes"])

def add_question(question_text, pub_date):
    question = Question.objects.get_or_create(question_text=
    question_text, date=date)[0]
    question.save()
    return question

def add_choice(question, choice_text, votes):
    choice = Choice.objects.get_or_create(question=question,
    choice_text=choice_text, votes=votes)[0]
    choice.save()
    return choice

## Execution starts here
if __name__ == '__main__':
    populate()
```

To run the script, we run `py manage.py`. We might need to makemigrations and migrate to run this.

2.7 Creating tests

Tests are simple routines that check the operation of your code. Testing operates at different levels. With automated tests, the testing work is done by the system. We create a set of tests once. As we make changes to our app, we can check the code still works as we originally intended.

Tests save us time. Tests do not just identify the problems. They also help us prevent them. Tests make the code more attractive (to others). Also, tests help teams work together.

In test-driven development, we write a test before coding. It might seem counter-intuitive, but it is similar to what we do already. We describe a problem and create some code to solve it. Test-driven development formalises the problem in a Python test case. It is easier to write tests as we create the code rather than later.

The polls app already has a bug! The function `was_published_recently` in the class `Question` within `models.py` is supposed to return `True` if it was published in the last day. However, it returns `True` even if the `date` is in the future.

Now, we add to the file `tests.py` to expose the bug:

```python
from datetime import timedelta

from django.utils import timezone
from django.test import TestCase
from .models import Question

class QuestionMethodTests(TestCase):
    def test_was_published_recently_with_future_question(self):
        ## was_published_recently() should return False for
        ## questions whose publication date is in the future
        time = timezone.now() + timedelta(days=30)
        future_question = Question(date=time)
        self.assertIs(future_question.was_published_recently(),
            False)
```

We can run this file using the command `py manage.py test polls.test`. This test will fail because the function returns `True`. Testing makes use of a dummy database for each class; it does not affect the pre-existing database.

When we call `py manage.py test polls.test`, Django found a subclass of `django.test.TestCase`. It then created a special database for the purpose of testing. It then looks for test methods. These are the methods whose name begins with test. The `test_was_published_recently_with_future_question` test created a `Question` whose `date` was 30 days in the future. Using the method `assertIs`, we then discovered that it returned `True` when we called the method `was_published_recently`, even though we wanted it to return `False`.

Now, we will fix the bug in `models.py`:

```python
def was_published_recently(self):
    now = timezone.now()
    return now - timedelta(days=1) <= self.date <= now
```

This passes the test. We can now add more tests:

```python
from datetime import timedelta

from django.utils import timezone
from django.test import TestCase
from .models import Question

class QuestionMethodTests(TestCase):
    def test_was_published_recently_with_future_question(self):
        ## was_published_recently() should return False for
        ## questions whose publication date is in the future
        time = timezone.now() + timedelta(days=30)
        future_question = Question(date=time)
        self.assertIs(future_question.was_published_recently(),
            False)

    def test_was_published_recently_with_old_question(self):
        ## was_published_recently() should return False for
        ## questions whose publication date is older than 1 day.
        time = timezone.now() - timedelta(days=30)
        future_question = Question(date=time)
        self.assertIs(future_question.was_published_recently(),
            False)

    def test_was_published_recently_with_recent_question(self):
        ## was_published_recently() should return False for
        ## questions whose publication date is within the last day
    .
        time = timezone.now() - timedelta(hours=1)
        future_question = Question(date=time)
        self.assertIs(future_question.was_published_recently(),
            True)
```

This still passes the test.

Testing Views

Django provides a test `Client` to simulate a user interacting with the code at the view level. We can use `Client` in `tests.py` or in the shell. We start by using the shell:

```
>>> from django.test.utils import setup_test_environment
>>> from django.test import Client
>>> setup_test_environment()
>>> client = Client()
>>> response = client.get("/")
Not Found: /
>>> response.status_code
404
>>> from django.urls import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.context['latest_questions_list']
<QuerySet [<Question: "What's up">]>
```

The content here will be operated on the same database unlike before since it is through the command line. Now, we update the file `tests.py`:

```
from django.shortcuts import reverse

def create_question(question_text, days):
    ## creates a question given question_text and
    ## published the given number of days offset to now
    ## (negative for past, positive for future)
    time = timezone.now() + timedelta(days=days)
    return Question.objects.create(question_text=question_text,
        date=time)

class QuestionViewTest(TestCase):
    def test_index_view_with_no_questions(self):
        ## If no questions exist, an appropriate message
        ## should be displayed
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(ressponse, "No polls are available.")
        self.assertQuerySetEqual(response.context
            ['latest_questions_list'], [])

    def test_index_with_a_past_question(self):
        ## Only past questions published should be displayed
        create_question(question_text="Past Question", days=-30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerySetEqual(request.context
            ['latest_question_list'], ['<Question: Past Question'
    ])

    def test_index_with_a_future_question(self):
        # Future questions should not be displayed
        create_question(question_text="Future Question", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(ressponse, "No polls are available.")
        self.assertQuerySetEqual(response.context
            ['latest_questions_list'], [])

    ## ... more tests
```

We can add more tests similar to those already present. We could also improve our application in other ways, while adding tests to check those features. For example, we could ensure that questions cannot be published without choices. Our views could check this, and exclude such questions. Our tests would create a question without choices and then test that it's not published. Also, perhaps logged-in admin users should be allowed to see unpublished questions, but not ordinary visitors. Again, whatever needs to be added to the software to accomplish this should be accompanied by a test. Having too many tests is not an issue.

We might need to update the test, e.g. if we amend our views so that only questions with choices are published. It doesn't matter if tests are redundant. Tests should be arranged so that they are manageable. There should be a separate TestClass for each model or view. We should also have a separate test method for each set of conditions we want to test. We should give test methods intuitive names that describe their function.

2.8 Static Files

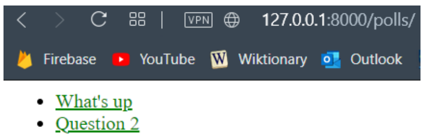
Static files correspond to images, JavaScript and CSS. We will now add a stylesheet (via a CSS file) and an image. We create a file at `static/polls/style.css` inside `polls` folder with the following CSS code:

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

To make use of static files in our templates, the top line should be `{% load staticfiles %}`. We also update the `settings.py` file with:

```python
STATIC_DIR = os.path.join(BASE_DIR, 'static')
STATICFILES_DIRS = [STATIC_DIR, ]
```

so that static files are hosted. Then, the final version of the project is:



- What's up
- Question 2