Создаём своё первое приложение с Django, часть 5

Мы создали веб-приложение для опросов, теперь надо создать несколько автоматизированных тестов для него.

Введение в автоматизированное тестирование

Что такое автоматизированные тесты?

По своей сути тесты – простые задачи, который проверяют работу вашего кода.

Тесты разделяются по уровням. Некоторые проверяют мелкие детали - *этот метод возвращает то, что я ожидаю?* другие более глобальны - *эта последовательность ввода пользователя приведёт к ожидаемому результату?* Нет никакой разницы как тестировать: через shell или через приложение, которое само вводит данные.

Отличие *автоматизированных* тестов заключается в том, что вся работа выполняется за вас системой. Вы один раз создаёте набор тестов, а затем, внеся некоторые изменения в код приложения, вы можете проверить, осталось ли приложение работоспособным в целом, без муторной ручной проверки.

Зачем вам нужны тесты

Так зачем создавать тесты и зачем прямо сейчас?

Может сложиться впечатление, что вы уже освоили Python/Django и изучение ещё каких-то вещей может быть ненужным. После того, как мы завершили разработку веб-приложения для проведения опроса (и оно даже работает!), написание тестов не сделает его лучше. Если эти уроки - последнее, что вы изучали по Django, то тесты вам действительно не нужны, в ином случае приступим к изучению.

Тесты сэкономят ваше время

До определённого момента проверки ‘оно всё ещё работает’ будет достаточно, однако, чем больше приложение, тем больше и сложнее будут связи между компонентами.

Изменения в любом из этих компонентов могут привести к неожиданным последствиям. Проверка в стиле ‘похоже, оно всё ещё работает’ может обернуться беглой проверкой выполнения вашего кода с двенадцатью различными вариациями тестовых данных, чтобы убедиться, что ничего не сломалось. Время можно потратить и с большей пользой.

Особенно важно, что автоматизированные тесты могут сделать это за секунды. Если же что-то пойдёт не так, тесты также помогут найти какая именно часть кода приводит к неожиданному поведению приложения.

Иногда это может выглядеть как случайный повод, чтобы не заниматься по-настоящему продуктивным творческим программированием вместо неблагодарного и неинтересного дела по написанию тестов, особенно, когда вы точно знаете, что ваш код и так работает правильно.

Тем не менее, задача по написанию тестов значительно более плодотворна, чем пустая трата часов ручного труда по поиску причин вновь возникших проблем в коде.

Тесты не находят проблемы, они их предотвращают

Будет ошибкой считать написание тестов негативной стороной разработки.

Без тестов цель и поведение приложения может быть совсем не прозрачным. Даже если это ваш код, вы можете часами биться над ошибкой, не понимая откуда она взялась.

С тестами такого не произойдёт. Они подсвечивают код изнутри, и если что-то пошло не так, они обратят на это ваше внимание - *даже если не видно, что что-то сломалось*.

Тесты делают код более привлекательным

Возможно, вы написали отличный кусок кода, однако многие разработчики могут отказаться с ним работать из-за того, что к нему нет тестов - они не будут доверять ему. Яков Каплан-Мос (один из разработчиков Django) говорит: “Код без тестов нарушает дизайн”.

То, что другие разработчики хотят видеть тесты в вашей программе для того, чтобы воспринимать её всерьёз - ещё одна причина, чтобы начать писать тесты.

Тесты упрощают совместную работу

Предыдущие пункты написаны с точки зрения одиночной разработки приложения. Сложные приложения разрабатываются командами. Тесты гарантируют, что коллеги ненароком не поломают ваш код, а вы их, даже не узнав об этом. Если вы хотите зарабатывать на жизнь в качестве разработчика на Django, вы обязаны быть хороши в написании тестов!

Базовые стратегии тестирования

Есть много подходов к написанию тестов.

Некоторые программисты следуют дисциплине, называемой “test-driven development”; они, на самом деле, сначала пишут тесты, прежде чем писать код. Это может показаться нелогичным, но, на самом деле это похоже на то, что большинство людей будут часто делать в любом случае: они описывают проблемы, а затем создают код, чтобы решить эту проблему. Разработка через тестирование просто формализует задачу в конкретном тесте.

Чаще всего новичок сначала пишет код, а потом тесты. Возможно, было бы лучше делать наоборот, но никогда не поздно дописать тесты.

Иногда трудно понять откуда начать написание тестов. Если вы уже написали несколько тысяч строк кода Python, выбор чего-нибудь для теста может быть непростым. В таком случае, довольно плодотворным будет написать ваш первый тест в следующий раз, когда вы внесёте изменения, добавите новую функцию или исправите баг.

Создаем наш первый тест

Мы нашли ошибку

К счастью, в нашем приложении есть небольшой баг, пригодный для исправления прямо сейчас: метод Question.was_published_recently() возвращает True, если Question был опубликован в последние сутки (что верно), но также если в поле pub_date стоит дата в будущем (что неверно).

Это можно увидеть в админке: создайте вопрос будущей датой и проверьте метод, используя shell:

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30 days in the future
>>> future_question = Question(pub_date=timezone.now() + datetime.timedelta(days=30))
>>> # was it published recently?
>>> future_question.was_published_recently()
True
```

Так как будущее - не ‘недавно’, то это явная ошибка.

Создадим тест на эту ошибку

То, что сделали в shell по сути повторяется в автоматизированном тесте, так что приступим.

Лучшее место для тестов приложения - в файле tests.py - система будет искать тесты во всех файлах, название которых начинается на test.

Скопируйте следующий код в файл tests.py в каталог приложения polls:

```
polls/tests.py
import datetime

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
        pub_date is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertEqual(future_question.was_published_recently(), False)
```

Здесь мы унаследовались от django.test.TestCase и добавили метод, который создаёт экземпляр Question со значением pub_date в будущем. Затем проверяем результат выполнения was_published_recently() - который *должен быть* False.

Выполнение тестов

Для запуска тестов в консоли:

```
$ python manage.py test polls
```

и вы увидите следующее:

```
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_was_published_recently_with_future_question (polls.tests.QuestionMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in
test_was_published_recently_with_future_question
    self.assertEqual(future_question.was_published_recently(), False)
AssertionError: True != False

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Вот что случилось:

- python manage.py test polls ищет тесты в каталоге приложения polls;
- находит класс, который наследуется от django.test.TestCase;
- создаёт специальную базу данных для тестирования;
- ищет тестовые методы - они должны начинаться с test;
- в методе test_was_published_recently_with_future_question создаётся экземпляр Question с pub_date в далёком будущем (аж на 30 дней);
- ... и с помощью метода assertEqual() проверяется что вернул was_published_recently(). Вернулось True, а мы указали, что хотим False.

Далее идёт информация, какой тест не прошёл и на какой строчке возникло расхождение.

Исправление ошибки

Мы уже знаем, что проблема в Question.was_published_recently(): он должен вернуть False, если pub_date находится в будущем. Дополним метод в models.py проверкой - возвращать True только если дата в прошлом:

polls/models.py

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

и заново запустим:

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

После того, как ошибка была выявлена, мы написали тест, который выявил и исправил ошибку в коде, поэтому наш тест был пройден успешно.

Много где ещё могут быть ошибки, но только не в этом месте, потому что простой запуск тестов немедленно нас об этом предупредит. Можно рассматривать этот маленький кусок кода как гарантию работы нашего приложения.

Более подробные тесты

Пока мы здесь, помучаем ещё метод was_published_recently(); действительно, как-то нехорошо при исправлении ошибки создавать другую.

Добавим ещё тестов для полноты проверки поведения метода:

polls/tests.py

```
def test_was_published_recently_with_old_question(self):
    """
    was_published_recently() should return False for questions whose
    pub_date is older than 1 day.
    """
    time = timezone.now() - datetime.timedelta(days=30)
    old_question = Question(pub_date=time)
    self.assertEqual(old_question.was_published_recently(), False)

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() should return True for questions whose
    pub_date is within the last day.
    """
    time = timezone.now() - datetime.timedelta(hours=1)
    recent_question = Question(pub_date=time)
    self.assertEqual(recent_question.was_published_recently(), True)
```

Теперь у нас есть 3 метода, которые проверяют, что Question.was_published_recently() возвращает ожидаемое значение для опросов в далёком прошлом, недавних и будущих.

polls - простое приложение, но каким бы сложным оно не стало в будущем, мы будем уверены, что этот метод будет возвращать ожидаемое значение.

Тестируем представление

Наше приложение довольно неразборчивое - оно публикует даже те вопросы, которые запланированы в будущем. Давайте это исправим. Значение pub_date в будущем должно означать, что до этого времени вопрос должен оставаться невидимым.

Тесты для представления

Когда мы исправляли ошибку выше, мы писали тест, а затем код. Это самый простой пример разработки через тестирование, но на самом деле нет разницы в каком порядке будут проделаны эти действия.

В нашем первом тесте мы фокусировались на коде функции. Сейчас же мы проверим поведение, как если бы пользователь использовал наше приложение через браузер.

Прежде чем мы попытаемся исправить что-либо, давайте взглянем на инструменты, которые есть в нашем распоряжении.

Клиент для тестирования Django

Django предоставляет класс для тестов Client для эмуляции пользовательских действий на уровне представления. Мы можем использовать его в tests.py или в shell.

Давайте начнем снова с shell, где мы должны сделать несколько вещей, которые не нужны в tests.py. Сначала проинициализируем тестовую среду в shell:

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

setup_test_environment() инициализирует шаблон, который позволяет обращаться к некоторым дополнительным атрибутам, например, response.context. Обратите внимание, что этот метод *не настраивает* базу данных, так что мы будем обращаться к реальным данным, и вывод может отличаться в зависимости от того, что там содержится.

Далее нужно импортировать класс тестового клиента (позже в tests.py мы будем использовать django.test.TestCase, который поставляется со своим клиентом):

```
>>> from django.test import Client
>>> # create an instance of the client for our use
>>> client = Client()
```

Теперь мы готовы для запуска:

```
>>> # get a response from '/'
>>> response = client.get('/')
>>> # we should expect a 404 from that address
>>> response.status_code
404
>>> # on the other hand we should expect to find something at '/polls/'
>>> # we'll use 'reverse()' rather than a hardcoded URL
>>> from django.core.urlresolvers import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
b'\n\n\n    <p>No polls are available.</p>\n\n'
>>> # note - you might get unexpected results if your ``TIME_ZONE``
>>> # in ``settings.py`` is not correct. If you need to change it,
>>> # you will also need to restart your shell session
>>> from polls.models import Question
>>> from django.utils import timezone
>>> # create a Question and save it
>>> q = Question(question_text="Who is your favorite Beatle?", pub_date=timezone.now())
>>> q.save()
>>> # check the response once again
>>> response = client.get('/polls/')
>>> response.content
b'\n\n\n      <ul>\n      \n            <li><a href="/polls/1/">Who is your favorite
Beatle?</a></li>\n    \n    </ul>\n\n'
>>> # If the following doesn't work, you probably omitted the call to
>>> # setup_test_environment() described above
>>> response.context['latest_question_list']
[<Question: Who is your favorite Beatle?>]
```

Улучшим наше представление

В список опросов попали и те, которые ещё не должны быть опубликованы (у которых pub_date в будущем). Исправим это.

В четвёртой главе мы унаследовались от ListView:

polls/views.py

```
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]
```

Нам нужно изменить метод get_queryset() так, чтобы он сравнивал дату опроса с timezone.now(). Сначала добавим импорт:

```
polls/views.py

from django.utils import timezone
```

далее изменим существующую функцию get_queryset на:

```
polls/views.py
def get_queryset(self):
    """
    Return the last five published questions (not including those set to be
    published in the future).
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

Question.objects.filter(pub_date__lte=timezone.now()) возвращает набор данных, содержащий те вопросы, у которых pub_date меньше или равна timezone.now.

Тестирование нового представления

Теперь поведение такое, как и задумывалось. Проверим это: запустим runserver, загрузим сайт, создадим Questions с датами в прошлом и будущем и убедимся, что видны только даты в прошлом. Но проделывать это вручную каждый раз накладно - давайте напишем тест, который будет выполнять такую последовательность действий.

Добавим следующий код в polls/tests.py:

```
polls/tests.py

from django.core.urlresolvers import reverse
```

и мы создадим метод-фабрику для создания опросов как новый тестовый класс:

```
polls/tests.py
def create_question(question_text, days):
    """
    Creates a question with the given `question_text` published the given
    number of `days` offset to now (negative for questions published
    in the past, positive for questions that have yet to be published).
    """
    time = timezone.now() + datetime.timedelta(days=days)
    return Question.objects.create(question_text=question_text,
                                   pub_date=time)


class QuestionViewTests(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(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_a_past_question(self):
```

```
            """
            Questions with a pub_date in the past should be displayed on the
            index page.
            """
            create_question(question_text="Past question.", days=-30)
            response = self.client.get(reverse('polls:index'))
            self.assertQuerysetEqual(
                response.context['latest_question_list'],
                ['<Question: Past question.>']
            )

        def test_index_view_with_a_future_question(self):
            """
            Questions with a pub_date in the future should not be displayed on
            the index page.
            """
            create_question(question_text="Future question.", days=30)
            response = self.client.get(reverse('polls:index'))
            self.assertContains(response, "No polls are available.",
                                status_code=200)
            self.assertQuerysetEqual(response.context['latest_question_list'], [])

        def test_index_view_with_future_question_and_past_question(self):
            """
            Even if both past and future questions exist, only past questions
            should be displayed.
            """
            create_question(question_text="Past question.", days=-30)
            create_question(question_text="Future question.", days=30)
            response = self.client.get(reverse('polls:index'))
            self.assertQuerysetEqual(
                response.context['latest_question_list'],
                ['<Question: Past question.>']
            )

        def test_index_view_with_two_past_questions(self):
            """
            The questions index page may display multiple questions.
            """
            create_question(question_text="Past question 1.", days=-30)
            create_question(question_text="Past question 2.", days=-5)
            response = self.client.get(reverse('polls:index'))
            self.assertQuerysetEqual(
                response.context['latest_question_list'],
                ['<Question: Past question 2.>', '<Question: Past question 1.>']
            )
```

Первый блок - это метод-фабрика create_question, который воспроизводит некоторые действия при создании опроса.

test_index_view_with_no_questions не создаёт никаких вопросов, но проверяет вывод сообщения “No polls are available.” и latest_question_list на отсутствие содержимого. Обратите внимание, что django.test.TestCase использует дополнительные методы для проверки утверждений: assertContains() и assertQuerysetEqual().

В test_index_view_with_a_past_question мы создали опрос и убедились, что он попал в список. В test_index_view_with_a_future_question мы создаём опрос с pub_date в будущем. База данных сбрасывается для каждого тестового метода, так что в списке опросов должно быть пусто. По сути, мы используем тесты как эмуляцию ввода данных администратора и для проверки, что на каждом шаге и для каждого изменения программа ведёт себя именно так, как задумывалось.

Тестирование DetailView

мы получили вполне рабочий результат, однако хоть запланированные опросы и не попадут в список, они будут доступны по прямой ссылке. Следовательно, нужно наложить некоторые ограничения на DetailView:

```
polls/views.py
class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        """
        Excludes any questions that aren't published yet.
        """
        return Question.objects.filter(pub_date__lte=timezone.now())
```

И, конечно, мы добавим тесты, проверяющие, что вопросы, у которых pub_date в прошлом, показываются, в то время как те, у которых pub_date в будущем - нет:

```
polls/tests.py
class QuestionIndexDetailTests(TestCase):
    def test_detail_view_with_a_future_question(self):
        """
        The detail view of a question with a pub_date in the future should
        return a 404 not found.
        """
        future_question = create_question(question_text='Future question.',
                                          days=5)
        response = self.client.get(reverse('polls:detail',
                                   args=(future_question.id,)))
        self.assertEqual(response.status_code, 404)

    def test_detail_view_with_a_past_question(self):
        """
        The detail view of a question with a pub_date in the past should
        display the question's text.
        """
        past_question = create_question(question_text='Past Question.',
                                        days=-5)
        response = self.client.get(reverse('polls:detail',
                                   args=(past_question.id,)))
        self.assertContains(response, past_question.question_text,
                            status_code=200)
```

Идеи для других тестов

Нам следовало бы добавить метод, подобный get_queryset, в ResultsView и создать для него новый класс тестов. Это несложно, но, в основном, повторит код, который уже был здесь написан.

Мы также могли бы улучшить наше приложение, покрыв всё тестами. Например, Questions не должен публиковаться, если у него нет Choices. Можно исключать такие опросы в представлении. Тест же может создать Question без Choices и проверить, что он не опубликован, а потом создать Question с Choices и проверить, что он *отображается*.

Возможно, администраторы и должны иметь возможность видеть неопубликованные вопросы, но не обычные посетители. Опять же: каждый новый функционал должен сопровождаться тестами. Неважно, пишутся ли они до или после кода.

В какой-то момент вы посмотрите на тесты, покрывающие функционал и удивитесь, что всё так слаженно работает. Поэтому:

Чем больше тестов, тем лучше

Может показаться, что количество тестов выходит из-под контроля. Их объём превысит размер кода, а методы будут ужасны, по сравнению с кодом приложения.

Это не важно! Пускай они растут. В основном, вы будете писать тест всего раз и забывать про него. Он же будет работать и помогать развивать приложение.

Иногда тесты должны быть обновлены. Например, когда меняется логика функций или объектов. В этом случае многие существующие тесты будут провалены - *это означает, что эти тесты необходимо обновить*, чтобы поддерживать в актуальном состоянии.

В худшем случае может оказаться, что тесты проверяют один и тот же функционал. Не обращайте на это внимание, в тестировании избыточность - *хорошая* вещь.

Пока ваши тесты логично организованы, они управляемы. Хорошие привычки подразумевают:

- разделять TestClass для каждой модели или представления;
- делать отдельный метод для каждого проверяемого условия;
- имена тестовых методов должны описывать их функционал.

Дальнейшее тестирование

Этот урок показывает только базовые техники тестирования. На самом деле много чего здесь не описано, например, полезные утилиты, которые помогают в разработке.

Например, пока наши тесты покрывают внутреннюю логику, вы можете использовать сторонние инструменты для тестирования, например, Selenium. Он позволяет проверять уже конечный результат генерации страницы непосредственно в браузере. Это очень пригодится при тестировании JavaScript. Выглядит это как будто человек открывает сайт и начинает с ним работать. У Django есть класс для интеграции - LiveServerTestCase.

Хороший способ определить какая часть не покрыта тестами - узнать покрытие кода. Это также позволяет выявить неиспользуемый код. Если вы не можете написать тест для тестирования какого-либо куска кода, это означает, что именно здесь требуется рефакторинг.

Когда вы освоите тестирование, прочитайте следующую главу: *«Создаем свое приложение с Django, часть 6»*, там разбирается работа со статическими файлами.