Создаём своё первое приложение с Django, часть 5

Продолжаем начатое в четвёртой части. Мы создали веб-приложение для опросов, теперь надо создать несколько автоматизированных тестов для него.

Where to get help:

If you're having trouble going through this tutorial, please head over to the Getting Help section of the FAQ.

Введение в автоматизированное тестирование

Что такое автоматизированные тесты?

Tests are routines that check the operation of your code.

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` (как показано в Части 2) 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.

A change in any of those components could have unexpected consequences on the application's behavior. Checking that it still „seems to work" could mean running through your code's functionality with twenty different variations of your test data to make sure you haven't broken something - not a good use of your time.

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

You might have created a brilliant piece of software, but you will find that many other developers will refuse to look at it because it lacks tests; without tests, they won't trust it. Jacob Kaplan-Moss, one of Django's original developers, says «Code without tests is broken by design.»

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

Предыдущий раздел

Создаём своё первое приложение с Django, часть 4

Следующий раздел

Создаём своё первое приложение с Django, часть 6

Эта страница

- Исходный текст

Быстрый поиск

Искать

Последнее обновление:

нояб. 23, 2021

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.

1

Some programmers follow a discipline called «test-driven development»; they actually write their tests before they write their code. This might seem counter-intuitive, but in fact it's similar to what most people will often do anyway: they describe a problem, then create some code to solve it. Test-driven development formalizes the problem in a Python test case.

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.

Давайте приступим!

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).

Confirm the bug by using the `shell` to check the method on a question whose date lies in the future:

```
$ python manage.py 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.test import TestCase
from django.utils import timezone

from .models import Question


class QuestionModelTests(TestCase):

    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() returns False for questions whose pub_date
        is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertIs(future_question.was_published_recently(), False)
```

1

Here we have created a `django.test.TestCase` subclass with a method that creates a `Question` instance with a `pub_date` in the future. We then check the output of `was_published_recently()` - which *ought* to be False.

Выполнение тестов

In the terminal, we can run our test:

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

и вы увидите следующее:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
F
======================================================================
FAIL: test_was_published_recently_with_future_question (polls.tests.QuestionModelTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in
test_was_published_recently_with_future_question
    self.assertIs(future_question.was_published_recently(), False)
AssertionError: True is not False

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

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

Different error?

If instead you're getting a `NameError` here, you may have missed a step in Part 2 where we added imports of `datetime` and `timezone` to `polls/models.py`. Copy the imports from that section, and try running your tests again.

Вот что случилось:

- `manage.py test polls` looked for tests in the `polls` application
- 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 `assertIs()` 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'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
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.

1

Many other things might go wrong with our application in the future, but we can be sure that we won't inadvertently reintroduce this bug, because running the test will warn us immediately. We can consider this little portion of the application pinned down safely forever.

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() returns False for questions 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 questions 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)
```

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

Again, `polls` is a minimal application, but however complex it grows in the future and whatever other code it interacts with, we now have some guarantee that the method we have written tests for will behave in expected ways.

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

When we fixed the bug above, we wrote the test first and then the code to fix it. In fact that was an example of test-driven development, but it doesn't really matter in which order we do the work.

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`.

We will start again with the `shell`, where we need to do a couple of things that won't be necessary in `tests.py`. The first is to set up the test environment in the `shell`:

```
$ python manage.py 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. Вы можете получить неожиданный результат, если `TIME_ZONE` в `settings.py` указана не верно. Если вы не установили эту настройку, проверьте перед тем, как продолжить.

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('/')
Not Found: /
>>> # we should expect a 404 from that address; if you instead see an
>>> # "Invalid HTTP_HOST header" error and a 400 response, you probably
>>> # omitted the setup_test_environment() call described earlier.
>>> 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.urls import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
b'\n    <ul>\n    \n        <li><a href="/polls/1/">What&#x27;s up?</a></li>\n    \n
</ul>\n\n'
>>> response.context['latest_question_list']
<QuerySet [<Question: What's up?>]>
```

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 части 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

1

Now you can satisfy yourself that this behaves as expected by firing up `runserver`, loading the site in your browser, creating `Questions` with dates in the past and future, and checking that only those that have been published are listed. You don't want to have to do that *every single time you make any change that might affect this* - so let's also create a test, based on our `shell` session above.

Добавим следующий код в `polls/tests.py`:

polls/tests.py

```
from django.urls import reverse
```

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

polls/tests.py

```
def create_question(question_text, days):
    """
    Create a question with the given `question_text` and 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 QuestionIndexViewTests(TestCase):
    def test_no_questions(self):
        """
        If no questions exist, an appropriate message is 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_past_question(self):
        """
        Questions with a pub_date in the past are 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_future_question(self):
        """
        Questions with a pub_date in the future aren't 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.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        are 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_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.>']
        )
```

Давайте подробнее разберём некоторые моменты.

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

`test_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_past_question` мы создали опрос и убедились, что он попал в список.

В `test_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.

И так далее. 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 QuestionDetailViewTests(TestCase):
    def test_future_question(self):
        """
        The detail view of a question with a pub_date in the future
        returns a 404 not found.
        """
        future_question = create_question(question_text='Future question.', days=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 a question with a pub_date in the past
        displays the question's text.
        """
        past_question = create_question(question_text='Past Question.', days=-5)
        url = reverse('polls:detail', args=(past_question.id,))
        response = self.client.get(url)
        self.assertContains(response, past_question.question_text)
```

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`.

Если у вас сложное приложение, то можно настроить запуск тестов на каждый коммит (часть continuous integration), что позволит автоматически следить за качеством кода.

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. Подробнее тут - Integration with coverage.py.

Больше информации по тестированию можно получить здесь: Testing Django applications

Что дальше?

За дополнительной информацией можно обратиться к документу Тестирование в Django.

Когда вы освоите тестирование, прочитайте шестую часть этого учебника, там разбирается работа со статическими файлами.