Внедрение автоматизированного тестирования¶

Что такое автоматические тесты?¶

Тесты — это процедуры, которые проверяют работу вашего кода.

Тестирование работает на разных уровнях. Некоторые тесты могут применяться к крошечной детали (*возвращает ли конкретный метод модели ожидаемые значения?*) в то время как другие Изучите общую работу программного обеспечения (*выполняет ли последовательность пользовательских вводов на сайте произвести желаемый результат?*). Это ничем не отличается от такого рода тестирование, выполненное ранее в учебнике 2, использование метода для проверки поведения метода или выполнение команды приложение и ввод данных, чтобы проверить, как он себя ведет.

Отличие *автоматических* тестов заключается в том, что работа по тестированию выполняется для Вы по системе. Набор тестов создается один раз, а затем по мере внесения изменений Чтобы убедиться в том, что ваш код по-прежнему работает так, как вы были изначально, вы можете убедиться в том, что ваш код работает так же, как и изначально предназначено, без необходимости выполнять трудоемкое ручное тестирование.

Зачем нужно создавать тесты¶

Так зачем создавать тесты, и почему именно сейчас?

Вы можете почувствовать, что у вас достаточно на тарелке, просто учась Python/Django, и может показаться, что нужно изучить и сделать еще одну вещь ошеломляющий и, возможно, ненужный. В конце концов, наше приложение для опросов Сейчас работаю вполне счастливо; Трудности с созданием автоматизированных Тесты не заставят его работать лучше. При создании опросов приложение - это последний бит программирования Django, который вы когда-либо будете делать, тогда true, Вам не нужно знать, как создавать автоматические тесты. Но, если это не так кейс, сейчас отличное время для обучения.

Тесты сэкономят ваше время¶

До определенного момента «проверка того, что кажется, что это работает», будет удовлетворительной тест. В более сложном приложении у вас могут быть десятки сложных взаимодействия между компонентами.

Изменение любого из этих компонентов может иметь неожиданные последствия для Поведение приложения. Проверка того, что он все еще «кажется, работает», может означать Прогон функциональности вашего кода с двадцатью различными вариантами Ваши тестовые данные, чтобы убедиться, что вы что-то не сломали - не очень хорошее использование вашего времени.

Это особенно верно, когда автоматические тесты могут сделать это за вас за считанные секунды. Если что-то пошло не так, тесты также помогут идентифицировать код Это вызывает неожиданное поведение.

Иногда может показаться рутиной оторваться от своей продуктивной, Творческая работа по программированию, чтобы противостоять не гламурному и не интересному бизнесу написания тестов, особенно когда вы знаете, что ваш код работает правильно.

Тем не менее, задача написания тестов приносит гораздо больше удовлетворения, чем тратить часы Тестирование приложения вручную или попытка определить причину Недавно введенная проблема.

Тесты не просто выявляют проблемы, они их предотвращают¶

Ошибочно думать о тестах просто как о негативном аспекте разработки.

Без тестов цель или предполагаемое поведение приложения может быть следующим довольно непрозрачный. Даже если это ваш собственный код, вы иногда окажетесь ковыряться в нем, пытаясь выяснить, что именно он делает.

Тесты меняют это; Они подсвечивают ваш код изнутри, и когда что-то идет не так, они фокусируют свет на той части, которая пошла не так, *даже если вы даже не понял, что все пошло не так*.

Тесты делают ваш код более привлекательным¶

Возможно, вы создали блестящее программное обеспечение, но вы обнаружите, что многие другие разработчики откажутся смотреть на него, потому что в нем отсутствуют тесты; без тесты, они не будут доверять этому. Джейкоб Каплан-Мосс, один из оригинальных Джанго разработчиков, говорит: «Код без тестов нарушается по дизайну».

Что другие разработчики хотят видеть тесты в вашем программном обеспечении, прежде чем они его выполнят Серьезно, это еще одна причина, по которой вы должны начать писать тесты.

Тесты помогают командам работать вместе¶

Предыдущие пункты написаны с точки зрения одного разработчика Сопровождение приложения. Сложные приложения будут поддерживаться командами. Тесты гарантируют, что коллеги случайно не нарушат ваш код (и что вы не сломаете их, не зная). Если вы хотите зарабатывать на жизнь Программист Django, вы должны хорошо писать тесты!

Основные стратегии тестирования¶

Есть много способов подойти к написанию тестов.

Некоторые программисты следуют дисциплине, называемой «разработка, основанная на тестировании»; они на самом деле пишут свои тесты до того, как они напишут свой код. Казалось бы, это Это противоречит здравому смыслу, но на самом деле это похоже на то, что часто делает большинство людей В любом случае: они описывают проблему, а затем создают какой-то код для ее решения. Тест-драйв разработка формализует проблему в тестовом примере Python.

Чаще всего новичок в тестировании создаст какой-то код, а потом решит, что У него должны быть некоторые тесты. Возможно, было бы лучше написать что-нибудь тестирует раньше, но никогда не поздно начать.

Иногда трудно понять, с чего начать написание тестов. Если вы написали несколько тысяч строк на Python, выбирая что-то для Тест может быть непростым. В таком случае полезно написать свой первый тест В следующий раз, когда вы внесете изменения, либо при добавлении новой функции, либо при исправлении ошибки.

Так что давайте сделаем это прямо сейчас.

Написание нашего первого теста¶

Выявляем баг¶

К счастью, в приложении есть небольшая ошибка, которую мы должны исправить Сразу же: метод возвращает, если был опубликован в течение последнего дня (что правильно), но также и если: Поле находится в будущем (что, конечно, не так).**`pollsQuestion.was_published_recently()TrueQuestionQuestionpub_date`**

Подтвердите ошибку с помощью метода проверки вопроса чья дата лежит в будущем:

```
/
...\> py 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
```

Поскольку события в будущем не являются «недавними», это явно неправильно.

Создание теста для выявления ошибки¶

То, что мы только что сделали для проверки проблемы, точно Что мы можем сделать в автоматическом тесте, поэтому давайте превратим это в автоматизированный тест.

Обычное место для тестов приложения находится в файле приложения; Система тестирования автоматически найдет тесты в любом файле чье имя начинается с **.tests.pytest**

Поместите в файл в приложении следующее:**tests.pypolls**

```
 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)
```

Здесь мы создали подкласс с методом, который Создает экземпляр с символом A в будущем. Затем мы проверяем вывод - который *должен* быть False.**Questionpub_datewas_published_recently()**

Выполнение тестов¶

В терминале мы можем запустить наш тест:

```
/
...\> py manage.py test polls
```

И вы увидите что-то вроде:

```
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'...
```

Другая ошибка?

Если вместо этого вы получаете здесь, возможно, вы пропустили шаг в части 2, где мы добавили импорт и в . Скопируйте импорт из и попробуйте выполнить тесты еще раз.**NameErrordatetimetimezonepolls/models.py**

Произошло вот что:

- **manage.py test polls** искал тесты в приложении**polls**
- Он нашел подкласс класса
- Была создана специальная база данных с целью тестирования
- Он искал методы испытаний - те, названия которых начинаются с **test**
- В нем создается экземпляр, поле которого находится на 30 дней впереди**test_was_published_recently_with_future_questionQuestionpub_date**
- ... И, используя метод, он обнаружил, что его возвращает , хотя мы хотели, чтобы он вернулся **assertIs()was_published_recently()TrueFalse**

Тест информирует нас о том, какой тест не удался, и даже о строке, на которой произошел сбой Произошло.

Исправление ошибки¶

We already know what the problem is: should return if its is in the future. Amend the method in , so that it will only return if the date is also in the past:**Question.was_published_recently()Falsepub_datemodels.pyTrue**

polls/models.py¶

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

and run the test again:

```
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 выявления ошибки мы написали тест, который ее выявляет, и исправили ошибку в коде так наш тест проходит.

Многое другое может пойти не так с нашим приложением в будущем, но мы можем Будьте уверены, что мы случайно не добавим эту ошибку снова, потому что запуск Тест предупредит нас немедленно. Мы можем рассмотреть эту маленькую часть Приложение надежно закреплено навсегда.

Более полные тесты¶

Пока мы здесь, мы можем уточнить метод; На самом деле, было бы положительно неловко, если бы при исправлении одной ошибки у нас было ввел другую.**`was_published_recently()`**

Добавьте еще два метода теста в тот же класс, чтобы проверить поведение метода Более подробно:

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)
```

И теперь у нас есть три теста, которые подтверждают, что возвращают разумные значения для прошлых, недавних и будущих вопросов.**`Question.was_published_recently()`**

Опять же, это минимальное приложение, но каким бы сложным оно ни было в future и с каким бы другим кодом он ни взаимодействовал, теперь у нас есть некоторая гарантия что метод, для которого мы написали тесты, будет вести себя ожидаемым образом.**`polls`**

Тестирование представления¶

Приложение опросов довольно неразборчиво: оно опубликует любой вопрос, в том числе и те, чья сфера лежит в будущем. Мы должны совершенствоваться этот. Постановка в будущем должна означать, что Вопрос опубликованный в тот момент, но невидимый до тех пор.**`pub_datepub_date`**

Тест для представления¶

Когда мы исправили ошибку выше, мы сначала написали тест, а затем код для исправления оно. На самом деле это был пример разработки, основанной на тестировании, но это не так действительно имеет значение, в каком порядке мы выполняем работу.

В нашем первом тесте мы сосредоточились на внутреннем поведении кода. Для В этом тесте мы хотим проверить его поведение так, как оно будет восприниматься пользователем через веб-браузер.

Прежде чем мы попытаемся что-то исправить, давайте посмотрим на инструменты, имеющиеся в нашем распоряжении.

Тестовый клиент Django¶

Django предоставляет тест для имитации пользователя взаимодействие с кодом на уровне представления. Мы можем использовать его в или даже в .**`tests.py`**

Мы начнем снова с , где нам нужно сделать пару вещи, которые не понадобятся в . Во-первых, настроить тест Окружающая среда в :**`tests.py`**

```
/
...\> py manage.py shell
```

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

Устанавливает модуль визуализации шаблонов что позволит нам изучить некоторые дополнительные атрибуты ответов, такие как те, которые в противном случае были бы недоступны. Обратите внимание, что это *Метод не настраивает* тестовую базу данных, поэтому будет выполнено следующее Существующая база данных и выходные данные могут незначительно отличаться в зависимости от того, что вопросы, которые вы уже создали. Вы можете получить неожиданные результаты, если ваш ввод неверен. Если вы не помните настройку Это раньше, проверьте это, прежде чем продолжить.**response.contextTIME_ZONEsettings.py**

Далее нам нужно импортировать тестовый класс клиента (далее мы будем использовать класс, который поставляется со своим клиентом, поэтому Это не потребуется):**tests.py**

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

Имея все готово, мы можем попросить клиента сделать за нас некоторую работу:

```
>>> # 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?>]>
```

Улучшение нашего обзора

В списке опросов отображаются опросы, которые еще не опубликованы (т.е. те, которые будут опубликованы в будущем). Давайте исправим это.**pub_date**

В уроке 4 мы представили представление на основе классов, По материалам:

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 внести поправки в метод и изменить его так, чтобы он также проверяет дату, сравнивая ее с . Для начала нам нужно добавить
Импорт:**get_queryset()timezone.now()**

polls/views.py

```
from django.utils import timezone
```

И тогда мы должны изменить метод следующим образом:**get_queryset**

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_da
te")[:5]
```

`Question.objects.filter(pub_date__lte=timezone.now())` Возвращает набор запросов. содержащий s, который меньше или равен - что является, более ранним или равным - .**`Questionpub_datetimezone.now`**

Тестирование нашего нового представления¶

Теперь вы можете убедиться в том, что это ведет себя так, как ожидалось, запустив, загрузив сайт в свой браузер, создав с помощью даты в прошлом и будущем, и проверяя, что только те, которые были опубликованные перечислены. Вы не хотите делать это *каждый раз, когда вы Внесите любые изменения, которые могут повлиять на это* - поэтому давайте также создадим тест, основанный на Наша сессия выше.**`runserverQuestions`**

Добавить следующее:

`polls/tests.py`

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

И мы создадим функцию быстрого доступа для создания вопросов, а также нового теста .class:

`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.
        """
        question = create_question(question_text="Past question.",
days=-30)
        response = self.client.get(reverse("polls:index"))
        self.assertQuerySetEqual(
            response.context["latest_question_list"],
            [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.
        """
        question = 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],
        )

    def test_two_past_questions(self):
        """
        The questions index page may display multiple questions.
        """
        question1 = create_question(question_text="Past question 1.",
days=-30)
        question2 = create_question(question_text="Past question 2.",
days=-5)
        response = self.client.get(reverse("polls:index"))
        self.assertQuerySetEqual(
            response.context["latest_question_list"],
            [question2, question1],
        )
```

Давайте посмотрим на некоторые из них более внимательно.

Во-первых, это функция ярлыка вопроса, чтобы взять некоторые Повторение вне процесса создания вопросов.**create_question**

test_no_questions не создает никаких вопросов, но проверяет сообщение: «Опросы недоступны» и проверяет, пуст. Обратите внимание, что класс предоставляет некоторые дополнительные Методы утверждения. В этих примерах мы используем и .**latest_question_list**

В , мы создаем вопрос и проверяем, отображается ли он в список.**test_past_question**

В , мы создаем вопрос с символом в будущее. База данных сбрасывается для каждого метода тестирования, поэтому первый вопрос - нет дольше там, и поэтому опять же в индексе не должно быть никаких вопросов.**test_future_questionpub_date**

И так далее. По сути, мы используем тесты, чтобы рассказать историю ввода администратора и пользовательский опыт на сайте, и проверка этого в каждом штате и для каждого Новые изменения в состоянии системы, ожидаемые результаты публикуются.

Тестирование `DetailView`¶

То, что у нас есть, работает хорошо; Однако, несмотря на то, что будущие вопросы не появляются в *индекс*, пользователи по-прежнему могут получить к ним доступ, если они знают или угадывают правильный URL-адрес. Так Нам нужно добавить аналогичное ограничение к :**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())
```

Затем мы должны добавить несколько тестов, чтобы проверить, что тот, кто в прошлом, может быть отображен, а тот, у кого в будущем, Нет:**Questionpub_datepub_date**

```
 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 дополнительных тестов¶

Мы должны добавить аналогичный метод к и Создайте новый тестовый класс для этого представления. Это будет очень похоже на то, что у нас есть только что созданный; На самом деле повторений будет много.**`get_querysetResultsView`**

Мы также могли бы улучшить наше приложение другими способами, добавив тесты вдоль способ. Например, это глупо, что может быть опубликовано на сайте которые не имеют . Так что, наши взгляды могли бы проверить это, и исключить такие. Наши тесты создадут без и Затем проверьте, что он не опубликован, а также создайте аналогичный *файл с* помощью , и проверьте, *что он опубликован*.**`QuestionsChoicesQuestionsQuestionChoicesQuestionChoices`**

Возможно, вошедшим в систему пользователям-администраторам следует разрешить видеть неопубликованных, но не обычных посетителей. Опять же: все, что нужно добавить Программное обеспечение для этого должно сопровождаться тестом, независимо от того, являетесь ли вы Сначала напишите тест, а затем заставьте код пройти тест или разработайте Сначала логика в коде, а затем напишите тест, чтобы доказать это.**`Questions`**

В какой-то момент вы обязательно посмотрите на свои тесты и зададитесь вопросом, является ли ваш Код страдает от раздувания тестов, что приводит нас к следующему:

При тестировании чем больше, тем лучше¶

It might seem that our tests are growing out of control. At this rate there will soon be more code in our tests than in our application, and the repetition is unaesthetic, compared to the elegant conciseness of the rest of our code.

It doesn't matter. Let them grow. For the most part, you can write a test once and then forget about it. It will continue performing its useful function as you continue to develop your program.

Sometimes tests will need to be updated. Suppose that we amend our views so that only with are published. In that case, many of our existing tests will fail - *telling us exactly which tests need to be amended to bring them up to date*, so to that extent tests help look after themselves.`QuestionsChoices`

At worst, as you continue developing, you might find that you have some tests that are now redundant. Even that's not a problem; in testing redundancy is a *good* thing.

Пока ваши тесты разумно организованы, они не станут неуправляемыми. Хорошие эмпирические правила включают в себя:

- Отдельно для каждой модели или вида`TestClass`
- Отдельный метод тестирования для каждого набора условий, которые необходимо проверить
- Имена тестовых методов, описывающие их функции

Дальнейшее тестирование¶

В этом учебном пособии представлены только некоторые основы тестирования. Есть отличный Сделайте больше, чем вы можете сделать, и ряд очень полезных инструментов в вашем распоряжении, чтобы Добейтесь некоторых очень умных вещей.

Например, в то время как наши тесты здесь охватывали некоторую внутреннюю логику Модель и способ, которым наши представления публикуют информацию, вы можете использовать «в браузере» фреймворк, такой как Selenium, для тестирования того, как ваш HTML на самом деле отображается в Обозреватель. Эти инструменты позволяют вам проверять не только поведение вашего Django код, а также, например, вашего JavaScript. Это довольно то, на что можно посмотреть Тесты запускают браузер и начинают взаимодействовать с вашим сайтом, как с человеком существа были за рулем! Django включает в себя для облегчения интеграции с такими инструментами, как Selenium.

Если у вас сложное приложение, вы можете захотеть запускать тесты автоматически с каждым коммитом в целях непрерывной интеграции, так что Контроль качества сам по себе - по крайней мере, частично - автоматизирован.