Оглавление

Предисловие ... 16

Предпосылки и предположения ... 21

Сопутствующее видео ... 30

Признательности ... 31

Часть I. Основы TDD и Django ... 33

Глава 1. Настройка Django с использованием функционального теста ... 34

Повинуйтесь Билли-тестировщику! Ничего не делайте, пока у вас не будет теста ... 34
Приведение Django в рабочее состояние ... 37
Запуск репозитория Git ... 40

Глава 2. Расширение функционального теста при помощи модуля unittestn ... 44

Использование функционального теста для определения минимального дееспособного приложения ... 44
Модуль unittest стандартной библиотеки Python ... 47
Фиксация ... 50

Глава 3. Тестирование простой домашней страницы при помощи модульных тестов ... 52

Первое приложение Django и первый модульный тест ... 53
Модульные тесты и их отличия от функциональных тестов ... 53
Модульное тестирование в Django ... 55
MVC в Django, URL-адреса и функции представления ... 56
Наконец-то! Мы на самом деле напишем прикладной код! ... 58
Файл urls.py ... 59
Модульное тестирование представления ... 62
Цикл «модульный-тест/программный-код» ... 64

Глава 4. И что же делать со всеми этими тестами (и рефакторизацией)? ... 68

Программировать – все равно что поднимать ведро с водой из колодца ... 69
Использование Selenium для тестирования взаимодействий пользователя ... 71
Правило «Не тестировать константы» и шаблоны во спасение ... 74
Перестройка программного кода для использования шаблона ... 75
Тестовый клиент Django ... 79
О рефакторизации ... 81
Еще немного о главной странице ... 83
Резюме: процесс TDD ... 85

Глава 5. Сохранение вводимых пользователем данных: тестирование базы данных ... 88

Подключение формы для отправки POST-запроса ... 89
Обработка POST-запроса на сервере ... 92
Передача переменных Python для вывода в шаблоне ... 93
Если клюнуло трижды, рефакторизуй ... 98
Django ORM и первая модель ... 100
Первая миграция базы данных ... 102
Тест продвинулся удивительно далеко ... 103
Новое поле означает новую миграцию ... 104
Сохранение POST-запроса в базу данных ... 105
Переадресация после POST-запроса ... 108
Лучшие приемы модульного тестирования: каждый тест должен проверять одну единицу кода ... 109
Генерирование элементов в шаблоне ... 110
Создание производственной базы данных с миграцией ... 112
Резюме ... 115

Глава 6. Усовершенствование функциональных тестов: обеспечение изоляции и удаление методов sleep ... 118
Обеспечение изоляции в функциональных тестах ... 118
Выполнение только модульных тестов ... 122
Ремарка: обновление Selenium и Geckodriver ... 123
О неявных и явных ожиданиях и вуду сне time.sleep ... 124

Глава 7. Работа в инкрементном режиме ... 130
Маломасштабные конструктивные изменения по мере необходимости ... 130
В первую очередь маломасштабные конструктивные изменения ... 131
Вам это никогда не понадобится! ... 131
REST (-овский) подход ... 132
Инкрементная реализация новой структуры кода на основе TDD ... 133
Обеспечение теста на наличие регрессии ... 134
Итеративное движение в сторону новой структуры кода ... 137
Первый самодостаточный шаг: один новый URL-адрес ... 139
Новый URL-адрес ... 140
Новая функция представления ... 140
Зеленый? Рефакторизуй! ... 142
Еще один шажок: отдельный шаблон для просмотра списков ... 143
Третий шажок: URL-адрес для добавления элементов списка ... 146
Тестовый класс для создания нового списка ... 146
URL-адрес и представление для создания нового списка ... 148
Удаление избыточного теперь кода и тестов ... 149
Регрессия! Наведение форм на новый URL-адрес ... 149
Стиснув зубы: корректировка моделей ... 151
Связь по внешнему ключу ... 153
Адаптация остальной части мира к новым моделям ... 155
Каждому списку – свой URL-адрес ... 157
Извлечение параметров из URL-адресов ... 158
Адаптирование new_list к новому миру ... 160

Функциональные тесты обнаруживают еще одну регрессию 161
Еще одно представление для добавления элементов в существующий список 162
Остерегайтесь «жадных» регулярных выражений! 163
Последний новый URL-адрес 164
Последнее новое представление 165
Тестирование контекстных объектов отклика напрямую 166
Финальная рефакторизация с использованием URL-включений 168

Часть II. Непременные условия веб-разработки 171

Глава 8. Придание привлекательного вида: макет, стилевое оформление сайта и что тут тестировать 172
Что функционально тестируется в макете и стилевом оформлении 172
Придание привлекательного вида: использование платформы CSS 176
Наследование шаблонов в Django 178
Интеграция платформы Bootstrap 180
Строки и столбцы 180
Статические файлы в Django 182
Переход на StaticLiveServerTestCase 183
Использование компонентов Bootstrap для улучшения внешнего вида сайта 184
Класс jumbotron 184
Большие поля ввода 185
Стилистическое оформление таблицы 185
Использование собственного CSS 185
О чем мы умолчали: collectstatic и другие статические каталоги 187
Несколько вещей, которые не удались 190

Глава 9. Тестирование развертывания с использованием промежуточного сайта 191
TDD и опасные зоны развертывания 192
Как всегда, начинайте с теста 194
Получение доменного имени 196
Ручное обеспечение работы сервера для размещения сайта 196
Выбор места размещения сайта 197
Запуск сервера 197
Учетные записи пользователей, SSH и полномочия 198
Инсталляция Nginx 198
Инсталляция Python 3.6 200
Конфигурирование доменов для промежуточного и реального серверов 200
Использование ФТ для подтверждения, что домен работает и Nginx выполняется 201
Развертывание исходного кода вручную 201
Корректировка расположения базы данных 202
Создание Virtualenv вручную и использование requirements.txt 204
Простое конфигурирование Nginx 206
Создание базы данных при помощи команды migrate 209
Успех! Наше хакерское развертывание работает 210

Глава 10. Переход к развертыванию, готовому к эксплуатации ... 212
Переход на Gunicorn ... 212
Настройка Nginx для раздачи статических файлов ... 214
Переход на использование сокетов Unix ... 215
Установка DEBUG в False и настройка ALLOWED_HOSTS ... 216
Применение Systemd для проверки, что Gunicorn запускается на начальной загрузке ... 217
Сохранение изменений: добавление Gunicorn в файл requirements.txt ... 218
Размышления об автоматизации ... 219
Сохранение шаблонов конфигурационных файлов этапа обеспечения работы ... 219
Сохранение хода выполнения ... 222

Глава 11. Автоматизация развертывания с помощью Fabric ... 224
Описание частей сценария Fabric для развертывания ... 225
Создание структуры каталогов ... 226
Получение исходного кода из репозитория командой git ... 226
Обновление файла settings.py ... 227
Обновление virtualenv ... 229
Миграция базы данных при необходимости ... 229
Испытание автоматизации ... 230
Развертывание на работающем сайте ... 231
Конфигурирование Nginx и Gunicorn при помощи sed ... 234
Маркировка релиза командой git tag ... 235
Дополнительные материалы для чтения ... 236

Глава 12. Разделение тестов на многочисленные файлы и обобщенный помощник ожидания ... 238
Начало с ФТ валидации данных: предотвращение пустых элементов ... 238
Пропуск теста ... 239
Разбиение функциональных тестов на несколько файлов ... 241
Выполнение только одного файла с тестами ... 244
Новый инструмент функционального тестирования: обобщенная вспомогательная функция явного ожидания ... 245
Завершение ФТ ... 249
Рефакторизация модульных тестов на несколько файлов ... 251

Глава 13. Валидация на уровне базы данных ... 254
Валидация на уровне модели ... 255
Контекстный менеджер self.assertRaises ... 255
Причуда Django: сохранение модели не выполняет валидацию ... 256
Выведение на поверхность ошибок валидации модели в представлении ... 257
Проверка, чтобы недопустимые входные данные не сохранялись в базе данных ... 261
Схема Django: обработка POST-запросов в том же представлении, которое генерирует форму ... 263
Рефакторизация: передача функциональности new_item в view_list ... 264
Обеспечение валидации модели в view_list ... 267
Рефакторизация: удаление жестко кодированных URL-адресов ... 269

Тег {% url %} шаблона ... 269
Использование get_absolute_url для переадресаций ... 270

Глава 14. Простая форма ... 274

Перемещение программной логики валидации из формы ... 274
Исследование API форм при помощи модульного теста ... 275
Переход на Django ModelForm ... 277
Тестирование и индивидуальная настройка валидации формы ... 278
Использование формы в представлениях ... 281
Использование формы в представлении с GET-запросом ... 281
Глобальный поиск и замена ... 282
Использование формы в представлении, принимающем POST-запросы ... 285
Адаптация модульных тестов к представлению new_list ... 285
Использование формы в представлении ... 287
Использование формы для отображения ошибок в шаблоне ... 287
Использование формы в другом представлении ... 288
Вспомогательный метод для нескольких коротких тестов ... 289
Неожиданное преимущество: бесплатная валидация на стороне клиента из HTML5 ... 291
Одобряющее похлопывание по спине ... 293
Не потратили ли мы уйму времени впустую? ... 294
Использование собственного для формы метода save ... 295

Глава 15. Более развитые формы ... 298

Еще один ФТ на наличие повторяющихся элементов ... 298
Предотвращение дубликатов на уровне модели ... 299
Небольшое отступление по поводу упорядочивания Queryset и представлений строковых значений ... 301
Новое написание старого теста модели ... 304
Некоторые ошибки целостности проявляются при сохранении ... 306
Экспериментирование с проверкой на наличие повторяющихся элементов на уровне представлений ... 307
Более сложная форма для проверки на наличие повторяющихся значений ... 308
Использование существующей формы для элемента списка в представлении для списка ... 310
Итоги: что мы узнали о тестировании Django ... 313

Глава 16 Пробуем окунуться, очень робко, в JavaScript ... 316

Начинаем с ФТ ... 316
Настройка элементарного исполнителя тестов на JavaScript ... 318
Использование элемента div для jQuery и фикстуры ... 320
Создание модульного теста на JavaScript для требуемой функциональности ... 324
Фикстуры, порядок выполнения и глобальное состояние: ключевые проблемы тестирования на JS ... 326
console.log для отладочной распечатки ... 326
Использование функции инициализации для большего контроля над временем выполнения ... 328
Коломбо говорит: стереотипный код для onload и организация пространства

имен ... 330
Тестирование на Javascript в цикле TDD ... 332
Несколько вещей, которые не удалось сделать ... 332

Глава 17. Развертывание нового программного кода ... 334
Развертывание на промежуточном сервере ... 334
Развертывание на реальном сервере ... 335
Что делать, если вы видите ошибку базы данных ... 335
Итоги: маркировка нового релиза командой git tag ... 335

Часть III. Основы TDD и Django ... 337

Глава 18. Аутентификация пользователя, импульсное исследование и внедрение его результатов ... 338
Беспарольная аутентификация ... 338
Разведочное программирование, или Импульсное исследование ... 339
Открытие ветки для результатов импульсного исследования ... 340
Авторизация на стороне клиента в пользовательском интерфейсе ... 341
Отправка электронных писем из Django ... 341
Использование переменных окружения для предотвращения секретов в исходном коде ... 344
Хранение маркеров в базе данных ... 344
Индивидуализированные модели аутентификации ... 345
Завершение индивидуализированной авторизации в Django ... 346
Внедрение результатов импульсного исследования ... 351
Возвращение импульсного исходного кода в прежний вид ... 353
Минимальная индивидуализированная модель пользователя ... 354
Тесты в качестве документирования ... 357
Модель маркера для соединения электронных адресов с уникальным идентификатором ... 358

Глава 19. Использование имитаций для тестирования внешних зависимостей или сокращения дублирования ... 362
Перед началом: получение базовой инфраструктуры ... 362
Создание имитаций вручную, или Обезьянья заплатка ... 363
Библиотека Mock ... 367
Использование unittest.patch ... 368
Продвижение ФТ чуть дальше вперед ... 371
Тестирование инфраструктуры сообщений Django ... 371
Добавление сообщений в HTML ... 374
Начало с URL-адреса для входа в систему ... 375
Подтверждение отправки пользователю ссылки с маркером ... 376
Создание индивидуализированного серверного процессора аутентификации на основе результатов импульсного исследования ... 378
Еще один тест для каждого случая ... 379
Метод get_user ... 382

Использование серверного процессора аутентификации в представлении входа в систему 384
Еще одна причина использовать имитации: устранение повторов 385
Использование mock.return_value 388
Установка заплатки на уровне класса 390
Момент истины: пройдет ли ФТ? 392
Теоретически – работает! Работает ли на практике? 394
Завершение ФТ, тестирование выхода из системы 396

Глава 20. Тестовые фикстуры и декоратор для явных ожиданий 399
Пропуск регистрации в системе путем предварительного создания сеанса 399
Проверка работоспособности решения 402
Финальный вспомогательный метод явного ожидания: декоратор ожидания 405

Глава 21. Отладка на стороне сервера 410
Чтобы убедиться в пудинге, надо его попробовать: использование предварительного сервера для отлавливания финальных дефектов 410
Настройка журналирования 411
Установка секретных переменных окружения на сервере 413
Адаптация ФТ для тестирования реальных электронных писем POP3 414
Управление тестовой базой данных на промежуточном сервере 418
Управляющая команда Django для создания сеансов 418
Настройка ФТ для выполнения управляющей команды на сервере 420
Использование fabric напрямую из Python 421
Резюме: создание сеансов локально по сравнению с промежуточным сервером 422
Внедрение программного кода журналирования 424
Итоги 425

Глава 22. Завершение приложения «Мои списки»: TDD с подходом «снаружи внутрь» 427
Альтернатива: «изнутри наружу» 427
Почему «снаружи внутрь» предпочтительнее? 428
ФТ для «Моих списков» 428
Внешний уровень: презентация и шаблоны 431
Спуск на один уровень вниз к функциям представления (к контроллеру) 431
Еще один проход снаружи внутрь 433
Быстрая реструктуризация иерархии наследования шаблонов 433
Конструирование API при помощи шаблона 434
Спуск к следующему уровню: что именно представление передает в шаблон 436
Следующее техническое требование из уровня представлений: новые списки должны записывать владельца 437
Момент принятия решения: перейти к следующему уровню с неработающим тестом или нет 438
Спуск к уровню модели 439
Финальный шаг: подача .name API из шаблона 441

Глава 23. Изоляция тестов и «слушание своих тестов» 444

Пересмотр точки принятия решения: уровень представлений зависит от ненаписанного кода моделей ... 444
Первая попытка использования имитаций для изоляции ... 446
Использование имитации side_effects для проверки последовательности событий ... 447
Слушайте свои тесты: уродливые тесты сигнализируют о необходимости рефакторизации ... 449
Написание тестов по-новому для представления, которое будет полностью изолировано ... 450
Держите рядом старый комплект интегрированных тестов в качестве проверки на токсичность ... 450
Новый комплект тестов с полной изоляцией ... 451
Мышление с точки зрения взаимодействующих объектов ... 452
Спуск вниз на уровень форм ... 457
Продолжайте слушать свои тесты: удаление программного кода ORM из нашего приложения ... 458
Наконец, спуск вниз на уровень моделей ... 462
Назад к представлениям ... 464
Момент истины (и риски имитации) ... 466
Точка зрения о взаимодействиях между уровнями как о контрактах ... 467
Идентификация неявных контрактов ... 468
Исправление недосмотра ... 469
Еще один тест ... 471
Наведение порядка: что убрать из комплекта интегрированных тестов ... 472
Удаление избыточного кода на уровне форм ... 472
Удаление старой реализации представления ... 473
Удаление избыточного кода на уровне форм ... 474
Выводы: когда писать изолированные тесты, а когда – интегрированные ... 475
Пусть вычислительная сложность будет вашим гидом ... 476
Следует ли делать оба типа тестов? ... 477
Вперед! ... 477

Глава 24. Непрерывная интеграция ... 479

Инсталляция сервера Jenkins ... 479
Конфигурирование сервера Jenkins ... 481
Первоначальная разблокировка ... 481
Набор плагинов на первое время ... 481
Конфигурирование пользователя-администратора ... 482
Добавление плагинов ... 483
Указание серверу Jenkins, где искать Python 3 и Xvfb ... 483
Завершение с HTTPS ... 484
Настройка проекта ... 484
Первая сборка! ... 485
Установка виртуального дисплея, чтобы ФТ выполнялись бездисплейно ... 487
Взятие снимков экрана ... 489
Если сомневаетесь – встряхните тайм-аут! ... 492
Выполнение тестов JavaScript QUnit на Jenkins вместе с PhantomJS ... 494
Установка node ... 494

Добавление шагов сборки в Jenkins ... 495
Больше возможностей с CI-сервером ... 497

Глава 25. Социально зачимый кусок, шаблон проектирования «Страница» и упражнение для читателя ... 499

ФТ с многочисленными пользователями и addCleanup ... 499
Страничный шаблон проектирования ... 501
Расширение ФТ до второго пользователя и страница «Мои списки» ... 504
Упражнение для читателя ... 506

Глава 26. Быстрые тесты, медленные тесты и горячий поля ... 509

Тезис: модульные тесты сверхбыстры и к тому же хороши ... 511
Более быстрые тесты означают более быструю разработку ... 511
Священное состояние потока ... 511
Медленные тесты не выполняются часто, что приводит к плохому коду ... 512
Теперь у нас все в порядке, но интегрированные тесты со временем становятся медленнее ... 512
Не верьте мне ... 512
И модульные тесты управляют хорошей структурой кода ... 512
Проблемы с «чистыми» модульными тестами ... 512
Изолированные тесты труднее читать и писать ... 512
Изолированные тесты не тестируют интеграцию автоматически ... 513
Модульные тесты редко отлавливают неожиданные дефекты ... 513
Тесты с имитациями могут стать близко привязанными к реализации ... 513
Но все эти проблемы могут быть преодолены ... 514
Синтез: что мы хотим от всех наших тестов? ... 514
Правильность ... 514
Чистый код, удобный в сопровождении ... 514
Продуктивный поток операций ... 515
Оценивайте свои тесты относительно преимуществ, которые вы хотите от них получить ... 515
Архитектурные решения ... 516
Порты и адаптеры/шестиугольная/чистая архитектура ... 516
Функциональное ядро, императивная оболочка ... 517
Заключение ... 518
Дополнительные материалы для чтения ... 518

Повинуйтесь Билли-тестировщику! ... 521

Тестировать очень тяжело ... 521
Держите свои сборки CI-сервера на зеленом уровне ... 521
Гордитесь своими тестами так же, как своим программным кодом ... 522
Не забудьте дать на чай персоналу заведения ... 522
Не пропадайте! ... 522

Приложение A. PythonAnywhere ... 523

Выполнение сеансов Firefox Selenium при помощи Xvfb ... 523
Настройка Django как веб-приложение PythonAnywhere ... 525
Очистка папки /tmp ... 526

Снимки экрана ... 526
Глава о развертывании ... 526

Приложение B. Представления на основе классов в Django ... 528

Обобщенные представления на основе классов ... 528
Домашняя страница как FormView ... 529
Использование form_valid для индивидуализации CreateView ... 530
Более сложное представление для обработки просмотра и добавления к списку ... 533
Сравните старую верию с новой ... 536
Лучшие приемы модульного тестирования обобщенных представлений на основе классов ... 537

Приложение C. Обеспечение работы серверной среды при помощи Ansible ... 539

Установка системных пакетов и Nginx ... 539
Конфигурирование Gunicorn и использование обработчиков для перезапуска служб ... 541
Что делать дальше ... 542

Приложение D. Тестирование миграций базы данных ... 544

Попытка развертывания на промежуточном сервере ... 544
Выполнение тестовой миграции локально ... 545
Вставка миграции данных ... 546
Совместное тестирование новых миграций ... 547
Выводы ... 548

Приложение E. Разработка на основе поведения (BDD) ... 550

Что такое BDD? ... 550
Базовые служебные операции ... 551
Написание ФТ как компонента при помощи синтаксиса языка Gherkin ... 552
Программирование шаговых функций ... 553
Определение первого шага ... 555
Эквиваленты setUp и tearDown в environment.py ... 555
Еще один прогон ... 556
Извлечение параметров в шагах ... 556
BDD по сравнению с ФТ с локальными комментариями ... 559
BDD способствует написанию структурированного тестового кода ... 561
Страничный шаблон проектирования как альтернатива ... 561
BDD может быть менее выразительным, чем локальные комментарии ... 563
Будут ли непрограммисты писать тесты? ... 563
Некоторые предварительные выводы ... 564

Приложение F. Создание REST API: JSON, Ajax и имитирование на JavaScript ... 565

Наш подход для этого раздела ... 565
Выбор подхода к тестированию ... 566
Организация базовой конвейерной передачи ... 566
Получение фактического отклика ... 568

Добавление POST-метода ... 569
Тестирование клиентского Ajax при помощи Sinon.js ... 570
Соединение всего в шаблоне, чтобы убедиться, что это реально работает ... 574
Реализация Ajax POST-запроса, включая маркер CSRF ... 576
Имитация в JavaScript ... 578
Валидация данных. Упражнение для читателя ... 582

Приложение G. Django-Rest-Framework ... 587

Инсталляция ... 587
Сериализаторы (на самом деле – объекты ModelSerializer) ... 588
Объекты Viewset (на самом деле – объекты ModelViewset) и объекты Router ... 589
Другой URL для элемента с POST-запросом ... 592
Адаптирование на стороне клиента ... 593
Что дает инфраструктура Django-Rest-Framework ... 595

Приложение H. Шпаргалка ... 599

Начальная настройка проекта ... 599
Основной поток операций TDD ... 599
Выход за пределы тестирования только на сервере разработки ... 600
Общие приемы тестирования ... 601
Лучшие приемы на основе Selenium / функциональных тестов ... 601
«Снаружи внутрь», изоляция тестов против интегрированных тестов и имитация. 602

Приложение I. Что делать дальше ... 603

Уведомления – на сайте и по электронной почте ... 603
Переходите на Postgres ... 604
Выполняйте тесты относительно разных браузеров ... 604
Тесты на коды состояния 404 и 500 ... 604
Сайт администратора Django ... 604
Напишите несколько тестов защищенности ... 605
Тест на мягкую деградацию ... 605
Кэширование и тестирование и производительности ... 605
MVC-инфраструктуры для JavaScript ... 605
Async и протокол Websocket ... 606
Перейдите на использование py.test ... 606
Попробуйте coverage.py ... 606
Шифрование на стороне клиента ... 606
Здесь место для ваших предложений ... 607

Приложение J. Примеры исходного кода ... 608

Полный список ссылок для каждой главы ... 608
Использование Git для проверки вашего прогресса ... 610
Скачивание ZIP-файла для главы ... 611
Не позволяйте этому превращаться в костыль! ... 611

Предметный указатель ... 612

Предисловие

Эта книга стала результатом моей попытки рассказать миру о путешествии, которое я начал с «чистого хакерства» и в итоге пришел к «программной инженерии». В основном речь пойдет о тестировании, но повествование коснется многих других аспектов, в чем вы скоро сами убедитесь.

Я хочу поблагодарить вас за то, что взялись прочитать мою книгу.

Если вы ее приобрели, то я очень благодарен. Если вы читаете бесплатную онлайн-версию, я *по-прежнему* благодарен за то, что вы сочли ее достойной потраченного времени. Кто знает, возможно, добравшись до конца, вы решите, что она достаточно хороша, чтобы приобрести печатное издание для себя или друзей.

Если у вас есть какие-либо комментарии, вопросы или предложения, буду рад получить от вас известие. Меня можно найти непосредственно по ссылке obeythetestinggoat@gmail.com либо в Твиттере @hjwp. Можно также зайти на мой веб-сайт и блог (http://www.obeythetestinggoat.com/), есть также список рассылки[1].

Надеюсь, от прочтения настоящей книги вы получите удовольствие в той же степени, что и я, когда писал ее.

Почему я посвятил книгу разработке на основе тестирования

«Кто ты такой, почему ты пишешь эту книгу и почему я должен ее прочитать?» – спросите вы.

Я все еще нахожусь в самом начале карьеры программиста. Говорят, в любой дисциплине человек проходит путь от ученика до подмастерья и в конечном счете (иногда) становится мастером. Должен сказать, что я, в лучшем случае, программист-путешественник. Но в самом начале своей карьеры мне посчастливилось попасть в группу фанатиков методологии TDD, и это оказало такое огромное влияние на мой стиль программирования, что я испытываю желанием поделиться им со всеми. Можно сказать, горю энтузиазмом новообращенного – опыт обучения свеж в моей памяти, поэтому я надеюсь, что все еще способен хорошо понимать новичков.

Когда я приступил к изучению Python (опираясь на превосходную книгу Марка Пилгрима *«Погружение в Python»*), я столкнулся с понятием TDD и подумал: «Определенно, в этом есть смысл». Возможно, у вас была подобная реакция, когда вы впервые услышали про TDD? Этот подход и правда выглядит разумным, действительно хорошая привычка – все равно что регулярно пользоваться зубной нитью или что-то в этом роде.

---

[1] См. https://groups.google.com/forum/%23!forum/obey-the-testing-goat-book

Затем подоспел мой первый большой проект. Вы можете догадаться, что произошло: клиент, сроки, уйма работы – и все мои благие намерения по поводу TDD отправились прямиком в урну.

По сути, все шло прекрасно. Я был в порядке.

Сначала.

Я понимал, что в сущности TDD мне не нужен, потому что это был небольшой веб-сайт и я мог в ручном режиме легко проверить, что все работает. Нажми на ссылку *здесь*, выбери из выпадающего списка *там* и получи *это* – все просто! Вся эта затея с написанием тестов, казалось, будет занимать *вечность*. Кроме того, с высоты своих трех недель зрелого опыта программирования я мнил себя довольно хорошим программистом. Я мог справляться с работой. Легко.

Затем пробил час грозной богини Сложности. Она-то и показала мне пределы моего опыта.

Проект рос. Одни части системы начинали зависеть от других. Я прилагал все усилия, чтобы следовать хорошим принципам, таким как DRY (Don't Repeat Yourself – не повторяйся), но в итоге он завел меня на довольно опасную территорию. Вскоре я уже был в окружении множественного наследования. Иерархии классов глубиной восемь уровней. Операторов `eval`.

Я начал испытывать страх перед внесением изменений в свой код. Я больше не был уверен, что понимаю, что от чего зависит и что может произойти, если изменить этот код в этом месте. Проклятие! Кажется, вот тот кусок наследует отсюда. Да нет же, он переопределен! Стоп, так он же зависит вон от той переменной класса. Верно! Ну, в общем, пока я переопределяю переопределение, все должно прекрасно работать. Просто проверю и все.

Но проверка становилась все труднее. Теперь у моего сайта было много разделов, и щелканье по ним вручную становилось непрактичным. Лучше все оставить в покое, забыть о рефакторизации и обойтись тем, что есть.

Вскоре у меня уже была отвратительная, уродливая мешанина кода. Внесение в него новых наработок стало болезненным.

Некоторое время спустя мне посчастливилось получить задание от компании Resolver Systems (теперь это PythonAnywhere[2]), где экстремальное программирование (XP) было нормой. Они представили меня строгой методологии TDD.

Несмотря на то что предыдущий опыт, конечно же, открыл мой ум для возможных преимуществ автоматизированного тестирования, я все еще волочил ноги на каждом этапе. «Признаю, тестирование в целом может быть неплохой идеей, но постойте! Все эти тесты? Некоторые из них выглядят как тотальная трата времени... Что? Функциональные тесты *вместе с* модульными? Да ладно вам, ну вы загнули! И весь этот цикл TDD «тест/

[2] См. https://www.pythonanywhere.com/

минимальное-изменение-кода/тест»... Это же просто глупо! Кому нужны все эти крохотные шажочки! Давайте-ка посмотрим, чей ответ верный и просто перейдем к финалу!»

Поверьте, я подвергал сомнению каждое правило, предлагал всякого рода короткие пути, требовал обоснований по любому, на первый взгляд бессмысленному, аспекту TDD... В итоге я увидел в этом здравый смысл. Я потерял счет числу раз, когда я ловил себя на мысли: «Спасибо вам, тесты», поскольку функциональный тест раскрывает регрессию, которую мы бы никогда не предсказали, и модульный тест спасает меня от действительно глупой логической ошибки. В психологическом отношении методология TDD сделала разработку гораздо менее напряженным процессом. Она порождает программный код, работать с которым одно удовольствие.

Так что позвольте рассказать вам об этом *все*, что я знаю!

Цели этой книги

Моя главная цель состоит в том, чтобы донести до вас методологию ОЛНЯТЬ веб-разработку, которая, думаю, позволяет создавать самые лучшие веб-приложения и делает разработчиков счастливее. Не много толка от книги, предлагающей материал, который вы легко можете найти в Гугле. Так что эта книга *как таковая* не является руководством по синтаксису Python или учебным руководством по веб-разработке. Вместо этого я надеюсь научить вас использовать методологию TDD, чтобы более надежным образом прийти к нашей общей священной цели: *чистому коду, который работает*.

С учетом всего сказанного, я постоянно буду обращаться к реальным практическим примерам, создавая веб-приложение с нуля с использованием таких инструментов, как Django, Selenium, jQuery и Mock. Я не исхожу из ваших предварительных знаний ни об одном из них, поэтому вам следует обратиться к концу этой книги, где вы найдете достаточное введение в эти инструменты, а также в дисциплину TDD.

В экстремальном программировании мы всегда имеем дело с парным программированием, и, занимаясь написанием данной книги, я представлял, будто общаюсь в паре с самим собой прошлым, объясняя, каким образом работают инструменты, и отвечая на вопросы, почему мы программируем именно так, а не иначе. Так что если я когда-нибудь приму нечто вроде покровительственного тона, это только потому, что я вовсе не такой умный и должен себя хорошенько сдерживать. А если начну оправдываться, то потому, что я похож на зануду, который систематически не соглашается, кто что бы ни говорил, поэтому иногда приходится много привести много обоснований, чтобы убедиться в чем-либо.

Схема

Я разделил эту книгу на три части:

Первая часть (главы 1–7) – основы

Погружает прямо в создание простого веб-приложения с использованием методологии TDD. Мы начинаем с написания функционального теста (при помощи Selenium), затем проходим основы Django – модели, представления, шаблоны – со строгим модульным тестированием на каждом этапе. Также я представляю Билли-тестировщика.

Вторая часть (главы 8–17) – существенные аспекты веб-разработки

Охватывает некоторые более сложные, но неизбежные аспекты веб-разработки; показывает, как тестирование может в этом помочь: статические файлы, развертывание в производственной среде, валидация данных в формах, миграция баз данных и ужасающий JavaScript.

Часть III (главы 18–26) – более продвинутые темы тестирования

Применение объектов-имитаций, интеграция сторонней системы, тестовые фикстуры, TDD с подходом «снаружи-внутрь» и непрерывная интеграция (CI).

Переходим к некоторым служебным операциям…

Условные обозначения, принятые в книге

В книге используются следующие условные обозначения:

Курсив

Указывает новые термины, URL-адреса, адреса электронной почты, имена и расширения файлов.

`Моноширный шрифт`

Используется для распечаток программ, а также внутри абзацев для ссылки на элементы программ, такие как переменные или имена функций, базы данных, типы данных, переменные окружающей среды, операторы и ключевые слова.

`Жирный моноширный шрифт`

Показывает команды либо другой текст, который должен быть напечатан пользователем, а также ключевые слова в программном коде.

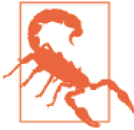
Иногда я буду использовать символ:

`[...]`

чтобы обозначить пропущенную часть содержимого с целью сократить куски результатов работы или перейти к соответствующему разделу.

Данный элемент обозначает подсказку или совет.

Общее замечание или ремарка.



Предупреждение или предостережение.

Предпосылки и предположения

Вот краткое описание того, что должен знать читатель этой книги и какое программное обеспечение ему потребуется.

Python 3 и программирование

Я попытался написать эту книгу, ориентируясь на начинающих программистов, но если вы в программировании новичок, то я исхожу из того, что вы уже изучили основы Python. Поэтому, если вы еще не приступили, убедительно советую прочесть учебник по Python для начинающих или ознакомительную книгу, например «*Погружение в Python*» (Dive Into Python) или «*Python на горьком опыте*» (Learn Python the Hard Way) или только для развлечения «*Придумывайте свои компьютерные игры вместе с Python*» (Invent Your Own Computer Games with Python) – все эти книги являются превосходными введениями[1].

Если вы – опытный программист, но новичок в Python, вы отлично поладите. Python удивительно доступен для понимания.

В настоящей книге я использую *Python 3*. На тот момент, когда я писал ее в 2013–2014 годах, Python 3 уже существовал несколько лет, и тогда мир стоял перед выбором: какой версии Python отдать предпочтение. Вы сможете руководствоваться текстом книги, работая в Mac, Windows или Linux. Подробные инструкции по инсталляции в каждой ОС следуют ниже.

Эта книга была протестирована для Python 3.6. Если у вас более ранняя версия, вы найдете незначительные различия (синтаксис f-string, например), поэтому будет лучше, если вы обновитесь, по возможности.

Я бы не рекомендовал пытаться использовать Python 2, поскольку различия между двумя версиями весьма существенные. Вы, конечно, сможете применить все знания, полученные из этой книги, если ваш следующий проект окажется на основе Python 2. Но временные затраты на выяснение, в чем причина того, что результаты работы вашей программы отличаются от моих – Python 2 или же все-таки фактическая ошибка, которую вы допустили сами, – не будут продуктивными.

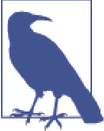
Если вы раздумываете, использовать *PythonAnywhere* (PaaS-стартап, на который я работаю)[2] или локально установленный Python, то перед началом работы вам следует пройти в конец книги и ознакомиться с Приложением А.

---

[1] См. библиографию – *Прим. перев.*

[2] См. https://www.pythonanywhere.com/

В любом случае вам потребуется доступ к Python и вы должны знать, как запускать его из командной строки, как редактировать файлы Python и их выполнять. И еще раз: взгляните на те три книги, которые я порекомендовал ранее, если вы находитесь в сомнении.



Если у вас уже установлен Python 2 и вы переживаете, что установка Python 3 как-то ему навредит, не беспокойтесь! Python 3 и 2 могут сосуществовать мирно в одной системе, особенно при использовании virtualenv, которым мы воспользуемся.

Как работает HTML

Также я исхожу, что у вас есть базовые знания о том, как работает веб: что такое HTML, POST-запрос и т. д. Если вы в этом не уверены, изучите элементарное руководство по HTML; некоторые можно найти на http://www.webplatform.org/. Если вы способны создать страницу HTML на ПК, просмотреть ее в браузере, а также понимаете, что такое форма и как она может работать, то у вас, похоже, все в порядке.

Django

Настоящая книга использует программную инфраструктуру (фреймворк) Django, которая является, пожалуй, самой известной в мире системой разработки веб-приложений на Python. Я написал книгу, исходя из того, что у читателя нет предварительных знаний о Django, но если вы новичок в Python *и* веб-разработке *и* тестировании, можете порой замечать, что приходится пробовать и принимать на борт слишком много тем и наборов понятий, чтобы их все охватить. Если это так, рекомендую делать перерывы в чтении книги и переключаться на учебники по Django. DjangoGirls – лучший из них, это самое дружественное пособие для новичков, которое я знаю. Официальное учебное руководство превосходно подойдет для более опытных программистов[3].

Смотрите ниже инструкции по установке Django.

JavaScript

Во второй половине книги мы кратко остановимся на JavaScript. Если вы новичок в JavaScript, не переживайте: если вы окажетесь в небольшом замешательстве, то в той части книги я порекомендую несколько соответствующих руководств.

[3] См. https://tutorial.djangogirls.org/ и https://docs.djangoproject.com/en/1.11/intro/tutorial01/

Примечание по IDE

Если вы пришли из мира Java или .NET, возможно, вам будет интересно использовать интегрированную среду разработки (IDE) для программирования на Python. Они располагают разнообразными полезными инструментами, включая интеграцию с системой управления версиями (VCS). Среди них есть несколько превосходных, предназначенных как раз для Python. Я сам использовал одну из них, когда только начинал, она очень пригодилась для моих первых двух проектов.

Могу ли я полагагать (и это только предположение), что вы не используете IDE, по крайней мере на время чтения этого учебного пособия? IDE гораздо менее востребованы в мире Python, и я написал об этом целую книгу, допустив, что вы просто используете элементарный текстовый редактор и командную оболочку. Иногда это все, что есть (когда вы работаете с сервером, например), поэтому всегда сначала стоит учиться использовать базовые инструменты, понимать, как они работают. Эти инструменты будут всегда под рукой, даже если после изучения этой книги вы решите вернуться к вашему IDE и всем его полезным инструментам.

Инсталлируемое программное обеспечение

Кроме Python, вам понадобятся:

Веб-браузер Firefox
: Selenium может управлять любым из основных браузеров, но для примера лучше всего воспользоваться Firefox, потому что он доказал свою надежную кросс-платформенность и, в качестве бонуса, меньше всего продается корпоративным клиентам.

Система управления версиями Git
: Она доступна для любой платформы и расположена на `http://git-scm.com/`. В Windows она поставляется вместе с командной оболочкой **Bash**, которая необходима для настоящей книги.

virtualenv с Python 3, Django 1.11 вместе с Selenium 3
: Python’овская среда окружения virtualenv и инструменты менеджера пакетов pip поставляются в комплекте начиная с Python 3.4+ (так было не всегда, так что гип-гип-ура). Подробные инструкции по подготовке virtualenv последуют далее.

Geckodriver
: Это драйвер, который позволит нам удаленно управлять Firefox через Selenium. Я дам ссылку на скачивание в разделе «Инсталляция Firefox» ниже.

Примечания по поводу Windows

В мире с открытым исходным кодом пользователи Windows иногда могут чувствовать себя немного обделенными вниманием, поскольку OS X и Linux там настолько доминируют, что легко забывается о существовании мира вне парадигмы Unix. Обратные каталожные разделители? Буквенные метки дисков? Что?! Тем не менее вполне можно следовать по тексту книги, работая в Windows. Вот несколько подсказок:

1. Устанавливая Git для Windows, убедитесь, что вы выбрали **Run Git and included Unix tools from the Windows command prompt** (Выполнять Git вместе с включенными средствами Unix для командной строки Windows). Далее вы получите доступ к программе Git Bash. Используйте ее в качестве основной командной оболочки, и вы получите все полезные инструменты командной строки GNU, такие как `ls`, `touch` и `grep`, плюс прямые каталожные разделители.
2. Также в установщике Git выберите **Use Windows' default console** (Использовать консоль Windows по умолчанию), иначе Python не будет работать должным образом в окне Git-Bash.
3. Когда вы устанавливаете Python 3 при уже установленном Python 2 и хотите его сохранить в качестве среды по умолчанию, выберите опцию **Add Python 3.6 to PATH** (Добавьте Python 3.6 к системному пути PATH), как на рис. р-1, чтобы можно было легко выполнять Python из командной строки.

Рис. р-1. Добавьте Python к системному пути из установщика

Проверьте, что все это сделано: вы можете открывать командную оболочку Git-Bash и запускать Python или pip из любой папки.

Примечания по поводу MacOS

MacOS немного более вменяема, чем Windows, несмотря на то что до сего времени установка `pip` по-прежнему представляла определенную сложность. С момента появления версии 3.4 все стало довольно прямолинейным:

- Python 3.6 установится без суеты при помощи загружаемого установщика (http://www.python.org/). Он автоматически установит `pip`.
- Установщик Git тоже должен сразу заработать.

Так же как в Windows, проверкой работоспособности будет то, что вы можете открыть терминал и просто выполнить `git`, `python3` или `pip` из любого места. Если вы столкнетесь с какой-либо проблемой, то поиск «системный путь» и «команда не найдена» должны обеспечить хорошие ресурсы для устранения неисправностей.

Возможно, вы захотите обратиться к Homebrew (http://brew.sh/). Раньше он был единственным надежным средством установки в Mac[4] большого количества инструментов Unixy (включая Python 3). Несмотря на то что с нормальным установщиком Python теперь все в порядке, Homebrew может пригодиться в будущем. Правда, он потребует скачать весь 1,1 Гб XCode, но он также предоставит вам компилятор C, который будет полезным побочным продуктом.

Редактор Git по умолчанию и другие базовые настройки Git

Я предоставлю пошаговые инструкции для Git, но сейчас предлагаю немного заняться конфигурированием. Например, когда вы в первый раз будете фиксировать (комитить) внесенные изменения в репозитории, по умолчанию появится текстовый редактор *vi*, и вы, возможно, не будете иметь понятия, что с ним сделать. Так вот, поскольку vi имеет два режима, у вас, соответственно, будет два варианта. Первый – изучить минимальное количество команд vi (*нажать клавишу i, чтобы войти в режим вставки, набрать текст, нажать* `<Esc>` *для возвращения в нормальный режим, затем записать файл и выйти при помощи* `:wq<Enter>`). В результате вы присоединитесь к большому братству людей, которые знают этот древний, уважаемый текстовый редактор.

Или же вы можете решительно отказаться участвовать в таком смехотворном откате к 1970-м и сконфигурировать Git для использования редактора по вашему выбору. Выйдите из vi при помощи клавиши `<Esc>`, со-

[4] Правда, я бы не рекомендовал устанавливать Firefox посредством Homebrew :`brew` помещает бинарный файл Firefox в непонятное место, и это приводит Selenium в замешательство. Вы сможете работать в обход этого, но проще просто установить Firefox обычным способом.

провождаемой `:q!`, затем поменяйте на свой редактор Git по умолчанию. Смотрите документацию по Git относительно процедуры базовой конфигурации Git[5].

Инсталляция Firefox и Geckodriver

Firefox можно скачать для Windows и OSX с https://www.mozilla.org/firefox/. В Linux вы, вероятно, уже его установили, в противном случае ваш диспетчер пакетов будет его иметь .

Geckodriver доступен на https://github.com/mozilla/geckodriver/releases. Вам нужно его скачать, извлечь и поместить где-нибудь в системном пути:

- в OSX или Linux я рекомендую положить его в `~/.local/bin`;
- в Windows положите его в вашу папку Python «Scripts».

Чтобы проверить, что все работает, откройте консоль Bash, и у вас должно получиться выполнить следующее:

```
geckodriver --version
geckodriver 0.17.0

The source code of this program is available at
https://github.com/mozilla/geckodriver.

This program is subject to the terms of the Mozilla Public License 2.0.
You can obtain a copy of the license at https://mozilla.org/MPL/2.0/.
```

Если это не работает, возможно, `~/.local/bin` не находится в вашем системном пути `PATH` (это относится к некоторым системам Mac и Linux). Неплохо иметь эту папку в вашем системном пути, потому что именно туда Python будет устанавливать компоненты, когда вы будете использовать команду `pip install --user`. Вот как добавить его в ваш `.bashrc`:

```
echo 'PATH=~/.local/bin:$PATH' >> ~/.bashrc
```

Закройте свой терминал, откройте снова и убедитесь, что `geckodriver --version` теперь работает.

Настройка виртуальной среды virtualenv

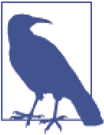
В Python виртуальная среда virtualenv (сокращенно от virtual environment) – это то, как вы настраиваете свою программную среду для различных проектов Python. Она позволяет использовать в каждом проекте разные пакеты: например, разные версии Django и даже разные версии Python.

[5] См. http://git-scm.com/book/en/Customizing-Git-Git-Configuration.

И поскольку вы не устанавливаете компоненты в масштабе всей системы, это означает, что вам не требуются полномочия root.

Virtualenv входит в Python с версии 3.4, но я всегда рекомендую вспомогательный инструмент под названием virtualenvwrapper. Давайте сначала установим его (не важно, для какой версии Python вы его устанавливаете):

```
# в Windows
pip install virtualenvwrapper
# в MacOS / Linux
pip install --user virtualenvwrapper
# затем дайте Bash загрузить virtualenvwrapper автоматически
echo "source virtualenvwrapper.sh" >> ~/.bashrc
source ~/.bashrc
```



В Windows `virtualenvwrapper` будет работать только в оболочке Git-Bash, не из обычной командной строки.

Virtualenvwrapper хранит все ваши виртуальные среды virtualenv в одном месте и обеспечивает удобные инструменты для их активации и деактивации.

Давайте создадим виртуальную среду superlists[6], в которой установлен Python 3:

```
# в MacOS/Linux:
mkvirtualenv --python=python3.6 superlists
# в Windows
mkvirtualenv --python=`py -3.6 -c"import sys; print(sys.executable)"`
superlists
# (небольшой трюк, чтобы убедиться, что у нас virtualenv в версии python 3.6)
```

Активация и деактивация virtualenv

Каждый раз при работе с книгой у вас будет возникать желание убедиться, что ваша виртуальная среда virtualenv активна. Об этом можно узнать по тому, что в командной строке вы увидите `(superlists)` в скобках. Что-то вроде этого:

```
$
(superlists) $
```

Сразу после создания виртуальной среды virtualenv она будет активной. Это можно перепроверить, выполнив `which python`:

[6] Слышу, как вы говорите: «Почему superlists?» Никаких спойлеров! Вы узнаете в следующей главе.

```
(superlists) $ which python
/home/harry/.virtualenvs/superlists/bin/python
# (в Windows это будет что-то вроде
# /C/Users/IEUser/.virtualenvs/superlists/Scripts/python)

(superlists) $ deactivate
$ which python
/usr/bin/python
$ python --version
Python 2.7.12 # у меня вне virtualenv команда python по умолчанию покажет Python 2.

$ workon superlists
(superlists) $ which python
/home/harry/.virtualenvs/superlists/bin/python
(superlists) $ python --version
Python 3.6.0
```

Чтобы активировать виртуальную среду virtualenv, нужно выполнить команду `workon superlists`. Чтобы проверить, активна ли она, ищите в командной строке `(superlists) $` либо выполните `which python`.

Инсталляция Django и Selenium

Теперь мы установим Django 1.11 и последнюю версию Selenium – Selenium3.

```
(superlists) $ pip install "django==1.12" "selenium<4"
Collecting django==1.11.3
  Using cached Django-1.11.3-py2.py3-none-any.whl
Collecting selenium>3
  Using cached selenium-3.4.3-py2.py3-none-any.whl
Installing collected packages: django, selenium
Successfully installed django-1.11.3 selenium-3.4.3
```

Некоторые сообщения об ошибках, которые вы, вероятно, увидите, когда *неминуемо* провалите активацию вашего virtualenv

Если вы новичок в virtualenv или, по правде говоря, даже если нет, в какой-то момент вы *гарантированно* забудете ее активировать и уставитесь в сообщение об ошибке. Со мной это случается постоянно. Вот примеры того, что вы увидите:

```
ImportError: No module named selenium
```

Или:

```
ImportError: No module named django.core.management
```

Как всегда, смотрите, чтобы в вашей командной оболочке было `(superlists)` и быстрая команда `workon superlists`, вероятно, позволит вернуть среду в рабочее состояние.

Вот еще парочка для пущей достоверности:

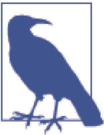
```
bash: workon: command not found
```

Это сообщение означает, что вы перешли на шаг раньше и не добавили virtualenvwrapper к своему *.bashrc*. Выше найдите команды `echo source virtualenvwrapper.sh` и выполните их повторно.

```
'workon' is not recognized as an internal or external command,
operable program or batch file.
```

Вы запустили стандартную командную оболочку Windows – `cmd` – вместо Git-Bash. Закройте ее и откройте последнюю.

Успешного программирования!



Помогли ли вам эти инструкции? Или у вас есть кое-что получше? Свяжитесь со мной по адресу: obeythetestinggoat@gmail.com!

Сопутствующее видео

В качестве сопровождения для настоящей книги я записал видео из 10 частей[1]. Видео охватывает содержание глав с 1 по 6. Если вы лучше усваиваете видеоматериал – приглашаю вас обратиться к нему. В дополнение к тому, что имеется в книге, этот материал позволит вам почувствовать, что собой представляет “поток” TDD, дав возможность переключаться между тестами и кодом, получать объяснения мыслительного процесса по мере продвижения.

Плюс я ношу восхитительную желтую футболку.

[1] См. http://oreil.ly/1svTFqB.

Признательности

Следует поблагодарить многих людей, без которых эта книга никогда бы не появилась и/или была бы еще хуже, чем есть.

Огромное спасибо в первую очередь Грегу в $OTHER_PUBLISHER. Он был первым, кто вселил в меня уверенность, что она может быть осуществлена. Несмотря на то что у его работодателей оказались чрезмерно регрессивные представления об авторском праве, я всегда буду благодарен, что он в меня поверил.

Спасибо Майклу Фурду – еще одному бывшему сотруднику Resolver Systems – за то, что он стал примером человека, который первым решился написать книгу, и спасибо за его непрекращающуюся поддержку данного проекта. Также хочу поблагодарить своего босса Гайлса Томаса за то, что сглупил, позволив еще одному из своих сотрудников написать книгу (хотя, полагаю, он теперь внес в стандартный трудовой договор поправку «никаких книг»). Ему также спасибо за непрекращающееся здравомыслие и за то, что поставил меня на путь тестирования.

Благодарю своих других коллег, Гленна Джоунса и Хансела Данлопа, за то, что были моими неоценимыми пробными слушателями, и за их терпение к моему заезженному разговору весь прошлый год.

Спасибо моей жене Клементине и обеим моим семьям, без поддержки и терпения которых я бы никогда не справился. Прошу прощения за все то время, когда я с головой погружался в компьютере, вместо участия в незабываемых семейных событиях. Когда я пустился во все тяжкие, я понятия не имел, что эта книга сделает с моей жизнью («Пиши ее в свое свободное время, говоришь? Надо подумать...»). Вероятно, я не написал бы ее без вас.

Благодарю своих технических рецензентов, Джонатана Хартли, Николаса Толлерви и Эмили Бах, за их поддержку и бесценную обратную связь. В особенности Эмили, которая в действительности добросовестно прочитала каждую главу. Частичная похвала Нику и Джону, но она все же должна рассматриваться как вечная благодарность. Ваше присутствие сделало всю эту затею не такой одинокой. Без вас эта книга была бы не более чем бессмысленной болтовней идиота.

Спасибо всем, кто уделил мне часть своего времени, чтобы поделиться мнением о книге просто так, по доброте душевной: Гэри Бернхард, Марк Лавин, Матт Одоннел, Майкл Фурд, Хинек Шлавак, Рассел Кейт-Магей, Эндрю Годуин, Кеннет Рейтц и Натан Стокс. Спасибо, что вы намного умнее меня и что не дали мне сказать несколько глупых вещей. Разумеется, в книге осталось еще много глупостей, за которые вы все не можете нести абсолютно никакой ответственности.

Благодарю своего редактора Меган Бланшетт за то, что была очень дружелюбным и симпатичным надзирателем, за то, что следила за процессом написания книги с точки зрения сроков и ограничения моих более глупых идей. Благодарю всех остальных в издательстве O'Reilly за их советы, в том числе Сару Шнайдер, Кару Эбраим и Дэна Фокссмита за том, что дали возможность сохранить британский вариант английского языка. Благодарю Чарльза Румелиотиса за его советы относительно стиля и грамматики. Мы никогда не сойдемся во взглядах насчет достоинств правил цитирования/пунктуации Чикагской школы, но, должен заве-

рить, я рад, что вы были рядом. И спасибо отделу дизайна за предоставленного нам животного для обложки!

Самую большую благодарность хочу высказать всем моим читателям предварительного выпуска книги за выявленные опечатки, советы и предложения, за все то, чем они помогли сгладить кривую обучения в книге, и больше всего – за их добрые слова ободрения и поддержки, которые помогали мне в работе. Спасибо Jason Wirth, Dave Pawson, Jeff Orr, Kevin De Baere, crainbf, dsisson, Galeran, Michael Allan, James O'Donnell, Marek Turnovec, SoonerBourne, julz, Cody Farmer, William Vincent, Trey Hunner, David Souther, Tom Perkin, Котелок Sorcha, Jon Poler, Charles Quast, Siddhartha Naithani, Steve Young, Roger Camargo, Wesley Hansen, Johansen Christian Vermeer, Ian Laurain, Sean Robertson, Хари Jayaram, Bayard Randel, Konrad Korżel, Matthew Waller, Julian Harley, Barry McClendon, Simon Jakobi, Angelo Cordon, Jyrki Kajala, джайн Manish, Mahadevan Sreenivasan, Konrad Korżel, Deric Crago, Cosmo Smith, Markus Kemmerling, Andrea Costantini, Daniel Patrick, Ryan Allen, Jason Selby, Greg Vaughan, Jonathan Sundqvist, Richard Bailey, Diane Soini, Dale Stewart, Mark Keaton, Johan Warlander, Simon Scarfe, Eric Grannan, MarcAnthony Taylor, Maria McKinley, John McKenna, Rafai Szymanski, Roel van der Goot, Ignacio Reguero, TJ Tolton, Jonathan Means, Theodor Nolte, Луна Jungsoo, Craig Cook, Gabriel Ewilazarus, Vincenzo Pandolfo, David "farbish2", Nico Coetzee, Daniel Gonzalez, Jared Contrascere, Zhao Mro и многих-многих других. Если я пропустил чье-то имя, вы имеете абсолютное право расстроиться; я невероятно благодарен и вам, так что напишите мне, и я попытаюсь загладить свою вину любым возможным способом.

И наконец благодарю вас, последний читатель, за то, что обратили внимание на эту книгу! Надеюсь, она вам понравится.

Дополнительные благодарности относительно второго издания

Большое спасибо моему замечательному редактору второго издания Нэну Барберу, а также Сьюзен Конант, Кристен Браун и всей команде O'Reilly. И еще раз выражаю свою благодарность Эмили и Джонатану за техническую рецензию, а также Эдварду Уонгу за его подробные замечания. Все остальные ошибки и несоответствия остаются на моей совести.

Также благодарю читателей свободного издания книги, которые внесли комментарии и предложения, а некоторые даже сделали запрос на включение кода. В этом списке я определенно мог кого-то из вас пропустить, так что прошу извинить, если вашего имени здесь нет, и тем не менее, благодарю Emre Gonulates, Jésus Gómez, Jordon Birk, James Evans, Iain Houston, Jason DeWitt, Ronnie Raney, Spencer Ogden, Suresh Nimbalkar, Darius, Caco, LeBodro, Jeff, wasabigeek, joegnis, Lars, Mustafa, Jared, Craig, Sorcha, TJ, Ignacio, Roel, Justyna, Nathan, Andrea, Alexandr, bilyanhadzhi, mosegontar, sfarzy, henziger, hunterji, das-g, juanriaza, GeoWill, Windsooon, gonulate и многих-многих других.

Часть I

Основы TDD и Django

В этой первой части я собираюсь представить азы разработки на основе тестирования (TDD от англ. Test-Driven Development). Мы разработаем реальное веб-приложение с нуля, на каждом этапе создавая сначала тесты.

Мы рассмотрим функциональное тестирование с использованием Selenium, а также модульное тестирование и увидим между ними разницу. Я представлю поток операций TDD – то, что я называю циклом «модульный-тест/программный-код». Мы выполним небольшую рефакторизацию и увидим, как она укладывается в TDD. Поскольку система управления версиями – абсолютно необходимый элемент серьезной программной инженерии, я буду также использовать Git. Мы обсудим, как и когда фиксировать изменения и интегрировать их с потоком операций веб-разработки и TDD.

Мы будем использовать Django – пожалуй, самую популярную в мире Python'овскую программную инфраструктуру для веб-разработки. Я старался представлять понятия Django медленно и по одному и приводить большое количество ссылок на дополнительные материалы для чтения. Если вы начинаете работать с Django с абсолютного нуля, убедительно рекомендую не торопиться с их изучением. Если вы почувствуете, что немного заблудились, уделите пару часов, чтобы просмотреть официальное учебное руководство по Django, а затем возвращайтесь к этой книге.

Вы также познакомитесь с Билли-тестировщиком…

Будьте осторожны с копированием и вставкой

Если вы работаете с цифровой версией книги, то вполне естественным будет желание копипастить распечатки программного кода из книги по мере того, как вы читаете. Будет намного лучше, если вы этого не будете делать: набор кода вручную остается в вашей мышечной памяти, да и сам по себе ощущается намного реальнее. Вы также неизбежно сделаете случайные опечатки, исправление которых является важной составной частью обучения.

Совершенно отдельно от этого вы обнаружите, что причуды формата PDF выражаются в том, что, когда вы пытаетесь из него скопировать/вставить, иногда происходят странные вещи…

Глава 1

Настройка Django с использованием функционального теста

TDD – это не то, что происходит естественным образом. Это дисциплина, подобная боевому искусству, и точно так же, как в фильме про кунг фу, вам нужен раздражительный и неблагоразумный наставник, который будет заставлять вас постигать дисциплину. Нашим наставником будет Билли-тестировщик.

Повинуйтесь Билли-тестировщику! Ничего не делайте, пока у вас не будет теста

Горный козел по прозвищу Билли-тестировщик – это неофициальный талисман TDD в сообществе тестирования Python. Для разных людей он, вероятно, имеет разное значение[1], но для меня Билли-тестировщик – это внутренний голос, который не дает мне сбиться с Истинного пути тестирования – подобно одному из тех ангелочков или дьяволят, которые выскакивают из-за плеча героев мультфильмов, но с очень своеобразным набором озабоченностей. Я надеюсь с помощью этой книги внедрить Билли-тестировщика и в вашу голову тоже.

Мы решили создать веб-сайт, хотя не совсем уверены, зачем он нужен. В веб-разработке обычно первый шаг состоит в инсталляции и конфигурировании вашей веб-платформы. *Скачайте это, установите то, сконфигурируйте другое, выполните сценарий*... Но TDD требует совсем другого образа мыслей. Когда вы выполняете TDD, у вас все время на уме Билли-тестировщик – целенаправленный, как все горные козлы, блеющий: «Сначала тест, сначала тест!»

В TDD первым шагом всегда является одно и то же: *написать тест*.

---

[1] Кашмирский козел является официальным талисманом пехотных подразделений Королевских валлийцев Британской армии – *Прим. перев.*

Сначала мы пишем тест, *затем* мы его выполняем и убеждаемся, что он не проходит, как ожидалось. И *только потом* мы идем вперед и создаем часть нашего приложения. Повторите это себе голосом Билли.

Еще одна вещь, которая касается сородичей нашего наставника, заключается в том, что они двигаются постепенно, шаг за шагом. Вот почему они редко падают с гор, гляньте: не важно, на какой крутизне они стоят. Как вы можете убедиться из рис. 1-1.

Рис. 1-1. Горные козлы гибче, чем вы думаете (источник: Caitlin Stewart, на Flickr, http://www.flickr.com/photos/caitlinstewart/2846642630/)

Мы будем продвигаться небольшими шажками и для создания нашего приложения будем использовать Django – популярную программную инфраструктуру для разработки веб-приложений на Python.

Первое, что мы хотим сделать, – это проверить, что Django у нас установлена и готова к работе. *Способ* проверки сводится к подтверждению, что мы можем завести сервер разработки Django и воочию убедиться, что

он раздает веб-страницу на наш веб-браузер на нашем локальном ПК[2]. Для этого воспользуемся средством автоматизации браузеров *Selenium*.

Создайте новый файл Python под названием *functional_tests.py* в любом месте, где вы хотите хранить программный код своего проекта, и введите следующий исходный код. Если во время работы вы испытываете желание издать несколько восклицаний голосом Билли, то это наверняка поможет:

functional_tests.py

```
from selenium import webdriver

browser = webdriver.Firefox()
browser.get('http://localhost:8000')

assert 'Django' in browser.title
```

Это наш первый *функциональный тест* (ФТ); о том, что я подразумеваю под функциональными тестами и как они контрастируют с модульными тестами, я буду рассказывать много. Пока же достаточно просто убедиться, что мы понимаем, что он делает:

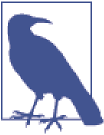
- Запускает «webdriver» Selenium, чтобы появилось окно реального браузера Firefox.
- Использует его для открытия веб-страницы, которая, как мы ожидаем, будет роздана из локального ПК.
- Проверяет (выполняя тестовое утверждение), что в заголовке страница есть слово «Django».

Попробуем его выполнить:

```
$ python functional_tests.py
  File ".../selenium/webdriver/remote/webdriver.py", line 268, in get
    self.execute(Command.GET, {'url': url})
  File ".../selenium/webdriver/remote/webdriver.py", line 256, in execute
    self.error_handler.check_response(response)
  File ".../selenium/webdriver/remote/errorhandler.py", line 194, in
check_response
    raise exception_class(message, screen, stacktrace)
selenium.common.exceptions.WebDriverException: Message: Reached error page: abo
ut:neterror?e=connectionFailure&u=http%3A//localhost%3A8000/[...]
```

[2] Здесь и далее в тексте процесс разработки конечного программного продукта подразделяется на три этапа: этап непосредственной разработки (development), этап подготовки к внедрению, или промежуточный этап (staging), и этап внедрения в производственную среду, или производственный этап (production). Соответственно, на каждом этапе используется свой сервер: сервер разработки, промежуточный сервер и производственный сервер. То же самое касается и версий проектируемого программного обеспечения, в данном случае веб-сайта: разрабатываемая, промежуточная и производственная версии – *Прим. перев.*

Вы увидите, как появится окно браузера и попытается открыть *localhost:8000* и показать страницу с ошибкой «Unable to connect» (Не в состоянии установить соединение). Если вы перейдете назад на свою консоль, то увидите большое уродливое сообщение об ошибке, говорящее о том, что Selenium попал на страницу ошибки. И затем вы, вероятно, будете раздражены тем, что этот тест оставил на вашем рабочем столе окно Firefox где попало, вынуждая вас наводить порядок. Этот вопрос мы уладим чуть позже!



Если вместо этого вы видите ошибку с попыткой импортировать Selenium или ошибку с попыткой найти "geckodriver", то вам, возможно, надо вернуться и еще раз взглянуть на главу «*Предпосылки и предположения*».

А пока у нас есть *неработающий тест*, а это значит, что мы можем приступить к созданию нашего приложения.

Прощайте, римские цифры!

Столько много предисловий к TDD используют римские цифры в качестве примера, что это стало ходячим анекдотом – одно такое я даже сам начинал писать. Если вам любопытно, вы можете найти его на моей странице в GitHub[3].

Римские цифры и хороши, и плохи одновременно. Это хорошая «игрушечная» задача, разумно ограниченная в объеме, и вы можете вполне хорошо объяснить ими TDD.

Проблема состоит в том, что ее бывает очень трудно увязать с реальным миром. Вот почему в качестве моего примера я решил обратиться к созданию реального веб-приложения, начиная с пустого места. Несмотря на то что это веб-приложение простое, надеюсь, вам будет проще его перенести в свой следующий реальный проект.

Приведение Django в рабочее состояние

Поскольку к настоящему моменту вы определенно прочли главу «*Предпосылки и предположения*», программная инфраструктура Django у вас уже инсталлирована. Первый шаг по приведению Django в рабочее состояние заключается в создании *проекта*, который будет основным контейнером для нашего сайта. Для этого Django обеспечивает небольшой инструмент командной строки:

```
$ django-admin.py startproject superlists
```

[3] См. https://github.com/hjwp/tdd-roman-numeral-calculator/

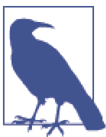
Эта команда создаст папку под названием *superlists* и ряд файлов и подпапок внутри нее:

```
.
├── functional_tests.py
├── geckodriver.log
└── superlists
        ├── manage.py
        └── superlists
                ├── __init__.py
                ├── settings.py
                ├── urls.py
                └── wsgi.py
```

Да, внутри папки *superlists* есть папка *superlists*. Немного сбивает с толку, но это только одна из таких вещей; тому существуют серьезные причины, если вы оглянетесь назад на историю Django. А пока важно знать, что папка *superlists/superlists* предназначена для вещей, которые относятся ко всему проекту – как, например, файл *settings.py*, используемый для хранения глобальной конфигурационной информации о сайте.

Вы также обратите внимание на файл *manage.py*. Для Django данный файл – это швейцарский нож, и часть из его функций заключается в выполнении сервера разработки. Теперь испытаем его. Командой **`cd superlists`** перейдите в корневую папку *superlists* (мы много будем работать из этой папки) и затем выполните:

```
$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until
they are applied. Run 'python manage.py migrate' to apply them.

Django version 1.11, using settings 'superlists.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```



Пока ничего страшного, если проигнорировать это сообщение об «unapplied migrations» (о незадействованных миграциях). Мы обратимся к миграциям в главе 5.

Это тот самый сервер разработки Django, теперь он приведен в рабочее состояние на нашей машине. Оставьте его там и откройте другую команд-

ную оболочку. В ней мы сможем попытаться выполнить наш тест еще раз (из папки, в которой мы запустили):

```
$ python functional_tests.py
$
```

Поскольку вы только что открыли новое окно терминала, чтобы все заработало, вам придется активировать вашу виртуальную среду командой `virtualenv workon superlists`.

Совсем немного действий с командной строкой, но вы должны заметить две вещи: во-первых, нет уродливой ошибки `AssertionError` и, во-вторых, окно Firefox, которое Selenium показал, содержит совсем другую страницу.

Конечно, не так уж и много, но это был наш самый первый успешно выполненный тест! Ура!

Если все это выглядит почти как волшебство, будто этого не должно быть, почему бы не пойти и не взглянуть на сервер разработки вручную, открыв веб-браузер самостоятельно и посетив http://localhost:8000? Вы увидите нечто, как на рис. 1-2.

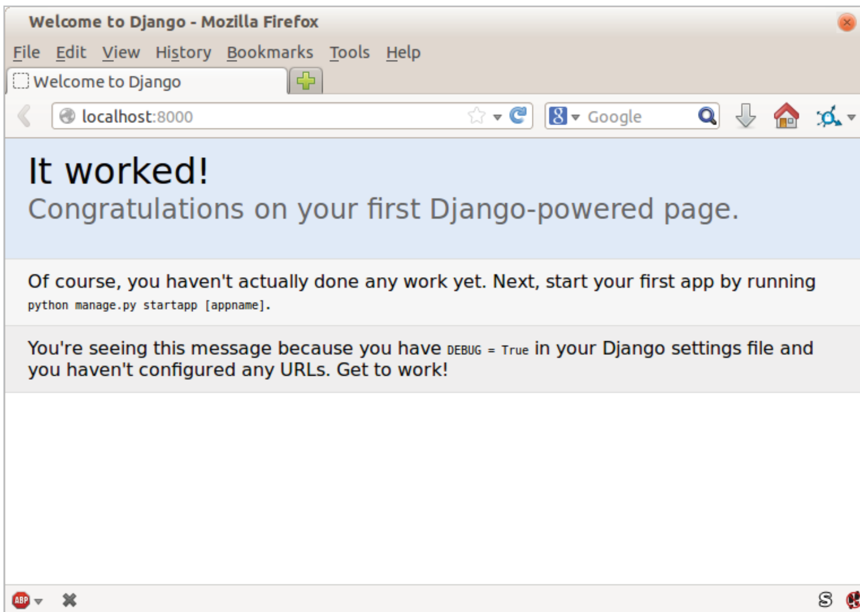


Рис. 1-2. Сработало!

Если хотите, можете вернуться из сервера разработки в исходную оболочку при помощи **Ctrl+C**.

Запуск репозитория Git

Прежде чем мы закончим главу, надо сделать еще одну вещь: начать передавать нашу работу *системе управления версиями* (VCS). Если вы опытный программист, вам не нужно слушать мои проповеди об управлении версиями. Но если вы плохо знакомы с этой системой, поверьте мне, что VCS – это необходимая вещь. Когда вашему проекту исполнится несколько недель и он вырастит до нескольких строк программного кода, иметь инструмент, который позволяет оглянуться назад, на старые версии кода, обратить изменения, безопасно исследовать новые идеи и даже просто выполнить резервное копирование... О боже! TDD идет рука об руку с управлением версиями, и я хочу быть уверенным, что правильно передаю мысль о том, как оно вписывается в поток операций.

Итак, наша первая фиксация изменений! Если немного поздновато, позор нам. В качестве системы управления версиями мы используем *Git*, потому что он – лучший.

Давайте начнем, переместив файл *functional_tests.py* в папку *superlists* и выполнив команду `git init` для запуска репозитория:

```
$ ls
superlists functional_tests.py geckodriver.log
$ mv functional_tests.py superlists/
$ cd superlists
$ git init .
Initialised empty Git repository in /.../superlists/.git/
```

Нашим рабочим каталогом с этого момента будет корневая папка superlists

С этой точки и впредь корневая папка *superlists* будет нашим рабочим каталогом.

(Для простоты в моих распечатках команд я буду всегда показывать ее как */.../ superlists/*, хотя, вероятно, на самом деле она будет чем-то вроде */home/kind-reader-username/my-python-projects/superlists/*.)

Всегда, когда я показываю команду для ввода, подразумевается, что мы находимся в этом каталоге. Аналогично, если я упомяну путь к файлу, то он будет относительно этого корневого каталога. Так, *superlists/settings.py* означает файл *settings.py* внутри папки *superlists* второго уровня.

Ясно как день? Если есть сомнения, отыщите файл *manage.py*; вы должны быть в том же каталоге, что и *manage.py*.

Теперь посмотрим, какие файлы мы хотим фиксировать:

```
$ ls
db.sqlite3 manage.py superlists functional_tests.py
```

db.sqlite3 – это файл базы данных. Он не нужен в управлении версиями. Ранее мы также встречали *geckodriver.log*, журнальный файл от Selenium, изменения которого мы тоже не хотим отслеживать. Оба эти файла мы добавим в специальный файл под названием *.gitignore*, который «говорит» Git, что именно следует проигнорировать:

```
$ echo "db.sqlite3" >> .gitignore
$ echo "geckodriver.log" >> .gitignore
```

Затем мы можем добавить остальную часть содержимого текущей папки, «.»:

```
$ git add .
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file: .gitignore
        new file: functional_tests.py
        new file: manage.py
        new file: superlists/__init__.py
        new file: superlists/__pycache__/__init__.cpython-36.pyc
        new file: superlists/__pycache__/settings.cpython-36.pyc
        new file: superlists/__pycache__/urls.cpython-36.pyc
        new file: superlists/__pycache__/wsgi.cpython-36.pyc
        new file: superlists/settings.py
        new file: superlists/urls.py
        new file: superlists/wsgi.py
```

Ух ты! У нас там целая куча файлов *.pyc*; бессмысленно их фиксировать. Удалим их из Git и тоже добавим в *.gitignore*:

```
$ git rm -r --cached superlists/__pycache__
rm 'superlists/__pycache__/__init__.cpython-36.pyc'
rm 'superlists/__pycache__/settings.cpython-36.pyc'
rm 'superlists/__pycache__/urls.cpython-36.pyc'
rm 'superlists/__pycache__/wsgi.cpython-36.pyc'
$ echo "__pycache__" >> .gitignore
$ echo "*.pyc" >> .gitignore
```

Теперь посмотрим, где мы находимся... (в дальнейшем вы увидите, что я много использую `git status` – да так много, что часто применяю псевдо-

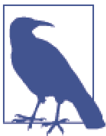
ним `git st`..., хотя я не скажу, как это делается; оставлю вам возможность обнаружить секреты псевдонимов Git самостоятельно!):

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file: .gitignore
        new file: functional_tests.py
        new file: manage.py
        new file: superlists/__init__.py
        new file: superlists/settings.py
        new file: superlists/urls.py
        new file: superlists/wsgi.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified: .gitignore
```

Выглядит неплохо, мы готовы сделать нашу первую фиксацию!

```
$ git add .gitignore
$ git commit
```

Когда вы наберете команду `git commit`, она раскроет окно редактора, чтобы вы написали в нем свое сообщение о фиксации. Мое выглядело как на рис. 1-3[4].



Если вы хотите оседлать Git по-настоящему, то пора узнать, как отправить вашу работу в облачную службу управления версиями, такую как GitHub или BitBucket. Они будут полезны, если вы намерены работать, следуя по тексту этой книги, на других ПК. Оставляю на вас обязанность узнать, как они работают; они располагают превосходной документацией. Как вариант, можно подождать до главы 9, когда мы будем использовать одну из них для развертывания.

[4] Редактор *vi* появился, и вы не представляете, что надо делать? Или же вы видите сообщение об идентичности регистрационной записи и `git config --global user.username`? Вернитесь и еще раз прочтите главу «*Предпосылки и предположения*»; там есть немного коротких инструкций.

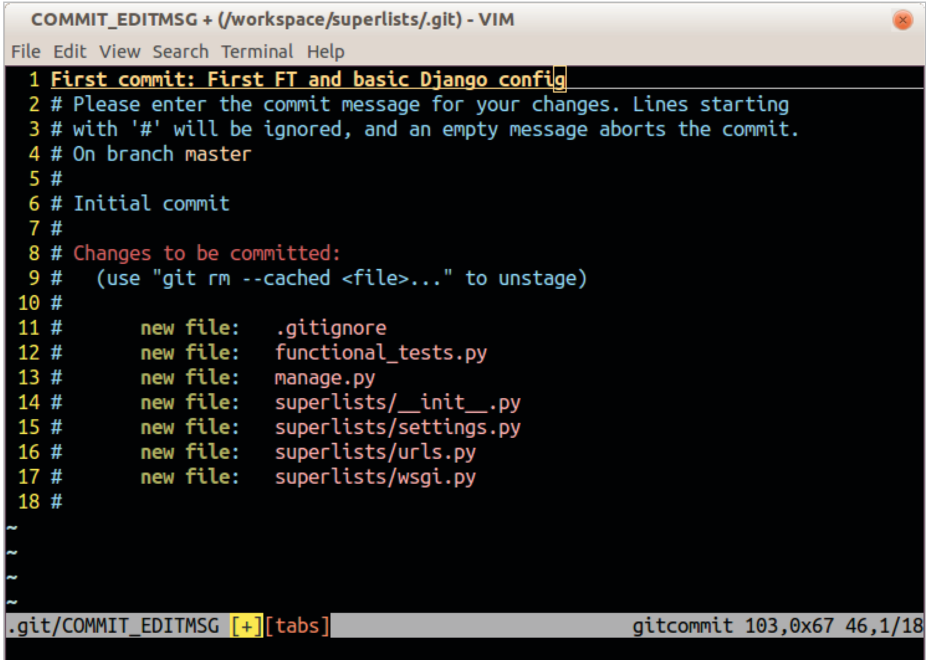


Рис. 1-3. Первая фиксация в Git

Вот и все, что касается лекции по системам управления версиями (VCS). Поздравляю! Вы написали функциональный тест при помощи Selenium, и у вас установлена платформа Django, которая работает самоочевидным, заверенным Билли способом, в стиле TDD и с первоочередным тестированием. Вы можете заслуженно похлопать себя по спине, перед тем как перейти дальше к главе 2.

Глава 2

Расширение функционального теста при помощи модуля unittest

Давайте адаптируем наш тест, который в настоящее время проверяет стандартную страницу Django «it worked» (сработало), и вместо нее проверим кое-какие вещи, которые мы хотим увидеть на реальной главной странице нашего сайта.

Пора раскрыть секрет по поводу того, какое веб-приложение мы создаем: сайт списков неотложных дел! При этом мы во многом следуем моде: несколько лет назад все учебные материалы по веб касались создания блога. Тогда это были форумы и опросы; сегодня – списки неотложных дел (to-do list).

Вся причина в том, что список неотложных дел – действительно удачный пример. По своей сути он очень прост – просто список текстовых строк – и поэтому «минимально дееспособное» приложение для обработки списка очень легко доводится до рабочего состояния. Помимо этого, его можно расширить – различными моделями долговременного хранения данных, добавлением сроков исполнения, напоминаний, обменом с другими пользователями и улучшением клиентского пользовательского интерфейса (UI). Кроме того, нет никаких причин ограничиваться только списками неотложных дел; это может быть любой вид списков. Но главное – это мне позволит продемонстрировать все основные аспекты веб-программирования и то, как применять к ним методологию TDD.

Использование функционального теста для определения минимального дееспособного приложения

Тесты, которые используют Selenium, позволяют управлять реальным веб-браузером, тем самым они дают нам возможность увидеть, как при-

ложение *функционирует* с точки зрения пользователя. Вот почему они называются *функциональными тестами*.

Это означает, что ФТ может выступать для вашего приложения своего рода спецификацией. Он помогает очертить контуры того, что вы, по-видимому, называете *историей пользователя*, и следит за тем, как пользователь мог бы работать с конкретным элементом и как приложение должно на них отвечать.

Терминология:
функциональный тест == приемочный тест == сквозной тест

То, что я называю функциональными тестами, некоторые специалисты предпочитают именовать приемочными либо сквозными тестами или испытаниями. Их суть заключается в том, что эти виды тестов смотрят на характер функционирования всего приложения с внешней стороны. Есть и другой термин – тест черного ящика, потому что он ничего не знает о внутреннем устройстве тестируемой системы.

Функциональные тесты должны иметь удобочитаемую историю, которую можно прослеживать. Мы определяем ее в явной форме, используя комментарии, которые сопровождают программный код теста. При создании нового ФТ мы сначала пишем комментарии, чтобы зафиксировать ключевые моменты истории пользователя. Поскольку они легко читаемы, вы даже можете обмениваться ими с непрограммистами, чтобы обсудить технические требования и функциональные особенности вашего приложения.

TDD и методологии гибкой разработки программного обеспечения часто сочетаются, и одной из наиболее обсуждаемых является тема минимально дееспособного приложения. Каким будет самое простое возможное приложение, которое по-прежнему останется полезным? Предлагаю начать с создания такого приложения, чтобы как можно быстрее прощупать почву.

Минимальный дееспособный список неотложных дел, в общем-то, нуждается только в том, чтобы обеспечить пользователю возможность вводить некоторые элементы списка и запоминать их для следующего посещения.

Откройте *functional_tests.py* и напишите историю, что-то вроде этой:

functional_tests.py

```
from selenium import webdriver

browser = webdriver.Firefox()

# Эдит слышала про крутое новое онлайн-приложение со списком
```

```
# неотложных дел. Она решает оценить его домашнюю страницу
browser.get('http://localhost:8000')

# Она видит, что заголовок и шапка страницы говорят о списках
# неотложных дел
assert 'To-Do' in browser.title

# Ей сразу же предлагается ввести элемент списка

# Она набирает в текстовом поле "Купить павлиньи перья" (ее хобби -
# вязание рыболовных мушек)

# Когда она нажимает enter, страница обновляется, и теперь страница
# содержит "1: Купить павлиньи перья" в качестве элемента списка

# Текстовое поле по-прежнему приглашает ее добавить еще один элемент.
# Она вводит "Сделать мушку из павлиньих перьев"
# (Эдит очень методична)

# Страница снова обновляется, и теперь показывает оба элемента ее списка

# Эдит интересно, запомнит ли сайт ее список. Далее она видит, что
# сайт сгенерировал для нее уникальный URL-адрес - об этом
# выводится небольшой текст с объяснениями.

# Она посещает этот URL-адрес - ее список по-прежнему там.

# Удовлетворенная, она снова ложится спать
browser.quit()
```

Вы заметите, что, кроме детального описания теста в комментариях, я обновил утверждение `assert` для поиска слова «To-Do» вместо слова «Django». Это означает, что теперь тест ожидаемо не сработает. Попробуем его выполнить.

Сначала запустите сервер:

```
$ python manage.py runserver
```

Затем в другой оболочке запустите тесты:

```
$ python functional_tests.py
Traceback (most recent call last):
  File "functional_tests.py", line 10, in <module>
    assert 'To-Do' in browser.title
AssertionError
```

У нас свой термин для комментариев...

Когда я только начал работу в Resolver, я имел обыкновение добродетельно приправлять свой код хорошими описательными комментариями. Коллеги сказали: «Гарри, у нас есть свой термин для комментариев. Мы называем их брехней». Я был потрясен! Ведь я же со школьной скамьи знал, что комментарии являются хорошей практикой!

Они преувеличивали для пущего словца. Безусловно, есть место для комментариев, которые добавляют контекст и намерение. Но коллеги имели в виду, что бессмысленно писать комментарий, который просто повторяет то, что вы делаете в коде:

```
# увеличить wibble на 1
wibble += 1
```

Мало того, что это бессмысленно, существует опасность, что вы забудете их обновить при обновлении исходного кода и в итоге они станут вводить в заблуждение. В идеале нужно стремиться делать код настолько читаемым, использовать настолько хорошие имена переменных и функций и структурировать его так хорошо, чтобы вам больше не нужны были никакие комментарии для объяснения того, что делает программный код. Оставить лишь несколько тут и там для объяснения *причины*.

Есть и другие места, где комментарии очень полезны. Мы увидим, что Django часто применяет их в файлах, которые генерирует для нас, чтобы мы использовали их в качестве подсказок по поводу полезных элементов ее API. И, конечно, мы используем комментарии, чтобы объяснить историю пользователя в своих функциональных тестах, составляя из теста связную историю. Это даст гарантию, что мы всегда будем тестировать с точки зрения пользователя.

В этой области есть много других забавных вещей – например, *Разработка на основе поведения* (см. приложение E) и предметно-ориентированные языки для тестирования (DSL), но это темы других книг.

Это то, что мы называем *ожидаемой неполадкой* (failure), что в сущности представляет собой хорошую новость – не столь хорошую, как тест, который проходит, но по крайней мере он не сработал по правильной причине; у нас может быть некоторая уверенность, что мы написали тест правильно.

Модуль unittest стандартной библиотеки Python

Есть пара досадных мелочей, с которыми мы должны разобраться. Прежде всего, сообщение «AssertionError» не очень полезное – было бы неплохо, если бы тест сообщал нам о том, что именно он нашел в качестве заголовка браузера. Кроме того, он оставил окно Firefox где попало на рабочем столе – было бы неплохо, если бы он автоматически его убирал.

Один из вариантов – использовать второй параметр ключевого слова `assert`, что-то вроде:

```
assert 'To-Do' in browser.title, "Browser title was " + browser.title
```

И мы могли бы также применить блок `try/finally`, чтобы убрать старое окно Firefox. Однако эти виды проблем встречаются в тестировании довольно часто, и в модуле `unittest` стандартной библиотеки у нас имеется несколько готовых решений. Давайте ими воспользуемся! В файле *functional_tests.py*:

functional_tests.py

```
from selenium import webdriver
import unittest

class NewVisitorTest(unittest.TestCase):  ❶
    '''тест нового посетителя'''

    def setUp(self):  ❸
        '''установка'''
        self.browser = webdriver.Firefox()

    def tearDown(self):  ❸
        '''демонтаж'''
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self):  ❷
        '''тест: можно начать список и получить его позже'''
        # Эдит слышала про крутое новое онлайн-приложение со
        # списком неотложных дел. Она решает оценить его
        # домашнюю страницу
        self.browser.get('http://localhost:8000')

        # Она видит, что заголовок и шапка страницы говорят о
        # списках неотложных дел
        self.assertIn('To-Do', self.browser.title)  ❹
        self.fail('Закончить тест!')  ❺

        # Ей сразу же предлагается ввести элемент списка
        [...остальные комментарии, как и прежде]

if __name__ == '__main__':  ❻
    unittest.main(warnings='ignore')  ❼
```

Здесь вы, вероятно, заметите несколько вещей:

❶ Тесты организованы в классы, которые наследуют от `unittest.TestCase`.

❷ Главная часть теста находится в методе под названием `test_can_start_a_list_and_retrieve_it_later`. Любой метод, имя которого начинается с `test`, является методом тестирования и будет выполнен исполнителем тестов. В одном классе может быть более одного метода `test_`. Неплохо также для методов тестирования иметь хорошо говорящие имена.

❸ `setUp` и `tearDown` – особые методы, которые выполняются перед и после каждого теста. Я их использую для запуска и остановки браузера – обратите внимание, что они немного походят на `try/except` в том, что `tearDown` будет выполняться, даже если во время самого теста произойдет ошибка[1]. Больше никаких окон Firefox, разбросанных где попало!

❹ Для создания тестовых утверждений мы используем `self.assertln` вместо просто `assert`. Модуль `unittest` предоставляет много подобных вспомогательных функций для создания тестовых утверждений – например, `assertEqual`, `assertTrue`, `assertFalse` и т. д. Подробности вы можете узнать в документации по `unittest`[2].

❺ `self.fail` никогда не срабатывает и всегда генерирует переданное сообщение об ошибке. Я использую его в качестве напоминания об окончании теста.

❻ В конце у нас оператор `if __name__ == '__main__'` (если вы с ним раньше не встречались, то знайте: так сценарий Python проверяет, был ли он выполнен из командной строки, а не просто импортирован другим сценарием). Мы вызываем `unittest.main()`, который запускает исполнителя тестов `unittest`, а он в свою очередь автоматически найдет в файле тестовые классы и методы и их выполнит.

❼ `warnings='ignore'` подавляет излишние предупреждающие сообщения `Resourcewarning`, которые выдавались во время написания. Возможно, к тому моменту, когда вы будете это читать, они уже не будут появляться; не стесняйтесь, попробуйте удалить эту инструкцию!

Попробуем выполнить тест!

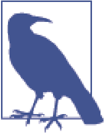
```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
 ---------------------------------------------------------------------
Traceback (most recent call last):
```

[1] Единственное исключение: когда у вас исключение внутри `setUp`, `tearDown` не работает.

[2] См. http://docs.python.org/3/library/unittest.html.

```
  File "functional_tests.py", line 18, in
test_can_start_a_list_and_retrieve_it_later
    self.assertIn('To-Do', self.browser.title)
AssertionError: 'To-Do' not found in 'Welcome to Django'

 ---------------------------------------------------------------------
Ran 1 test in 1.747s
FAILED (failures=1)
```

Теперь немного лучше, не правда ли? Окно Firefox убрано, и мы получили отформатированный отчет о том, сколько тестов было запущено и сколько из них не сработало, а `assertIn` выдало полезное сообщение об ошибке с информацией об отладке. Шикарно!



Во время чтения документации Django по тестированию вы, возможно, видели что-то похожее под названием `LiveServerTestCase` и интересуетесь, следует ли нам теперь его использовать. Вам 5 баллов за то, что прочли дружелюбное руководство! Пока `LiveServerTestCase` будет немного сложноват, но обещаю, что воспользуюсь им в одной из следующих глав.

Фиксация

Сейчас самое время выполнить фиксацию изменений; точнее, приятное автономное изменение. Мы развернули функциональный тест, включив комментарии, которые описывают поставленную задачу – минимальный дееспособный список неотложных дел. Мы также переписали этот тест для использования Python'овского модуля `unittest` и его разнообразных вспомогательных функций тестирования.

Выполните команду `git status` – она подтвердит, что единственный изменившийся файл – *functional_tests.py*. Затем выполните команду `git diff`, которая покажет разницу (дельту) между последней фиксацией и тем, что в настоящее время находится на диске. Она должна сообщить, что *functional_tests.py* достаточно существенно изменился:

```
$ git diff
diff --git a/functional_tests.py b/functional_tests.py
index d333591..b0f22dc 100644
--- a/functional_tests.py
+++ b/functional_tests.py
@@ -1,6 +1,45 @@
from selenium import webdriver
```

```
+import unittest

-browser = webdriver.Firefox()
-browser.get('http://localhost:8000')
+class NewVisitorTest(unittest.TestCase):

-assert 'Django' in browser.title
+    def setUp(self):
+        self.browser = webdriver.Firefox()
+
+    def tearDown(self):
+        self.browser.quit()
[...]
```

Теперь выполним:

```
$ git commit -a
```

`-a` означает «автоматически добавить любые изменения в отслеживаемых файлах» (то есть в любых файлах, которые мы фиксировали прежде). Эта опция не добавит совершенно новые файлы (вам придется самостоятельно добавлять их в явном виде с помощью команды `git`), но часто, как в данном случае, нет никаких новых файлов, так что это будет полезное сокращение.

Когда раскроется редактор, добавьте описательное сообщение о фиксации, например: «Расширен первый ФТ в комментариях, теперь использует unittest»[3].

Теперь мы находимся в превосходном положении, чтобы начать писать реальный код для приложения со списками. Продолжайте читать!

Полезные понятия TDD

История пользователя

Описание того, как приложение будет работать с точки зрения пользователя. Используется для структурирования функционального теста.

Ожидаемая неполадка

Когда тест не срабатывает ожидаемым для нас образом.

[3] См. http://pr0git.blogspot.ru/2015/02/git_4.html по поводу кодировки и комментариев на русском языке в Git – *Прим. перев.*