O'REILLY®

TDD w praktyce

Niezawodny kod w języku Python

TWÓRZ NIEZAWODNE APLIKACJE W JĘZYKU PYTHON!

Helion

Harry J.W. Percival

Tytuł oryginału: Test-Driven Development with Python

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-1377-4

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/tddwpr.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/tddwpr
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Wprowadzenie **13**
Przygotowania i założenia **19**
Podziękowania **25**

I Podstawy TDD i Django **27**

1. Konfiguracja Django za pomocą testu funkcjonalnego **29**
Słuchaj Testing Goat! Nie rób nic, dopóki nie przygotujesz testu 29
Rozpoczęcie pracy z frameworkiem Django 32
Utworzenie repozytorium Git 33

2. Rozszerzenie testu funkcjonalnego za pomocą modułu unittest **37**
Użycie testu funkcjonalnego do przygotowania minimalnej aplikacji 37
Moduł unittest ze standardowej biblioteki Pythona 40
Ukryte oczekiwanie 42
Przekazanie plików do repozytorium 42

3. Testowanie prostej strony głównej za pomocą testów jednostkowych **45**
Nasza pierwsza aplikacja Django i test jednostkowy 46
Testy jednostkowe i różnice dzielące je od testów funkcjonalnych 46
Testy jednostkowe w Django 47
MVC w Django, adresy URL i funkcje widoku 48
Wreszcie zaczynamy tworzyć kod aplikacji 49
urls.py 51
Testy jednostkowe widoku 53
Cykl test jednostkowy — tworzenie kodu 54

4. Do czego służą te wszystkie testy? **57**
Programowanie przypomina wyciąganie wiadrem wody ze studni 58
Użycie Selenium do testowania interakcji użytkownika 59
Reguła „nie testuj stałych” i szablony na ratunek 62
Refaktoryzacja w celu użycia szablonu 62

Refaktoryzacja ... 65
Nieco więcej o stronie głównej ... 67
Przypomnienie — proces TDD ... 68

5. Zapis danych wejściowych użytkownika ... 73
Od formularza sieciowego do wykonania żądania POST ... 73
Przetwarzanie żądania POST w serwerze ... 76
Przekazanie zmiennych Pythona do wygenerowania w szablonie ... 77
Do trzech razy sztuka, a później refaktoryzacja ... 81
Django ORM i nasz pierwszy model ... 82
Pierwsza migracja bazy danych ... 84
Zdumiewająco duży postęp w teście ... 85
Nowa kolumna oznacza nową migrację ... 85
Zapis w bazie danych informacji z żądania POST ... 86
Przekierowanie po wykonaniu żądania POST ... 89
Poszczególne testy powinny testować pojedyncze rzeczy ... 89
Wygenerowanie elementów w szablonie ... 90
Utworzenie produkcyjnej bazy danych za pomocą polecenia migrate ... 92

6. Przygotowanie minimalnej działającej wersji witryny ... 97
Gwarancja izolacji testu w testach funkcjonalnych ... 97
Wykonanie tylko testów jednostkowych ... 100
Stawiaj na małe projekty ... 101
YAGNI! ... 102
REST ... 102
Implementacja nowego projektu za pomocą TDD ... 103
Iteracja w kierunku nowego projektu ... 105
Testowanie widoków, szablonów i adresów URL za pomocą testu klienta Django ... 107
Nowa klasa testowa ... 107
Nowy adres URL ... 108
Nowa funkcja widoku ... 108
Oddzielny szablon do wyświetlania list ... 109
Kolejny adres URL i widok pozwalający na dodanie elementów listy ... 112
Klasa testowa dla operacji tworzenia nowej listy ... 112
Adres URL i widok przeznaczony do tworzenia nowej listy ... 113
Usunięcie zbędnego kodu i dalsze testy ... 114
Wskazanie formularzy w nowym adresie URL ... 115
Dostosowanie modeli ... 116
Związek klucza zewnętrznego ... 117
Dostosowanie reszty świata do naszych nowych modeli ... 118
Każda lista powinna mieć własny adres URL ... 120
Przechwytywanie parametrów z adresów URL ... 121
Dostosowanie new_list do nowego świata ... 122
Jeszcze jeden widok pozwalający na dodanie elementu do istniejącej listy ... 123
Uwaga na żarłoczne wyrażenia regularne! ... 124
Ostatni nowy adres URL ... 124

Ostatni nowy widok ... 125
Jak można użyć adresu URL w formularzu? ... 126
Ostatnia refaktoryzacja za pomocą polecenia include ... 128

II Programowanie sieciowe ... 131

7. Upiększanie — jak przetestować układ i style? ... 133
Jaką funkcjonalność należy testować w przypadku układu i stylów? ... 133
Upiększanie za pomocą frameworka CSS ... 136
Dziedziczenie szablonu w Django ... 137
Integracja z frameworkiem Bootstrap ... 139
Wiersze i kolumny ... 139
Pliki statyczne w Django ... 140
Zaczynamy używać klasy StaticLiveServerCase ... 141
Użycie komponentów Bootstrap do poprawy wyglądu witryny ... 142
Jumbotron ... 142
Ogromne pola danych wejściowych ... 143
Nadanie stylu tabeli ... 143
Użycie własnych arkuszy stylów CSS ... 143
Co zostało zatuszowane — polecenie collectstatic i inne katalogi statyczne ... 144
Kilka tematów, które nie zostały omówione ... 147

8. TDD na przykładzie witryny prowizorycznej ... 149
Techniki TDD i niebezpieczeństwa związane z wdrożeniem ... 150
Jak zwykle zaczynamy od testu ... 151
Pobranie nazwy domeny ... 153
Ręczne przygotowanie serwera do hostingu naszej witryny ... 153
Wybór hostingu dla witryny ... 154
Uruchomienie serwera ... 154
Konto użytkownika, SSH i uprawnienia ... 155
Instalacja Nginx ... 155
Konfiguracja domen dla witryn prowizorycznej i rzeczywistej ... 156
Użycie testów funkcjonalnych do potwierdzenia działania domeny i serwera Nginx ... 157
Ręczne wdrożenie kodu ... 157
Dostosowanie położenia bazy danych ... 158
Utworzenie virtualenv ... 159
Prosta konfiguracja Nginx ... 162
Utworzenie bazy danych za pomocą polecenia migrate ... 164
Wdrożenie w środowisku produkcyjnym ... 164
Użycie Gunicorn ... 164
Użycie Nginx do obsługi plików statycznych ... 165
Użycie gniazd systemu Unix ... 166
Przypisanie opcji DEBUG wartości False i ustawienie ALLOWED_HOSTS ... 167
Użycie Upstart do uruchamiania Gunicorn wraz z systemem ... 168
Zachowanie wprowadzonych zmian — dodanie Gunicorn do pliku requirements.txt ... 168
Automatyzacja ... 169
Zachowanie informacji o postępie ... 172

9. Zautomatyzowane wdrożenie za pomocą Fabric 173
Analiza skryptu Fabric dla naszego wdrożenia 174
Wypróbowanie rozwiązania 177
Wdrożenie w środowisku produkcyjnym 179
Pliki konfiguracyjne Nginx i Gunicorn odtworzone za pomocą sed 180
Użycie polecenia git tag do oznaczenia wydania 181
Dalsza lektura 181

10. Weryfikacja danych wejściowych i organizacja testu 183
Testy funkcjonalne weryfikacji danych — ochrona przed pustymi elementami 183
Pominięcie testu 184
Podział testów funkcjonalnych na wiele plików 185
Wykonanie pojedynczego pliku testu 187
Podparcie testów funkcjonalnych 188
Sprawdzenie warstwy modelu 189
Refaktoryzacja testów jednostkowych na oddzielne pliki 189
Testy jednostkowe sprawdzania modelu oraz menedżer kontekstu self.assertRaises() 190
Dziwactwo Django — zapis modelu nie wywołuje operacji sprawdzenia poprawności 191
Wyświetlanie w widoku błędów z weryfikacji modelu 192
Upewnienie się, że nieprawidłowe dane nie zostaną zapisane w bazie danych 194
Wzorzec Django — przetwarzanie żądań POST w widoku generującym formularz 196
Refaktoryzacja — przekształcenie funkcjonalności new_item na view_list 197
Egzekwowanie w widoku view_list weryfikacji modelu 199
Refaktoryzacja — usunięcie na stałe zdefiniowanych adresów URL 200
Znacznik szablonu {% url %} 200
Użycie get_absolute_url w przekierowaniach 201

11. Prosty formularz 205
Przeniesienie do formularza logiki odpowiedzialnej za sprawdzanie poprawności danych 205
Użycie testu jednostkowego do analizy API formularzy 206
Przejście do Django ModelForm 208
Testowanie i dostosowanie do własnych potrzeb logiki weryfikacji formularza 209
Użycie formularza w widokach 210
Użycie formularza w widoku za pomocą żądania GET 211
Duża operacja znajdź i zastąp 213
Użycie formularza w widoku obsługującym żądania POST 215
Adaptacja testów jednostkowych dla widoku new_list 215
Użycie formularza w widoku 216
Użycie formularza w celu wyświetlenia błędów w szablonie 216
Użycie formularza w innym widoku 217
Metoda pomocnicza dla wielu krótkich testów 218
Użycie metody save() formularza 220

12. Bardziej skomplikowane formularze 223
Kolejny test funkcjonalny dotyczący powielonych elementów 223
Ochrona przed duplikatami w warstwie modelu 224
Mała dygresja dotycząca kolejności API Querystring i przedstawiania ciągu tekstowego 226

Przepisanie testu starego modelu ... 228
Pewne błędy spójności ujawniają się podczas zapisu ... 229
Eksperymenty w warstwie widoku sprawdzające, czy są powielone elementy ... 230
Bardziej skomplikowany formularz do obsługi unikalności elementów ... 231
Użycie istniejącego formularza w widoku listy ... 232

13. Zagłębiamy się ostrożnie w JavaScript ... 237
Rozpoczynamy od testów funkcjonalnych ... 237
Konfiguracja prostego silnika wykonywania testów JavaScript ... 238
Użycie jQuery i stałych elementów ... 240
Utworzenie testu jednostkowego JavaScript dla żądanej funkcjonalności ... 243
Testowanie JavaScript w cyklu TDD ... 245
Zdarzenie onload i przestrzenie nazw ... 245
Kilka rozwiązań, które się nie sprawdzają ... 246

14. Wdrożenie nowego kodu ... 247
Wdrożenie prowizoryczne ... 247
Wdrożenie rzeczywiste ... 247
A jeśli wystąpi błąd bazy danych? ... 248
Podsumowanie — git tag i nowe wydanie ... 248

III Bardziej zaawansowane zagadnienia ... 249

15. Użycie JavaScript do uwierzytelniania użytkownika, integracji wtyczek i przygotowania imitacji ... 251
Mozilla Persona (BrowserID) ... 252
Kod eksperymentalny, czyli „Spiking” ... 252
Utworzenie nowej gałęzi dla Spike ... 253
Łączenie kodu JavaScript i interfejsu użytkownika ... 253
Protokół Browser-ID ... 254
Kod po stronie serwera — niestandardowe uwierzytelnienie ... 255
Zamiana rozwiązania eksperymentalnego na zwykłe ... 260
Często stosowana technika Selenium — wyraźne oczekiwanie ... 262
Wycofanie kodu eksperymentalnego ... 264
Testy jednostkowe JavaScript obejmujące komponenty zewnętrzne — nasze pierwsze imitacje ... 265
Porządkowanie — katalog plików statycznych dla całej witryny ... 265
Imitacja: kto, co i dlaczego? ... 266
Przestrzenie nazw ... 267
Prosta imitacja dla testów jednostkowych dla naszej funkcji inicjującej ... 267
Bardziej zaawansowane imitacje ... 272
Sprawdzenie wywołania argumentów ... 275
Konfiguracja QUnit i testowanie żądań Ajax ... 276
Więcej zagnieżdżonych wywołań zwrotnych! Testowanie kodu asynchronicznego ... 280

16. Uwierzytelnianie po stronie serwera i imitacje w Pythonie 283
Rzut oka na wersję eksperymentalną widoku logowania 283
Imitacje w Pythonie 284
Testowanie widoku za pomocą imitacji funkcji uwierzytelnienia 284
Sprawdzenie, czy widok faktycznie loguje użytkownika 286
Zmiana eksperymentalnej wersji uwierzytelniania na zwykłą — imitacja żądania internetowego 290
Polecenie if oznacza więcej testów 291
Poprawki na poziomie klasy 292
Strzeż się imitacji w porównaniach wartości boolowskich 295
Utworzenie użytkownika, jeśli to konieczne 296
Metoda get_user() 296
Minimalny niestandardowy model użytkownika 298
Małe rozczarowanie 300
Testy jako dokumentacja 301
Użytkownicy są uwierzytelnieni 301
Chwila prawdy — czy testy funkcjonalne zostaną zaliczone? 302
Zakończenie testu funkcjonalnego, przetestowanie wylogowania 303

17. Konfiguracja testu, rejestracja i debugowanie po stronie serwera 307
Pominięcie procesu logowania przez wstępne utworzenie sesji 307
Sprawdzamy rozwiązanie 309
Dowód znajdziesz w praktyce — użycie wersji prowizorycznej do wychwycenia błędów 310
Konfiguracja rejestracji danych 311
Usunięcie błędu systemu Persona 312
Zarządzanie testową bazą danych w serwerze prowizorycznym 314
Polecenie Django służące do tworzenia sesji 314
Test funkcjonalny uruchamiający w serwerze narzędzie zarządzania 315
Dodatkowy krok za pomocą modułu subprocess 317
Zachowanie kodu odpowiedzialnego za rejestrację danych 320
Użycie konfiguracji hierarchicznej rejestracji danych 320
Podsumowanie 322

18. Kończymy „Moje listy" — podejście Outside-In 325
Alternatywa, czyli podejście Inside-Out 325
Dlaczego preferowane jest podejście Outside-In? 326
Test funkcjonalny dla strony Moje listy 326
Warstwa zewnętrzna — prezentacja i szablony 327
Przejście o jedną warstwę w dół do funkcji widoku (kontroler) 328
Kolejne zaliczenie — podejście Outside-In 329
Szybka restrukturyzacja hierarchii dziedziczenia szablonu 329
Projektowanie API za pomocą szablonu 330
Przejście w dół do kolejnej warstwy — co widok przekazuje szablonowi? 331
Kolejne „wymaganie" z warstwy widoku — nowe listy powinny „zapamiętywać" swego właściciela 332
Czy przejść do kolejnej warstwy, gdy test kończy się niepowodzeniem? 333

Przejście w dół do warstwy modelu ... 333
Ostatni krok — uzyskanie z poziomu szablonu dostępu do właściciela za pomocą API .name ... 335

19. Izolacja i „słuchanie" testów ... 337

Powrót do miejsca, w którym podjęliśmy decyzję — warstwa widoku zależy od nieutworzonego jeszcze kodu modelu ... 337
Pierwsza próba użycia imitacji w celu zapewnienia izolacji ... 338
Użycie side_effect do sprawdzenia sekwencji zdarzeń ... 339
Posłuchaj testu — brzydki test oznacza konieczność refaktoryzacji ... 341
Ponowne utworzenie testów dla widoku, tym razem w pełni odizolowanych ... 342
Pozostawienie starych zintegrowanych testów jako punktu odniesienia ... 342
Nowy zestaw w pełni odizolowanych testów ... 342
Myślimy w kategoriach współpracy ... 343
Przejście w dół do warstwy formularzy ... 347
Nadal słuchaj testów — usunięcie kodu ORM z aplikacji ... 348
Wreszcie przechodzimy w dół do warstwy modelu ... 350
Powrót do widoków ... 352
Moment prawdy (i ryzyko związane z imitacjami) ... 353
Potraktowanie interakcji między warstwami jak kontraktów ... 354
Identyfikacja niejawnych kontraktów ... 355
Usunięcie przeoczonego problemu ... 356
Jeszcze jeden test ... 357
Porządkowanie, czyli co zachować z pakietu testów zintegrowanych? ... 358
Usunięcie powielonego kodu w warstwie formularzy ... 358
Usunięcie starej implementacji widoku ... 359
Usunięcie zbędnego kodu w warstwie formularzy ... 359
Podsumowanie — testy odizolowane kontra zintegrowane ... 360
Niech poziom skomplikowania będzie Twoim przewodnikiem ... 361
Czy powinienem tworzyć oba rodzaje testów? ... 361
Do przodu! ... 362

20. Ciągła integracja ... 363

Instalacja serwera Jenkins ... 363
Konfiguracja zabezpieczeń w Jenkins ... 365
Dodanie wymaganych wtyczek ... 365
Konfiguracja projektu ... 367
Pierwsza kompilacja ... 368
Konfiguracja ekranu wirtualnego, aby testy funkcjonalne można było wykonywać bez monitora ... 370
Wykonanie zrzutów ekranu ... 371
Najczęstszy problem w Selenium — stan wyścigu ... 374
Wykonanie testów QUnit w Jenkins za pomocą PhantomJS ... 376
Instalacja node ... 377
Dodanie kolejnych kroków kompilacji w Jenkins ... 378
Więcej zadań do wykonania za pomocą serwera ciągłej integracji ... 380

21. Token serwisów społecznościowych, wzorzec strony i ćwiczenie dla czytelnika ... 381
Test funkcjonalny z wieloma użytkownikami i funkcja addCleanup() ... 381
Implementacja w Selenium wzorca interakcja-oczekiwanie ... 383
Wzorzec strony ... 384
Rozszerzenie testu funkcjonalnego na drugiego użytkownika i stronę „Moje listy” ... 386
Ćwiczenie dla czytelnika ... 388

22. Szybkie testy, wolne testy i gorąca lawa ... 391
Teza — testy jednostkowe są niezwykle szybkie, mają także inne zalety ... 392
Szybsze testy oznaczają szybsze tworzenie kodu ... 392
Uczucie błogostanu ... 393
Wolne testy nie są wykonywane zbyt często, co przekłada się na gorszej jakości kod ... 393
Teraz jest dobrze, ale wraz z upływem czasu testy zintegrowane są wykonywane coraz wolniej ... 393
Nie zabieraj mi tego ... 393
Testy jednostkowe pozwalają przygotować dobry projekt ... 394
Problemy związane z czystymi testami jednostkowymi ... 394
Testy odizolowane mogą być trudniejsze w odczycie i zapisie ... 394
Testy odizolowane nie testują automatycznie integracji ... 394
Testy jednostkowe rzadko przechwytują nieoczekiwane błędy ... 394
Testy oparte na imitacji stają się ściśle powiązane z implementacją ... 394
Jednak wszystkie wymienione problemy można pokonać ... 395
Synteza — jakie mamy oczekiwania wobec testów? ... 395
Poprawność ... 395
Czytelny, łatwy w obsłudze kod ... 395
Produktywna praca ... 395
Oceń testy pod kątem korzyści, jakich oczekujesz dzięki ich użyciu ... 396
Rozwiązania architektoniczne ... 396
Porty i adaptery, czysta architektura i architektura heksagonalna ... 397
Architektura Functional Core, Imperative Shell ... 398
Podsumowanie ... 398

Kieruj się Testing Goat! ... 401

Dodatki ... 403

A PythonAnywhere ... 405

B Widoki oparte na klasach Django ... 409

C Przygotowanie serwera za pomocą Ansible ... 419

D Testowanie migracji bazy danych ... 423

E Co dalej? ... 429

F Ściąga ... 433

G Bibliografia ... 437

Skorowidz ... 439

ROZDZIAŁ 5.

Zapis danych wejściowych użytkownika

Nasza aplikacja pobiera wpisany przez użytkownika element listy rzeczy do zrobienia i przekazuje go do serwera. Dlatego też możemy gdzieś zapisać wspomniany element i wczytać go później, gdy zajdzie potrzeba.

Kiedy zaczynałem pisać ten rozdział, od początku zastanawiałem się nad odpowiednim rozwiązaniem dla naszej aplikacji: wiele modeli dla list i elementów listy, wiele różnych adresów URL przeznaczonych do dodawania nowych list i elementów, trzy nowe funkcje widoku i kilka nowych testów jednostkowych dla wymienionych wcześniej komponentów. W pewnym momencie zatrzymałem się. Wprawdzie uznałem się za wystarczająco sprytnego, aby jednocześnie ogarnąć wszystkie wymienione problemy, ale istota technik TDD polega na umożliwieniu programiście wykonywania zadań pojedynczo, gdy zachodzi taka potrzeba. Zdecydowałem się więc na celowy skrót i wykonywanie w danym momencie tylko tych zadań, które są niezbędne do posunięcia nieco naprzód testów funkcjonalnych.

Pokazuję tutaj, jak techniki TDD mogą obsługiwać iteracyjny styl programowania. Nie jest to najszybsza droga, ale ostatecznie i tak byś się spotkał z tego rodzaju podejściem. Pozytywnym efektem ubocznym przyjętego podejścia będzie możliwość wprowadzenia nowych koncepcji, takich jak modele, praca z żądaniami `POST`, znaczniki szablonów Django itd. Wymienione koncepcje będę mógł przedstawiać *pojedynczo*, zamiast jednocześnie zarzucić Cię nimi wszystkimi.

To oczywiście nie oznacza, że *nie powinieneś* próbować myśleć z wyprzedzeniem i być sprytnym. W następnym rozdziale w znacznie większym stopniu wykorzystamy projektowanie oraz przygotowywanie interfejsu i przekonasz się, jak to się wpisuje w stosowanie technik TDD. W tym rozdziale możesz się nad tym jeszcze nie zastanawiać i po prostu robić to, czego wymagają od nas testy.

Od formularza sieciowego do wykonania żądania POST

Na końcu poprzedniego rozdziału komunikaty generowane przez testy wskazują na brak możliwości zapisu danych wejściowych użytkownika. Teraz wykorzystamy standardowe żądanie `POST` w HTML. Wprawdzie to nieco nudne, ale jednocześnie elegancie i łatwe do osiągnięcia rozwiązanie, a ponadto pozwoli nam na użycie kodu HTML5 i JavaScript w dalszej części książki.

Aby przeglądarka internetowa wygenerowała żądanie POST, element <input> musi posiadać atrybut name= i zostać opakowany znacznikiem <form> wraz z atrybutem method="POST". W takim przypadku przeglądarka internetowa automatycznie zajmie się wygenerowaniem żądania POST. Spróbujmy dostosować szablon *lists/templates/home.html* do wymienionych wymagań.

Plik *lists/templates/home.html*:

```
<h1>Twoja lista rzeczy do zrobienia</h1>
<form method="POST">
    <input name="item_text" id="id_new_item" placeholder="Wpisz rzecz do zrobienia" />
</form>
<table id="id_list_table">
```

Teraz po wykonaniu naszych testów funkcjonalnych otrzymamy nieco zawiły i nieoczekiwany błąd:

```
$ python3 functional_tests.py
[...]
Traceback (most recent call last):
  File "functional_tests.py", line 39, in
test_can_start_a_list_and_retrieve_it_later
    table = self.browser.find_element_by_id('id_list_table')
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_list_table"}' ; Stacktrace [...]
```

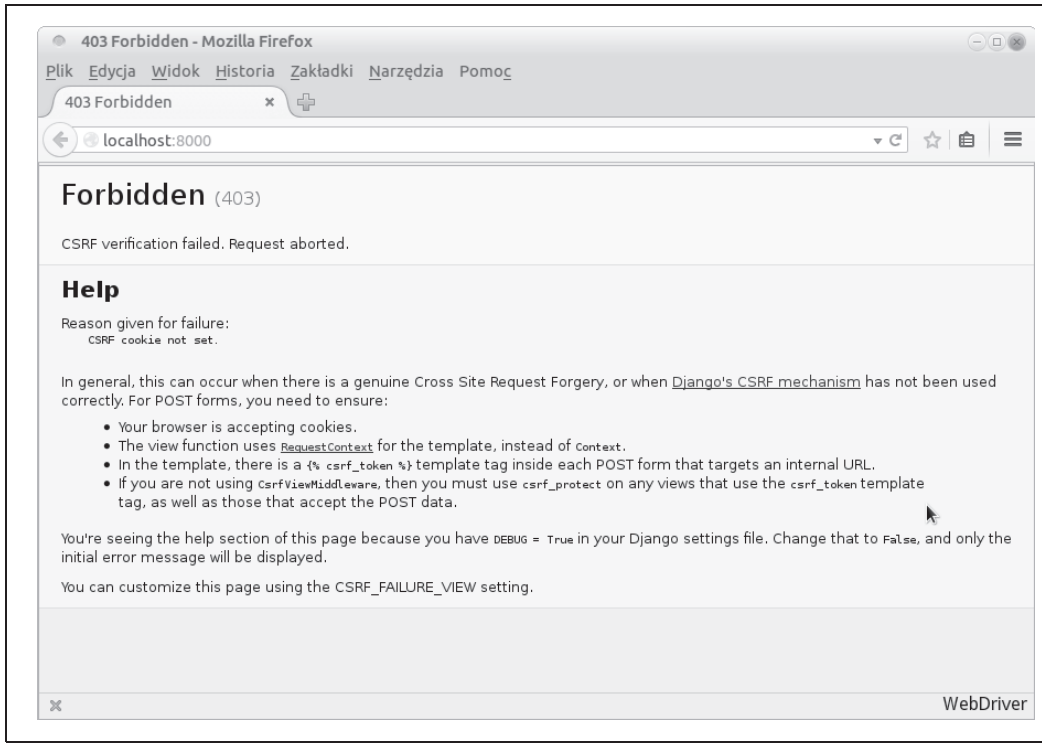
Kiedy test funkcjonalny kończy się nieoczekiwanym niepowodzeniem, wówczas możemy podjąć wiele różnych działań, aby odszukać źródło błędu:

- Dodanie poleceń print w celu wyświetlania na przykład tekstu bieżącej strony.
- Usprawnienie *komunikatu błędu*, aby wyświetlał więcej informacji o bieżącym stanie.
- Ręczne odwiedzenie problematycznej witryny.
- Użycie wywołania time.sleep() w celu zrobienia przerwy podczas wykonywania testu.

W trakcie lektury książki spotkasz się z wszystkimi wymienionymi powyżej działaniami. Wywołanie time.sleep() to opcja, z której osobiście korzystam bardzo często. Wypróbujmy więc to rozwiązanie w omawianym przykładzie. Przerwę dodamy przed wystąpieniem błędu.

```
# Po naciśnięciu klawisza Enter strona została uaktualniona i wyświetla
# "1: Kupić pawie pióra" jako element listy rzeczy do zrobienia.
inputbox.send_keys(Keys.ENTER)

import time
time.sleep(10)
table = self.browser.find_element_by_id('id_list_table')
```

W zależności od szybkości działania narzędzia Selenium w Twoim komputerze źródło problemu mogłeś dostrzec już wcześniej. Jednak po ponownym wykonaniu testów funkcjonalnych masz wystarczająco dużo czasu na zobaczenie, co tak naprawdę się dzieje. Na ekranie powinieneś zobaczyć stronę podobną do pokazanej na rysunku 5.1 i wyświetlającą wiele wygenerowanych przez Django informacji o błędzie.



Rysunek 5.1. Framework Django wyświetla informacje o błędzie CSRF

Zabezpieczenia — zaskakująco zabawne!

Jeżeli nigdy wcześniej nie słyszałeś o atakach typu CSRF (ang. *cross-site request forgery*), to warto teraz nadrobić tę zaległość. Podobnie jak w przypadku wszystkich luk w zabezpieczeniach lektura o genialnych rozwiązaniach pozwalających na wykorzystanie systemu w nieoczekiwany sposób może dostarczyć wiele radości…

Kiedy wróciłem na uniwersytet, aby uzyskać dyplom z informatyki, zapisałem się na zajęcia z zakresu bezpieczeństwa. Pomyślałem wtedy: *cóż, to prawdopodobnie będą bardzo nudne zajęcia, ale lepiej, jeśli będę w nich uczestniczył*. Okazało się, że to były najbardziej fascynujące zajęcia, jakie miałem na całych studiach. W takcie tych zajęć dużo czasu poświęciliśmy na hacking oraz rozważania, jak systemy można wykorzystywać na zupełnie nieoczekiwane sposoby.

Jako lekturę mogę polecić pozycję, z której korzystałem podczas moich zajęć — *Inżynieria zabezpieczeń*[1] napisana przez Rossa Andersona. W wymienionej książce nie znajdziesz zbyt wiele o samej kryptografii, ale została wypełniona omówieniem innych interesujących tematów, takich jak wybór zabezpieczeń, podrabianie informacji generowanych przez bank, ekonomia kartridżów w drukarkach atramentowych, a także oszukiwanie myśliwców RPA za pomocą ataków metodą powtórzenia. To jest dość obszerna pozycja, ale zapewniam Cię, że nie będziesz mógł się od niej oderwać.

[1] *http://helion.pl/ksiazki/inzynieria-zabezpieczen-ross-anderson,a_000w.htm*

Oferowane przez framework Django zabezpieczenia przed atakami typu CSRF obejmują między innymi umieszczenie w każdym wygenerowanym formularzu niewielkiego tokenu. Dzięki temu żądanie `POST` można zidentyfikować jako pochodzące z pierwotnej witryny. Jak dotąd omawiany szablon składa się wyłącznie z kodu HTML. Kolejnym krokiem jest więc wykorzystanie po raz pierwszy oferowanej przez Django magii szablonów. W celu dodania tokenu CSRF użyjemy tak zwanego *znacznika szablonu* składającego się z nawiasu klamrowego i znaku procentu. Wymieniony znacznik jest znany jako najbardziej irytująca na świecie kombinacja dwóch klawiszy.

Plik *lists/templates/home.html*:

```
<form method="POST">
    <input name="item_text" id="id_new_item" placeholder="Wpisz rzecz do zrobienia" />
    {% csrf_token %}
</form>
```

Podczas generowania strony Django zastąpi dodany znacznik elementem `<input type="hidden">` zawierającym token CSRF. Ponowne wykonanie testu zakończy się teraz oczekiwanym niepowodzeniem:

```
AssertionError: False is not true : Nowy element nie znajduje się w tabeli.
```

Ponieważ w kodzie nadal znajduje się wywołanie `time.sleep()`, wykonywanie testu zostanie przerwane na pewien czas i będziesz mógł zobaczyć, że nowy element listy znika po wysłaniu formularza, a strona zostaje odświeżona i ponownie wyświetla pusty formularz. Po prostu nie skonfigurowaliśmy jeszcze serwera do obsługi żądań `POST`. Dlatego też są one ignorowane, a przeglądarka internetowa wyświetla zwykłą stronę główną.

Teraz możemy już usunąć wywołanie `time.sleep()`.

Plik *functional_tests.py*:

```
    # "1: Kupić pawie pióra" jako element listy rzeczy do zrobienia.
    inputbox.send_keys(Keys.ENTER)
    table = self.browser.find_element_by_id('id_list_table')
```

Przetwarzanie żądania POST w serwerze

Ponieważ w formularzu sieciowym nie użyliśmy atrybutu `action=`, po jego wysłaniu następuje domyślnie przejście do adresu URL, w którym nastąpiło wygenerowanie formularza (na przykład /). W omawianym przypadku wyświetlanie strony jest obsługiwane przez funkcję `home_page()`. Przystosujmy więc widok do obsługi żądania `POST`.

To oznacza konieczność utworzenia nowego testu jednostkowego dla widoku `home_page`. Otwórz plik *lists/tests.py* i dodaj nową metodę do `HomePageTest`. Osobiście skopiowałem poprzednią metodę, a następnie przystosowałem ją do obsługi żądania `POST` i upewniłem się, że zwrócony kod HTML zawiera tekst nowego elementu listy rzeczy do zrobienia.

Plik *lists/tests.py* (ch05l005):

```
def test_home_page_returns_correct_html(self):
    [...]

def test_home_page_can_save_a_POST_request(self):
    request = HttpRequest()
    request.method = 'POST'
    request.POST['item_text'] = 'Nowy element listy'

    response = home_page(request)

self.assertIn('Nowy element listy', response.content.decode())
```

Czy zastanowiły Cię puste wiersze w kodzie nowej metody? Na początku zgrupowałem trzy wiersze zawierające konfigurację testu. Jeden wiersz w środku to faktyczne wywołanie testowanej funkcji, natomiast na końcu mamy asercję. Wprawdzie stosowanie takiego rozwiązania nie jest obligatoryjne, ale pomaga w dostrzeżeniu struktury testu. Konfiguracja, sprawdzenie i asercja to typowa struktura testu jednostkowego.

Jak możesz zobaczyć, w kodzie użyliśmy dwóch atrybutów specjalnych obiektu `HttpRequest`: `.method` i `.POST` (ich nazwy powinny wyraźnie wskazywać przeznaczenie atrybutów, ale i tak warto zajrzeć do *poświęconej im dokumentacji*[2] w witrynie Django). Następnie sprawdzamy, czy tekst przekazany w żądaniu `POST` znajduje się w wygenerowanym kodzie HTML. Wynikiem jest oczekiwane niepowodzenie testu:

```
$ python3 manage.py test
[...]
AssertionError: 'Nowy element listy' not found in '<html> [...]
```

Test może zostać zaliczony przez dodanie polecenia `if` i dostarczenie zupełnie innego kodu przeznaczonego do obsługi żądań `POST`. W typowym stylu TDD rozpoczynamy od celowego zdefiniowania zupełnie nieprawidłowej wartości zwrotnej.

Plik *lists/views.py*:

```
from django.http import HttpResponse
from django.shortcuts import render

def home_page(request):
    if request.method == 'POST':
        return HttpResponse(request.POST['item_text'])
    return render(request, 'home.html')
```

W ten sposób test jednostkowy zostaje zaliczony, ale tak naprawdę to nie jest rozwiązanie, które nas interesuje. Naszym celem jest umieszczenie w tabeli wyświetlanej na stronie głównej danych przekazanych przez żądanie `POST`.

Przekazanie zmiennych Pythona do wygenerowania w szablonie

Mieliśmy już przedsmak, a teraz zaczniemy w pełni wykorzystywać potężne możliwości składni szablonów Django, co pozwoli na przekazywanie zmiennych z kodu widoku Pythona do szablonów HTML.

Na początek musisz zobaczyć, jak składnia szablonu pozwala na umieszczenie w nim obiektu Pythona. Notacja w postaci `{{ ... }}` wyświetla obiekt jako ciąg tekstowy.

Plik *lists/templates/home.html*:

```
<body>
    <h1>Twoja lista rzeczy do zrobienia</h1>
    <form method="POST">
        <input name="item_text" id="id_new_item" placeholder="Wpisz rzecz do zrobienia" />
        {% csrf_token %}
    </form>
```

[2] *https://docs.djangoproject.com/en/1.7/ref/request-response/*

```
        <table id="id_list_table">
            <tr><td>{{ new_item_text }}</td></tr>
        </table>
    </body>
```

W jaki sposób można sprawdzić, czy widok otrzymuje prawidłową wartość dla `new_item_text`? Jak w ogóle można przekazać zmienną do szablonu? Możemy się o tym przekonać, faktycznie wykonując test jednostkowy. Funkcji `render_to_string()` użyliśmy w poprzednim teście jednostkowym w celu ręcznego wygenerowania szablonu i porównania go z kodem HTML wygenerowanym przez widok. Teraz dodamy zmienną, która ma zostać przekazana.

Plik *lists/tests.py*:

```
    self.assertIn('Nowy element listy', response.content.decode())
    expected_html = render_to_string(
        'home.html',
        {'new_item_text': 'Nowy element listy'}
    )
    self.assertEqual(response.content.decode(), expected_html)
```

Jak możesz zobaczyć, funkcja `render_to_string()` pobiera jako drugi argument mapowanie nazw zmiennych na wartości. Szablon otrzymuje zmienną o nazwie `new_item_text`, której wartością powinien być tekst pobrany z żądania `POST`.

Po wykonaniu testu jednostkowego funkcja `render_to_string()` zastąpi wewnątrz elementu `<td>` notację `{{ new_item_text }}` wartością `Nowy element listy`. Rzeczywisty widok nie wykonuje takiej operacji, a więc powinniśmy spodziewać się niepowodzenia testu:

```
        self.assertEqual(response.content.decode(), expected_html)
    AssertionError: 'Nowy element listy' != '<html>\n    <head>\n [...]
```

Doskonale. Celowo ustalona wcześniej nieprawdziwa wartość zwrotna nie będzie dłużej oszukiwała testów. Możemy więc zmodyfikować kod widoku i nakazać mu przekazanie parametru `POST` do szablonu.

Plik *lists/views.py* (ch05l009):

```
    def home_page(request):
        return render(request, 'home.html', {
            'new_item_text': request.POST['item_text'],
        })
```

Teraz ponownie wykonujemy test jednostkowy:

```
    ERROR: test_home_page_returns_correct_html (lists.tests.HomePageTest)
    [...]
        'new_item_text': request.POST['item_text'],
    KeyError: 'item_text'
```

Wynikiem testu jest *oczekiwane niepowodzenie*.

Jeżeli przypomnisz sobie reguły odczytu stosu wywołań, to zauważysz, że niepowodzeniem zakończył się *inny* test. Udało nam się osiągnąć zaliczenie testu, nad którym pracowaliśmy. Natomiast wybrane testy jednostkowe spowodowały powstanie nieoczekiwanych konsekwencji w postaci regresji — uszkodziliśmy funkcjonalność kodu przeznaczonego do obsługi sytuacji, gdy nie występuje żądanie `POST`.

Teraz już widzisz, jak ważne jest przygotowywanie testów. Wprawdzie w omawianym przypadku można się było spodziewać uszkodzenia funkcjonalności, ale wyobraź sobie sytuację, gdy masz zły dzień lub po prostu nie zwróciłeś wystarczającej uwagi. Wówczas testy

uchronią przed uszkodzeniem funkcjonalności aplikacji, a ponieważ stosujemy techniki TDD, to dowiemy się o tym natychmiast. Nie trzeba czekać na reakcję działu odpowiedzialnego za kontrolę jakości lub też przechodzić do przeglądarki internetowej i ręcznie sprawdzać witrynę. Problem można usunąć od razu, a rozwiązanie przedstawiono poniżej.

Plik *lists/views.py*:

```
def home_page(request):
    return render(request, 'home.html', {
        'new_item_text': request.POST.get('item_text', ''),
    })
```

Jeżeli nie jesteś pewien, jak działa powyższe rozwiązanie, spójrz na `dict.get()`.

Test jednostkowy powinien zostać teraz zaliczony. Zobaczmy, jak przedstawia się wynik testu funkcjonalnego:

```
AssertionError: False is not true : Nowy element nie znajduje się w tabeli.
```

Cóż, komunikat błędu nie okazuje się szczególnie użyteczny. Wykorzystamy więc kolejną technikę usuwania błędów z testów funkcjonalnych, czyli poprawimy komunikat błędu. To jest prawdopodobnie najbardziej konstruktywna technika, ponieważ poprawione komunikaty błędów już pozostaną w aplikacji i będą pomocne podczas usuwania błędów w przyszłości.

Plik *functional_tests.py*:

```
self.assertTrue(
    any(row.text == '1: Kupić pawie pióra' for row in rows),
    "Nowy element nie znajduje się w tabeli -- jego tekst to:\n%s" % (
        table.text,
    )
)
```

W ten sposób otrzymujemy znacznie użyteczniejszy komunikat błędu:

```
AssertionError: False is not true : Nowy element nie znajduje się w tabeli --
jego tekst to:
Kupić pawie pióra
```

Czy wiesz, że można przygotować jeszcze lepsze rozwiązanie niż obecne? Zastosowanie wskazanej asercji nie jest aż tak sprytnym podejściem. Jak zapewne pamiętasz, byłem zadowolony z wykorzystania funkcji `any()`. Jednak jeden z pierwszych czytelników książki (dziękuję, Jasonie!) zasugerował mi użycie znacznie prostszej implementacji. Wszystkie sześć wierszy `assertTrue()` można zastąpić pojedynczym wierszem `assertIn()`.

Plik *functional_tests.py*:

```
self.assertIn('1: Kupić pawie pióra', [row.text for row in rows])
```

Znacznie lepiej. Zawsze warto zastanowić się, czy można zastosować inne, sprytniejsze rozwiązanie, ponieważ prawdopodobnie używasz niepotrzebnie *zbyt skomplikowanego*. Bonusem będzie otrzymanie odpowiedniego komunikatu błędu:

```
        self.assertIn('1: Kupić pawie pióra', [row.text for row in rows])
AssertionError: '1: Kupić pawie pióra' not found in ['Kupić pawie pióra']
```

Mam nadzieję, że dobrze się tutaj zrozumiemy. Test funkcjonalny oczekuje otrzymania elementu numerowanej listy wraz ze znakami `1:` na początku pierwszego elementu. Najszybszym sposobem zaliczenia testu będzie małe „oszustwo” w postaci modyfikacji wprowadzonej w szablonie.

Plik *lists/templates/home.html*:

```
<tr><td>1: {{ new_item_text }}</td></tr>
```

Czerwony/zielony/refaktoryzacja i triangulacja

Cykl test jednostkowy i tworzenie kodu jest czasami określany mianem *czerwony, zielony, refaktoryzacja*:

- Zaczynamy od utworzenia testu jednostkowego, którego wykonanie kończy się niepowodzeniem (*czerwony*).
- Dodajemy minimalną możliwą ilość kodu w celu zaliczenia testu (*zielony*), *nawet jeśli oznacza to konieczność ucieknięcia się do oszustwa*.
- Przeprowadzamy *refaktoryzację*, aby otrzymać kod lepszej jakości i bardziej przejrzysty.

Jakie działania podejmujemy na etapie refaktoryzacji? Co usprawiedliwia przejście od implementacji, w której „oszukujemy", do implementacji, z której jesteśmy zadowoleni?

Jedną z metodologii jest *eliminacja powielania*. Jeżeli test używa magicznej stałej (na przykład znaków `1:` na początku elementu listy, jak w omawianej sytuacji), wówczas kod aplikacji również korzysta z tej stałej. Mamy więc do czynienia z powielaniem kodu, co jest wystarczającym powodem do przeprowadzenia refaktoryzacji. Usunięcie magicznej stałej z kodu aplikacji zwykle oznacza konieczność zaprzestania oszukiwania.

Przekonałem się, że powyższe rozwiązanie oznacza powstanie nieco niejednoznacznego kodu. Dlatego też najczęściej decyduję się na drugą technikę, o nazwie *triangulacja*. Jeżeli test można zaliczyć, tworząc „oszukujący" kod, z którego nie jesteś zadowolony (na przykład zwrot magicznej stałej), wówczas *utwórz kolejny test* wymuszający opracowanie lepszego kodu. Takie podejście zastosujemy podczas rozbudowy testu funkcjonalnego o sprawdzenie, czy znaki `2:` znajdują się na początku *drugiego* elementu listy rzeczy do zrobienia.

Teraz docieramy do wywołania `self.fail('Zakończenie testu!')`. Jeżeli rozbudujemy test funkcjonalny (metodą kopiuj i wklej, mój przyjacielu) o sprawdzenie, czy drugi element listy został dodany do tabeli, to przekonamy się, że nasze rozwiązanie tak naprawdę jest niezbyt dobre.

Plik *functional_tests.py*:

```
    # Na stronie nadal znajduje się pole tekstowe zachęcające do podania kolejnego zadania.
    # Edyta wpisała "Użyć pawich piór do zrobienia przynęty" (Edyta jest niezwykle skrupulatna).
    inputbox = self.browser.find_element_by_id('id_new_item')
    inputbox.send_keys('Użyć pawich piór do zrobienia przynęty')
    inputbox.send_keys(Keys.ENTER)

    # Strona została ponownie uaktualniona i teraz wyświetla dwa elementy na liście rzeczy do zrobienia.
    table = self.browser.find_element_by_id('id_list_table')
    rows = table.find_elements_by_tag_name('tr')
    self.assertIn('1: Kupić pawie pióra', [row.text for row in rows])
    self.assertIn(
        '2: Użyć pawich piór do zrobienia przynęty' , [row.text for row in rows]
    )
    # Edyta była ciekawa, czy witryna zapamięta jej listę. Zwróciła uwagę na
    # wygenerowany dla niej unikatowy adres URL, obok którego znajduje się
    # pewien tekst z wyjaśnieniem.
    self.fail('Zakończenie testu!')

    # Przechodzi pod podany adres URL i widzi wyświetloną swoją listę rzeczy do zrobienia.
```

Zgodnie z oczekiwaniami nasz test funkcjonalny powoduje wygenerowanie błędu:

```
AssertionError: '1: Kupić pawie pióra' not found in ['1: Use peacock
feathers to make a fly']
```

Do trzech razy sztuka, a później refaktoryzacja

Zanim przejdziemy dalej, mamy nieprzyjemny *zapach kodu*[3] w przedstawionym teście funkcjonalnym. Istnieją trzy prawie identyczne bloki kodu odpowiedzialne za sprawdzanie nowych elementów na liście. Warto przypomnieć sobie regułę DRY (ang. *don't repeat yourself*, nie powtarzaj się), którą możemy tutaj wykorzystać przez zastosowanie mantry *do trzech razy sztuka, a później refaktoryzacja*. Raz można skopiować kod i wkleić go w innym miejscu; usunięcie tego rodzaju powielenia może okazać się przedwczesne. Jednak mając już trzy wystąpienia danego kodu, najwyższa pora na pozbycie się powielonego kodu.

Pracę rozpoczynamy od przekazania do repozytorium zmodyfikowanych dotąd plików. Decydujemy się na to, wiedząc, że tworzona witryna zawiera poważny błąd (możliwość obsługi tylko jednego elementu listy), ale i tak poczyniliśmy postępy względem ostatniej wersji znajdującej się w repozytorium. Być może utworzymy cały kod od początku, a może nie, ale regułą jest, aby przed przeprowadzeniem jakiejkolwiek refaktoryzacji zawsze przekazać kod do repozytorium:

```
$ git diff
# Polecenie powinno wyświetlić zmiany wprowadzone w plikach
# functional_tests.py, home.html, tests.py i views.py.
$ git commit -a
```

Powracamy teraz do refaktoryzacji testu funkcjonalnego. Wprawdzie można użyć funkcji typu inline, ale to na pewno nieco zakłóci przebieg testu. Dlatego też wykorzystamy metodę pomocniczą — pamiętaj, że tylko metody o nazwach rozpoczynających się od `test_` zostaną wykonane jako testy. Pozostałych metod można więc użyć do własnych celów.

Plik *functional_tests.py*:

```
    def tearDown(self):
        self.browser.quit()

    def check_for_row_in_list_table(self, row_text):
        table = self.browser.find_element_by_id('id_list_table')
        rows = table.find_elements_by_tag_name('tr')

        self.assertIn(row_text, [row.text for row in rows])

    def test_can_start_a_list_and_retrieve_it_later(self):
        [...]
```

Metody pomocnicze lubię umieszczać gdzieś na początku klasy, między metodą `tearDown()` i pierwszym testem. Umieśćmy ją w naszym teście funkcjonalnym.

Plik *functional_tests.py*:

```
    # Po naciśnięciu klawisza Enter strona została uaktualniona i wyświetla
    # "1: Kupić pawie pióra" jako element listy rzeczy do zrobienia.
    inputbox.send_keys(Keys.ENTER)
    self.check_for_row_in_list_table('1: Kupić pawie pióra')

    # Na stronie nadal znajduje się pole tekstowe zachęcające do podania kolejnego zadania.
    # Edyta wpisała "Użyć pawich piór do zrobienia przynęty" (Edyta jest niezwykle skrupulatna).
```

[3] Jeżeli nie znasz koncepcji *zapachu kodu*, to powinieneś wiedzieć, że oznacza ona ten fragment kodu, który zmusza Cię do jego refaktoryzacji. Jeff Atwood napisał interesujący artykuł (*http://blog.codinghorror.com/code-smells/*) na ten temat. Im większe doświadczenie będziesz zdobywał w obszarze programowania, tym bardziej będziesz miał wyczulony nos na zapach kodu…

```
    inputbox = self.browser.find_element_by_id('id_new_item')
    inputbox.send_keys('Użyć pawich piór do zrobienia przynęty')
    inputbox.send_keys(Keys.ENTER)

    # Strona została ponownie uaktualniona i teraz wyświetla dwa elementy na liście rzeczy do zrobienia.
    self.check_for_row_in_list_table('1: Kupić pawie pióra')
    self.check_for_row_in_list_table('2: Użyć pawich piór do zrobienia przynęty')

    # Edyta była ciekawa, czy witryna zapamięta jej listę. Zwróciła uwagę na
    [...]
```

Po ponownym wykonaniu testu funkcjonalnego okazuje się, że jego zachowanie nie uległo zmianie…

```
AssertionError: '1: Kupić pawie pióra' not found in ['1: Użyć pawich
piór do zrobienia przynęty']
```

Dobrze. Teraz możemy przekazać pliki do repozytorium i potraktować refaktoryzację jako małą, niezależną zmianę:

```
$ git diff # Przejrzenie zmian wprowadzonych w pliku functional_tests.py.
$ git commit -a
```

Powracamy do pracy. Jeżeli kiedykolwiek zajdzie potrzeba obsługi więcej niż tylko jednego elementu listy, wtedy będziemy potrzebować pewnego rodzaju trwałego magazynu danych. Baza danych to niezawodne rozwiązanie, które możemy zastosować w tym obszarze.

Django ORM i nasz pierwszy model

ORM (ang. *object-relational mapper*, mapowanie obiektowo-relacyjne) to warstwa abstrakcji przeznaczona dla danych przechowywanych w tabelach, rekordach i kolumnach bazy danych. Pozwala na pracę z bazami danych za pomocą doskonale znanych metafor zorientowanych obiektowo, które sprawdzają się w kodzie aplikacji. Klasy są mapowane na tabele, atrybuty na kolumny, a poszczególne egzemplarze klasy przedstawiają rekordy danych przechowywanych w bazie danych.

Framework Django jest dostarczany wraz z doskonałą warstwą ORM. Utworzenie testu jednostkowego opartego na tej warstwie będzie najlepszym sposobem jej poznania, ponieważ przeanalizujemy kod, przygotowując go do działania w interesujący nas sposób.

Rozpoczynamy od utworzenia nowej klasy w pliku *lists/tests.py*.

Plik *lists/tests.py*:

```
from lists.models import Item
[...]

class ItemModelTest(TestCase):

    def test_saving_and_retrieving_items(self):
        first_item = Item()
        first_item.text = 'Absolutnie pierwszy element listy'
        first_item.save()

        second_item = Item()
        second_item.text = 'Drugi element'
        second_item.save()
```

```
    saved_items = Item.objects.all()
    self.assertEqual(saved_items.count(), 2)

    first_saved_item = saved_items[0]
    second_saved_item = saved_items[1]
    self.assertEqual(first_saved_item.text, 'Absolutnie pierwszy element listy')
    self.assertEqual(second_saved_item.text, 'Drugi element')
```

Jak możesz zobaczyć, utworzenie nowego rekordu w bazie danych to względnie proste zadanie. Sprowadza się do utworzenia obiektu, przypisania mu pewnych atrybutów, a następnie wywołania funkcji `save()`. Django oferuje API przeznaczone do wykonywania zapytań w bazie danych za pomocą atrybutu klasy `.objects`. W omawianym przykładzie używamy najprostszego z możliwych zapytań `.all()`, które pobiera wszystkie rekordy ze wskazanej tabeli. Wynikiem jest przypominający listę obiekt o nazwie `QuerySet`, z którego można wyodrębniać poszczególne obiekty, a ponadto wywoływać inne funkcje, na przykład `count()`. Następnie sprawdzamy, czy obiekty zostały zapisane w bazie danych, aby upewnić się o zachowaniu w niej właściwych informacji.

Oferowana przez Django warstwa ORM zawiera jeszcze wiele innych intuicyjnych w użyciu funkcji. Warto przynajmniej przejrzeć *samouczek Django*[4] poświęcony ORM, który stanowi doskonałe wprowadzenie do wspomnianych funkcji.

Przedstawiony powyżej test jednostkowy utworzyłem w bardzo rozwlekłym stylu i potraktowałem go jako wprowadzenie do warstwy ORM w Django. Dla klasy modelu możesz przygotować znacznie krótszy test, o czym się przekonasz w rozdziale 11.

Terminologia 2: test jednostkowy kontra test integracji a baza danych

Puryści będą się upierać, że „rzeczywisty" test jednostkowy nigdy nie powinien opierać się na bazie danych, a przygotowany tutaj test powinien być nazwany testem integracji, ponieważ nie sprawdza jedynie kodu, ale opiera się również na systemie zewnętrznym, jakim tutaj jest baza danych.

Na chwilę obecną możemy zrezygnować z rozróżniania wymienionych testów. Mamy dwa typy testów. Pierwszy to wysokiego poziomu testy funkcjonalne przeznaczone do testowania aplikacji z perspektywy użytkownika. Drugi to niskiego poziomu testy przeznaczone do testowania aplikacji z perspektywy programisty.

Do tego tematu oraz omówienia testów jednostkowych i integracji powrócimy jeszcze w rozdziale 19. i dalej w książce.

Spróbujmy teraz wykonać test jednostkowy. Mamy do czynienia z kolejnym cyklem test jednostkowy — tworzenie kodu:

```
ImportError: cannot import name 'Item'
```

Doskonale. Zacznijmy od możliwości zaimportowania czegokolwiek z pliku *lists/models.py*. Czujemy, że możemy pominąć krok `Item = None` i od razu przejść do utworzenia klasy.

[4] *https://docs.djangoproject.com/en/1.7/intro/tutorial01/*

Plik *lists/models.py*:

```
from django.db import models

class Item(object):
    pass
```

Wykonanie testu kończy się w następujący sposób:

```
        first_item.save()
AttributeError: 'Item' object has no attribute 'save'
```

Aby można było udostępnić klasie `Item` metodę `save()` i tym samym zmienić ją w prawdziwy model Django, musi ona dziedziczyć po klasie `Model`.

Plik *lists/models.py*:

```
from django.db import models

class Item(models.Model):
    pass
```

Pierwsza migracja bazy danych

Możemy się teraz spodziewać błędu wygenerowanego przez bazę danych:

```
first_item.save()
  File "/usr/local/lib/python3.4/dist-packages/django/db/models/base.py", line
593, in save
[...]
    return Database.Cursor.execute(self, query, params)
django.db.utils.OperationalError: no such table: lists_item
```

W Django zadaniem warstwy ORM jest modelowanie bazy danych. Istnieje jeszcze drugi system, o nazwie *migracje*, odpowiedzialny za faktyczne tworzenie bazy danych. Jego zadaniem jest umożliwienie programiście dodawania i usuwania tabel oraz kolumn na podstawie zmian wprowadzanych w pliku *models.py*.

Migracje możesz potraktować jako system kontroli wersji dla bazy danych. Jak się wkrótce przekonasz, okazuje się to szczególnie użyteczne, gdy zachodzi potrzeba uaktualnienia bazy danych wdrożonej w działającym serwerze.

Na obecnym etapie musisz jedynie wiedzieć, jak przygotować pierwszą migrację bazy danych. Do tego celu wykorzystamy polecenie `makemigrations`:

```
$ python3 manage.py makemigrations
Migrations for 'lists':
  0001_initial.py:
    - Create model Item
$ ls lists/migrations
0001_initial.py  __init__.py  __pycache__
```

Jeżeli jesteś ciekaw, zajrzyj do plików migracji. Przekonasz się, że stanowią one reprezentację zmian wprowadzonych w pliku *models.py*.

W międzyczasie powinniśmy posunąć nasze testy nieco do przodu.

Zdumiewająco duży postęp w teście

Osiągnęliśmy zdumiewający postęp w teście:

```
$ python3 manage.py test lists
[...]
    self.assertEqual(first_saved_item.text, 'Absolutnie pierwszy element listy')
AttributeError: 'Item' object has no attribute 'text'
```

W porównaniu z poprzednim niepowodzeniem udało nam się przejść o osiem wierszy kodu do przodu. Poczyniliśmy kroki mające na celu zachowanie w bazie danych dwóch obiektów `Item` i upewniliśmy się o ich zapisaniu w bazie danych. Wydaje się jednak, że Django nie zachowało atrybutu `text`.

Nawiasem mówiąc, jeżeli dopiero rozpoczynasz programowanie w Pythonie, to w ogóle możesz być zaskoczony możliwością przypisania wartości atrybutowi `text`. W językach takich jak Java tego rodzaju operacja prawdopodobnie zakończy się wygenerowaniem błędu kompilacji. Pod tym względem Python jest znacznie elastyczniejszy.

Klasy dziedziczące po `models.Model` odpowiadają za mapowanie obiektów na tabele w bazie danych. Domyślnie otrzymują automatycznie wygenerowany atrybut `id` będący kolumną klucza podstawowego w bazie danych. Jednak wszystkie pozostałe kolumny musisz wyraźnie zdefiniować. Poniżej przedstawiono przykład pokazujący konfigurację kolumny tekstowej.

Plik *lists/models.py*:

```
class Item(models.Model):
    text = models.TextField()
```

Django oferuje wiele innych typów kolumn, na przykład `IntegerField`, `CharField`, `DateField` itd. Zdecydowałem się na `TextField` zamiast `CharField`, ponieważ drugi z wymienionych typów wymaga zastosowania obszernych ograniczeń, co na obecnym etapie wydaje się niepotrzebne. Więcej informacji dotyczących różnych typów kolumn znajdziesz w *samouczku*[5] Django oraz w *dokumentacji*[6].

Nowa kolumna oznacza nową migrację

Wykonanie testów powoduje wygenerowanie kolejnego błędu bazy danych:

```
django.db.utils.OperationalError: table lists_item has no column named text
```

Błąd jest skutkiem dodania nowej kolumny do bazy danych, co oznacza konieczność przeprowadzenia kolejnej migracji. Dobrze, że testy mogą nas o tym poinformować!

Spróbujmy więc przeprowadzić migrację:

```
$ python3 manage.py makemigrations
You are trying to add a non-nullable field 'text' to item without a default;
we can't do that (the database needs something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows)
 2) Quit, and let me add a default in models.py
Select an option:2
```

[5] *https://docs.djangoproject.com/en/1.7/intro/tutorial01/#creating-models*

[6] *https://docs.djangoproject.com/en/1.7/ref/models/fields/*

Hm, nie możemy dodać kolumny bez wartości domyślnej. Wybieramy więc opcję drugą i ustalamy wartość domyślną w pliku *models.py*. Jak sądzę, składnia nie wymaga dodatkowych wyjaśnień.

Plik *lists/models.py*:

```
class Item(models.Model):
    text = models.TextField(default='')
```

Teraz migracja powinna zakończyć się powodzeniem:

```
$ python3 manage.py makemigrations
Migrations for 'lists':
  0002_item_text.py:
    - Add field text to item
```

Po dodaniu dwóch nowych wierszy kodu w pliku *models.py* oraz po przeprowadzeniu dwóch migracji bazy atrybut `text` obiektów modelu zostaje wreszcie uznany za atrybut specjalny i zapisany w bazie danych. Skutkiem jest zaliczenie testu…

```
$ python3 manage.py test lists
[...]
Ran 4 tests in 0.010s
OK
```

Do repozytorium możemy więc przekazać nasz pierwszy model!

```
$ git status # Polecenie wyświetla pliki tests.py, models.py oraz dwie niemonitorowane migracje.
$ git diff # Polecenie pozwala na przejrzenie zmian wprowadzonych w plikach tests.py i models.py.
$ git add lists
$ git commit -m"Model dla obiektów Items oraz powiązane z nim migracje."
```

Zapis w bazie danych informacji z żądania POST

Dostosujmy teraz test żądania `POST` strony głównej. Przyjmujemy założenie, że celem jest zapisanie przez widok nowego elementu w bazie danych zamiast po prostu umieszczenie go w odpowiedzi. Wystarczy dodać trzy nowe wiersze do istniejącego testu `test_home_page_can_save_a_POST_request()`.

Plik *lists/tests.py*:

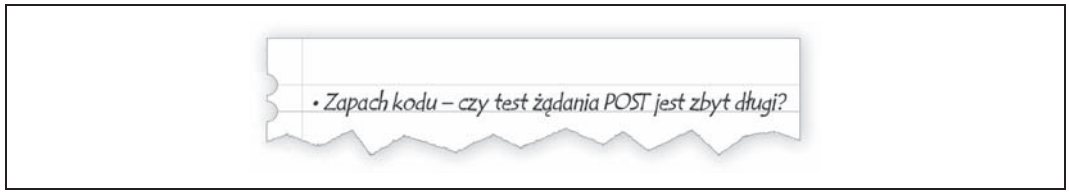
```
def test_home_page_can_save_a_POST_request(self):
    request = HttpRequest()
    request.method = 'POST'
    request.POST['item_text'] = 'Nowy element listy'

    response = home_page(request)

    self.assertEqual(Item.objects.count(), 1)  #❶
    new_item = Item.objects.first()  #❷
    self.assertEqual(new_item.text, 'Nowy element listy')  #❸

    self.assertIn('Nowy element listy', response.content.decode())
    expected_html = render_to_string(
        'home.html',
        {'new_item_text':  'Nowy element listy'}
    )
    self.assertEqual(response.content.decode(), expected_html)
```

❶ Sprawdzamy, czy nowy obiekt `Item` został zapisany w bazie danych. Wywołanie `objects.↪count()` to skrócona forma wywołania `objects.all().count()`.

❷ Wywołanie `objects.first()` ma taki sam efekt jak `objects.all()[0]`.

❸ Sprawdzamy, czy tekst elementu jest prawidłowy.

Ten test jest nieco rozwlekły. Wydaje się, że testowana jest duża ilość różnych rzeczy. Mamy więc kolejny przykład *zapachu kodu* — długi test jednostkowy, który powinien zostać podzielony na dwa lub jest sygnałem, że istnieje zbyt skomplikowany przedmiot testu. Tę kwestię dodajmy do naszej krótkiej, osobistej listy rzeczy do zrobienia, zapisanej na przykład na kartce papieru (patrz rysunek 5.2).



Rysunek 5.2. Nasza osobista lista rzeczy do zrobienia

Zapisanie wymienionej kwestii na kartce papieru gwarantuje, że o niej nie zapomnimy. Teraz możemy spokojnie powrócić do przerwanej pracy. Ponownie wykonujemy testy i otrzymujemy wynik w postaci oczekiwanego niepowodzenia:

```
        self.assertEqual(Item.objects.count(), 1)
    AssertionError: 0 != 1
```

Przystępujemy do modyfikacji widoku.

Plik *lists/views.py*:

```
    from django.shortcuts import render
    from lists.models import Item

    def home_page(request):
        item = Item()
        item.text = request.POST.get('item_text', '')
        item.save()

        return render(request, 'home.html', {
            'new_item_text': request.POST.get('item_text', ''),
        })
```

Przygotowałem bardzo naiwne rozwiązanie i prawdopodobnie od razu dostrzeżesz niezwykle oczywisty problem polegający na próbie zapisu pustych elementów w trakcie każdego żądania wykonywanego względem strony głównej. Dodajmy rozwiązanie tego problemu do naszej osobistej listy rzeczy do zrobienia. Na razie oczywiste jest, że nie mamy żadnej możliwości przechowywania różnych list dla poszczególnych osób. Spróbujmy to teraz zignorować.

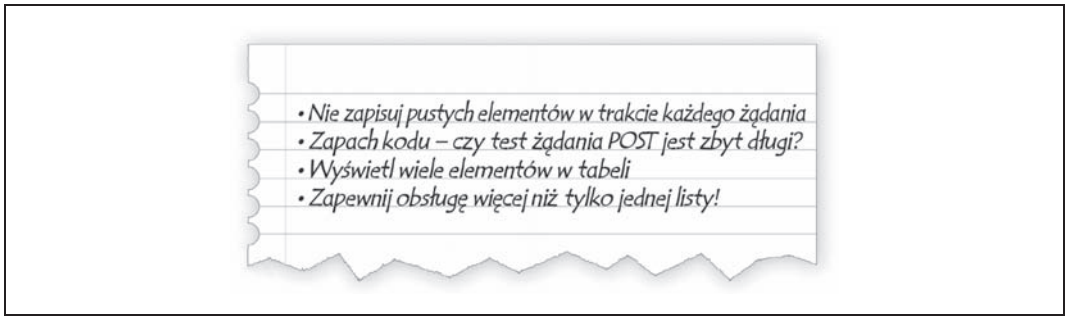
To oczywiście nie oznacza, że „w rzeczywistych projektach" zawsze powinniśmy ignorować tak rażące problemy. Kiedy problem zostanie wcześnie dostrzeżony, wtedy trzeba rozważyć, czy przerwać dotychczas wykonywane zadanie i zacząć raz jeszcze, czy jednak można odłożyć rozwiązanie problemu na później. Czasami dokończenie aktualnego zadania jest korzystne, z kolei innym razem problem jest na tyle poważny, że usprawiedliwia zatrzymanie prac i ponowne przemyślenie projektu.

Zobaczmy, jaki będzie wynik wykonania testów jednostkowych. Zaliczone! To dobrze, możemy przystąpić do odrobiny refaktoryzacji.

Plik *lists/views.py*:

```
    return render(request, 'home.html', {
        'new_item_text': item.text
    })
```

Spójrz na naszą osobistą listę rzeczy do zrobienia (patrz rysunek 5.3). Dodałem kilka innych problemów, z którymi trzeba będzie się zmierzyć.



Rysunek 5.3. Nasza osobista, uzupełniona lista rzeczy do zrobienia

Zacznijmy od pierwszego elementu na naszej osobistej liście. Wprawdzie można dodać kolejną asercję do istniejącego testu, ale lepszym rozwiązaniem będzie testowanie jednej rzeczy w danym momencie. Dlatego też tworzymy nowy test.

Plik *lists/tests.py*:

```
class HomePageTest(TestCase):
    [...]

    def test_home_page_only_saves_items_when_necessary(self):
        request = HttpRequest()
        home_page(request)
        self.assertEqual(Item.objects.count(), 0)
```

Wykonanie testu kończy się oczekiwanym niepowodzeniem, ponieważ `1 != 0`. Musimy to poprawić. Uważaj, przeprowadzamy niewielką zmianę w logice widoku plus kilka dodatkowych w kodzie implementacji.

Plik *lists/views.py*:

```
def home_page(request):
    if request.method == 'POST':
        new_item_text = request.POST['item_text']  #❶
        Item.objects.create(text=new_item_text)  #❷
    else:
        new_item_text = ''  #❸

    return render(request, 'home.html', {
        'new_item_text': new_item_text,  #❹
    })
```

❶❸❹ Używamy zmiennej o nazwie `new_item_text`, która będzie przechowywała zawartość żądania `POST` lub pusty ciąg tekstowy.

❷ Wywołanie `objects.create()` to skrót pozwalający na utworzenie nowego obiektu `Item` bez konieczności wywołania `save()`.

Po wprowadzonych zmianach wszystkie testy zostają zaliczone:

```
Ran 5 tests in 0.010s
OK
```

Przekierowanie po wykonaniu żądania POST

Fuj, rozwiązanie oparte na użyciu `new_item_text = ''` jest zupełnie niesatysfakcjonujące. Na szczęście kolejny element naszej osobistej listy rzeczy do zrobienia daje możliwość poprawienia rozwiązania. Można się spotkać ze stwierdzeniem, że należy *zawsze stosować przekierowanie po wykonaniu żądania POST*[7], a więc tak właśnie zrobimy. Ponownie modyfikujemy nasz test jednostkowych dotyczący zapisu danych żądania `POST`. Zamiast wygenerować odpowiedź zawierającą nowy element listy, zastosujemy przekierowanie z powrotem na stronę główną.

Plik *lists/tests.py*:

```
    def test_home_page_can_save_a_POST_request(self):
        request = HttpRequest()
        request.method = 'POST'
        request.POST['item_text'] = 'Nowy element listy'

        response = home_page(request)

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'Nowy element listy')

        self.assertEqual(response.status_code, 302)
        self.assertEqual(response['location'], '/')
```

Nie oczekujemy już odpowiedzi zawierającej treść (`content`) wygenerowaną przez szablon, więc możemy się pozbyć związanych z nią asercji. Zamiast tego odpowiedź przedstawia *przekierowanie* HTTP, które powinno mieć kod stanu 302 i wskazywać przeglądarce internetowej inną lokalizację.

W ten sposób otrzymujemy błąd wynikający z polecenia `200 != 302`. Możemy zdecydowanie uporządkować nasz widok.

Plik *lists/views.py* (ch05l028):

```
    from django.shortcuts import redirect, render
    from lists.models import Item

    def home_page(request):
        if request.method == 'POST':
            Item.objects.create(text=request.POST['item_text'])
            return redirect('/')

        return render(request, 'home.html')
```

Wszystkie testy powinny zostać teraz zaliczone:

```
    Ran 5 tests in 0.010s
    OK
```

Poszczególne testy powinny testować pojedyncze rzeczy

Omawiany widok wykonuje teraz przekierowanie po żądaniu `POST`, co jest dobrą praktyką. Udało się skrócić test jednostkowy, ale nadal pozostało miejsce na kolejne usprawnienia. W świecie dobrych testów jednostkowych poszczególne testy powinny sprawdzać jedynie pojedyncze rzeczy. To znacznie ułatwia wyśledzenie błędów. Umieszczenie w teście wielu

[7] *https://en.wikipedia.org/wiki/Post/Redirect/Get*

asercji oznacza, że jeśli jedna z pierwszych spowoduje niepowodzenie, to nie będziesz wiedział, jak przedstawia się stan dalszych asercji. W kolejnym rozdziale zobaczysz, że jeśli kiedykolwiek przez przypadek uszkodzisz funkcjonalność widoku, to będziesz chciał wiedzieć, czy nie działa zapisywanie obiektów, czy mamy nieprawidłowy typ odpowiedzi.

Nie zawsze za pierwszym razem uda się przygotować doskonały test jednostkowy z pojedynczą asercją, ale teraz nadeszła odpowiednia chwila na podział omawianego testu.

Plik *lists/tests.py*:

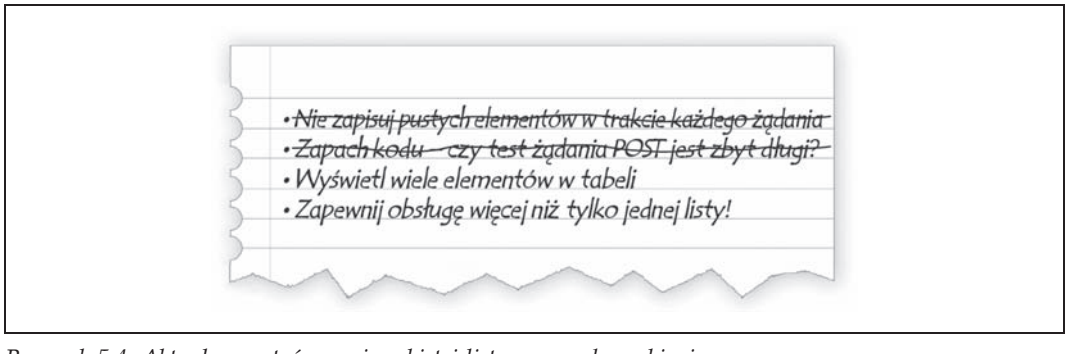
```
    def test_home_page_can_save_a_POST_request(self):
        request = HttpRequest()
        request.method = 'POST'
        request.POST['item_text'] = 'Nowy element listy'

        response = home_page(request)

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'Nowy element listy')

    def test_home_page_redirects_after_POST(self):
        request = HttpRequest()
        request.method = 'POST'
        request.POST['item_text'] = 'Nowy element listy'

        response = home_page(request)

        self.assertEqual(response.status_code, 302)
        self.assertEqual(response['location'], '/')
```

Teraz mamy zaliczonych sześć testów zamiast pięciu:

```
Ran 6 tests in 0.010s

OK
```

Wygenerowanie elementów w szablonie

Teraz już znacznie lepiej! Powracamy do naszej osobistej listy rzeczy do zrobienia (patrz rysunek 5.4).



Rysunek 5.4. Aktualna postać naszej osobistej listy rzeczy do zrobienia

Skreślanie kolejnych pozycji z listy przynosi niemal taką samą satysfakcję jak obserwacja zaliczanych testów!

Trzeci element na liście to jednocześnie ostatni z tych „najłatwiejszych”. Przygotujemy nowy test jednostkowy sprawdzający, czy szablon może wyświetlić wiele elementów listy.

Plik *lists/tests.py*:

```
class HomePageTest(TestCase):
    [...]

    def test_home_page_displays_all_list_items(self):
        Item.objects.create(text='itemey 1')
        Item.objects.create(text='itemey 2')

        request = HttpRequest()
        response = home_page(request)

        self.assertIn('itemey 1', response.content.decode())
        self.assertIn('itemey 2', response.content.decode())
```

Wykonanie testu zgodnie z oczekiwaniem kończy się niepowodzeniem:

```
AssertionError: 'itemey 1' not found in '<html>\n    <head>\n [...]
```

Składnia szablonów Django oferuje znacznik przeznaczony do iteracji list — `{% for .. in .. %}`. Wymienionego znacznika można użyć w poniższy sposób.

Plik *lists/templates/home.html*:

```
<table id="id_list_table">
    {% for item in items %}
        <tr><td>1: {{ item.text }}</td></tr>
    {% endfor %}
</table>
```

To jest jedna z największych zalet systemu szablonów. Teraz szablon spowoduje wygenerowanie wielu wierszy tabeli (`<tr>`), po jednym dla każdego elementu zmiennej `items`. Całkiem świetnie! Wprawdzie na stronach książki przedstawię jeszcze wiele innych możliwości szablonów w Django, ale nadejdzie chwila, gdy będziesz musiał sięgnąć do *dokumentacji Django*[8] i zapoznać się z pozostałymi.

Modyfikacja szablonu nie powoduje zaliczenia testu. Konieczne jest rzeczywiste przekazanie szablonowi elementów z widoku strony głównej.

Plik *lists/views.py*:

def home_page(request):

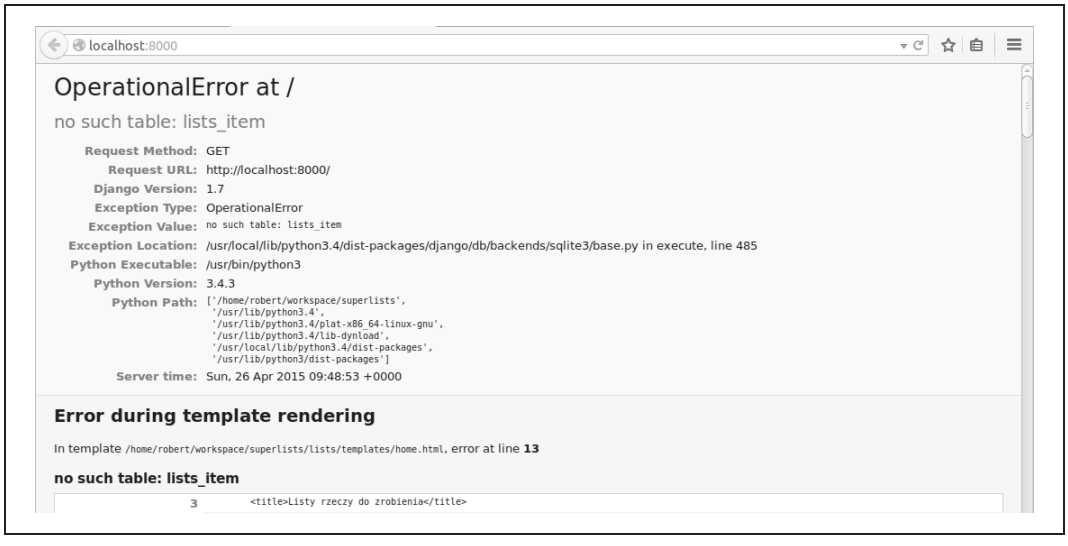
```
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/')

    items = Item.objects.all()
    return render(request, 'home.html', {'items': items})
```

To powinno pozwolić na zaliczenie testu jednostkowego… Nadchodzi moment prawdy, czy zaliczony będzie test funkcjonalny?

```
$ python3 functional_tests.py
[...]
AssertionError: 'To-Do' not found in 'OperationalError at /'
```

[8] *https://docs.djangoproject.com/en/1.7/topics/templates/*

Ups, niestety nie. Wykorzystamy więc inną technikę usuwania błędów z testu funkcjonalnego. To będzie jedna z najprostszych możliwych metod, czyli ręczne przejście do omawianej witryny. W przeglądarce internetowej przejdź pod adres *http://localhost:8000*, a zobaczysz wyświetlony komunikat błędu Django informujący o nieznalezieniu tabeli `list_item`, jak pokazano na rysunku 5.5.



Rysunek 5.5. Kolejny użyteczny komunikat wyświetlony w trakcie procesu usuwania błędów

Utworzenie produkcyjnej bazy danych za pomocą polecenia migrate

Otrzymaliśmy kolejny użyteczny komunikat wygenerowany przez Django, który wskazuje na brak prawidłowo skonfigurowanej bazy danych. Już słyszę Twoje pytanie, jak to możliwe, że wszystko działa doskonale w testach jednostkowych? Odpowiedź jest prosta — Django tworzy specjalną testową bazę danych dla testów jednostkowych; jest to jedno z magicznych działań podejmowanych przez klasę `TestCase`.

W celu przygotowania „rzeczywistej" bazy danych musimy ją utworzyć. Bazy danych SQLite to po prostu plik na dysku. Jak możesz się przekonać, analizując plik *settings.py*, Django domyślnie umieszcza w katalogu bazowym projektu plik o nazwie *db.sqlite3*.

Plik *superlists/settings.py*:

```
[...]
# Baza danych.
# https://docs.djangoproject.com/en/1.7/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Framework Django otrzymał od nas wszystkie informacje niezbędne do utworzenia bazy danych. Najpierw za pomocą pliku *models.py*, a następnie podczas tworzenia pliku migracji. W celu wykorzystania wspomnianych informacji do utworzenia rzeczywistej bazy danych musimy użyć kolejnego wszechstronnego polecenia Django zdefiniowanego w *manage.py*, czyli `migrate`:

```
$ python3 manage.py migrate
Operations to perform:
  Synchronize unmigrated apps: contenttypes, sessions, admin, auth
  Apply all migrations: lists
Synchronizing apps without migrations:
  Creating tables...
    Creating table django_admin_log
    Creating table auth_permission
    Creating table auth_group_permissions
    Creating table auth_group
    Creating table auth_user_groups
    Creating table auth_user_user_permissions
    Creating table auth_user
    Creating table django_content_type
    Creating table django_session
  Installing custom SQL...
  Installing indexes...
Running migrations:
  Applying lists.0001_initial... OK
  Applying lists.0002_item_text... OK

You have installed Django's auth system, and don't have any superusers defined.
Would you like to create one now? (yes/no):
no
```

Na wyświetlone pytanie udzieliłem odpowiedzi `no`, nie potrzebujemy jeszcze superużytkownika, ale przyjrzymy się mu w późniejszych rozdziałach. Teraz po odświeżeniu strony w przeglądarce zauważysz, że błąd zniknął. Spróbuj ponownie wykonać testy funkcjonalne[9]:

```
AssertionError: '2: Użyć pawich piór do zrobienia przynęty' not found in ['1: Kupić
pawie pióra', '1: Użyć pawich piór do zrobienia przynęty']
```

Jesteśmy już bardzo blisko celu! Musimy jeszcze zapewnić prawidłową numerację listy rzeczy do zrobienia. W tym zadaniu pomocny okaże się kolejny, wspaniały znacznik szablonów w Django, czyli `forloop.counter`.

Plik *lists/templates/home.html*:

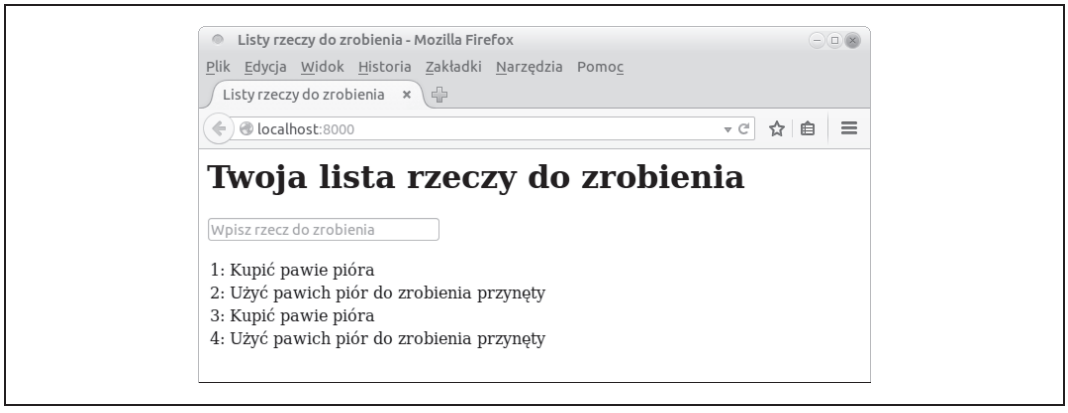
```
{% for item in items %}
    <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
{% endfor %}
```

Jeżeli ponownie spróbujesz wykonać test, to zauważysz, że dotarliśmy do końca testu funkcjonalnego:

```
self.fail('Zakończenie testu!')
AssertionError: Zakończenie testu!
```

Jednak po uruchomieniu aplikacji można zauważyć coś niepokojącego (patrz rysunek 5.6).

[9] Jeżeli na tym etapie otrzymasz kolejny komunikat błędu, spróbuj ponownie uruchomić serwer. Zmiany wprowadzone w bazie danych mogły spowodować drobne zakłócenia w pracy serwera.



Rysunek 5.6. Lista zawiera elementy dodane podczas poprzedniego wykonania testu

Ojej! Wygląda na to, że poprzednia operacja wykonania testów pozostawiła w bazie danych pewne informacje. Po ponownym wykonaniu testów sytuacja staje się jeszcze gorsza:

```
1: Kupić pawie pióra
2: Użyć pawich piór do zrobienia przynęty
3: Kupić pawie pióra
4: Użyć pawich piór do zrobienia przynęty
5: Kupić pawie pióra
6: Użyć pawich piór do zrobienia przynęty
```

Grr. Jesteśmy już tak blisko! Potrzebne jest nam rozwiązanie pozwalające na automatyczne usuwanie danych po zakończeniu testu. Teraz możesz to zrobić ręcznie przez usunięcie bazy danych, a następnie jej ponowne utworzenie za pomocą polecenia `migrate`:

```
$ rm db.sqlite3
$ python3 manage.py migrate --noinput
```

Następnie upewnij się, że testy funkcjonalne nadal są zaliczane.

Pomijając ten niewielki błąd w teście funkcjonalnym, przygotowany kod działa lepiej lub gorzej. Warto więc przekazać pliki do repozytorium.

Rozpocznij od wydania poleceń `git status` i `git diff`, a zobaczysz zmiany wprowadzone w plikach home.html, test.py i views.py. Dodajmy wymienione pliki do repozytorium:

```
$ git add lists
$ git commit -m"Przekierowanie po żądaniu POST, wyświetlenie wszystkich elementów w szablonie."
```

Być może uznasz za użyteczne dodanie znaczników na końcu poszczególnych rozdziałów, na przykład `git tag koniec-rozdziału-5`.

Na jakim etapie jesteśmy?

- Skonfigurowaliśmy formularz pozwalający na dodanie nowych elementów do listy za pomocą żądań `POST`.
- W bazie danych skonfigurowaliśmy prosty model pozwalający na zapisywanie elementów listy.
- Wykorzystaliśmy co najmniej trzy różne techniki usuwania błędów w testach funkcjonalnych.

Ale nasza osobista lista rzeczy do zrobienia zawiera jeszcze kilka pozycji. Przede wszystkim test funkcjonalny powinien usunąć wygenerowane w jego trakcie dane. Jednak najważniejszą kwestią jest dodanie obsługi więcej niż tylko jednej listy rzeczy do zrobienia.

Wprawdzie *można* udostępnić aplikację sieciową w obecnej postaci, ale użytkownicy uznają za dziwne fakt, że cała populacja korzysta z tylko jednej listy rzeczy do zrobienia. Sądzę jednak, że to mogłoby zmusić ludzi do zatrzymania się i zastanowienia, w jaki sposób są ze sobą powiązani, jak razem dzielą to samo miejsce na Matce Ziemi i jak powinni ze sobą współpracować, aby rozwiązać globalne problemy.

Jednak w kategoriach praktycznych tego rodzaju witryna na pewno nie będzie uznana za użyteczną.

No cóż.

Użyteczne koncepcje TDD

Regresja
: Gdy nowy kod powoduje uszkodzenie pewnych aspektów aplikacji, które wcześniej działały prawidłowo.

Nieoczekiwane niepowodzenie
: Kiedy test kończy się niepowodzeniem w sposób inny niż oczekiwany. Taka sytuacja może oznaczać popełnienie błędu w testach, odkrycie regresji za pomocą testów i konieczność wprowadzenia poprawek w kodzie.

Czerwony/zielony/refaktoryzacja
: To jest jeszcze inny sposób opisania procesu TDD. Utwórz test i zobacz, jak kończy się niepowodzeniem (czerwony). Utwórz minimalną ilość kodu potrzebną do zaliczenia testu (zielony). Następnie przeprowadź refaktoryzację w celu poprawienia implementacji.

Triangulacja
: Dodanie testu wraz z nowym konkretnym przykładem dla istniejącego kodu, aby tym samym uzasadnić generalizację implementacji (która do tej chwili mogła być pewnym „oszustwem").

Do trzech razy sztuka, a później refaktoryzacja
: Reguła określająca, kiedy należy przystąpić do usuwania powielonego kodu. Gdy dwa fragmenty kodu przedstawiają się niezwykle podobnie, często rozsądne będzie poczekanie aż do trzeciego wystąpienia danego bloku kodu. W ten sposób będzie można określić, który jego fragment występuje najczęściej, jest gotowy do ponownego użycia i stanowi dobrego kandydata do refaktoryzacji.

Papierowa lista rzeczy do zrobienia
: Miejsce do zapisywania kwestii pojawiających się podczas tworzenia kodu. Tego rodzaju lista pozwala na dokończenie aktualnie wykonywanego zadania, a następnie powrót do wcześniej zapisanych kwestii.

Skorowidz

A

adaptery, 397
adres URL, 48, 108, 112, 120, 124, 200
ainstalacja PhantomJS, 377
Ajax, 276
aktywowanie widoku, 353
analiza
  API formularzy, 206
  infrastruktury testowej, 261
  pliku cookie, 288
  skryptu Fabric, 174
Ansible, 419
API Querystring, 226
architektura
  Functional Core, 398
  heksagonalna, 397
arkusze stylów CSS, 143
asercja assertAlmostEqual(), 134
Async, 432
atak typu CSRF, 75
automatyzacja, 23, 169
automatyzacja wdrożenia, 173

B

baza danych, 83, 150
  bezpieczeństwo, 319
  dostosowanie położenia, 158
  migracja, 84
  nowa kolumna, 85
  testowanie migracji, 423
  zapis z żądania POST, 86
BDD, behavior-driven development, 39
BDUF, Big Design Up Front, 101
bezpieczeństwo, 431
biblioteka
  imitacji, 305
  jQuery, 241
błąd
  404, 52, 121, 430
  500, 430
błędy
  bazy danych, 248
  systemu Persona, 312
  spójności, 229
  weryfikacji modelu, 192

C

ciągła integracja, 363, 380
cookie, 288
CSRF, cross-site request forgery, 75
CSS, 134, 136
cykl TDD, 245

D

dane
  migracji, 425
  problematyczne, 424
  testowe, 425
debugowanie
  po stronie serwera, 307
  żądań Ajax, 259
degradacja, 431
dekorator
  @property, 335
  patch, 305

Django ModelForm, 208
Django ORM, 82
dodanie elementu do listy, 112, 123
dostęp do właściciela, 335
dostosowanie
  modeli, 116
  new_list, 122
  położenia bazy danych, 158
  widoku CreateView, 411
DRY, don't repeat yourself, 81
duplikat, 224
dziedziczenie szablonu, 137, 329

E

edytor
  Git, 23
  vi, 35
ekran QUnit, 240
element
  , 240
  <form>, 241
elementy powielone, 230
eliminacja powielania, 80

F

Fabric, 173
  automatyzacja wdrożenia, 173
  instalacja, 174
  konfiguracja, 178
formularz, 73, 196
  metoda save(), 220
  obsługa unikalności elementów, 231
  obsługa żądania POST, 215
  prosty, 205
  skomplikowany, 223
  sprawdzanie poprawności danych, 205
  użycie w widokach, 210, 217
  w widoku listy, 232
  weryfikacja, 209
  wyświetlenie błędów w szablonie, 216
framework
  Bootstrap, 136, 139
  Django, 32
  sieciowy, 23
frameworki
  CSS, 136
  JavaScript MVC, 431
funkcja
  addCleanup(), 381
  any(), 61, 79
  application(), 165
  assertTrue(), 68
  authenticate(), 283, 286
  create_pre_authenticated_session(), 315
  get_absolute_url(), 202
  home_page(), 49, 64
  initialize(), 267
  is_displayed(), 238
  login(), 283
  redirect(), 201
  render(), 63
  render_to_string(), 63, 65
  send_keys(), 61
  view_list(), 111, 197
  watch(), 279
funkcje widoku, 48, 108
funkcjonalność
  new_item, 197
  view_list, 197

G

generowanie
  elementów w szablonie, 90
  szablonu, 78
Git, 21, 71
git tag, 248
gniazda systemu Unix, 166
gorąca lawa, 399
Gunicorn, 164
  konfiguracja, 421
  uruchamianie, 168

H

hierarchiczna rejestracja danych, 320
hosting, 153
HTML, 20

I

IDE, 24
identyfikacja niejawnych kontraktów, 355
imitacja, 251, 265, 435
  funkcji uwierzytelnienia, 284
  sinon.js, 273
  żądania internetowego, 290

imitacje
  w JavaScript, 282
  w Pythonie, 283, 284, 305
  zaawansowane, 272
Imperative Shell, 398
implementacja nowego projektu, 103
informacje
  o błędzie CSRF, 75
  o postępie, 172
  o testowaniu migracji, 427
instalacja
  Fabric, 174
  Nginx, 155, 419
  node, 377
  pakietów systemowych, 419
  serwera Jenkins, 363
integracja
  wtyczek, 251
  z frameworkiem, 139
interakcja między warstwami, 354
interfejs użytkownika, 253
iteracja, 105
iteracja list, 91
izolacja, 338
izolacja testu, 97, 337

J

JavaScript, 20, 237
Jenkins, 363
  instalacja, 363
  konfiguracja zabezpieczeń, 365
  wtyczki, 365
jQuery, 240

K

kaskadowe arkusze stylów, 134
katalog
  superlists, 32
  tmp, 406
katalogi plików statycznych, 144, 265
klasa
  FunctionalTest, 186
  jumbotron, 142
  ListAndItemModelTest, 190
  ListViewTest, 107, 197
  LiveServerTestCase, 97, 98, 323
  Meta, 209
  ModelForm, 209
  NewItemTests, 197
  StaticLiveServerCase, 141
  text-center, 139
klasy
  Django, 409
  testowe, 107
klucz zewnętrzny, 117
kod
  asynchroniczny, 280
  eksperymentalny, 252, 264, 282
  ORM, 348, 349
  stanu 302, 89
kolejność API Querystring, 226
komentarze, 39
kompilacja, 368
kompilacja w Jenkins, 378
komunikat
  błędu, 50, 92, 106, 227, 331
  ImportError, 160
koncepcje TDD, 95, 129
konfiguracja
  Django, 29
  domen, 156
  ekranu wirtualnego, 370
  Fabric, 178
  Git, 23
  Gunicorn, 421
  Nginx, 162
  projektu, 367, 433
  QUnit, 276
  rejestracji danych, 311, 320
  serwera, 171
  testu, 307, 308, 323
  witryn, 157
konsola
  Firefox, 259
  JavaScript, 270
konto użytkownika, 155
kontrakt, 354
kontroler, 328

L

lista, 120
lista składana, 61
localhost, 33
logika weryfikacji formularza, 209
logowanie, 283, 286, 290

Ł

łączenie testów funkcjonalnych, 98

M

mechanizm ORM, 348
menedżer kontekstu self.assertRaises(), 190
metoda
  form.as_p(), 206
  FunctionalTest.setUp(), 263
  get_user(), 296
  handle(), 315
  prób i błędów, 414
  save(), 220
  setUp(), 41, 42
  tearDown(), 81
  test_displays_all_items(), 120
metody zwinne, 101
migracja bazy danych, 84, 93
model, 82
model użytkownika, 298
moduł
  subprocess, 317, 318
  unittest, 37, 40, 41
moduły Pythona, 23
Mozilla Persona, 252
MVC, 48

N

najlepsze praktyki
  ogólne testowanie, 435
  Selenium, 435
  testy funkcjonalne, 435
narzędzia debugowania przeglądarki, 262
narzędzie
  BDD, 431
  pip, 21
  Selenium, 23
  zarządzania, 315
nazwa domeny, 153
Nginx
  instalacja, 155, 419
  konfiguracja, 162
notacja {{ ... }}, 77

O

obsługa
  maszyn wirtualnych, 422
  plików statycznych, 165
  wyświetlania listy, 413
  żądania POST, 215
  unikalności elementów, 231
ochrona przed duplikatami, 224
odczyt stosu wywołań, 50
odwrócona piramida, 399
okno edytora, 35
opcja
  ALLOWED_HOSTS, 167
  DEBUG, 167
operacja znajdź i zastąp, 213
oprogramowanie, 20
organizacja
  refaktoryzacji, 203
  testów, 183, 203
ORM, object-relational mapper, 82, 348
OS X, 22
oznaczenie wydania, 181

P

pakiet PyTest, 432
PhantomJS, 376
plik
  .gitignore, 35
  accounts.js, 272
  authentication.py, 256
  authentication.py, 294
  base.html, 139, 142
  base.py, 186, 309, 375
  create_session.py, 315
  db.sqlite3, 34, 92
  est_models.py, 333
  fabfile.py, 173–177, 313, 318
  forms.py, 207
  functional_tests.py, 30–34, 38–42, 47, 60, 80, 97
  gunicorn-superlists-staging.ottg.eu.conf, 168
  gunicorn-upstart.template.conf, 170
  home.html, 63
  home_and_list_pages.py, 384, 385
  list.html, 138
  models.py, 86, 118, 256
  my_lists.html, 330
  nginx.config, 162
  nginx.template.conf, 169
  requirements.txt, 168
  runner.js, 377
  settings.py, 64, 92, 145, 146, 258
  superlists-staging.ottg.eu, 162
  templates/base.html, 302

test_authentication.py, 292
test_forms.py, 206, 348
test_layout_and_styling.py, 187
test_models.py, 189, 202, 335
test_my_lists.py, 310, 326, 374
test_sharing.py, 383
test_simple_list_creation.py, 186
test_views.py, 192, 197, 287, 328, 343, 354
tests.py, 47, 49, 54, 104, 116, 120, 122, 142, 151
urls.py, 51, 108, 124, 198, 257
views.py, 50, 52, 106, 109, 121, 193, 257
pliki
.pyc, 35
Bootstrap CSS, 265
konfiguracyjne Gunicorn, 180
konfiguracyjne Nginx, 169, 180
konfiguracyjne Upstart, 169
statyczne, 140, 150, 165
pobranie nazwy domeny, 153
podejście
Outside-In, 325, 329, 336, 435
pragmatyczne, 399
podparcie testów funkcjonalnych, 188
podział testów funkcjonalnych, 185
pola danych wejściowych, 143
polecenie
apt-get, 156
assert, 39
collectstatic, 144
git push, 161
git tag, 181
if, 291
include, 128
makemigrations, 258
manage.py test, 100
migrate, 92, 164
sed, 180
pominięcie testu, 184
poprawa wyglądu witryny, 142
poprawki, 292
porty, 397
potwierdzenie
działania domeny, 157
istnienia błędu, 425
powielanie elementów, 223
prezentacja, 327
proces TDD, 68, 69, 103, 434
produkcyjna baza danych, 92
programowanie
ekstremalne, XP, 14
sieciowe, 131
sterowane testami, TDD, 13, 27, 101, 207
zwinne, 102
projektowanie API, 330
protokół Browser-ID, 254
przechwytywanie parametrów, 121
przeglądarka Firefox, 20
przekazywanie zmiennych, 77
przekierowanie, 89, 114, 201
przestrzenie nazw, 245, 267
przetwarzanie żądania POST, 76, 196
puste elementy, 183
Python 3, 19
PythonAnywhere, 405, 406

R

refaktoryzacja, 62, 65, 80, 109, 128, 189, 200, 341
reguła DRY, 81
reguły, 129
rejestracja, 307
aplikacji, 64, 320
danych, 311, 323
repozytorium, 35, 42
repozytorium Git, 33
REST, representational state transfer, 102
restrukturyzacja hierarchii dziedziczenia szablonu, 329
ręczne wdrożenie kodu, 157
rozszerzenie testu funkcjonalnego, 37, 386
rozwiązania architektoniczne, 396

S

Salt, 419
serwer
ciągłej integracji, 380, 401
Jenkins, 363
Nginx, 155, 157
prowizoryczny, 314, 423
WSGI, 165
serwis GitHub, 17
silnik testów
Django, 238
JavaScript, 238
Spike, 253
sprawdzanie
logowania, 286
poprawności danych, 205
postępu prac, 71
poprawności, 191, 194

sprawdzanie
  sekwencji zdarzeń, 339
  warstwy modelu, 189
  wywołania argumentów, 275
SSH, 155
stan wyścigu, 374
standardowe wyjście błędów, 260
sterowana testami konfiguracja serwera, 171
stos wywołań, 50
strona główna, 67
styl tabeli, 143
synteza, 395
system kontroli wersji, VCS, 33
system kontroli wersji Git, 21, 71
szablon, 327, 330
szablon do wyświetlania list, 109
szyfrowanie, 432

Ś

ścieżka dostępu, 21
środowisko
  produkcyjne, 164
  wirtualne, 160

T

TDD, test-driven development, 27
technologia Ajax, 276
test
  czarnej skrzynki, 38
  E2E, 38
  jako dokumentacja, 301
  kończący się niepowodzeniem, 31
Testing Goat, 401
testowanie
  adresów URL, 107
  buforowania, 431
  interakcji użytkownika, 59
  JavaScript, 245
  klienta Django, 107
  kodu asynchronicznego, 280
  logowania, 290
  migracji, 424, 426
  migracji bazy danych, 423
  modelu, 118, 228
  strony głównej, 45
  stylów, 133
  systemu, 392
  szablonów, 107
  układu graficznego, 133, 147
  w JavaScript, 246
  widoku, 53, 107, 235, 284, 331
  wydajności, 431
  wylogowania, 303
  żądań Ajax, 276
testy
  akceptacji, 38, 392
  dotyczące bezpieczeństwa, 431
  funkcjonalne, 29, 37, 46, 60, 93, 99, 362, 392
    dla strony, 326
    elementów powielonych, 223
    weryfikacji danych, 183
    z wieloma użytkownikami, 381
  integracji, 83, 392
  JavaScript, 238
  jednostkowe, 45–47, 83, 392, 394
    JavaScript, 243, 265
    sprawdzania modelu, 190
  jQuery, 243
  odizolowane, 342, 360, 362, 392, 394, 435
  QUnit, 242, 376
  zintegrowane, 360, 362, 435
token
  CSRF, 273
  serwisów społecznościowych, 381
triangulacja, 80
tworzenie
  bazy danych, 164
  kodu, 392
  kodu aplikacji, 49
  nowej listy, 112, 113
  pliku migracji, 93
  produkcyjnej bazy danych, 92
  repozytorium Git, 33
  sesji, 307, 314
  środowiska wirtualnego, 177
  testowego kodu, 207
  testów dla widoku, 342
  testu funkcjonalnego, 46
  użytkownika, 296
  virtualenv, 159
typy testów, 392

U

ukrycie kodu ORM, 349
upiększanie, 133, 136
uprawnienia, 155
uruchamianie
  Gunicorn, 168
  serwera, 56, 154

usługa
AWS, 156
Persona, 254
usuwanie
błędów, 92, 270
błędu systemu Persona, 312
kodu ORM, 348, 362
powielonego kodu, 358
przeoczonego problemu, 356
starej implementacji widoku, 359
zbędnego kodu, 114, 359
uwierzytelnianie
po stronie serwera, 283
użytkownika, 251
niestandardowe, 255
użycie
adresu URL, 126
form_valid, 411
formularza w widokach, 210, 216
get_absolute_url, 201
gniazd, 166
Gunicorn, 164
imitacji, 338
jQuery, 240
komponentów Bootstrap, 142
Nginx, 165
PhantomJS, 376
Selenium, 59, 380
side_effect, 339
systemu Git, 71
szablonu, 62
technik TDD, 433
testów funkcjonalnych, 37, 157
Upstart, 168
własnych arkuszy stylów, 143
Xvfb, 405
użytkownicy uwierzytelnieni, 301

V

Vagrant, 422
VCS, version control system, 33

W

warstwa
formularzy, 347
modelu, 189, 333, 350
widoku, 332
wartości boolowskie, 295
wdrożenie, 150, 171, 407
nowego kodu, 247
prowizoryczne, 247
rzeczywiste, 247
w środowisku produkcyjnym, 164, 179
za pomocą Fabric, 173
WebSocket, 432
weryfikacja
danych wejściowych, 183
formularza, 209
modelu, 192, 199
widok, 125
CreateView, 411
FormView, 410
listy, 232
logowania, 283
new_list, 215
view_list, 199
widoki
oparte na klasach, 409, 416
złożone, 413
Windows, 21
witryna
node.js, 377
prowizoryczna, 149, 163
właściciel listy, 352
wskazanie formularzy, 115
wybór hostingu, 154
wychwycenie błędów, 310
wycofanie kodu eksperymentalnego, 264
wygląd witryny, 142
wykonanie
pojedynczego pliku testu, 187
testów funkcjonalnych, 56, 100
testów jednostkowych, 56, 100
wylogowanie, 279, 303
wyrażenia regularne, 124
wyświetlanie
błędów, 260
błędów w szablonie, 216
list, 109
wywołania zwrotne, 280
wzorzec
Django, 196
interakcja-oczekiwanie, 383
strony, 381, 384, 389

X

XP, extreme programming, 14
Xvfb, 405

Y

YAGNI, 102
YAML, 419
YUI, 238

Z

zabezpieczenia, 75
zależności, 150
zapis
  danych użytkownika, 73
  modelu, 191
zarządzanie
  pakietami, 21
  testową bazą danych, 314
zdarzenie onload, 245
zmienna
  DOMAIN, 312
  INSTALLED_APPS, 64
znacznik
  {% for .. in .. %}, 91
  {% url %}, 200
  <form>, 74
  szablonu, 76, 200
znak
  apostrofu, 194
  hash, 105
  nowego wiersza, 65
zrzut ekranu, 371, 407
związek klucza zewnętrznego, 117

Ż

żądanie
  GET, 102, 211
  POST, 73, 76, 86, 89, 196, 215

O'REILLY®

Poznaj techniki TDD w połączeniu z Pythonem!

Każdy programista marzy o pracy z przejrzystym kodem, który został w całości pokryty testami. Niestety, rzeczywistość bywa często daleka od ideału. A może da się go jednak osiągnąć? Odpowiedzią na to pytanie jest TDD (ang. *Test-Driven Development*), czyli wytwarzanie oprogramowania sterowane testami. Jak zacząć stosować tę technikę? Na to i wiele innych pytań odpowiada ta książka.

Zacznij w praktyce realizować koncepcje płynące z TDD w połączeniu z językiem Python. Na początku dowiedz się, jak skonfigurować Django za pomocą testu funkcjonalnego, oraz skorzystaj z modułu unittest. Zdobądź też bezcenną wiedzę na temat testowania widoków, szablonów i adresów URL oraz naucz się testować układy strony i style. Sprawdź, jak zapewnić ciągłą integrację z wykorzystaniem systemu Jenkins oraz najlepszych praktyk w tworzeniu testowalnego kodu. Książka ta jest doskonałą lekturą dla wszystkich programistów tworzących aplikacje internetowe w języku Python. Twój kod może być naprawdę łatwy w utrzymaniu!

Dzięki tej książce:

- poznasz techniki wytwarzania oprogramowania sterowanego testami
- odkryjesz najlepsze narzędzia do zapewnienia ciągłej integracji oraz testów
- nauczysz się testować widoki, style oraz logikę Twojej aplikacji
- stworzysz niezawodny, łatwy w utrzymaniu kod

Harry J.W. Percival — pracuje dla firmy Python Anywhere LLP. Swoją przygodę z programowaniem (język BASIC) zaczął we wczesnym dzieciństwie na 8-bitowym komputerze Thomson T-07. Często występuje jako prelegent na konferencjach oraz prowadzi warsztaty. W trakcie swoich wystąpień zaraża programistów pasją wytwarzania czystego kodu oraz stosowania TDD.

Helion

35746 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Informatyka w najlepszym wydaniu

Sprawdź najnowsze promocje:
● http://helion.pl/promocje
Książki najchętniej czytane:
● http://helion.pl/bestsellery
Zamów informacje o nowościach:
● http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-283-1377-4

9 788328 313774

cena: 69,00 zł