

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: 978-83-283-1380-4

© 2015 Helion S.A.

Authorized Polish translation of the English edition of Test-Driven Development with Python, ISBN: 9781449364823 © 2014 Harry Percival.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Polish edition copyright © 2015 by Helion S.A.

All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiekolwiek postaci jest zabronione.

Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książek na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicielami.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Pochwały dla książki *TDD w praktyce*

„W tej książce Harry zabiera nas w podróz mającą na celu odkrycie Pythona i testowania.

To doskonała książka, przyjemna w lekturze i wypełniona przydatnymi informacjami. Gorąco polecam ją każdemu zainteresowanemu tematem testowania w Pythonie, poznaniem framework'a Django lub użycia narzędzi Selenium. Testowanie ma niezwykle istotne znaczenie

w pracy programisty, to niewątpliwie trudny obszar, pełen kompromisów. Harry spisał się doskonale — przyciągnął naszą uwagę, wyjaśniając praktyki stosowane podczas testowania”.

— Michael Foord

Programista Pythona i jednocześnie osoba odpowiedzialna za rozwój modułu unittest

„Ta książka to znacznie więcej niż tylko wprowadzenie do programowania sterowanego testami w Pythonie. To jest pełny kurs przedstawiający najlepsze praktyki od początku do końca na przykładzie nowoczesnego programowania aplikacji sieciowej w Pythonie.

Każdy programista sieciowy powinien zapoznać się z tą książką”.

— Kenneth Reitz

Członek Python Software Foundation

„Żałujemy, że nie mieliśmy do dyspozycji książki Harry'ego, gdy poznawaliśmy Django. Zachowuje ona odpowiednie tempo, a przy tym jest nieco wymagająca. Stanowi doskonałe wprowadzenie do framework'a Django oraz różnych praktyk z zakresu testowania.

Tę książkę warto kupić choćby dla samego materiału poświęconego Selenium, choć oczywiście znajdziesz w niej dużo więcej!”

— Daniel i Audrey Greenfieldowie

autorzy książki *Two Scoops of Django* (Two Scoops Press)



Spis treści

Wprowadzenie	13
Przygotowania i założenia	19
Podziękowania	25
I Podstawy TDD i Django	27
1. Konfiguracja Django za pomocą testu funkcjonalnego	29
Sluchaj Testing Goat! Nie rób nic, dopóki nie przygotujesz testu	29
Rozpoczęcie pracy z frameworkm 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 framework'a 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 <div>	240
Utworzenie testu jednostkowego JavaScript dla żądanej funkcjonalności	243
Testowanie JavaScript w cyklu TDD	245
Zdarzenie onload i przestrzenie nazw	245
Kilką rozwiązania, 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
Przestrzeń 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 roczarowanie	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 rozwiążanie	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 niewyutworzonego jeszcze kodu modelu	337
Pierwsza próba użycia imitacji w celu zapewnienia izolacji	338
Użycie side_effect do sprawdzenia sekwencji zdarzeń	339
Posłuchaj testu — brzydkie 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 powinieneś 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 komplikacja	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 komplikacji 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 lava	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 przechwytyują 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

Wprowadzenie

Niniejszą książkę traktuję jako swego rodzaju próbę podzielenia się z czytelnikami moimi doświadczeniami, jakie zdobyłem w trakcie podróży od „hakera” do „inżyniera oprogramowania”. Znajdziesz tutaj informacje przede wszystkim z zakresu testowania, ale jak się wkrótce przekonasz, także z wielu innych dziedzin.

Na początek dziękuję, że sięgnąłeś po tę książkę.

Jeśli kupiłeś jej egzemplarz, tym bardziej jestem wdzięczny. Natomiast jeśli czytasz bezpłatną wersję opublikowaną w internecie, to *nadal* jestem wdzięczny, że zdecydowałeś się poświęcić swój czas na lekturę. Kto wie, być może gdy dotrzesz do końca książki, uznasz ją zawartą zakupu dla siebie lub przyjaciela.

Jeżeli masz jakiekolwiek komentarze, pytania lub sugestie, podziel się nimi ze mną. Jestem dostępny bezpośrednio poprzez e-mail obeythetestinggoat@gmail.com lub w serwisie Twitter (@hjwp)¹. Znajdziesz mnie również na *forum tematycznym książki*². Zajrzyj także na moją *witrynę i bloga*³.

Mam nadzieję, że lektura książki sprawiła Ci tyle radości, ile mnie jej napisanie.

Dlaczego napisałem książkę o programowaniu sterowanym testami?

Już słyszę, jak pytasz: „Kim jesteś, dlaczego napisałeś tę książkę i dlaczego miałbym ją przeczytać?”.

Nadal jestem na początku mojej kariery programisty. Można spotkać się ze stwierdzeniem, że w każdej dyscyplinie zaczyna się od etapu ucznia, następnie przechodzi do czeladnika i na koniec czasami staje się mistrzem. O sobie mógłbym powiedzieć, że jestem (co najwyżej) czeladnikiem programowania. Na wczesnym etapie kariery miałem dużo szczęścia i zetknąłem się z wieloma fanatykami stosowania technik TDD w programowaniu, co oczywiście wywarło ogromny wpływ na sposób, w jaki programuję. Zapragnąłem więc podzielić się swoimi doświadczeniami z innymi. Mógłbym powiedzieć, że mój entuzjazm jest wynikiem ostatnich zmian i nadal zachowuję świeże wspomnienia z niedawnej nauki, więc łatwo mogę wczuć się w położenie początkujących.

¹ <https://twitter.com/hjwp>

² <https://groups.google.com/forum/#!forum/obey-the-testing-goat-book>

³ <http://www.obeythetestinggoat.com/>

Kiedy dopiero poznawałem Pythona (na podstawie doskonałej książki Marka Pilgrima zatytułowanej *Dive Into Python*), natknąłem się na koncepcję stosowania technik TDD i stwierdziłem: „tak, zdecydowanie widzę w tym sens”. Prawdopodobnie miałeś podobne odczucie, gdy po raz pierwszy zetknąłeś się z TDD. Techniki TDD wyglądają na całkiem rozsądne podeście i naprawdę stanowią dobry nawyk wart stosowania, podobnie jak na przykład regularne szczotkowanie zębów.

Następnie pojawił się mój pierwszy duży projekt i możesz zgadnąć, co się stało — miałem klienta, napięte terminy, dużo pracy do wykonania i wszelkie dobre intencje dotyczące technik TDD szybko odeszły w kąt.

Wszystko było w porządku, a ja czułem się dobrze.

Przynajmniej na początku.

Na początku wiedziałem, że tak naprawdę nie potrzebuję technik TDD, ponieważ pracuję nad małą witryną internetową. Bardzo łatwo mogłem ręcznie sprawdzić działanie poszczególnych składników witryny. Kliknij tutaj łącze, wybierz tam element z rozwijanej listy, a wtedy powinno wydarzyć się to i to. Całkiem łatwe. Tworzenie testów do sprawdzania wymienionych rzeczy trwałoby wieki lub jeszcze dłużej. Poza tym na podstawie pełnych trzech tygodni tworzenia kodu wydawało mi się, że stałem się całkiem dobrym programistą i mogę sobie poradzić z projektem. Łatwizna.

Następnie nadszedł strach przed boginią Złożoności, która wkrótce się pojawiła i pokazała mi braki w moim doświadczeniu.

Projekt rozrastał się i pewne komponenty systemu zaczęły zależeć od innych. Robiłem wszystko, co w mojej mocy, aby kierować się dobrymi zasadami, takimi jak DRY (nie powtarzaj się), ale to zaprowadziło mnie na całkiem niebezpieczne tereny. Wkrótce doszło wielokrotne dziedziczenie, hierarchie klas na ośmiu poziomach głębokości i polecenia eval.

Zacząłem być przerażony na myśl o wprowadzeniu jakiejkolwiek zmiany w kodzie. Nie byłem już pewien, co jest zależne od czego i co może się stać, gdy wprowadzę zmianę tego fragmentu kodu — jeju, myślałem, że ten kod dziedziczy po tamtym. Nie, tutaj został nadpisany. O nie, jego działanie zależy od zmiennej klasy. Dobrze, jeżeli nadpiszę nadpisany wcześniej fragment, to wszystko będzie dobrze. Tylko to sprawdzę. Jednak operacja sprawdzania okazała się dużo bardziej skomplikowana, niż sądziłem. Budowana witryna internetowa zawiera teraz wiele sekcji i ręczne ich sprawdzanie zaczęło być po prostu niepraktyczne. Lepiej pozostawić wszystko, jak jest, zapomnieć o refaktoryzacji i skoncentrować się na tworzeniu.

Bardzo szybko doprowadziłem do powstania okropnego kodu, a tworzenie nowego stało się prawdziwym utrapieniem.

Niedługo potem miałem szczęście i otrzymałem pracę w firmie Resolver Systemem (obecnie *PythonAnywhere*⁴), w której stosowanie programowania ekstremalnego (ang. *extreme programming* — XP) było normą. Pracownicy firmy przyuczyli mnie do rygorystycznego stosowania technik TDD.

⁴ <https://www.pythonanywhere.com/>

Wprawdzie wcześniej zdobyte doświadczenie na pewno otworzyło mój umysł na potencjalne korzyści wynikające z automatyzowanego testowania, ale nadal na każdym etapie pojawiały się pewne wątpliwości. „Ogólnie rzecz biorąc, testowanie może być dobrym pomysłem, ale *naprawdę*?” Wszystkie te testy? Niektóre z nich wydawały się zupełną stratą czasu... Co takiego? Testy funkcjonalne *oraz* testy jednostkowe? Dajcie spokój, to już przesada! I na dodatek ten cykl TDD: testowanie, minimalna zmiana w kodzie i znów testowanie. To już naprawdę głupota! Nie musimy wykonywać tych nieciecznych kroków. Dajcie spokój, przecież widać, jak wygląda prawidłowa odpowiedź. Dlaczego nie możemy pominąć tych testów do końca?

Możesz mi uwierzyć: przewidywałem każdą regułę, sugerowałem każdy skrót, domagałem się uzasadnienia dla każdego aspektu TDD z pozoru wydającego się bezsensownym i poddałem się, widząc sens w tym wszystkim. Niezliczoną ilość razy powtarzałem sobie „dzięki wam, testy”, gdy test funkcjonalny wskazał regresję, której nigdy nie przewidywałem, lub test jednostkowy uchronił mnie przed popełnieniem naprawdę głupiego błędu logicznego. Pod względem psychologicznym tego rodzaju programowanie jest mniej stresującym procesem. Efektem jest kod, z którym praca będzie przyjemnością.

Dlatego też pozwól mi o tym *wszystkim* Ci opowiedzieć.

Cel książki

Moim podstawowym celem jest przekazanie metodologii — sposobu programowania sieciowego, który prowadzi do powstania lepszych aplikacji sieciowych i jednocześnie zapewnia więcej radości ich programistom. Nie ma zbyt dużego sensu w pisaniu książki przedstawiającej materiał, który można znaleźć, korzystając z ulubionej wyszukiwarki internetowej. Dlatego też to nie jest przewodnik omawiający składnię Pythona lub programowanie sieciowe *jako takie*. Zamiast tego mam nadzieję nauczyć Cię, jak wykorzystać techniki TDD do osiągnięcia wymarzonego celu: *przejrzystego kodu, który po prostu działa*.

Mając to na uwadze, nieustannie odwołuję się do rzeczywistego, praktycznego projektu i buduję aplikację sieciową zupełnie od początku, używając do tego narzędzi takich jak Django, Selenium, jQuery i Mock. Nie przyjąłem żadnych założeń dotyczących Twojej wiedzy z zakresu wymienionych narzędzi i dlatego po zakończeniu lektury będziesz miał solidne podstawy dotyczące tych narzędzi, a także opanowane stosowanie technik TDD.

Programowanie ekstremalne zawsze przeprowadzane jest w parach. Dlatego też pisząc tę książkę, wyobrażałem sobie parę z samym sobą, starałem się wytłumaczyć sposób działania narzędzi oraz odpowiadać na pytania dotyczące określonych sposobów działania kodu. Jeżeli gdziekolwiek piszę zbyt protekcyjnie, wynika to z niewystarczającego sprytu oraz braku zachowania cierpliwości względem samego siebie. Natomiast jeśli piszę zbyt defensywnie, wynika to z faktu, że jestem irytującą osobą, która systematycznie nie zgadza się z innymi. Dlatego czasami potrzebuję wielu uzasadnień, zanim przekonam się do czegoś.

Zarys książki

Książkę podzieliłem na trzy części.

Część I (rozdziały od 1. do 6.): Podstawy

Od razu przystępujemy do budowy prostej aplikacji sieciowej, wykorzystując przy tym techniki TDD. Pracę rozpoczynamy od utworzenia testu funkcjonalnego (za pomocą narzędzia Selenium), a następnie przechodzimy przez podstawy Django — modele, widoki i szablony — na każdym etapie rygorystycznie stosując testy jednostkowe. Ponadto dowiesz się, czym jest Testing Goat⁵.

Część II (rozdziały od 7. do 14.): Programowanie sieciowe

W tej części znajdziesz omówienie trudniejszych, choć niemożliwych do uniknięcia aspektów programowania sieciowego. Zobaczysz także, jak testy pomagają radzić sobie ze wspomnianymi aspektami: plikami statycznymi, wdrożeniem w środowisku produkcyjnym, weryfikacją danych formularza, migracjami bazy danych i budzącym lęk JavaScriptem.

Część III (rozdziały od 15. do 20.): Bardziej zaawansowane zagadnienia

Omówiono zagadnienia takie jak imitacje, integracja opracowanego przez firmę trzecią systemu uwierzytelniania, Ajax, konfiguracja testów, podejście Outside-In i ciągła integracja (ang. *continuous integration* — CI).

A teraz kilka słów w kwestiach porządkowych...

Konwencje zastosowane w książce

W tej książce zastosowano następujące konwencje typograficzne.

Kursywa

Wskazuje na nowe pojęcia, adresy URL i e-mail, nazwy plików, rozszerzenia plików i tak dalej.

Czcionka o stałej szerokości

Użyta jest w przykładowych fragmentach kodu, a także w samym tekście w celu odwołania się do pewnych poleceń bądź innych elementów programistycznych, takich jak nazwy zmiennych lub funkcji, bazy danych, typy danych, zmienne środowiskowe, poleceń i słowa kluczowe.

Pogrubiona czcionka o stałej szerokości

Użyta w celu wyekspozowania poleceń bądź innego tekstu, który powinien być wprowadzony przez czytelnika.

Czasami używam znaków:

[...]

do wskazania, że pewna zawartość została pominięta, nastąpiło skrócenie danych wyjściowych lub przejście do ważniejszych informacji w danym fragmencie.

⁵ Testing Goat to nieoficjalna maskotka TDD, więcej na ten temat znajdziesz w rozdziale 1. — przyp. tłum.



Taka ikona oznacza wskazówkę lub sugestię.



Taka ikona oznacza ogólną uwagę.



Ta ikona oznacza ostrzeżenie.

Użycie przykładowych kodów

Przykładowe fragmenty kodów znajdziesz w serwisie GitHub (<https://github.com/hjwp/book-example/>). Kody dla poszczególnych rozdziałów są również dostępne oddzielnie, na przykład pod adresem https://github.com/hjwp/book-example/tree/chapter_03. Na końcu rozdziałów znajdziesz pewne sugestie dotyczące sposobów pracy z repozytorium.

Książka ta ma na celu pomóc Ci w pracy. Ogólnie rzecz biorąc, można wykorzystywać przykłady z niej w swoich programach i w dokumentacji. Nie trzeba kontaktować się z nami w celu uzyskania zezwolenia, dopóki nie powiela się znaczących ilości kodu. Na przykład pisanie programu, w którym znajdzie się kilka fragmentów kodu z tej książki, nie wymaga zezwolenia, jednak sprzedawanie lub rozpowszechnianie płyty CD-ROM zawierającej przykłady z książki wydawnictwa O'Reilly już tak. Odpowiedź na pytanie przez cytowanie tej książki lub przykładowego kodu nie wymaga zezwolenia, ale włączenie wielu przykładowych kodów z tej książki do dokumentacji produktu czytelnika już tak.

Jesteśmy wdzięczni za umieszczanie przypisów, ale nie wymagamy tego. Przypis zwykle zawiera tytuł, autora, wydawcę i ISBN. Na przykład: Harry Percival, *TDD w praktyce. Niezawodny kod w języku Python*, ISBN 978-1-449-36482-3, Helion, Gliwice 2015.

Przygotowania i założenia

W tym miejscu znajdziesz ogólny zarys przyjętych przeze mnie założeń dotyczących Twojej wiedzy, a także oprogramowania, jakie powinieneś mieć zainstalowane w komputerze.

Python 3 i programowanie

Wprawdzie niniejszą książkę napisałem, nie zapominając o początkujących, ale jeżeli dopiero zaczynasz przygodę z programowaniem, to powinieneś przynajmniej poznać podstawy języka Python. Jeśli jeszcze nie opanowałeś podstaw, zachęcam Cię do lektury samouczka dla początkujących programistów Pythona, książki przeznaczonej dla początkujących (na przykład *Dive Into Python*¹ lub *Learn Python the Hard Way*²) bądź też tak dla rozrywki do zapoznania się z materiałami w witrynie *Invent Your Own Computer Games*³. Wszystkie wymienione zasoby stanowią doskonałe wprowadzenie do programowania w Pythonie.

Jeżeli masz doświadczenie w programowaniu w innych językach niż Python, powinieneś sobie poradzić bez problemów, ponieważ Python jest niezwykle łatwy do zrozumienia.

Kod przedstawiony w książce został utworzony w języku Python 3. Kiedy pisałem tę książkę w latach 2013 – 2014, wersja 3 była dostępna już od dłuższego czasu. Osiagnęliśmy więc punkt, w którym to preferowana wersja Pythona. Możesz korzystać z dowolnego systemu operacyjnego: OS X, Windows lub Linux. Nieco dalej znajdziesz informacje szczegółowe przydatne dla użytkowników poszczególnych systemów operacyjnych.



Kod przedstawiony w książce został przetestowany za pomocą Pythona w wersjach 3.3 i 3.4. Jeżeli z jakiegokolwiek powodu używasz wersji 3.2, możesz zauważyc drobne różnice. Dlatego też najlepszym rozwiązaniem będzie uaktualnienie używanej wersji, o ile masz taką możliwość.

Nie zalecam podejmowania prób użycia Pythona 2, ponieważ różnice względem nowszych wersji są już znaczące. Jeżeli Twój kolejny projekt ma powstać właśnie w Pythonie 2, to materiał przedstawiony w książce nadal może być przydatny. Jednak czas poświęcony na ustalenie, czy różnice między otrzymanymi przez Ciebie danymi wyjściowymi i przedstawionymi

¹ <http://www.diveintopython.net/>

² <http://learnpythonthehardway.org/>

³ <http://inventwithpython.com/>

w książce wynikają z faktu użycia Pythona 2, czy jednak są skutkiem błędu, tak naprawdę będzie czasem zmarnowanym.

Jeżeli planujesz wykorzystanie *PythonAnywhere*⁴ (startup PaaS, dla którego pracuję) zamiast lokalnie zainstalowanego Pythona, wówczas przed rozpoczęciem pracy powinieneś zapoznać się z dodatkiem A.

Niezależnie od stosowanego środowiska pracy oczekuję, że masz dostęp do Pythona, wiesz, jak uruchomić go z poziomu powłoki (najczęściej za pomocą polecenia `python3`), a także wiesz, jak edytować i uruchamiać pliki Pythona. Jeżeli masz jakiekolwiek wątpliwości, zawsze możesz sięgnąć do wymienionych wcześniej książek i zasobów.



Jeżeli masz zainstalowanego Pythona w wersji 2 i obawiasz się, że instalacja wersji 3 może uszkodzić istniejącą, Twoje obawy są zupełnie niepotrzebne! Obie wersje Pythona mogą bez problemu funkcjonować w jednym systemie, a swoje pakiety przechowują w zupełnie różnych miejscach. Musisz się jedynie upewnić o przygotowaniu oddzielnych polecień do uruchamiania poszczególnych wersji Pythona. Jednego przeznaczonego dla Pythona 3 (`python3`) i drugiego dla Pythona 2 (to najczęściej po prostu `python`). Podobnie podczas instalacji narzędzia `pip` dla Pythona 3 upewniamy się, że uruchamiające go polecenie (zwykle `pip3`) jest inne od polecenia przeznaczonego do uruchamiania narzędzia `pip` w Pythonie 2.

Jak działa HTML?

Przyjęłem również założenie, że masz ogólną wiedzę dotyczącą sposobu działania sieci — czym jest HTML, żądania POST itd. Jeżeli nie dysponujesz wspomnianymi podstawami, powinieneś poszukać odpowiednich źródeł dotyczących HTML; kilka znajdziesz w witrynie <http://www.webplatform.org/>. Jeżeli potrafisz utworzyć w komputerze stronę HTML, wyświetlić ją w przeglądarce internetowej, a także rozumiesz sposób działania formularzy sieciowych, to masz odpowiednią wiedzę.

JavaScript

W części drugiej książki pojawi się odrobina kodu JavaScript. Jeżeli nie znasz języka JavaScript, nie przejmuj się tym teraz. Gdy na późniejszym etapie poczujesz się nieco zagubiony, wskażę Ci kilka przydatnych zasobów.

Wymagane oprogramowanie

Poza językiem Python będziemy potrzebowali jeszcze kilku innych komponentów.

Przeglądarka internetowa Firefox

Jeżeli skorzystasz z ulubionej wyszukiwarki internetowej, szybko znajdziesz instalator przeglądarki Firefox dla używanego systemu operacyjnego. Wprawdzie narzędzie Selenium współpracuje z wszystkimi najważniejszymi przeglądarkami internetowymi, ale Firefox

⁴ <https://www.pythonanywhere.com/>

pozostaje najłatwiejszą w użyciu, ponieważ jest dostępna na różne platformy sprzętowe. Dodatkowym atutem pozostaje mniejsza podatność twórców przeglądarki Firefox na żądania korporacji.

System kontroli wersji Git

Ten system jest dostępny dla dowolnej platformy sprzętowej i znajdziesz go w witrynie <http://git-scm.com/>.

Narzędzie pip przeznaczone do zarządzania pakietami Pythona

Wymienione narzędzie zostało dołączone do Pythona 3.4 (nie zawsze znajdowało się w pakiecie, więc to zmiana na plus). Aby upewnić się że używamy narzędzia pip w wersji dla Pythona 3, w zaprezentowanych przykładach zawsze stosuję polecenie pip3. W zależności od platformy sprzętowej to może być również pip-3.4 lub pip-3.3. Informacje szczegółowe na ten temat znajdziesz w ramkach poświęconych poszczególnym systemom operacyjnym.

Informacje dla użytkowników Windows

Użytkownicy Windows mogą czasami czuć się zaniedbywani, ponieważ systemy OS X i Linux pomagają zapomnieć o istnieniu świata poza paradygmatem systemów pochodnych dla systemu Unix. Lewe ukośniki jako separatory katalogów? Litery oznaczające napędy? Co takiego? Mimo wszystko materiał przedstawiony w książce może być wykorzystany także przez użytkowników systemu Windows. Oto garść podpowiedzi ułatwiających pracę.

1. Podczas instalacji systemu kontroli wersji Git w Windows upewnij się o wyborze opcji *Run Git and included Unix tools from the Windows command prompt*. W ten sposób uzyskasz dostęp do programu o nazwie Git Bash. Wykorzystaj go jako podstawowy wiersz poleceń, a będziesz mógł korzystać ze wszystkich użytecznych narzędzi powłoki GNU, takich jak ls, touch i grep. Bonusem będzie możliwość użycia zwykłych ukośników jako separatorów katalogów.
2. Podczas instalacji Pythona 3 upewnij się o zaznaczeniu opcji *Add python.exe to Path*, jak pokazano na rysunku 0.1. W ten sposób będziesz mógł uruchamiać Pythona z poziomu powłoki.



Rysunek 0.1. Dodanie Pythona do systemowej ścieżki dostępu

3. W systemie Windows plik wykonywalny Pythona 3 nosi nazwę *python.exe*. To jest dokładnie taka sama nazwa jak w przypadku Pythona 2. W celu uniknięcia ewentualnych niejasności w katalogu plików binarnych Git Bash utwórz następujące dowiązanie symboliczne:

```
ln -s /c/Python34/python.exe /bin/python3.exe
```

Może wystąpić konieczność kliknięcia Git Bash prawym przyciskiem myszy i wybrania opcji *Uruchom jako administrator*. Zwróć uwagę, że utworzone tutaj dowiązanie symboliczne działa jedynie w powłoce Git Bash, a nie w zwykłym wierszu poleceń systemu Windows.

4. Wydanie Python 3.4 jest dostarczane wraz z narzędziem pip. Możesz sprawdzić, czy narzędzie pip zostało zainstalowane, wydając polecenie `which pip3`. Wynikiem wykonania wymienionego polecenia powinno być `/c/Python34/Scripts/pip3`.

Jeżeli z jakiegokolwiek powodu musisz korzystać z wydania Python 3.3 i nie masz narzędzia `pip3`, informacje dotyczące jego instalacji znajdziesz w witrynie <https://pip.pypa.io/en/latest/>. W trakcie pisania książki procedura instalacji wymagała pobrania pliku, a następnie uruchomienia go za pomocą polecenia `python3 get-pip.py`. Upewnij się o użyciu polecenia `python3` podczas uruchamiania skryptu konfiguracyjnego.



Po uwzględnieniu powyższych wskazówek powinieneś mieć możliwość przejścia do powłoki Git Bash i wydania polecień `python3` i `pip3` z poziomu dowolnego katalogu.

Informacje dla użytkowników OS X

Wprawdzie system OS X jest lepiej niż Windows przystosowany do obsługi Pythona, ale do niedawna instalacja narzędzia `pip3` była niełatwym zadaniem. Wraz z wydaniem wersji Python 3.4 wszystko uległo zmianie i proces instalacji jest wręcz trywialny.

- Wydanie Python 3.4 można zainstalować bez żadnych problemów za pomocą *dostępnego instalatora*⁵. Narzędzie pip zostanie zainstalowane automatycznie.
- Instalator systemu kontroli wersji Git również działa bez problemów.

Podobnie jak w przypadku Windows po przeprowadzeniu instalacji oprogramowania będziesz mógł otworzyć okno powłoki i z poziomu dowolnego katalogu wydawać polecenia `git`, `python3` i `pip3`. Jeżeli wystąpią jakiekolwiek problemy, w ulubionej wyszukiwarce internetowej wpisz „ścieżka systemowa” lub „polecenie nieznalezione”, a otrzymasz listę wielu zasobów, w których znajdziesz informacje niezbędne do rozwiązania problemu.



Warto również zwrócić uwagę na menedżera pakietów *Homebrew*⁶, za pomocą którego w systemie OS X można w niezawodny sposób zainstalować wiele narzędzi pochodzących z systemu Unix (w tym również Pythona 3). Wprawdzie oficjalny instalator Pythona działa już bez problemów, ale wymieniony menedżer może okazać się użyteczny w przyszłości. Jego użycie wymaga pobrania Xcode (jego wielkość to obecnie około 2,5 GB), ale w ten sposób zyskujesz kompilator C, co będzie przydatnym „efektem ubocznym”.

⁵ <https://www.python.org/downloads/mac-osx/>

⁶ <http://brew.sh/>

Domyślny edytor Git oraz pozostała podstawowa konfiguracja Git

W dalszej części książki przedstawię polecenia pokazujące krok po kroku użycie Git, choć dobrym rozwiązaniem będzie przeprowadzenie już teraz pewnej konfiguracji. Na przykład w trakcie pierwszego przekazywania plików do repozytorium zostanie wyświetcone okno edytora vi. W takiej sytuacji być może nie będziesz wiedział, co należy dalej zrobić. Cóż, edytor vi ma dwa tryby pracy, więc masz dwa wyjścia do wyboru. Pierwsze polega na poznaniu kilku najpotrzebniejszych poleceń edytora (*naciśnij klawisz i przechodząc w ten sposób do trybu wstawiania, wprowadź odpowiedni tekst; naciśnij Esc, aby powrócić do zwykłego trybu edytora, a następnie zapisz plik i zakończ pracę z vi, wpisując :wq i naciskając Enter*). W ten sposób dołączysz do wspaniałego środowiska osób, które znają ten stary i niezwykle ceniony edytor.

Drugie rozwiązanie polega na zupełnym odrzuceniu jakiegokolwiek powrotu do lat siedemdziesiątych ubiegłego wieku i skonfigurowaniu Git do użycia innego, dowolnie wybranego edytora. Zamknij vi, naciskając Esc i następnie wpisując :q!, a następnie zmień domyślny edytor Git. W dokumentacji Git znajdziesz informacje dotyczące *podstawowej konfiguracji Git*⁷.

Wymagane moduły Pythona

Po zainstalowaniu menedżera pip instalacja nowych modułów Pythona jest niezwykle łatwym zadaniem. Nowe moduły będziemy instalować, gdy okażą się niezbędne. Jednak już kilka powinniśmy mieć zainstalowanych na samym początku.

- **Django 1.7 — sudo pip3 install django==1.7** (w systemie Windows pomiń sudo). Django to wybrany przez nas framework sieciowy. Upewnij się o instalacji wersji 1.7. Dzięki wymienionemu modułowi uzyskasz dostęp do django-admin.py z poziomu powłoki. Jeżeli potrzebujesz pomocy, pewne informacje dotyczące instalacji znajdziesz w *dokumentacji Django*⁸.



W maju 2014 roku framework Django 1.7 nadal pozostawał w wersji beta. Jeżeli wymienione powyżej polecenie nie działa, użyj następującego: **sudo pip3 install https://github.com/django/django/archive/stable/1.7.x.zip**.

- **Selenium — sudo pip3 install --upgrade selenium** (w systemie Windows pomiń sudo). Selenium to narzędzie automatyzacji pozwalające nam na wykonywanie tak zwanych testów funkcjonalnych. Upewnij się o instalacji absolutnie najnowszej dostępnej wersji narzędzia. Selenium bierze udział w nieustającym wyścigu z najważniejszymi przeglądarkami internetowymi, próbując zachować zgodność z najnowszymi funkcjami, które są w nich implementowane. Jeżeli kiedykolwiek zauważysz dziwne zachowanie Selenium, powodem będzie instalacja nowej wersji przeglądarki Firefox. W takim przypadku rozwiązaniem będzie uaktualnienie do najnowszej wersji Selenium...



O ile doskonale nie wiesz, co robisz, *nie używaj narzędzia virtualenv*. Tego rodzaju narzędzie zaczniemy wykorzystywać w rozdziale 8.

⁷ <http://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration>

⁸ <https://docs.djangoproject.com/en/1.7/intro/install/>

Informacje dotyczące IDE

Jeżeli masz doświadczenie w programowaniu w języku Java lub .NET, to być może myślisz o wykorzystaniu środowiska IDE podczas pracy nad projektami w Pythonie. Wspomniane środowiska oferują wiele użytecznych narzędzi, w tym również integrację z VCS. Na rynku dostępnych jest sporo doskonałych rozwiązań IDE przeznaczonych dla Pythona. Kiedy začynałem pracę z Pythonem, sam zacząłem używać środowiska IDE i uznawałem je za niezwykle użyteczne w kilku pierwszych projektach.

Jednak sugeruję (to jedynie propozycja) *rezygnację* z użycia środowiska IDE, przynajmniej podczas lektury niniejszej książki. W świecie Pythona nie istnieje aż tak duża potrzeba użycia IDE. Tę książkę napisałem z założeniem, że będziesz korzystał z prostego edytora tekstu oraz powłoki. Czasami do dyspozycji masz właśnie tylko tyle — na przykład podczas pracy z serwerem. Dlatego też warto nauczyć się użycia podstawowych narzędzi i poznać sposoby ich działania. W ten sposób zdobędziesz przydatną wiedzę, nawet jeśli po zakończeniu lektury książki postanowisz powrócić do środowiska IDE i wszystkich oferowanych przez nie użytecznych narzędzi.



Jeżeli przedstawione tutaj informacje nie sprawdzają się lub chcesz zaproponować lepsze rozwiązania, zawsze możesz do mnie napisać na adres obeythetestinggoat@gmail.com.

Podziękowania

Pragnę podziękować wielu osobom, bez których ta książka nigdy by nie powstała lub byłaby gorsza niż w obecnej postaci.

Przede wszystkim dziękuję „Gregowi” z \$OTHER_PUBLISHER, który był pierwszą osobą zachęcającą do uwierzenia, że tę książkę naprawdę można napisać. Wprawdzie Twoi pracownicy okazali się mieć przesadnie rygorystyczne podejście do praw autorskich, ale jestem Ci niezmiernie wdzięczny za wiarę we mnie.

Podziękowania składam Michaelowi Foordowi, kolejnemu byłemu pracownikowi Resolver Systems, za początkową inspirację do napisania książki oraz nieustanną pomoc podczas realizacji tego projektu. Dziękuję także mojemu szefowi Gilesowi Thomasowi, że naivnie pozwolił kolejnemu pracownikowi na napisanie książki. (Jestem przekonany, że teraz już zmienił tekst standardowej umowy o pracę i zamieścił w niej klauzulę „żadnych książek”). Dziękuję Ci za nieustającą mądrość i skierowanie mnie na ścieżkę testów.

Dziękuję moim pozostałym kolegom, Glennowi Jonesowi i Hanselowi Dunlopowi, za nieocenioną pomoc i cierpliwość na przestrzeni ostatniego roku.

Dziękuję mojej żonie Clementine oraz obu rodzinom — bez ich wsparcia i cierpliwości nigdy nie napisałbym tej książki. Jednocześnie przepraszam Was za cały czas, który spędziłem z nosem w komputerze w chwilach, które powinny być jednak niezapomnianymi chwilami rodzinnymi. Nie miałem pojęcia, jak ta książka wpłynie na moje życie („Chcesz ją napisać w wolnych chwilach? Brzmi rozsądnie...”). Clementine, nie zrobiłbym tego bez Ciebie.

Dziękuję recenzentom technicznym Jonathanowi Hartleyowi, Nicholasowi Tollerveyowi i Emily Bache za słowa zachęty i nieocenione komentarze. Podziękowania kieruję zwłaszcza pod adresem Emily, która uważnie przeczytała wszystkie rozdziały. Oczywiście jestem także niezmiernie wdzięczny Nickowi i Jonowi. Dzięki Waszej obecności nie miałem poczucia, że pracuję zupełnie sam nad tym projektem. Bez pomocy wymienionych recenzentów technicznych ta książka byłaby tylko mniej lub bardziej sensownym wywodem idiota.

Dziękuję również każdemu, kto poświęcił swój czas na dostarczanie informacji zwrotnych o książce, nawet w postaci kilku dobrych życzeń; są to: Gary Bernhardt, Mark Lavin, Matt O’Donnell, Michael Foord, Hynek Schlawack, Russell Keith-Magee, Andrew Godwin i Kenneth Reitz. Dziękuję, że okazałyście się sprytniejsi ode mnie i uchroniłyście mnie przed napisaniem wielu głupstw. Oczywiście w książce pozostało wiele innych głupstw, za które ponoszę wyjątkową odpowiedzialność.

Dziękuję mojej redaktor Meghan Blanchette za okazaną przyjaźń i sympatię, za kierowanie projektem w zakresie zarówno ram czasowych, jak i ograniczania moich kolejnych głupich pomysłów. Dziękuję pozostałym pracownikom wydawnictwa O'Reilly za pomoc, między innymi Sarze Schneider, Karze Ebrahim i Danowi Fauxsmithowi za możliwość pozostać przy języku angielskim, którym posługujemy się w Wielkiej Brytanii. Dziękuję Charlesowi Roumeliotisowi za pomoc nad stylem i gramatyką. Wprawdzie możesz nigdy nie dostrzec meritum chicagowskiej szkoły stosowania znaków cytowania i punktowania, ale jestem pewien, że cieszyłeś się z jej istnienia. Dziękuję także działowi graficznemu za umieszczenie kozy na okładce książki!

Specjalne podziękowania kieruję do wszystkich czytelników wczesnej wersji książki za pomoc w wychwytywaniu literówek, informacje, sugestie oraz wszelkie inne sposoby, na jakie pomogliście w usprawnieniu książki. Przede wszystkim jestem wdzięczny za ciepłe słowa zachęty i nieustające wsparcie. Osoby, którym dziękuję szczególnie, to: 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, Sorcha Bowler, Jon Poler, Charles Quast, Siddhartha Naithani, Steve Young, Roger Camargo, Wesley Hansen, Johansen Christian Vermeer, Ian Laurain, Sean Robertson, Hari Jayaram, Bayard Randel, Konrad Korżel, Matthew Waller, Julian Harley, Barry McClendon, Simon Jakobi, Angelo Cordon, Jyrki Kajala, Manish Jain, 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 Wärlander, Simon Scarfe, Eric Grannan, Marc-Anthony Taylor, Maria McKinley, John McKenna i wiele innych. Jeżeli pominiałem Cię tutaj, masz absolutne prawo do złości. Jestem niezwykle wdzięczny także Tobie, napisz do mnie, a postaram się jakoś naprawić moje przeoczenie.

Na końcu dziękuję czytelnikom za decyzję o sprawdzeniu tej książki. Mam nadzieję, że jej lektura okaże się przyjemnością!

Podstawy TDD i Django

W tej części poznasz podstawy *programowania sterowanego testami* (ang. *test-driven development*, TDD). Całkowicie od podstaw opracujemy prawdziwą aplikację sieciową i na każdym etapie prac będziemy przygotowywać dla niej testy.

Dokładnie omówiony będzie temat testowania funkcjonalnego za pomocą Selenium i testy jednostkowe, a także poznasz występujące między nimi różnice. Przedstawię tutaj sposób pracy w trakcie programowania sterowanego testami, który określam mianem cyklu „test jednostkowy i tworzenie kodu”. Przeprowadzimy również refactoring i zobaczymy, jak go stosować z technikami TDD. Ponieważ do tworzenia oprogramowania podchodzę niezwykle poważnie, podczas pracy będziemy używać systemu kontroli wersji (Git). Dowiesz się więc, jak i kiedy przekazywać kod do repozytorium oraz integrować go z technikami TDD i sposobem pracy w trakcie programowania sieciowego.

W książce wykorzystamy Django, czyli (prawdopodobnie) najpopularniejszy na świecie framework sieciowy dla Pythona. Postaram się powoli i pojedynczo wprowadzać nowe koncepcje Django oraz podać wiele łączy prowadzących do innych źródeł. Jeżeli nie miałeś wcześniej styczności z Django, to gorąco zachęcam Cię do zapoznania się z materiałem przedstawionym we wspomnianych źródłach. Jeżeli poczujesz się nieco zagubiony w trakcie lektury książki, zrób sobie przerwę, poświęć kilka godzin na przejrzenie oficjalnych samouczków Django, a następnie powróć do lektury książki.

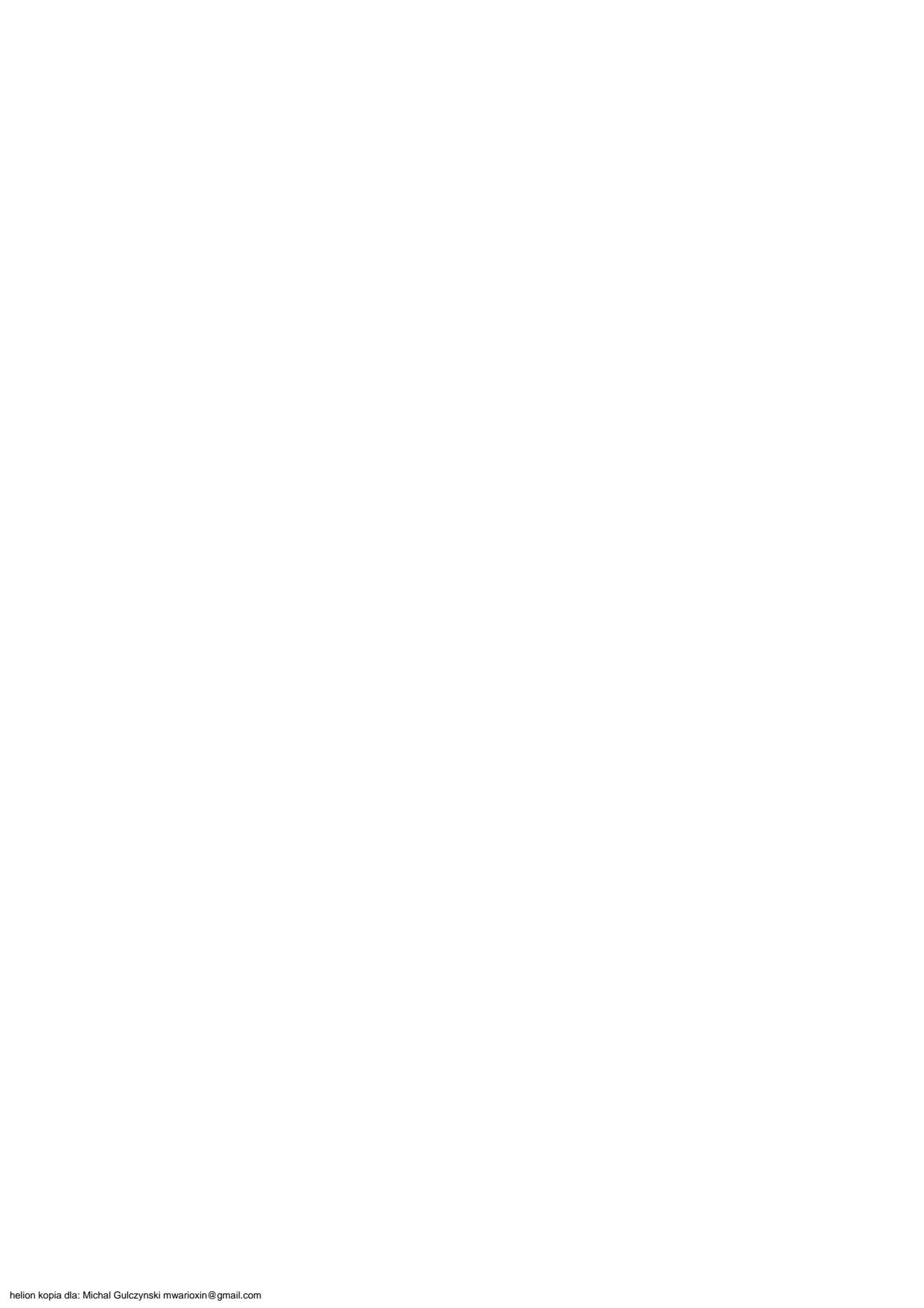
Oczywiście spotkasz również Testing Goat...



Zachowaj ostrożność podczas kopiowania i wklejania

Jeżeli czytasz książkę w wersji elektronicznej, naturalne jest, że będziesz chciał kopiować listingi i wklejać je w używanym środowisku pracy. Jednak znacznie lepszym rozwiązaniem będzie, jeśli zrezygnujesz z kopiowania kodu. Samodzielne wprowadzanie kodu zmusza mózg do efektywniejszego działania, a ponadto takie rozwiązanie jest bliższe rzeczywistemu sposobowi tworzenia kodu. Po drodze niewątpliwie popełnisz pewne błędy, a ich samodzielne wyszukanie okaże się cenną lekcją.

Poza wszystkim warto również pamiętać o niedoskonałościach formatu PDF. Oznacza to, że podczas kopiowania i wklejania kodu mogą działać się różne dziwne rzeczy...



Konfiguracja Django za pomocą testu funkcjonalnego

Programowanie sterowane testami nie jest czymś, co przychodzi naturalnie. To rodzaj dyscypliny, jak sztuki walki. Podobnie jak w filmie kung-fu będziesz potrzebował wymagającego mistrza, który zmusi Cię do zachowania dyscypliny. W naszym przypadku to będzie Testing Goat.

Słuchaj Testing Goat! Nie rób nic, dopóki nie przygotujesz testu

Wspomniany Testing Goat¹ to nieoficjalna maskotka technik TDD w społeczności Pythona i prawdopodobnie ma różne znaczenie dla poszczególnych osób. Dla mnie Testing Goat to głos wewnętrzny, który trzyma mnie w pionie i nakazuje pozostać na „właściwej ścieżce testowania”. Przypomina małe aniołki lub demony, które w kreskówkach pojawiają się na barkach bohaterów, ale mają niewiele zmartwień. Mam nadzieję, że dzięki niniejszej książce Testing Goat pojawi się także i w Twojej głowie.

Postanowiliśmy zbudować witrynę internetową, nawet jeśli jeszcze nie całkiem dokładnie wiemy, co należy zrobić. W świecie programowania sieciowego pierwszym krokiem jest wybór frameworka, jego instalacja i konfiguracja. *Pobierz to, zainstaluj tak, skonfiguruj to, uruchom skrypt...* Jednak programowanie sterowane testami wymaga zupełnie innego podejścia. Kiedy stosujesz techniki TDD, zawsze powinieneś mieć w sobie Testing Goat wołający „najpierw test, najpierw test!” i uparty niczym koza.

W programowaniu sterowanym testami krok pierwszy zawsze jest taki sam: *utwórz test*.

Dlatego też *najpierw* tworzymy test, *następnie* go wykonujemy i sprawdzamy, czy zgodnie z oczekiwaniemi zakończył się niepowodzeniem. *Tylko wtedy* możemy przejść dalej, czyli do tworzenia aplikacji. Powinieneś o tym pamiętać i nieustannie sobie powtarzać.

Kolejną cechą kozy jest wykonywanie pojedynczych kroków. Dlatego też kozy rzadko spa- dają z gór, niezależnie od tego, jak strome mogą one być (patrz rysunek 1.1).

¹ Wymieniony zwrot dosłownie oznacza testującą kożę — przyp. tłum.



Rysunek 1.1. Kozy są o wiele zwinniejsze, niż sądzisz (źródło: Caitlin Stewart, Flickr)²

Będziemy poruszać się eleganckimi, małymi krokami. Do zbudowania aplikacji wykorzystamy Django, czyli popularny framework sieciowy dla Pythona.

Pierwszym zadaniem jest sprawdzenie, czy w środowisku pracy mamy zainstalowany framework Django oraz czy jest on gotowy do pracy. Wspomniane sprawdzenie będzie polegało na próbie uruchomienia serwera Django, aby przekonać się, czy udostępnia on stronę internetową w przeglądarce uruchomionej w komputerze lokalnym. Do wymienionego zadania wykorzystamy narzędzie automatyzacji o nazwie Selenium.

W katalogu przeznaczonym na budowany projekt utwórz nowy plik Pythona o nazwie *functional_tests.py*, a następnie umieść w nim przedstawiony poniżej kod. Jeżeli masz wrażenie, że słyszysz głosy kilku małych kóz, to może być pomocne.

```
from selenium import webdriver

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

assert 'Django' in browser.title
```

² <https://www.flickr.com/photos/caitlinstewart/2846642630/>

Pożegnanie z liczbami rzymskimi!

W wielu wprowadzeniach do technik TDD stosowane są przykłady liczb rzymskich, co wydaje się być żartem. Nawet ja się temu poddałem. Jeżeli jesteś ciekaw, możesz się o tym przekonać, odwiedzając moją stronę w serwisie *Github*³.

Użycie przykładu opartego na liczbach rzymskich ma zarówno dobre, jak i złe strony. To jest przykład teoretycznego problemu o ograniczonym zakresie, a na jego podstawie można dość dobrze wyjaśnić techniki TDD.

Problem polega na tym, że trudno to później powiązać z rzeczywistymi rozwiązaniami. Dlatego też zdecydowałem się na opracowanie zupełnie od zera przykładowej aplikacji sieciowej. Wprawdzie to będzie bardzo prosta aplikacja, ale mam nadzieję, że ułatwi Ci ona później przejście do kolejnego, rzeczywistego projektu.

Przedstawiony powyżej kod to nasz pierwszy *test funkcjonalny* (ang. *functional test*). W dalszej części książki dowiesz się więcej na temat testów funkcjonalnych oraz cech odróżniających je od testów jednostkowych. Teraz wystarczy upewnić się o zrozumieniu sposobu działania przedstawionego testu:

- Uruchomienie modułu *webdriver* narzędzia Selenium w celu wyświetlenia okna przeglądarki internetowej Firefox.
- Użycie wymienionego modułu do wyświetlenia strony internetowej, która powinna być pobrana z komputera lokalnego.
- Sprawdzenie (za pomocą asercji), czy tytuł wczytanej strony internetowej zawiera słowo *Django*.

Spróbujmy uruchomić przygotowany plik:

```
$ python3 functional_tests.py
Traceback (most recent call last):
  File "functional_tests.py", line 6, in <module>
    assert 'Django' in browser.title
AssertionError
```

Na ekranie powinieneś zobaczyć okno przeglądarki internetowej, w którym następuje próba przejścia do adresu *localhost:8000*. Następnie wyświetlany jest komunikat błędu Pythona. Prawdopodobnie będziesz poirytowany pozostawionym na ekranie oknem przeglądarki Firefox. Tym zajmiemy się nieco później.



Jeżeli zamiast wspomnianego powyżej komunikatu otrzymasz informację o błędzie dotyczącym importu narzędzia Selenium, powinieneś powrócić do „Wprowadzenia” i raz jeszcze zapoznać się z podrozdziałem omawiającym wymagania dotyczące oprogramowania, jakie powinieneś mieć zainstalowane w komputerze.

Na tym etapie mamy *test kończący się niepowodzeniem*, co oznacza, że możemy przystąpić do budowy naszej aplikacji.

³ <https://github.com/hjwp/>

Rozpoczęcie pracy z frameworkiem Django

Ponieważ jesteś już po lekturze przedstawionego we „Wprowadzeniu” podrozdziału poświęconego wymaganiom i przyjętym założeniom, to powinieneś mieć zainstalowany framework Django. Pierwszym krokiem podczas rozpoczęcia pracy z Django jest utworzenie *projektu*, czyli kontenera dla naszej witryny internetowej. Do tego celu Django udostępnia niewielkie narzędzie działające w powłoce:

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

Wynikiem wydania powyższego polecenia będzie utworzenie katalogu o nazwie *superlists*, zawierającego pewną liczbę podkatalogów i plików:

```
functional_tests.py
superlists
    manage.py
    superlists
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

Jak możesz zobaczyć, katalog projektu *superlists* zawiera również podkatalog o tej samej nazwie. Wprawdzie to może być nieco mylące, ale to drobiazg. Jeżeli chcesz wiedzieć, skąd wziął się podkatalog o nazwie takiej samej jak projekt, będziesz się musiał cofnąć do historii Django. W tym momencie najważniejsze jest, aby zapamiętać, że katalog *superlists/superlists* jest przeznaczony dla komponentów stosowanych w całym projekcie. Przykładem może być plik *settings.py*, który jest używany do przechowywania konfiguracji globalnej witryny internetowej.

Zwróć również uwagę na plik o nazwie *manage.py*. To wszechstronne narzędzie Django, wykorzystasz je do uruchomienia serwera. Wypróbujmy je teraz. Wydaj polecenie **cd superlists**, aby przejść do katalogu projektu *superlists* (na poziomie tego katalogu będziemy wykonywali wiele zadań). Następnie wydaj poniższe polecenie:

```
$ python3 manage.py runserver
Validating models...
```

```
0 errors found
Django version 1.7, using settings 'superlists.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

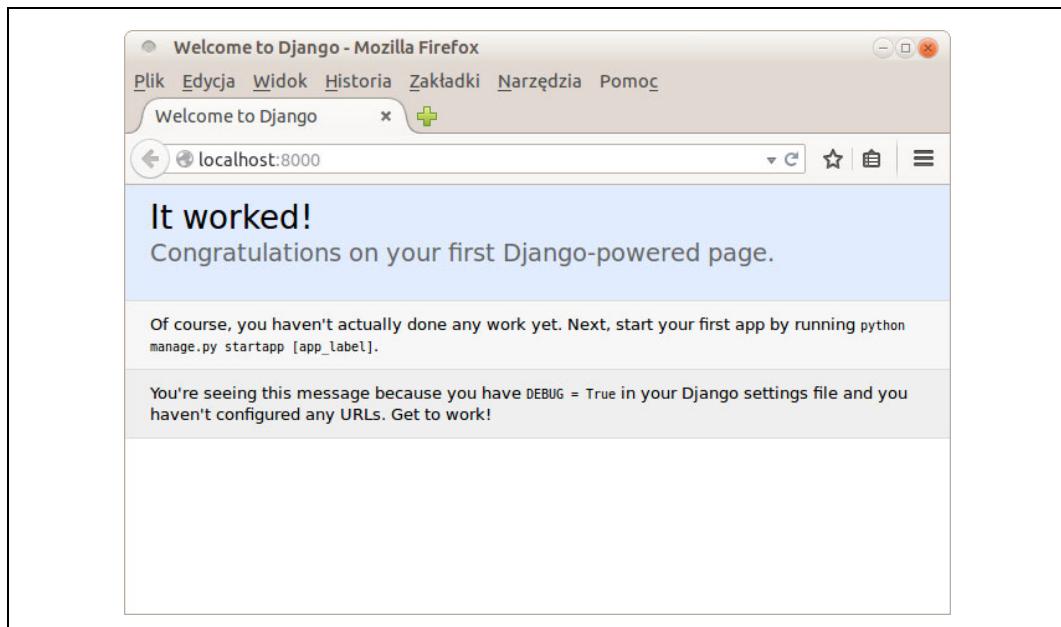
Pozostaw uruchomiony serwer i otwórz kolejne okno powłoki. W nowo otwartym oknie ponownie spróbuj wykonać nasz test (z poziomu katalogu, w którym znajduje się plik *functional_tests.py*):

```
$ python3 functional_tests.py
$
```

W powłoce nie zobaczysz nic ciekawego, ale powinieneś zwrócić uwagę na dwie kwestie. Pierwsza to brak niemiłego komunikatu *AssertionError*. Natomiast druga to wyświetcone przez narzędzie Selenium okno przeglądarki internetowej Firefox wyświetlające dziwnie wyglądającą stronę.

Strona nie prezentuje się zbyt zachęcająco, ale jest to nasz pierwszy zaliczony test. Hura!

Jeżeli masz wrażenie, że to jest magia, spójrz na uruchomiony serwer. W tym celu samodzielnie otwórz nowe okno przeglądarki internetowej i przejdź pod adres <http://localhost:8000/>. Powinieneś otrzymać wynik podobny do pokazanego na rysunku 1.2.



Rysunek 1.2. Serwer framework Django działa

Możesz teraz zakończyć działanie serwera. Wystarczy powrócić do początkowego okna powłoki i nacisnąć klawisze *Ctrl+C*.

Utworzenie repozytorium Git

Zanim zakończymy ten rozdział, do wykonania pozostało nam jeszcze jedno zadanie — umieszczenie w systemie kontroli wersji (ang. *version control system*, VCS) utworzonego do tą kodu. Jeżeli jesteś doświadczonym programistą, nie muszę Cię przekonywać o zaletach systemu kontroli wersji. Natomiast jeśli dopiero rozpoczynasz przygodę z programowaniem, to musisz mi uwierzyć na słowo, że system kontroli wersji to po prostu konieczność. Gdy nad projektem pracujesz przez kilka tygodni i zawiera on setki wierszy kodu, wówczas narzędzie pozwalające na przeglądanie jego wcześniejszych wersji, wycofywanie wprowadzonych zmian czy bezpieczne wypróbowanie nowych koncepcji jest wręcz nieocenione, a kod umieszczony w VCS stanowi rodzaj kopii zapasowej. Techniki TDD doskonale sprawdzają się w połączeniu z systemem kontroli wersji i dlatego chcę pokazać, jak system VCS można wykorzystać we własnym środowisku pracy.

Najwyższa pora na umieszczenie pierwszego kodu w repozytorium. Jeżeli to nieco za późno, możesz mnie za to winić. W książce będziemy używać systemu kontroli Git, ponieważ uznaje go za najlepszy z dostępnych.

Rozpoczynamy od przeniesienia pliku *functional_tests.py* do podkatalogu *superlists*, a następnie wydamy polecenie `git init` w celu zainicjowania repozytorium.

```
$ ls  
superlists      functional_tests.py  
$ mv functional_tests.py superlists/  
$ cd superlists  
$ git init .  
Initialized empty Git repository in /workspace/superlists/.git/
```



Od tej chwili katalog główny *superlists* staje się naszym katalogiem roboczym. Gdy zobaczyś polecenie do wykonania, możesz przyjąć założenie, że powinno być wydane w wymienionym katalogu. Podobnie jeśli w tekście podajesz ścieżkę dostępu do pliku, potraktuj ją jako względową dla katalogu głównego *superlists*. Dlatego też *superlists/settings.py* oznacza plik *settings.py* znajdujący się w podkatalogu *superlists* katalogu głównego o tej samej nazwie. Czy wszystko jasne? Jeżeli będziesz miał jakiekolwiek wątpliwości, szukaj pliku *manage.py*, ponieważ powinieneś być w tym samym katalogu, w którym jest wymieniony plik.

Teraz wybierzemy pliki, które mają zostać umieszczone w repozytorium — w tym przypadku to prawie wszystkie pliki.

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

Plik o nazwie *db.sqlite3* jest plikiem bazy danych. Ponieważ nie chcemy go umieszczać w systemie kontroli wersji, do projektu dodajemy plik *.gitignore* wskazujący pliki, które mają być ignorowane przez Git:

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

W tym momencie pozostała zawartość katalogu bieżącego (wskaazywanego przez kropkę) możemy dodać do repozytorium:

```
$ 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-34.pyc  
    new file:   superlists/_pycache_/_settings.cpython-34.pyc  
    new file:   superlists/_pycache_/_urls.cpython-34.pyc  
    new file:   superlists/_pycache_/_wsgi.cpython-34.pyc  
    new file:   superlists/settings.py  
    new file:   superlists/urls.py  
    new file:   superlists/wsgi.py
```

Psiakrew! W projekcie znajduje się mnóstwo plików *.pyc*, których umieszczanie w repozytorium jest bezcelowe. Musimy je więc usunąć z repozytorium oraz dodać do listy w pliku *.gitignore*:

```
$ git rm -r --cached superlists/__pycache__  
rm 'superlists/__pycache__/_init_.cpython-34.pyc'  
rm 'superlists/__pycache__/settings.cpython-34.pyc'  
rm 'superlists/__pycache__/urls.cpython-34.pyc'  
rm 'superlists/__pycache__/wsgi.cpython-34.pyc'  
$ echo "__pycache__" >> .gitignore  
$ echo "*.pyc" >> .gitignore
```

Przekonajmy się, co dotąd osiągnęliśmy... (Jak możesz zobaczyć, dość często korzystam z polecenia `git status`. Ponieważ wydaje je naprawdę często, zdefiniowałem dla niego alias `git st`. Oczywiście nie zmuszam Cię do tego samego i dlatego zachęcam do samodzielnego definiowania własnych aliasów systemu kontroli Git).

```
$ 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
```

Wszystko wygląda dobrze, więc jesteśmy gotowi do pierwszego umieszczenia plików w repozytorium:

```
$ git commit
```

Po wydaniu powyższego polecenia zostanie wyświetlone okno edytora⁴ (podobne do pokazanego na rysunku 1.3) pozwalające na wprowadzenie opisu dotyczącego plików umieszczanych w repozytorium.

⁴ Czy na ekranie zobaczyłeś okno edytora vi i zupełnie nie wiesz, co należy teraz zrobić? A może wyświetlony został komunikat dotyczący konta użytkownika i polecenie `git config --global user.username`? W takiej sytuacji powróć do przedstawionego we „Wprowadzeniu” podrozdziału poświęconego wymaganiom dotyczącym niezbędnego oprogramowania zainstalowanego w komputerze lokalnym. Znajdziesz tam kilka podpowiedzi pomocnych w wymienionych powyżej sytuacjach.

```
robert@tolumnie:~/workspace/superlists
Plik Edycja Widok Wyszukiwanie Terminal Pomoc
1 Pierwsze umieszczenie plików w repozytorium: test funkcjonalny i podstawowa konfiguracja Django.
2 # Please enter the commit message for your changes. Lines starting
3 # with '#' will be ignored, and an empty message aborts the commit.
4 # On branch master
5 #
6 # Initial commit
7 #
8 # Changes to be committed:
9 #   new file:  .gitignore
10 #   new file: functional_tests.py
11 #   new file: manage.py
12 #   new file: superlists/_intt_.py
13 #   new file: superlists/settings.py
14 #   new file: superlists/urls.py
15 #   new file: superlists/wsgi.py
16 #
17 # Changes not staged for commit:
18 #   modified:  .gitignore
19 #

.git/COMMIT_EDITMSG" 19L, 585C zapisano
```

Rysunek 1.3. Pierwsze umieszczenie plików w repozytorium



Jeżeli naprawdę chcesz poznać Git, warto w tym miejscu poświęcić nieco czasu na opanowanie umiejętności przekazywania plików do repozytoriów umieszczonych w chmurze, oferowanych przez serwisy takie jak GitHub i BitBucket. Wykorzystanie repozytorium umieszczonego w chmurze będzie użyteczne, jeśli podczas lektury książki będziesz pracował na więcej niż tylko jednym komputerze. Opanowanie umiejętności pracy ze zdalnymi repozytoriami pozostawiam już Tobie, wymienione serwisy oferują w tym zakresie doskonałą dokumentację. Ewentualnie możesz się z tym wstrzymać aż do rozdziału 8., w którym zdalne repozytorium wykorzystamy do wdrożenia naszego projektu.

To już wszystko w kwestii systemu kontroli wersji. Gratulacje! Utworzyleś test funkcjonalny za pomocą narzędzia Selenium, a także rozpoczęłeś pracę z frameworkm Django w sposób certyfikowany i zgodny z filozofią TDD. Zrób sobie zasłużony odpoczynek, zanim przejdziesz do lektury rozdziału 2.

Rozszerzenie testu funkcjonalnego za pomocą modułu unittest

Przystąpimy teraz do modyfikacji naszego testu, który na obecnym etapie po prostu sprawdza istnienie domyślnej strony Django. Zamiast tego chcemy wyszukać tekst, który będzie znajdował się na stronie głównej naszej witryny.

Najwyższy czas na ujawnienie rodzaju budowanej tutaj aplikacji sieciowej — to lista rzeczy do zrobienia. W ten sposób wpisujemy się w obecną modę. Kilka lat temu ogromna część samouczków z zakresu programowania sieciowego była poświęcona budowie bloga. Następnie mieliśmy fora i ankiety. Teraz nadeszła pora na listy rzeczy do zrobienia.

Warto w tym miejscu dodać, że lista rzeczy do zrobienia to naprawdę użyteczny przykład. W zasadzie to bardzo prosta konstrukcja — lista ciągów tekstowych — co niezwykle ułatwia przygotowanie minimalnej, działającej wersji aplikacji. Jednak tego rodzaju aplikację można rozbudować na wiele różnych sposobów, na przykład zastosować odmienne modele trwałych magazynów danych, dodać terminy wykonania poszczególnych zadań, przypomnienia, zapewnić możliwość współdzielenia danych z innymi użytkownikami, a także usprawnić interfejs użytkownika wyświetlany po stronie klienta. Nie ma żadnego powodu, aby ograniczać się jedynie do „listy rzeczy do zrobienia”, to może być dowolnego rodzaju lista. Moim celem jest pokazanie wszystkich najważniejszych aspektów programowania sieciowego oraz zastosowanie w nich technik TDD.

Użycie testu funkcjonalnego do przygotowania minimalnej aplikacji

Testy oparte na narzędziu Selenium pozwalają na użycie prawdziwej przeglądarki internetowej, a tym samym na sprawdzenie *funkcjonowania* aplikacji z perspektywy użytkownika. Dlatego też są nazywane *testami funkcjonalnymi*.

Oznacza to, że test funkcjonalny może być dowolnego rodzaju specyfikacją aplikacji. Jest przeznaczony do monitorowania tego, co można nazwać *informacjami od użytkownika*, i opiera się na sposobie, w jaki użytkownik może korzystać z danej funkcji, a także na tym, jak aplikacja powinna reagować na działania użytkownika.

Terminologia: test funkcjonalny == test akceptacji == test E2E

To, co ja nazywam testem funkcjonalnym, inni wolą określać mianem *testu akceptacji* lub testu *E2E*. Najważniejsze pozostaje to, że tego rodzaju testy sprawdzają z zewnątrz sposób działania całej aplikacji. Innym spotykany tutaj pojęciem jest *test czarnej skrzynki*, ponieważ test nie ma żadnych informacji dotyczących wewnętrznych komponentów sprawdzanego systemu.

Testy funkcjonalne powinny zapewniać czytelne dla człowieka informacje, którymi można się kierować. Dodamy więc jasne komentarze do przygotowanego wcześniej kodu testu. Podczas tworzenia nowego testu funkcjonalnego możesz najpierw przygotować odpowiedni komentarz przeznaczony do przechwycenia kluczowych punktów informacji od użytkownika. Ponieważ testy funkcjonalne mają opisy czytelne dla człowieka, to można je przekazywać także osobom niebędącym programistami i w taki sposób dyskutować nad wymaganiami oraz funkcjami aplikacji.

Techniki TDD oraz metodologie programowania zwinnego bardzo często są stosowane razem. Jedną z częściej pojawiających się kwestii będzie to, jaka jest najprostsza możliwa wersja zapewniająca użyteczność aplikacji. Rozpoczynamy więc od jej utworzenia, aby jak najszybciej móc przystąpić do testowania.

Minimalna działająca wersja aplikacji listy rzeczy do zrobienia musi tak naprawdę pozwalać użytkownikowi na wprowadzanie rzeczy do zrobienia oraz przechowywać tę listę aż do kolejnego uruchomienia aplikacji przez użytkownika.

Otwórz plik *functional_tests.py* i zmodyfikuj go do przedstawionej poniżej postaci.

Plik *functional_tests.py*:

```
from selenium import webdriver

browser = webdriver.Firefox()

# Edyta dowiedziała się o nowej, wspaniałej aplikacji w postaci listy rzeczy do zrobienia.
# Postanowiła więc przejść na stronę główną tej aplikacji.
browser.get('http://localhost:8000')

# Zwróciła uwagę, że tytuł strony i nagłówek zawierają słowo Listy.
assert 'Listy' in browser.title

# Od razu zostaje zachęcona, aby wpisać rzecz do zrobienia.

# W polu tekstowym wpisała "Kupić pawie pióra" (hobby Edyty
# polega na tworzeniu ozdobnych przynęt).

# Po naciśnięciu klawisza Enter strona została uaktualniona i wyświetla
# "1: Kupić pawie pióra" jako element listy rzeczy do zrobienia.

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

# Strona została ponownie uaktualniona i teraz wyświetla dwa elementy na liście rzeczy do zrobienia.

# Edyta była ciekawa, czy witryna zapamięta jej listę. Zwróciła uwagę na wygenerowany dla niej
# unikalny adres URL, obok którego znajduje się pewien tekst z wyjaśnieniem

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

# Usatysfakcjonowana kładzie się spać.

browser.quit()
```

Mamy określenie na komentarze...

Kiedy zacząłem pracę w firmie Resolver, tworzony przez siebie kod wręcz naszpicowałem opisowymi komentarzami. Moi współpracownicy powiedzieli mi: „Harry, mamy określenie na komentarze. Nazywamy je kłamstwami”. Byłem szokowany! Przecież w szkole dowiedziałem się, że umieszczanie komentarzy należy do dobrej praktyki.

Jednak nie należy z nimi przesadzać. W kodzie na pewno znajdzie się miejsce na komentarze, które mają wskazać kontekst i zamierzenia autora. Natomiast zupełnie bezcelowe jest umieszczanie komentarzy powtarzających to, co można bez problemu odczytać na podstawie kodu:

```
# Inkrementacja zmiennej wibble o 1.  
wibble += 1
```

Powyższy komentarz nie tylko jest bezcelowy, ale również niebezpieczny. Jeżeli zapomnisz o jego uaktualnieniu po modyfikacji kodu, wówczas komentarz stanie się mylący. Idealnym rozwiązaniem jest dążenie do zapewnienia maksymalnej czytelności kodu, na przykład przez użycie dobrych nazw dla zmiennych i funkcji. W połączeniu z doskonale przemyślaną strukturą nie będziesz potrzebował żadnych komentarzy wyjaśniających sposób działania kodu. Wówczas w różnych miejscach wystarczy jedynie dodanie komentarzy wyjaśniających *powody* użycia danego kodu.

Istnieje jeszcze wiele innych miejsc, w których komentarze są bardzo użyteczne. Przekonasz się, że framework Django wstawia je w wielu plikach generowanych dla użytkownika i używa w charakterze podpowiedzi dotyczących API framework'a. Ponadto komentarze wykorzystujemy w celu wyjaśnienia otrzymanych informacji od użytkownika umieszczanych w testach funkcjonalnych. Wymuszając przygotowanie spójnych informacji dotyczących testu, zawsze mamy pewność, że test będzie przeprowadzony z uwzględnieniem punktu widzenia użytkownika.

To jest tylko wierzchołek góry lodowej. Mamy jeszcze BDD (ang. *behavior-driven development*) i testowanie za pomocą DSL, ale to są tematy na zupełnie inne książki.

Zwróć uwagę, że poza zapisem testów w postaci komentarzy zmieniłem także polecenie assert, które teraz wyszukuje słowa *Listy* zamiast *Django*. To oznacza, że teraz oczekujemy zakończenia testu niepowodzeniem. Spróbujmy więc wykonać nasz test.

Najpierw uruchom serwer:

```
$ python3 manage.py runserver
```

Następnie w innej powłoce wykonaj test:

```
$ python3 functional_tests.py  
Traceback (most recent call last):  
  File "functional_tests.py", line 10, in <module>  
    assert 'Listy' in browser.title  
AssertionError
```

W ten sposób otrzymaliśmy tak zwane *oczekiwane niepowodzenie*, które wbrew nazwie jest tak naprawdę dobrą wiadomością. Wprawdzie nie tak dobrą, jak zakończenie testu powodzeniem, ale przynajmniej doskonale znamy przyczynę niepowodzenia. Mamy więc potwierdzenie prawidłowego utworzenia testu.

Moduł unittest ze standardowej biblioteki Pythona

Istnieje kilka drobnych, choć irytujących problemów, z którymi prawdopodobnie będziemy musieli sobie poradzić. Pierwszy z nich to niezbyt użyteczny komunikat o treści `AssertionError`. Byłoby dobrze, gdyby test podał rzeczywisty tytuł okna przeglądarki internetowej. Ponadto pozostawienie okna przeglądarki Firefox po zakończeniu testu jest niepotrzebne, więc jest pożądane, aby wspomniane okno zostało automatycznie zamknięte.

Jedną z możliwości jest użycie drugiego parametru w słowie kluczowym `assert`, na przykład w następujący sposób:

```
assert 'Listy' in browser.title, "Tytuł okna przeglądarki: " + browser.title
```

Do zamknięcia niepotrzebnego nam okna przeglądarki internetowej Firefox możemy wykorzystać konstrukcję `try-finally`. Jednak wspomniane wcześniej problemy pojawiają się tak często w trakcie przeprowadzania testów, że opracowano dla nich gotowe rozwiązania. Umieszczono je w module `unittest` znajdującym się w standardowej bibliotece Pythona. Wykorzystajmy więc gotowe rozwiązania! Zmodyfikuj plik `functional_tests.py`, aby przedstawiał się, jak pokazano poniżej.

Plik `functional_tests.py`:

```
from selenium import webdriver
import unittest

class NewVisitorTest(unittest.TestCase): #❶

    def setUp(self): #❷
        self.browser = webdriver.Firefox()

    def tearDown(self): #❸
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self): #❹
        # Edyta dowiedziała się o nowej, wspaniałej aplikacji w postaci listy rzeczy do zrobienia.
        # Postanowiła więc przejść na stronę główną tej aplikacji.
        self.browser.get('http://localhost:8000')

        # Zwróciła uwagę, że tytuł strony i nagłówek zawierają słowo Listy.
        self.assertIn('Listy', self.browser.title) #❺
        self.fail('Zakończenie testu!') #❻

        # Od razu zostaje zachęcona, aby wpisać rzecz do zrobienia.
        # [...] pozostałe komentarze jak we wcześniejszym przykładzie...]

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

Prawdopodobnie zwróciłeś uwagę na kilka kwestii.

- ❶ Testy zostały zorganizowane w klasy dziedziczące po `unittest.TestCase`.
- ❷ Część główna testu znajduje się w metodzie o nazwie `test_can_start_a_list_and_retrieve_it_later()`. Każda metoda o nazwie rozpoczynającej się od `test_` jest metodą testową i będzie wykonana przez silnik testów. W klasie możesz mieć więcej niż tylko jedną metodę tego typu. Dobrym rozwiązaniem będzie stosowanie opisowych nazw dla metod testowych.

- ❸ `setUp()` i `tearDown()` to metody specjalne wykonywane odpowiednio przed i po każdym teście. Wykorzystuję je do uruchomienia i zamknięcia przeglądarki internetowej. Zwróć uwagę, że przypominają one nieco konstrukcję `try-catch`, ponieważ metoda `tearDown()` zostanie wykonana nawet wtedy, gdy w trakcie testu wystąpi błąd¹. Po zakończeniu testu na ekranie nie będą dłużej pozostały niepotrzebne nam okna przeglądarki internetowej Firefox.
- ❹ Do przeprowadzenia asercji wykorzystujemy `self.assertEqual` zamiast po prostu słowa kluczowego `assert`. Moduł `unittest` oferuje wiele funkcji pomocniczych przydatnych podczas stosowania asercji, na przykład `assertEqual`, `assertTrue`, `assertFalse` itd.Więcej informacji na ten temat znajdziesz w *dokumentacji modułu unittest*².
- ❺ Polecenie `self.fail` niezależnie od wszystkiego kończy się niepowodzeniem i wyświetla wskazany komunikat błędu. W omawianym kodzie to przypomnienie o zakończeniu testu.
- ❻ Na końcu mamy klauzulę `if __name__ == '__main__'` (jeśli wcześniej nie spotkałeś się z tego rodzaju kodem, to powinieneś wiedzieć, że za jego pomocą skrypt Pythona sprawdza, czy został uruchomiony z poziomu powłoki, a nie zimportowany przez inny skrypt). Wywołujemy `unittest.main()`, co powoduje uruchomienie silnika testów `unittest`, który z kolei automatycznie wyszukuje w pliku klasy i metody testowe, a następnie je wykonuje.
- ❻ Argument `warnings='ignore'` powoduje zawieszenie wyświetlania zbędnego komunikatu `ResourceWarning` dodanego w trakcie powstawania tej książki. Być może zniknie on w chwili, gdy będziesz miał tę książkę w rękach — w takim przypadku możesz spokojnie usunąć z kodu wymieniony argument.



Jeżeli zapoznałeś się z poświęconą testom dokumentacją Django, mogłeś się natknąć na klasę `LiveServerTestCase` i być może zastanawiasz się, czy powinieneś jej używać. Należą Ci się brawa za lekturę podręcznika. Objaśnienie działania wymienionej klasy jest teraz zbyt skomplikowane, ale obiecuje, że wykorzystamy ją w późniejszych rozdziałach książki.

Wypróbujmy teraz zmodyfikowany test!

```
$ python3 functional_tests.py
F
=====
FAIL: test_can_start_a_list_and_retrieve_it_later (_main_.NewVisitorTest)
-----
Traceback (most recent call last):
  File "functional_tests.py", line 18, in
    test_can_start_a_list_and_retrieve_it_later
      self.assertEqual('Listy', self.browser.title)
AssertionError: 'Listy' not found in 'Welcome to Django'

-----
Ran 1 test in 1.747s

FAILED (failures=1)
```

¹ Jedynym wyjątkiem jest sytuacja, gdy w metodzie `setUp()` nastąpi zgłoszenie wyjątku. W takim przypadku metoda `tearDown()` nie zostanie wykonana.

² <https://docs.python.org/3/library/unittest.html>

Otrzymane dane wyjściowe są teraz znacznie czytelniejsze. Okno przeglądarki internetowej Firefox zostało zamknięte, a na ekranie jest wyświetlany komunikat informujący o liczbie wykonanych testów i liczbie testów zakończonych niepowodzeniem. Ponadto `assertIn` wyświetla jasny komunikat błędu. Wspaniale!

Ukryte oczekiwanie

Na obecnym etapie mamy do wykonania jeszcze jedno zadanie: dodanie wywołania `implicitly_wait()` do metody `setUp()`:

```
[...]
def setUp(self):
    self.browser = webdriver.Firefox()
    self.browser.implicitly_wait(3)

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

To jest standardowe podejście stosowane w testach Selenium. Wprawdzie narzędzie Selenium całkiem dobrze radzi sobie z oczekiwaniem na pełne wczytanie strony, zanim spróbuje przeprowadzić na niej jakiekolwiek działania, ale jednak nie jest idealne na tym polu. Metoda `implicitly_wait()` nakazuje oczekивание przez podaną liczbę sekund, jeśli zachodzi potrzeba. W omawianym przykładzie narzędzie Selenium będzie czekało przez maksymalnie trzy sekundy, zanim rozpoczęcie wyszukiwanie tekstu na stronie.



Nie polegaj jedynie na metodzie `implicitly_wait()`, ponieważ nie sprawdza się ona w każdej sytuacji. Doskonale spełnia swoje zadanie w przypadku prostych aplikacji. Jak się jednak przekonasz w wyniku lektury części III książki (na przykład rozdziałów 15. i 20.), gdy poziom skomplikowana aplikacji przekroczy pewną wartość, wówczas konieczne jest opracowanie bardziej złożonych algorytmów *jawnego* oczekiwania przeznaczonych do użycia w testach.

Przekazanie plików do repozytorium

Teraz jest doskonały moment na umieszczenie kodu w repozytorium, ponieważ wprowadziliśmy pewną odizolowaną zmianę. Nasz test funkcjonalny został rozbudowany o komentarze opisujące zadania, które mamy do wykonania, aby powstała minimalna działająca wersja aplikacji w postaci listy rzeczy do zrobienia. Ponadto zmodyfikowaliśmy test, przystosowując go do użycia oferowanego przez Python modułu `unittest` oraz jego różnych funkcji pomocniczych z zakresu testów.

Wydaj polecenie `git status`, a przekonasz się, że jedynym zmodyfikowanym plikiem jest `functional_tests.py`. Następnie wydaj polecenie `git diff`, które pokaże różnice między projektem na dysku lokalnym a ostatnio umieszczonym w repozytorium. Dane wyjściowe drugiego z wymienionych poleceń powinny wskazać, że plik `functional_tests.py` uległ dość znacznym modyfikacjom:

```
$ 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()
+        self.browser.implicitly_wait(3)
+
+    def tearDown(self):
+        self.browser.quit()
[...]
```

Teraz wydaj następujące polecenie:

```
$ git commit -a
```

Opcja `-a` oznacza „automatycznie dodaj wszelkie zmiany wprowadzone w monitorowanych plikach”, czyli wszystkich plikach umieszczonych wcześniej w repozytorium. Powyższe polecenie nie spowoduje dodania do repozytorium nowych plików (w takim przypadku trzeba wyraźnie wydać polecenie `git add`). Jednak często — jak w omawianym przykładzie — nie ma żadnych nowych plików, więc wymienione polecenie stanowi użyteczny skrót.

Po wyświetleniu okna edytora tekstów dodaj opisowy komunikat dotyczący przekazywanych plików, na przykład „W komentarzach podano pierwszą specyfikację testu funkcjonalnego, a ponadto przystosowano go do użycia modułu `unittest`”.

To jest doskonały moment na rozpoczęcie tworzenia faktycznego kodu naszej aplikacji w postaci listy rzeczy do zrobienia. Kontynuuj więc lekturę!

Użyteczne koncepcje TDD

Informacje od użytkownika

Opis przedstawiający sposób działania aplikacji z punktu widzenia użytkownika. Ten opis służy do nadania struktury testowi funkcjonalnemu.

Oczekiwane niepowodzenie

Test kończący się niepowodzeniem w oczekiwany sposób.

Testowanie prostej strony głównej za pomocą testów jednostkowych

Na końcu poprzedniego rozdziału przygotowaliśmy test funkcjonalny, którego wykonanie kończy się niepowodzeniem. Celem testu jest sprawdzenie, czy strona główna zawiera w tytule słowo *Listy*. Najwyższy czas rozpocząć prace nad właściwą aplikacją.

Ostrzeżenie: zaczynamy działać na poważnie

Pierwsze dwa rozdziały celowo były dość proste. Jednak od tej chwili poważnie bierzemy się za tworzenie kodu. Możemy przewidzieć jedno: w pewnym momencie coś na pewno pojedzie nie tak. Istnieje więc prawdopodobieństwo, że otrzymane przez Ciebie wyniki będą inne niż przedstawione w książce. To jest dobre, ponieważ stanowi niezwykle ważny element nauki.

Jednym z powodów mogą być niejasne wyjaśnienia z mojej strony, które spowodują, że zrobisz zupełnie co innego, niż miałem na myśli. W takim przypadku zrób krok wstecz i zastanów się, co chciałeś osiągnąć w danym miejscu. Zadaj sobie kilka pytań, na przykład: które pliki były edytowane, do czego chcesz skłonić użytkownika aplikacji, co jest testowane i dla czego? Być może przeprowadziłeś edycję niewłaściwego pliku bądź funkcji lub też wykonałeś inne testy, niż powinieneś. Jestem przekonany, że w tego rodzaju sytuacjach i momentach zastanowienia możesz dowiedzieć się więcej o technikach TDD niż w pozostałych miejscach, gdy po prostu wykonujesz kolejne wskazywane polecenia.

Nie można również wykluczyć powstania faktycznego błędu w kodzie i wówczas musisz być nieustępliwy. Dokładnie przeczytaj cały komunikat błędu (patrz przedstawiona w dalszej części rozdziału ramka dotycząca stosu wywołań) od początku do końca. Najczęstsze przyczyny błędów to brak przecinka czy ukośnika lub litera s w nazwie jednej z metod narzędzia Selenium. Jak to trafnie ujął Zed Shaw¹, tego rodzaju usuwanie błędów stanowi absolutnie fantastyczną część nauki, więc działaj wytrwale.

Jeżeli wypróbowałeś różne możliwości i nadal nie znajdujesz rozwiązania, zawsze możesz wysłać do mnie wiadomość e-mail (lub zajrzeć na *forum poświęcone książce*²). Powodzenia podczas usuwania błędów!

¹ Zed A. Shaw, *Learn Python The Hard Way*: <http://learnpythonthehardway.org/>

² <https://groups.google.com/forum/#forum/obey-the-testing-goat-book>

Nasza pierwsza aplikacja Django i test jednostkowy

Framework Django zachęca do zastosowania struktury *aplikacji* dla tworzonego kodu. Teoretycznie jeden projekt może mieć wiele aplikacji, można też wykorzystać aplikacje opracowane przez firmy trzecie, a nawet ponownie używać własnych aplikacji w różnych projektach. Muszę przyznać, że w moim przypadku ten sposób się nigdy nie sprawdził. Aplikacje to jednak dobre rozwiązanie w zakresie organizacji kodu.

Rozpoczynamy od utworzenia aplikacji dla naszych list rzeczy do zrobienia:

```
$ python3 manage.py startapp lists
```

Powyzsze polecenie powoduje utworzenie podkatalogu *lists* w katalogu *superlists* (obok *superlists/superlists*) oraz umieszczenie w nim kilku plików przeznaczonych na modele, widoki i najbardziej interesujące nas teraz testy:

```
superlists/
    db.sqlite3
    functional_tests.py
    lists
        admin.py
        __init__.py
        migrations
            __init__.py
        models.py
        tests.py
        views.py
    manage.py
    superlists
        __init__.py
        __pycache__
        settings.py
        urls.py
        wsgi.py
```

Testy jednostkowe i różnice dzielące je od testów funkcjonalnych

Podobnie jak w wielu innych obszarach granica między testami jednostkowymi i funkcjonalnymi czasami się zaciera. Podstawowa różnica polega na tym, że testy funkcjonalne są przeznaczone do testowania aplikacji z zewnątrz, z perspektywy jej użytkownika. Z kolei testy jednostkowe sprawdzają aplikację z wewnętrz, z perspektywy programisty.

Stosowane przez nas podejście z użyciem technik TDD wymaga przetestowania aplikacji za pomocą obu wymienionych rodzajów testów. Sposób pracy będzie więc przedstawał się następująco:

1. Rozpoczynamy od utworzenia *testu funkcjonalnego*, opisując nową funkcjonalność z perspektywy użytkownika.
2. Gdy wykonanie testu funkcjonalnego zakończy się niepowodzeniem, wtedy zaczynamy się zastanawiać, jak powinien wyglądać kod pozwalający na zaliczenie danego testu (lub przynajmniej aktualnego niepowodzenia). W tym momencie wykorzystujemy jeden lub więcej *testów jednostkowych* do zdefiniowania sposobu, w jaki powinien zachowywać się kod. Idea polega na tym, że każdy wiersz kodu produkcyjnego powinien być przetestowany przez (przynajmniej) jeden z testów jednostkowych.

3. Gdy mamy test jednostkowy zakończony niepowodzeniem, przystępujemy do utworzenia najmniejszej ilości kodu aplikacji wystarczającej do zaliczenia testu jednostkowego. Między krokami 2. i 3. można przechodzić kilkakrotnie aż do chwili, gdy będziesz uważać, że nastąpił choć niewielki postęp w zaliczeniu testu funkcjonalnego.
4. Teraz ponownie wykonujemy testy funkcjonalne i sprawdzamy, czy zostały zaliczone lub przynajmniej nastąpił jakikolwiek postęp w trakcie ich wykonywania. To może wymagać przygotowania kolejnych testów jednostkowych, dodania nowego kodu itd.

Jak możesz zobaczyć, w trakcie wymienionego procesu testy funkcjonalne są tym, co kieruje programowaniem na wysokim poziomie, natomiast testy jednostkowe kierują programowaniem na niskim poziomie.

Czy takie rozwiązanie nie oznacza, że testy się wzajemnie nakładają? Czasami można odnieść takie wrażenie, ale testy funkcjonalne i jednostkowe naprawdę mają odmienne cele i zwykle przedstawiają się całkiem odmiennie.



Testy funkcjonalne powinny pomóc w przygotowaniu aplikacji oferującej odpowiednią funkcjonalność i gwarantują, że nigdy przypadkowo jej nie zepsujesz. Z kolei testy jednostkowe powinny pomóc w utworzeniu kodu przejrzystego i pozbawionego błędów.

Wystarczy już teorii, przekonajmy się teraz, jak można ją zastosować w praktyce.

Testy jednostkowe w Django

Zobaczmy, jak można utworzyć test jednostkowy dla widoku naszej strony głównej. Otwórz plik `lists/tests.py` i umieść w nim przedstawiony poniżej kod.

Plik `lists/tests.py`:

```
from django.test import TestCase

# Miejsce na utworzenie testów.
```

Django użytecznie podpowiada użycie specjalnej wersji oferowanej przez framework klasy `TestCase`. To rozszerzona wersja standardowej klasy `unittest.TestCase`, rozbudowana o pewne funkcje charakterystyczne dla Django, które będziemy odkrywać w kilku kolejnych rozdziałach.

Wcześniej zobaczyłeś, że cykl TDD rozpoczyna się od testu, którego wykonanie kończy się niepowodzeniem. Następnie tworzymy kod pozwalający na zaliczenie wspomnianego kodu. Jednak zanim przejdziemy dalej, powinniśmy się upewnić, że tworzone przez nas testy jednostkowe będą wykonywane przez zautomatyzowany silnik testów niezależnie od miejsca jego położenia. W przypadku pliku `functional_tests.py` wykonujemy go bezpośrednio, ale plik wygenerowany przez Django ma znacznie ciekawsze możliwości. Przygotowujemy więc test, który na pewno zakończy się niepowodzeniem:

```
from django.test import TestCase

class SmokeTest(TestCase):

    def test_bad_maths(self):
        self.assertEqual(1 + 1, 3)
```

Teraz wywołujemy mistyczny silnik testów w Django. Jak zwykle służy do tego po prostu polecenie `manage.py`:

```
$ python3 manage.py test
Creating test database for alias 'default'...
F
=====
FAIL: test_bad_maths (lists.tests.SmokeTest)
-----
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests.py", line 6, in test_bad_maths
    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3
-----
Ran 1 test in 0.001s

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

Doskonale. Wydaje się, że maszyna działa. Zatem to dobry czas na przekazanie plików do repozytorium:

```
$ git status # Powinieneś zobaczyć, że katalog lists/ nie jest monitorowany.
$ git add lists
$ git diff --staged # Powinieneś zobaczyć różnice między kodem w komputerze lokalnym i repozytorium.
$ git commit -m "Dodanie aplikacji i testu, który na pewno kończy się niepowodzeniem."
```

Jak zapewne się domyślisz, opcja `-m` pozwala na podanie dla przekazywanych plików opisu bezpośrednio w powłoce bez konieczności użycia edytora. Od Ciebie zależy sposób, w jaki będziesz używał Git w powłoce. Tutaj pokazuję rozwiązania, które sam najczęściej stosuję. Podstawowa zasada brzmi: *zanim przekażesz kod do repozytorium, upewnij się, że dokładnie go przejrzałeś.*

MVC w Django, adresy URL i funkcje widoku

Ogólnie rzecz biorąc, w Django jest stosowany klasyczny wzorzec model-widok-kontroler (MVC). Ale tylko *ogólnie*. Na pewno istnieją modele, widoki bardziej przypominają kontroler oraz mamy szablony, które tak naprawdę działają w charakterze widoku. Jednak ogólna idea MVC została zachowana. Jeżeli jesteś zainteresowany, więcej dokładnych informacji na ten temat znajdziesz w *dokumencie FAQ poświęconym Django*³.

Niezależnie od zastosowanej struktury podobnie jak w innych serwerach WWW głównym zadaniem Django jest wykonywanie odpowiednich zadań, gdy użytkownik zażąda przejścia pod dany adres URL w witrynie. Sposób działania Django przedstawia się mniej więcej następująco:

1. Otrzymano żądanie HTTP do określonego adresu URL.
2. Wykorzystując pewne reguły, Django próbuje ustalić, która funkcja widoku powinna zająć się obsługą żądania (to nosi nazwę *analizy* adresu URL).
3. Funkcja widoku przetwarza żądanie i zwraca odpowiedź HTTP.

³ <https://docs.djangoproject.com/en/1.7/faq/general/>

Wobec tego konieczne jest przetestowanie dwóch rzeczy:

- Czy adres URL dla katalogu głównego witryny (/) można przełożyć na konkretną funkcję widoku?
- Czy wspomniana funkcja widoku może zwrócić kod HTML, który pozwoli na zaliczenie testu funkcjonalnego?

Rozpoczynamy od sprawdzenia punktu pierwszego. Otwórz plik *lists/tests.py* i zmień znajdujący się tam kod na przedstawiony poniżej.

Plik *lists/tests.py*:

```
from django.core.urlresolvers import resolve
from django.test import TestCase
from lists.views import home_page #❶

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/') #❷
        self.assertEqual(found.func, home_page) #❸
```

Jak przedstawia się sposób działania powyższego kodu?

- ❶❷❸ Funkcja `resolve()` jest przez Django używana wewnętrznie do analizy adresów URL i ustalenia, na które funkcje widoku powinny być one mapowane. Sprawdzamy, czy funkcja `resolve()` wywołana dla katalogu głównego witryny znajduje funkcję o nazwie `home_page()`.
- ❶ Co to jest za funkcja? To jest funkcja widoku, którą wkrótce utworzymy. Jej zadaniem jest wygenerowanie potrzebnego nam kodu HTML. Na podstawie tego polecenia `import` można się dowiedzieć, że funkcja będzie przechowywana w pliku *lists/views.py*.

Jak sądzisz, jaki będzie wynik uruchomienia testów?

```
$ python3 manage.py test
ImportError: cannot import name 'home_page'
```

Otrzymujemy łatwy do przewidzenia i niezbyt interesujący błąd: spróbowaliśmy zaimportować kod, którego nawet jeszcze nie utworzyliśmy. Jednak to nadal są dobre wiadomości, ponieważ z perspektywy technik TDD możliwy do przewidzenia wyjątek jest uznawany za oczekiwane niepowodzenie. Skoro mamy zakończone niepowodzeniem testy funkcjonalny i jednostkowy, możemy przystąpić do tworzenia rzeczywistego kodu aplikacji.

Wreszcie zaczynamy tworzyć kod aplikacji

Wprawdzie nadeszła ekscytująca chwila, ale musisz jednak pamiętać o jednym. Stosowanie technik TDD oznacza, że po długim wyczekiwaniu wykonamy tylko niewielki krok. Tym bardziej że dopiero się uczymy i zaczynamy pracę. Dlatego też pozwalamy sobie na jednorazową zmianę (lub dodanie) jednego wiersza kodu. Ponadto za każdym razem wprowadzamy minimalną zmianę wymaganą do pokonania bieżącego testu zakończonego niepowodzeniem.

Celowo przyjąłem tutaj tak ekstremalne założenia. Mógłbyś w tym miejscu zapytać, na czym polega niepowodzenie bieżącego testu. Odpowiedź jest prosta: nie możemy zaimportować funkcji `home_page()` z pliku *lists/views.py*. Dobre, poprawimy to, ale usuniemy tylko i wyłącznie ten problem. W pliku *lists/views.py* umieść przedstawiony poniżej kod.

Plik *lists/views.py*:

```
from django.shortcuts import render

# Miejsce na utworzenie widoków.
home_page = None
```

Już słyszę, jak mówisz: „chyba żartujesz”.

Wiem, co myślisz, ponieważ przeżyłem to sam, gdy po raz pierwszy zademonstrowano mi przykład użycia technik TDD. Proszę Cię o chwilę cierpliwości, wkrótce wszystko się wyjaśni. Teraz po prostu wykonuj polecenia i przekonaj się, dokąd nas to zaprowadzi.

Wykonajmy ponownie testy:

```
$ python3 manage.py test
Creating test database for alias 'default'...
E
=====
ERROR: test_root_url_resolves_to_home_page_view (lists.tests.HomePageTest)
-----
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests.py", line 8, in
    test_root_url_resolves_to_home_page_view
    found = resolve('/')
  File "/usr/local/lib/python3.4/dist-packages/django/core/urlresolvers.py",
line 485, in resolve
    return get_resolver(urlconf).resolve(path)
  File "/usr/local/lib/python3.4/dist-packages/django/core/urlresolvers.py",
line 353, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
django.core.urlresolvers.Resolver404: {'tried': [<RegexURLResolver
<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''}

-----
Ran 1 test in 0.002s

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

Odczyt stosu wywołań

Poświeć chwilę na opanowanie umiejętności odczytu stosu wywołań, ponieważ to zadanie bardzo często wykonywane podczas stosowania technik TDD. Powinieneś potrafić przejrzeć stos wywołań i odszukać istotne wskazówki:

```
=====
ERROR: test_root_url_resolves_to_home_page_view (lists.tests.HomePageTest) #❶
-----
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests.py", line 8, in
    test_root_url_resolves_to_home_page_view
    found = resolve('/') #❷
  File "/usr/local/lib/python3.4/dist-packages/django/core/urlresolvers.py",
line 485, in resolve
    return get_resolver(urlconf).resolve(path)
  File "/usr/local/lib/python3.4/dist-packages/django/core/urlresolvers.py",
line 353, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
django.core.urlresolvers.Resolver404: {'tried': [<RegexURLResolver #❸
<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''} #❹

[...]
```

3.4 Podczas analizy stosu wywołań z reguły najpierw czyta się właściwy komunikat błędu. Czasami okazuje się on być wystarczający i pozwala na natychmiastową identyfikację problemu. Jednak nie zawsze tak się dzieje. Na przykład w omawianej sytuacji komunikat pozostaje niejasny.

- ① Następnie trzeba dokładnie sprawdzić, wykonanie którego testu kończy się niepowodzeniem. Czy niepowodzenie dotyczy spodziewanego testu, na przykład utworzonego przed chwilą? W omawianym przypadku odpowiedź brzmi: tak.
- ② Następnie w kodzie testu przechodzimy do miejsca, które spowodowało niepowodzenie. Stos wywołań trzeba przejrzeć od początku, szukając nazwy pliku testów, oraz sprawdzić, w której funkcji testu i w którym wierszu kodu nastąpiło niepowodzenie. W omawianym przykładzie to wiersz wywołujący funkcję `resolve()` wraz z adresem URL w postaci `/`.

Istnieje jeszcze kolejny krok, w którym analizujemy kod aplikacji zaangażowany w rozwiązywanego problemu. W omawianym przypadku to będzie kod Django. W prawdzie nie pokazuję tutaj tego kroku, ale w dalszej części książki znajdziesz mnóstwo przykładów dla wspomnianego kroku.

Teraz połączymy wszystko w całość. Analizując stos wywołań, widzimy, że próba przetwarzania adresu URL `/` powoduje zgłoszenie przez Django błędu o kodzie 404. Innymi słowy, Django nie znajduje mapowania dla wymienionego adresu URL. Musimy to więc naprawić.

urls.py

Django używa pliku o nazwie `urls.py` w celu określenia mapowań adresów URL na poszczególne funkcje widoku. W katalogu `superlists/superlists` znajduje się główny plik `urls.py` przeznaczony dla całej witryny. Poniżej przedstawiono zawartość wymienionego pliku.

Plik `superlists/urls.py`:

```
from django.conf.urls import patterns, include, url
from django.contrib import admin

urlpatterns = patterns('',
    # Przykłady:
    # url(r'^$', 'superlists.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),

    url(r'^admin/', include(admin.site.urls)),
)
```

Jak zwykle framework Django dostarcza wiele komentarzy oraz sugestii.

Wpis `url` rozpoczyna się od wyrażenia regularnego definiującego adres URL, do którego ma zastosowanie. Celem jest wskazanie miejsca, do którego ma być skierowane żądanie. To może być funkcja podana za pomocą notacji z użyciem kropki (na przykład `superlists.views.home`) lub inny plik `urls.py` wczytany za pomocą polecenia `include`.

Jak możesz zobaczyć, omawiany plik zawiera domyślnie jeden wpis przeznaczony dla administratora witryny. Ponieważ ten wpis nie jest nam teraz potrzebny, więc poprzedzamy go znakiem komentarza.

Plik `superlists/urls.py`:

```
from django.conf.urls import patterns, include, url
from django.contrib import admin

urlpatterns = patterns('',
    # Przykłady:
    # url(r'^$', 'superlists.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),

    # url(r'^admin/', include(admin.site.urls)),
)
```

Pierwszy wpis w `urlpatterns` zawiera wyrażenie regularne `^$` oznaczające pusty ciąg tekstowy. Czy może ono dopasować katalog główny naszej witryny, który testujemy, podając ukośnik `(/)`? Przekonajmy się o tym, usuwając znak komentarza z początku wiersza.



Jeżeli nie znasz wyrażeń regularnych, możesz nadal od nich uciekać i po prostu uwierzyć mi na słowo. Warto jednak zmusić się i przynajmniej ogólnie je poznać.

```
urlpatterns = patterns('',
    # Przykłady:
    url(r'^$', 'superlists.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),
    # url(r'^admin/', include(admin.site.urls)),
)
```

Ponownie uruchamiamy test jednostkowy, wydając polecenie `python3 manage.py test`:

```
ImportError: No module named 'superlists.views'
[...]
django.core.exceptions.ViewDoesNotExist: Could not import
superlists.views.home. Parent module superlists.views does not exist.
```

Osiagnęliśmy pewien postęp! Nie otrzymujemy już błędu o kodzie 404. Zamiast tego Django narzeka, że podana za pomocą notacji z użyciem kropki funkcja `superlists.views.home` nie prowadzi do rzeczywistego widoku. Naprawimy to, wskazując przygotowane przez nas miejsce zarezerwowane na obiekt `home_page` wewnątrz pliku `lists`, a nie `superlists`.

Plik `superlists/urls.py`:

```
urlpatterns = patterns('',
    # Przykłady:
    url(r'^$', 'lists.views.home_page', name='home'),
```

Teraz ponownie przeprowadzamy test:

```
django.core.exceptions.ViewDoesNotExist: Could not import
lists.views.home_page. View is not callable.
```

Test jednostkowy utworzył połączenie między adresem URL w postaci `/` i wierszem `home_page = None` w pliku `lists/views.py`, a teraz narzeka na brak możliwości wywołania `home_page`, ponieważ to nie jest funkcja. Mamy więc powód do przekształcenia tego wiersza na postać rzeczywistej funkcji. Każda pojedyncza zmiana w kodzie jest sterowana testami. Wprowadź więc kolejną zmianę w pliku `lists/views.py`.

Plik *lists/views.py*:

```
from django.shortcuts import render

# Miejsce na utworzenie widoków.
def home_page():
    pass

I co teraz?
```

```
$ python3 manage.py test
Creating test database for alias 'default'...
.

-----
Ran 1 test in 0.003s

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

Hura! Udało nam się zaliczyć pierwszy test jednostkowy. To jest niewątpliwie moment, w którym warto umieścić pliki w repozytorium.

```
$ git diff # Powinieneś zobaczyć zmiany wprowadzone w plikach urls.py, tests.py i views.py.
$ git commit -am"Pierwszy test jednostkowy, mapowanie url i atrapa widoku."
```

To już ostatni wariant polecenia `git commit`, który chcę pokazać. Jednoczesne użycie opcji `a` i `m` oznacza wprowadzenie wszystkich zmian do monitorowanych plików oraz podanie dla przekazywanych plików opisu bezpośrednio w powłoce.



Wprawdzie polecenie `git commit -am` to najszybszy sposób przekazania plików do repozytorium, ale jednocześnie dostarcza najmniej informacji o całym procesie. Dlatego też wcześniej warto wydać polecenia `git status` i `git diff`, aby tym samym dokładnie dowiedzieć się, jakie zmiany zostaną wprowadzone.

Testy jednostkowe widoku

Przystępujemy do utworzenia testu dla widoku, aby mógł mieć znacznie bardziej rozbudowaną postać niż jedynie funkcji, która nic nie robi. Zadaniem funkcji widoku ma być wygenerowanie rzeczywistej odpowiedzi HTML dla przeglądarki internetowej. Otwórz plik *lists/tests.py* i dodaj nową metodę *testową*. Poniżej wszystko dokładnie wyjaśnię.

Plik *lists/tests.py*:

```
from django.core.urlresolvers import resolve
from django.test import TestCase
from django.http import HttpRequest

from lists.views import home_page

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/')
        self.assertEqual(found.func, home_page)

    def test_home_page_returns_correct_html(self):
        request = HttpRequest() #❶
        response = home_page(request) #❷
        self.assertTrue(response.content.startswith(b'<html>'))
        self.assertIn(b'<title>Listy rzeczy do zrobienia</title>', response.content) #❸
        self.assertTrue(response.content.endswith(b'</html>')) #❹
```

Jak przedstawia się sposób działania tej nowej metody testowej?

- ❶ Tworzymy obiekt `HttpRequest`, który jest widziany przez Django, gdy użytkownik wykonuje żądanie do strony internetowej.
- ❷ Wymieniony obiekt jest przekazywany widokowi `home_page` odpowiedzialnemu za wygenerowanie odpowiedzi. Nie będziesz zaskoczony, słysząc, że ten obiekt jest egzemplarzem klasy o nazwie `HttpResponse`. Następnie definiujemy asercję w celu sprawdzenia, czy zawartość odpowiedzi (czyli kod HTML przekazywany użytkownikowi) zawiera pewne właściwości.
- ❸ Na początku odpowiedzi ma znajdować się znacznik `<html>`, który zostanie zamknięty na jej końcu. Zwróć uwagę na fakt, że `response.content` to ciąg niezmodyfikowanych bajtów, a nie ciąg tekstowy Pythona. Dlatego też w celu przeprowadzenia porównania trzeba użyć składni `b''`. Więcej informacji na ten temat znajdziesz w *dokumentacji Pythona 3⁴*.
- ❹ Gdzieś w odpowiedzi powinien znajdować się znacznik `<title>` wraz ze słowem *Listy*, ponieważ taki warunek zdefiniowaliśmy wcześniej w teście funkcjonalnym.

Warto przypomnieć ponownie, że test jednostkowy jest sterowany testem funkcjonalnym. Jednak obecnie znacznie bardziej przypomina rzeczywisty kod i można stwierdzić, że myślimy teraz jak programiści.

Wykonajmy więc test jednostkowy i przekonajmy się, jakie otrzymamy dane wyjściowe:

```
TypeError: home_page() takes 0 positional arguments but 1 was given
```

Cykl test jednostkowy — tworzenie kodu

Teraz możemy zacząć przyzwyczajać się do stosowanego w programowaniu sterowanym testami cyklu test jednostkowy — tworzenie kodu:

1. W powłoce wykonaj test jednostkowy i zobacz, dlaczego kończy się niepowodzeniem.
2. W edytorze wprowadź minimalną zmianę w kodzie, aby usunąć bieżące niepowodzenie testu jednostkowego.

Cały cykl powtórz!

Im większe dążenie do otrzymania prawidłowego kodu, tym mniejsze i bardziej minimalne zmiany należy wprowadzać w trakcie każdego cyklu. Idea polega na uzyskaniu pewności, że każdy fragment kodu został sprawdzony za pomocą testów. Wprawdzie takie podejście może wydawać się pracochłonne, ale kiedy nabierzesz większej wprawy, prace będziesz mógł prowadzić dość szybko. Prace mogą odbywać się na tyle szybko, że zwykle wprowadza się jedynie mikroskopijne zmiany w kodzie pomimo istnienia możliwości wprowadzenia większych zmian.

Przekonajmy się, jak szybko można zastosować omówiony powyżej cykl.

- Minimalna zmiana w kodzie.

Plik `lists/tests.py`:

```
def home_page(request):  
    pass
```

⁴ <https://docs.djangoproject.com/en/1.7/topics/python3/>

- Wykonanie testów.

```
self.assertTrue(response.content.startswith(b'<html>'))
AttributeError: 'NoneType' object has no attribute 'content'
```

- Zmiana w kodzie — użycie django.http.HttpResponse (zgodnie z oczekiwaniami).

Plik *lists/tests.py*:

```
from django.http import HttpResponse

# Miejsce na utworzenie widoków.
def home_page(request):
    return HttpResponse()
```

- Ponowne wykonanie testów.

```
self.assertTrue(response.content.startswith(b'<html>'))
AssertionError: False is not true
```

- Wprowadzenie zmiany w kodzie.

Plik *lists/tests.py*:

```
def home_page(request):
    return HttpResponse('<html>')
```

- Wykonanie testów.

```
AssertionError: b'<title>Listy rzeczy do zrobienia</title>' not found in b'<html>'
```

- Zmiana w kodzie.

Plik *lists/tests.py*:

```
def home_page(request):
    return HttpResponse('<html><title>Listy rzeczy do zrobienia</title>')
```

- Wykonanie testów — czy już prawie skończyliśmy?

```
self.assertTrue(response.content.endswith(b'</html>'))
AssertionError: False is not true
```

- Jeszcze jedna, ostatnia próba.

Plik *lists/tests.py*:

```
def home_page(request):
    return HttpResponse('<html><title>Listy rzeczy do zrobienia</title></html>')
```

- Na pewno?

```
$ python3 manage.py test
Creating test database for alias 'default'...
..
-----
Ran 2 tests in 0.001s

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

Tak! Teraz możemy wykonać nasze testy funkcjonalne. Jeżeli wcześniej tego nie zrobileś, nie zapomnij o uruchomieniu serwera. Wydaje się, że jesteśmy już na ostatniej prostej... czy na pewno?

```
$ python3 functional_tests.py
F
=====
FAIL: test_can_start_a_list_and_retrieve_it_later (_main_.NewVisitorTest)
-----
Traceback (most recent call last):
  File "functional_tests.py", line 20, in
```

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

Ran 1 test in 1.609s

FAILED (failures=1)

Zakończony niepowodzeniem? Co takiego? Aha, to tylko małe przypomnienie. Tak? Oczywiście, mamy stronę internetową!

Hm. Cóż, pomyślałem, że to będzie ekscytujące zakończenie rozdziału. Nadal możesz czuć się nieco zbitą z tropu, być może oczekując wyjaśnienia wszystkich wykonanych testów. Nie przejmuj się, wszystko stanie się wkrótce jasne. Mam nadzieję, że kończąc lekturę tego rozdziału, poczułeś dreszczyk emociji.

Dla uspokojenia warto przekazać pliki do repozytorium i zobaczyć, co udało nam się zrobić:

```
$ git diff # Powinieneś zobaczyć nowy test w pliku tests.py oraz widok w pliku views.py.
$ git commit -am "Prosty widok generuje teraz minimalną ilość kodu HTML."
```

To całkiem sporo jak na ten rozdział. Możesz wydać polecenie `git log` wraz z opcją `--oneline`, aby zobaczyć poczynione postępy:

```
$ git log --oneline
a6e6cc9 Prosty widok generuje teraz minimalną ilość kodu HTML.
450c0f3 Pierwszy test jednostkowy, mapowanie url i atrapa widoku.
ea2b037 Dodanie aplikacji i testu, który na pewno kończy się niepowodzeniem.
[...]
```

Całkiem nieźle; zajęliśmy się następującymi kwestiami:

- Uruchomienie aplikacji Django.
- Silnik testów jednostkowych w Django.
- Różnice między testami funkcjonalnymi i testami jednostkowymi.
- Analiza adresów URL w Django oraz plik `urls.py`.
- Funkcje widoku w Django, obiekty żądania i odpowiedzi.
- Wygenerowanie podstawowego kodu HTML.

Użyteczne polecenia i koncepcje

Uruchomienie serwera Django:

```
python3 manage.py runserver
```

Wykonanie testów funkcjonalnych:

```
python3 functional_tests.py
```

Wykonanie testów jednostkowych:

```
python3 manage.py test
```

Cykl test jednostkowy i tworzenie kodu:

1. Wykonaj test jednostkowy w powłoce.
2. Wprowadź minimalną zmianę w kodzie.
3. Powtórz proces!

Do czego służą te wszystkie testy?

Zobaczyłeś już w działaniu podstawowe techniki TDD, więc warto zrobić przerwę i dowieź się, dlaczego to wszystko robimy.

Wyobrażam sobie wielu czytelników bliskich frustracji, część z Was miała już wcześniej styczność z testami jednostkowymi, z kolei inni twierdzą, że nie mają czasu na ich stosowanie. Być może chciałbyś zadać mi następujące pytania:

- Czy te wszystkie testy to jednak nie przesada?
- Dlaczego część z nich się zazębia? Zauważylem powielanie kodu między testami funkcjonalnymi i jednostkowymi.
- Jaki jest cel importu `django.core.urlresolvers` w testach jednostkowych? Zajmujemy się testowaniem kodu Django, czy opracowanego przez firmy trzecie?
- Przedstawione testy jednostkowe wydają się zbyt łatwe — testujemy jeden wiersz deklaracji i jednowierszową funkcję, której wartością zwrotną jest stała. Czy to nie jest marnowanie czasu? Czy nie powinniśmy wykorzystywać testów do znacznie bardziej skomplikowanych zadań?
- Co z tymi wszystkimi drobnymi zmianami wprowadzanymi w trakcie cyklu test jednostkowy i tworzenie kodu? Czy możemy pominać małe kroki i przejść od razu na koniec, mam tutaj na myśli `home_page = None`? Naprawdę?
- Nie odpowiedziałeś mi na następujące pytanie: czy w rzeczywistych projektach *naprawdę* tworzysz kod w taki właśnie sposób?

Racja, młody podróżniku. Ja również zadawałem dużo tego rodzaju pytań, ponieważ to są bardzo dobre pytania. W rzeczywistości nadal je sobie zadaję, praktycznie przez cały czas. Czy tworzenie kodu w taki właśnie sposób ma wartość i znaczenie? Czy to nie za bardzo przypomina rodzaj kultu?

Programowanie przypomina wyciąganie wiadrem wody ze studni

Nie ulega wątpliwości, że programowanie to trudna sztuka. Bardzo często jesteśmy sprytni i osiągamy sukces. Techniki TDD zostały opracowane po to, aby pomóc nam, gdy nie jesteśmy aż tak sprytni. Kent Beck (wynalazca technik TDD) posługuje się metaforą wyciągania wody ze studni za pomocą wiadra przywiązanego do liny. Kiedy studnia jest płytka, a wiadro zapelnione jedynie w części, wtedy operacja będzie łatwa. Wciagnięcie pierwszego pełnego wiadra również wydaje się łatwe. Jednak po wyciągnięciu kolejnych wiader szybko stajesz się zmęczony. Techniki TDD są jak koło zapadkowe, które pozwala na zachowanie postępu podczas pracy i krótki odpoczynek oraz daje gwarancję, że wiadro nie zsunie się z powrotem w dół. W ten sposób nie musisz być sprytny przez cały czas (patrz rysunek 4.1).



Rysunek 4.1. Testuj wszystko (oryginalna ilustracja: Allie Brosh, Hyperbole and a Half¹)

Ogólnie rzecz biorąc, prawdopodobnie jesteś gotowy przyznać, że techniki TDD to dobra idea. Jednocześnie być może uważasz, że nieco przesadzam. Testować nawet najmniejszą rzecz i wykonywać absurdalnie wiele małych kroków?

Programowanie sterowane testami to *dyscyplina*, a to oznacza, że wspomniana umiejętność nie przychodzi naturalnie. Ponieważ wiele plonów będących skutkiem jej stosowania nie pojawi się od razu, ale dopiero po dłuższym czasie, więc trzeba się też zmuszać do stosowania testów w danej chwili. Teraz już wiesz, co ilustruje nieoficjalna maskotka technik TDD (czyli Testing Goat) — uparte dążenie do wykonywania testów.

¹ <http://hyperboleandahalf.blogspot.co.uk/2010/06/this-is-why-ill-never-be-adult.html>

Rozważania dotyczące wartości prostych testów i funkcji

Na krótką metę może wydawać się nieco głupie tworzenie testów przeznaczonych do sprawdzania prostych funkcji i stałych. Można sobie wyobrazić stosowanie technik „prawie” TDD, czyli opartych na znacznie luźniejszych regułach, według których testy jednostkowe nie są używane do testowania *absolutnie wszystkiego*. Jednak w tej książce moim celem jest zadeemonstrowanie pełnego, rygorystycznego podejścia do stosowania technik TDD. Podobnie jak ciosy w sztukach walki idea polega na poznaniu ciosów w kontrolowanym środowisku, w którym nie trzeba mierzyć się z żadnymi przeciwnościami, a zapamiętanie technik nie sprawia większych trudności. Na razie zadanie wydaje się łatwe, ponieważ rozpoczynamy od bardzo prostego przykładu. Problemy pojawiają się dopiero wraz ze wzrostem poziomu skomplikowania aplikacji. A niebezpieczeństwo polega na tym, że poziom skomplikowania najczęściej „podkrada się” stopniowo. Możesz nawet tego nie zauważać, ale bardzo szybko zostaniesz ugotowany.

Mamy jeszcze dwie inne rzeczy przemawiające za małymi, prostymi testami przeznaczonymi do sprawdzania prostych funkcji.

Po pierwsze, skoro to ma być naprawdę prosty test, jego utworzenie nie powinno zajmować wiele czasu. Przestań więc narzekać, a zacznij tworzyć te testy.

Po drugie, zawsze warto mieć przygotowane miejsce zarezerwowane. Przygotowany *test* dla prostej funkcji oznacza znacznie mniejszą barierę psychologiczną do przewyciężenia, gdy ta prosta funkcja staje się znacznie bardziej skomplikowana — prawdopodobnie rozrasta się o konstrukcję if. Następnie kilka tygodni później zostaje rozbudowana o pętlę for. Zanim zdążysz się zorientować, funkcja stanie się rekurencyjną i polimorficzną metaklasą. Ponieważ była testowana od samego początku, dodanie nowego testu na kolejnych etapach rozbudowy wydaje się całkiem naturalne, a sama funkcja jest doskonale przetestowana. Alternatywą jest próba ustalenia, kiedy funkcja staje się „wystarczająco skomplikowana”, co oczywiście jest bardzo subiektywne. Co gorsza, brak miejsca zarezerwowanego na test sprawia wrażenie konieczności włożenia sporego wysiłku w przetestowanie funkcji. To powoduje odwlekanie tej operacji i całkiem szybko stajesz się ugotowany.

Zamiast próbować określać niezwykle subiektywne reguły wskazujące moment rozpoczęcia testów, osobiście sugeruję stosowanie dyscypliny od samego początku. Podobnie jak ma to miejsce w wielu innych obszarach, konieczne będzie poświęcenie pewnej ilości czasu na poznanie reguł, zanim można będzie je złamać.

Teraz powracamy do naszych testów.

Użycie Selenium do testowania interakcji użytkownika

Na jakim etapie zakończyliśmy pracę w poprzednim rozdziale? Wykonajmy ponownie test i przekonajmy się o tym.

```
$ python3 functional_tests.py
F
=====
FAIL: test_can_start_a_list_and_retrieve_it_later (_main_.NewVisitorTest)
-----
Traceback (most recent call last):
  File "functional_tests.py", line 20, in
```

```
test_can_start_a_list_and_retrieve_it_later
    self.fail('Zakończenie testu!')
AssertionError: Zakończenie testu!
-----
Ran 1 test in 1.609s
FAILED (failures=1)
```

Czy wykonałeś test i otrzymałes komunikat błędu informujący o problemie z wczytywaniem strony (Problem loading page) lub o braku możliwości nawiązania połączenia (Unable to connect)? Podobnie było u mnie. Przyczyna leży w tym, że nie uruchomiles serwera za pomocą polecenia manage.py runserver. Uruchom więc serwer, wykonaj test, a otrzymasz odpowiedni komunikat o niepowodzeniu testu.



Oto jedna z doskonałych cech technik TDD: nie musisz martwić się, że zapomnisz, co powinno zostać zrobione w następnej kolejności. Wystarczy ponownie wykonać testy, a wyświetlane przez nie komunikaty dokładnie wskażą to, nad czym pracujesz.

Otrzymaliśmy komunikat o zakończeniu testu i tak też postępujemy. Otwórz teraz plik *functional_tests.py* i rozbuduj nasz test funkcjonalny.

Plik *functional_tests.py*:

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import unittest

class NewVisitorTest(unittest.TestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(3)

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

    def test_can_start_a_list_and_retrieve_it_later(self):
        # Edyta dowiedziała się o nowej, wspaniałej aplikacji w postaci listy rzeczy do zrobienia.
        # Postanowiła więc przejść na stronę główną tej aplikacji.
        self.browser.get('http://localhost:8000')

        # Zwróciła uwagę, że tytuł strony i nagłówek zawierają słowo Listy.
        self.assertIn('Listy', self.browser.title)
        header_text = self.browser.find_element_by_tag_name('h1').text
        self.assertIn('Listy', header_text)

        # Od razu zostaje zachęcona, aby wpisać rzecz do zrobienia.
        inputbox = self.browser.find_element_by_id('id_new_item')
        self.assertEqual(
            inputbox.get_attribute('placeholder'),
            'Wpisz rzecz do zrobienia'
        )

        # W polu tekstowym wpisala "Kupić pawie pióra" (hobby Edyty
        # polega na tworzeniu ozdobnych przyńet).
        inputbox.send_keys('Kupić pawie pióra')

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

```

table = self.browser.find_element_by_id('id_list_table')
rows = table.find_elements_by_tag_name('tr')
self.assertTrue(
    any(row.text == '1: Kupić pawie pióra' for row in rows)
)

# Na stronie nadal znajduje się pole tekstowe zachęcające do podania kolejnego zadania.
# Edyta wpisala "Użyć pawich piór do zrobienia przyęty" (Edyta jest niezwykle skrupulatna).
self.fail('Zakończenie testu!')

# Strona została ponownie uaktualniona i teraz wyświetla dwa elementy na liście rzeczy do zrobienia.
[...]

```

Wykorzystujemy kilka metod, które Selenium dostarcza w celu przeanalizowania stron internetowych: `find_element_by_tag_name()`, `find_element_by_id()` i `find_elements_by_tag_name()`. Zwróć uwagę na oznaczającą liczbę mnogą literę s w nazwie ostatniej wymienionej metody. Oznacza to, że wartością zwrotną metody jest kilka elementów zamiast po prostu jednego. Używamy także funkcji `send_keys()`, za pomocą której Selenium wstawia dane do elementów danych wejściowych. Zauważ także klasę `Keys` (nie zapomnij jej zimportować), która pozwala na wysyłanie sygnałów generowanych nie tylko przez klawisze specjalne, takie jak `Enter`, ale również przez modyfikatory, takie jak `Ctrl`.



Zaobserwuj różnicę między funkcjami Selenium `find_element_by...` i `find_elements_by....`. Wartością zwrotną pierwszej z wymienionych funkcji jest element i zgłasza ona wyjątek, jeśli nie może znaleźć wspomnianego elementu. Z kolei druga z wymienionych funkcji zwraca listę, która może być pusta.

Spójrz także na funkcję `any()`. To jest niezbyt znana, wbudowana funkcja Pythona. Zastanawiam się nawet, czy powiniem ją omawiać. Programowanie w Pythonie dostarcza wiele radości.

Jeżeli jesteś jednym z czytelników nieznających Pythona, to śpiesz z wyjaśnieniem, że wewnętrz funkcji `any()` znajduje się generator wyrażenia, który przypomina listę składaną, ale jest znacznie wspanialszy. Koniecznie musisz przeczytać opis wymienionej funkcji. Jeżeli użyjesz wyszukiwarki internetowej, trafisz na eleganckie wyjaśnienie przygotowane przez Guido². Wtedy stwierdzisz, że celem funkcji `any()` nie jest zabawa!

Zobaczmy, jak to się robi:

```

$ python3 functional_tests.py
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"tag name","selector":"h1"}' ; Stacktrace: [...]

```

Wyświetlony komunikat informuje o niemożliwości odnalezienia elementu `<h1>` na stronie. Zobaczmy, co można zrobić, aby wymieniony element dodać do kodu HTML naszej strony głównej.

Duże zmiany wprowadzone w teście funkcjonalnym stwarzają doskonałą okazję do przekazania plików do repozytorium. Zapomniałem o tym w pierwszym szkicu i żałowałem tego później, gdy zmieniłem zdanie i musiałem kombinować z kilkoma innymi zmianami. Im bardziej niepodzielne pozostałą operacje przekazania plików do repozytorium, tym lepiej.

```

$ git diff # Polecenie powinno wyświetlić zmiany wprowadzone w pliku functional_tests.py.
$ git commit -am "Test funkcjonalny teraz sprawdza możliwość dodania elementu listy."

```

² <http://python-history.blogspot.co.uk/2010/06/from-list-comprehensions-to-generator.html>

Reguła „nie testuj stałych” i szablony na ratunek

Spójrzmy na nasze testy jednostkowe zdefiniowane w pliku *lists/tests.py*. Aktualnie ich działanie polega na wyszukiwaniu konkretnych ciągów tekstowych HTML, choć to nie jest szczególnie efektywny sposób na testowanie kodu HTML. Ogólnie rzecz biorąc, jedna z reguł testów jednostkowych brzmi: *nie testuj stałych*. Testowanie HTML jako tekstu przypomina testowanie stałej.

Innymi słowy, jeżeli w dowolnym miejscu kodu znajduje się polecenie:

```
wibble = 3
```

wówczas nie ma większego sensu w przygotowaniu następującego testu:

```
from myprogram import wibble
assert wibble == 3
```

Testy jednostkowe są przeznaczone do testowania logiki, kontroli przepływu programu i konfiguracji. Przeprowadzanie asercji dotyczących dokładnej sekwencji znaków w ciągach tekstowych HTML nie wpisuje się dobrze w przeznaczenie testów jednostkowych.

Co więcej, maglowanie niezmodyfikowanych ciągów tekstowych w Pythonie tak naprawdę nie należy do dobrego sposobu pracy z kodem HTML. Istnieje znacznie lepsze rozwiązanie polegające na użyciu szablonów. Pomijając wszystko inne, jeżeli cały kod HTML uda się umieścić w pliku o rozszerzeniu *.html*, otrzymamy wówczas lepsze podświetlanie składni. Dostępnych jest wiele frameworków szablonów w Pythonie, a Django ma własny, który sprawdza się doskonale. Użyjmy go więc.

Refaktoryzacja w celu użycia szablonu

Chcemy, aby funkcja widoku zwracała dokładnie ten sam kod HTML, ale wykorzystamy zupełnie inny proces. To jest refaktoryzacja, czyli próba ulepszenia kodu *bez wprowadzania zmian w jego funkcjonalności*.

Ostatni fragment w poprzednim akapicie jest niezwykle ważny. Jeżeli w trakcie refaktoryzacji dadasz nową funkcjonalność, wówczas prawdopodobnie wpadniesz w kłopoty. Refaktoryzacja jest dyscypliną, na temat której więcej dowiesz się z książki *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*³ napisanej przez Martina Fowlera.

Podstawowa zasada brzmi: nie można przeprowadzać refaktoryzacji bez testów. Na szczęście stosujemy techniki TDD, więc pozostajemy w grze. Sprawdzamy teraz, czy testy zostaną zaliczone. Dzięki wspomnianym testom będziemy mieli pewność, że refaktoryzacja nie zmienia zachowania programu.

```
$ python3 manage.py test
[...]
OK
```

³ <http://helion.pl/ksiazki/refaktoryzacja-ulepszanie-struktury-istniejacego-kodu-martin-fowler-kent-beck-john-brant-william-opdyrefuko.htm>

Doskonale! Pracę zaczynamy od wyodrębnienia kodu HTML i umieszczenia go w oddzielnym pliku. Utwórz katalog *lists/templates* przeznaczony dla szablonów, a następnie w nim plik *home.html* przeznaczony na kod HTML⁴.

Plik *lists/templates/home.html*:

```
<html>
    <title>Listy rzeczy do zrobienia</title>
</html>
```

Uzyskaliśmy możliwość podświetlania składni, jak miło! Teraz przystępujemy do zmiany funkcji widoku.

Plik *lists/views.py*:

```
from django.shortcuts import render

def home_page(request):
    return render(request, 'home.html')
```

Zamiast tworzyć własny obiekt `HttpResponse`, wykorzystujemy oferowaną przez Django funkcję `render()`. Wymieniona funkcja pobiera żądanie jako pierwszy parametr (z powodów, które zostaną przedstawione nieco później) oraz nazwę widoku do wygenerowania. Django automatycznie szuka katalogów o nazwie *templates* we wszystkich katalogach aplikacji. Następnie w oparciu o zawartość szablonu tworzony jest obiekt `HttpResponse`.



Szablony to oferująca potężne możliwości cecha framework'a Django. Najważniejszą zaletą szablonu jest możliwość zastąpienia zmiennych Pythona kodem HTML. Wprawdzie teraz nie korzystamy z tej funkcji, ale będziemy to robić w kolejnych rozdziałach. Dlatego też używamy funkcji `render()` i (nieco później) `render_to_string()`, zamiast na przykład ręcznie odczytywać zawartość pliku z dysku za pomocą wbudowanego polecenia `open`.

Sprawdźmy, czy zastosowane rozwiązanie działa:

```
$ python3 manage.py test
[...]
=====
ERROR: test_home_page_returns_correct_html (lists.tests.HomePageTest) #❶
-----
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests.py", line 17, in
test_home_page_returns_correct_html
    response = home_page(request) #❷
  File "/workspace/superlists/lists/views.py", line 5, in home_page
    return render(request, 'home.html') #❸
  File "/usr/local/lib/python3.3/dist-packages/django/shortcuts.py", line 48,
in render
    return HttpResponseRedirect(loader.render_to_string(*args, **kwargs),
  File "/usr/local/lib/python3.3/dist-packages/django/template/loader.py", line
170, in render_to_string
    t = get_template(template_name, dirs)
  File "/usr/local/lib/python3.3/dist-packages/django/template/loader.py", line
```

⁴ Niektórzy lubią także dodać kolejny podkatalog o nazwie odpowiadającej aplikacji (na przykład *lists/templates/lists*), a następnie odwołują się do szablonu jako *lists/home.html*. To nosi nazwę „przestrzeni nazw szablonu”. Uznałem to za nadmiernie skomplikowane rozwiązanie dla tak małego projektu, ale warto je rozważyć w większych. Więcej informacji na ten temat znajdziesz w samouczku Django (<https://docs.djangoproject.com/en/1.7/intro/tutorial03/#writing-views-that-actually-do-something>).

```
144, in get_template
    template, origin = find_template(template_name, dirs)
    File "/usr/local/lib/python3.3/dist-packages/django/template/loader.py", line
136, in find_template
    raise TemplateDoesNotExist(name)
django.template.base.TemplateDoesNotExist: home.html #❶
```

```
Ran 2 tests in 0.004s
```

W ten sposób masz kolejną szansę na przeprowadzenie analizy stosu wywołań:

- ❶ Rozpoczynamy od błędu — nie znaleziono szablonu.
- ❷ Dokładnie sprawdzamy, który test kończy się niepowodzeniem — mamy pewność, że to jest test sprawdzający widok HTML.
- ❸ Odszukujemy numer wiersza powodujący niepowodzenie testu — w omawianym przykładzie to wiersz, w którym wywoływana jest funkcja `home_page()`.
- ❹ Na końcu sprawdzamy, która część kodu aplikacji jest odpowiedzialna za niepowodzenie testu — w omawianym przykładzie to wywołanie funkcji `render()`.

Powstaje pytanie: dlaczego Django nie może znaleźć szablonu? Przecież umieściliśmy go w odpowiednim miejscu, czyli w katalogu `lists/templates`.

Powodem jest brak *oficjalnej* rejestracji naszej aplikacji w Django. Niestety umieszczenie niezbędnych komponentów w katalogu projektu i po prostu wydanie polecenia `startapp` okazuje się niewystarczające. Konieczne jest jeszcze wskazanie Django, o co nam tak *naprawdę* chodzi, i wprowadzenie odpowiednich zmian w pliku `settings.py`. Zabezpieczamy się ze wszystkich stron. Otwórz wymieniony plik i odszukaj zmienną o nazwie `INSTALLED_APPS`, do której musimy jeszcze dodać naszą aplikację `lists`.

Plik `superlists/settings.py`.

```
# Definicja aplikacji.
```

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
)
```

Jak możesz zobaczyć, domyślnie zainstalowanych jest wiele aplikacji. Na końcu listy musimy dodać naszą aplikację `lists`. Nie zapomnij o przecinku na końcu. Wprawdzie przecinek może nie być wymagany, ale pewnego dnia możesz być naprawdę poirytowany, gdy o nim zapomnisz, a Python połączy dwa ciągi tekstowe z oddzielnymi wierszy...

Teraz możemy ponownie wykonać testy:

```
$ python3 manage.py test
[...]
self.assertTrue(response.content.endswith(b'</html>'))
AssertionError: False is not true
```

Psiakrew, nie całkiem.



Jeśli używany edytor tekstów dodaje nowy wiersz na końcu pliku, powyższy błąd nie musi wystąpić. W takim przypadku możesz bezpiecznie zignorować poniższe wyjaśnienie i od razu przejść do miejsca, w którym widzisz komunikat OK.

Okazuje się, że brniemy dalej. Framework odnalazł przygotowany przez nas szablon, ale ostatnia z trzech asercji kończy się niepowodzeniem. Nie ulega wątpliwości, że coś jest nie tak z końcówką wygenerowanych danych wyjściowych. Musiałem użyć polecenia `print repr(response.content)` w celu ustalenia przyczyny błędu. Okazało się, że w szablonie na końcu został umieszczony znak nowego wiersza (`\n`). Możemy sobie z tym poradzić w przedstawiony poniżej sposób.

Plik `lists/tests.py`:

```
self.assertTrue(response.content.strip().endswith(b'</html>'))
```

Wprawdzie to małe oszustwo, ale znaki odstępu na końcu pliku HTML tak naprawdę nie powinny mieć dla nas znaczenia. Ponownie wykonujemy testy:

```
$ python3 manage.py test
[...]
OK
```

Nasza refaktoryzacja kodu na tym się kończy, a wynik wykonania testów oznacza, że zachowanie aplikacji nie uległo zmianie. Teraz możemy zmienić już testy, aby dłużej nie testować stałych. Zamiast tego test powinien sprawdzić, czy generowany jest odpowiedni szablon. Przyda nam się kolejna funkcja pomocnicza Django o nazwie `render_to_string()`.

Plik `lists/tests.py`:

```
from django.template.loader import render_to_string
[...]

def test_home_page_returns_correct_html(self):
    request = HttpRequest()
    response = home_page(request)
    expected_html = render_to_string('home.html')
    self.assertEqual(response.content.decode(), expected_html)
```

Funkcja `decode()` została użyta w celu konwersji bajtów `response.content` na ciąg tekstowy Pythona w kodowaniu Unicode. W ten sposób zyskujemy możliwość porównywania ciągów tekstowych zamiast bajtów, jak to odbywało się wcześniej.

Najważniejsze jest, że zamiast testować stałe, teraz testujemy implementację. Doskonale!



Django oferuje klienta testów wraz z narzędziami przeznaczonymi do testowania szablonów. Skorzystamy z nich w dalszych rozdziałach. Teraz używamy działających na niskim poziomie narzędzi, aby zobaczyć, jak to wszystko działa. Nie ma tutaj żadnej magii!

Refaktoryzacja

W poprzednim podrozdziale przedstawiłem niezwykle prosty przykład refaktoryzacji. W swojej książce zatytułowanej *TDD. Sztuka tworzenia dobrego kodu*⁵ Kent Beck stwierdził: „Czy naprawdę zalecam Ci pracować w taki sposób? Nie, jedynie zachęcam do tego, abyś potrafił tak pracować”.

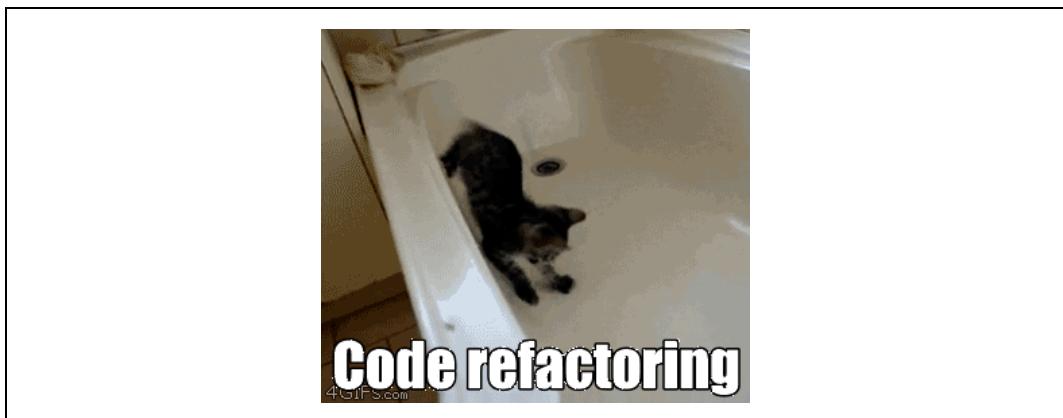
⁵ <http://helion.pl/ksiazki/tdd-sztuka-tworzenia-dobrego-kodu-kent-beck,tddszt.htm>

Kiedy pisałem poprzedni podrozdział poświęcony refaktoryzacji, moim pierwszym pomysłem była zmiana testu na początku, aby od razu upewnić się o użyciu funkcji `render_to_string()`, usunięcie trzech niepotrzebnych asercji, pozostawienie operacji sprawdzenia zawartości względem oczekiwanych danych wyjściowych, a następnie wprowadzenie zmian w kodzie. Zwróć uwagę, że takie podejście tworzyło wiele okazji do uszkodzenia kodu. Na przykład mógłbym zdefiniować szablon zawierający *dowolny* ciąg tekstowy zamiast zawierającego odpowiednie znaczniki `<html>` i `<title>`.



Podczas refaktoryzacji skoncentruj się na kodzie lub testach, nigdy na obu obszarach jednocześnie.

Zawsze istnieje pokusa pominięcia kilku kroków i tym samym wprowadzenia w zachowaniu aplikacji kilku usprawnień podczas refaktoryzacji. Jednak bardzo szybko może się okazać, że wprowadziłeś zmiany w wielu plikach, utraciłeś nad nimi kontrolę i nic już nie działa tak, jak powinno. Jeżeli nie chcesz skończyć jak *kot refaktoryzacji*⁶ (patrz rysunek 4.2), to lepiej pozostań przy małych krokach. Zachowaj całkowitą separację między refaktoryzacją i zmianami w funkcjonalności.



Rysunek 4.2. Kot refaktoryzacji — koniecznie obejrzyj cały animowany plik GIF (źródło: 4GIFs.com)



Wymieniony powyżej „kot refaktoryzacji” jeszcze pojawi się w książce jako przykład tego, co może się stać, gdy spróbujesz jednocześnie zrobić zbyt wiele rzeczy. Potraktuj go jako przeciwieństwo Testing Goat; to raczej demon pojawiający się nad drugim ramieniem i udzielający złej rady.

Po przeprowadzeniu refaktoryzacji dobrym pomysłem będzie przekazanie plików do repozytorium.

```
$ git status # Polecenie wyświetla pliki tests.py, views.py, settings.py plus nowy katalog templates.  
$ git add . # Polecenie spowoduje dodanie do repozytorium niemonitorowanego jeszcze katalogu templates.  
$ git diff --staged # Przejrzanie wprowadzonych zmian.  
$ git commit -m"Refaktoryzacja widoku strony głównej w celu użycia szablonu"
```

⁶ http://www.obeythetestinggoat.com/static/images/refactoring_cat.gif

Nieco więcej o stronie głównej

Nasz test funkcjonalny nadal kończy się niepowodzeniem. Wprowadzimy teraz zmianę w kodzie pozwalającą na zaliczenie testu. Ponieważ kod HTML znajduje się w szablonie, możemy więc go modyfikować bez konieczności tworzenia jakichkolwiek dodatkowych testów jednostkowych. Chcemy dodać znacznik <h1>.

Plik: *lists/templates/home.html*:

```
<html>
    <head>
        <title>Listy rzeczy do zrobienia</title>
    </head>
    <body>
        <h1>Twoja lista rzeczy do zrobienia</h1>
    </body>
</html>
```

Zobaczmy teraz, czy wprowadzona zmiana zmieniła cokolwiek na plus w teście funkcjonalnym:

```
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_new_item"}' ; Stacktrace: [...]
```

OK...

Plik *lists/templates/home.html*:

```
[...]
    <h1>Twoja lista rzeczy do zrobienia</h1>
    <input id="id_new_item" />
</body>
[...]
```

A teraz?

```
AssertionError: '' != 'Wpisz rzecz do zrobienia'
```

Dodajemy więc tekst podpowiedzi wyświetlany w elemencie formularza.

Plik *lists/templates/home.html*:

```
<input id="id_new_item" placeholder="Wpisz rzecz do zrobienia" />
```

Otrzymujemy następujący wynik:

```
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_list_table"}' ; Stacktrace: [...]
```

Możemy więc przejść do umieszczenia tabeli na stronie. Na tym etapie tabela pozostanie pusta.

Plik *lists/templates/home.html*:

```
<input id="id_new_item" placeholder="Wpisz rzecz do zrobienia" />
<table id="id_list_table">
</table>
</body>
```

Jaki jest teraz wynik testu funkcjonalnego?

```
File "functional_tests.py", line 42, in
test_can_start_a_list_and_retrieve_it_later
    any(row.text == '1: Kupić pawie pióra' for row in rows)
AssertionError: False is not true
```

Jak możesz zobaczyć, komunikat pozostaje niejasny. Opierając się na podanym numerze wiersza, odkrywamy, że wspomniana wcześniej funkcja `any()` — a dokładniej `assertTrue()` — nie generuje odpowiedniego komunikatu dotyczącego niepowodzenia. Własny komunikat błędu można w większości metod typu `assertX` modułu `unittest` zdefiniować jako argument:

Plik `functional_tests.py`:

```
self.assertTrue(  
    any(row.text == '1: Kupić pawie pióra' for row in rows),  
    "Nowy element nie znajduje się w tabeli."  
)
```

Jeżeli ponownie wykonasz test funkcjonalny, wówczas powinieneś zobaczyć zdefiniowany wcześniej komunikat:

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

Aby móc rozwiązać problem i zaliczyć test, konieczne będzie faktyczne przetworzenie informacji wprowadzonych przez użytkownika w formularzu. To jednak jest tematem następnego rozdziału.

W tym momencie przekazujemy pliki do repozytorium:

```
$ git diff  
$ git commit -am"Kod HTML strony głównej jest teraz generowany na podstawie szablonu."
```

Dzięki przeprowadzeniu refaktoryzacji omawiany widok został przygotowany do wygenerowania na podstawie szablonu. Przystaliśmy testować stałe i jesteśmy teraz gotowi do rozpoczęcia przetwarzania danych wejściowych użytkownika.

Przypomnienie — proces TDD

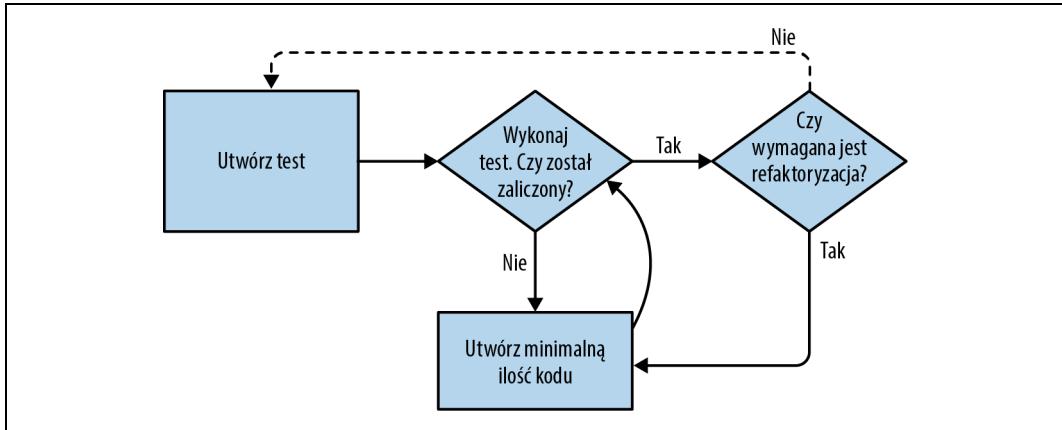
W ten sposób poznaleś i zobaczyłeś w praktyce wszystkie najważniejsze aspekty procesu TDD:

- testy funkcjonalne,
- testy jednostkowe,
- cykl test jednostkowy i tworzenie kodu,
- refaktoryzacja.

Warto więc poświęcić chwilę na małe przypomnienie, a być może nawet spojrzeć na pewne wykresy. Wybacz mi, ale lata spędzone jako konsultant do spraw zarządzania zepsuły mnie. Plusem obecnej sytuacji będzie to, że omówię rekurencję.

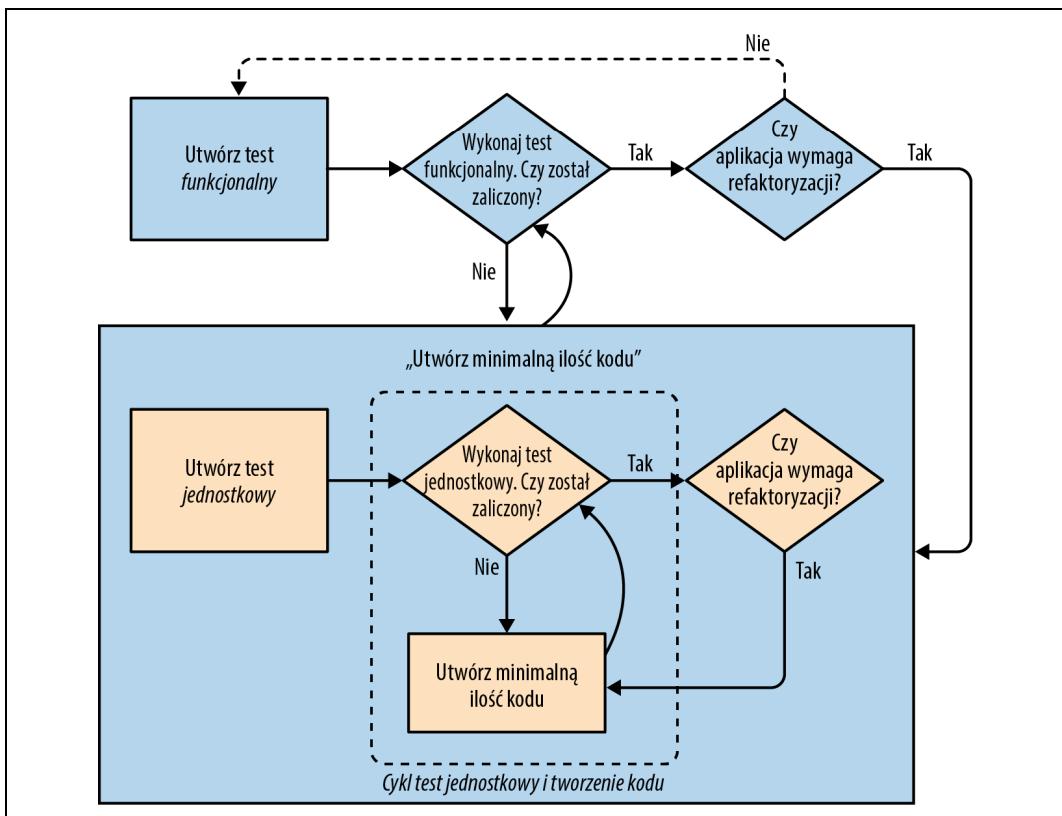
Jak więc wygląda ogólny proces TDD? Spójrz na rysunek 4.3.

Tworzymy test, który następnie wykonujemy i sprawdzamy, czy zakończył się niepowodzeniem. Kolejnym krokiem jest utworzenie minimalnej ilości kodu pozwalającej posunąć się nieco do przodu. Ponownie wykonujemy test i powtarzamy cały cykl aż do chwili, gdy test zostanie zaliczony. Następnie można opcjonalnie przeprowadzić refaktoryzację kodu i wykorzystując testy, upewnić się, że w jej trakcie nie nastąpiło zepsucie żadnej zaimplementowanej funkcjonalności.



Rysunek 4.3. Ogólny proces TDD

Jak to wszystko wygląda, gdy mamy przygotowane testy funkcjonalne *oraz* jednostkowe? Cóż, test funkcjonalny możesz potraktować jako wysokiego poziomu widok cyklu, w którym „utworzenie kodu” mającego na celu zaliczenie testu funkcjonalnego tak naprawdę oznacza użycie kolejnego, mniejszego cyklu TDD opartego na testach jednostkowych. Spójrz na rysunek 4.4.



Rysunek 4.4. Proces TDD obejmujący testy funkcjonalne i jednostkowe

Tworzymy test funkcjonalny i sprawdzamy, czy jego wykonanie kończy się niepowodzeniem. Następnie proces „utworzenia kodu” niezbędnego do zaliczenia testu staje się sam w sobie minicyklem TDD. Tworzymy więc jeden lub więcej testów jednostkowych, przeprowadzamy cykl test jednostkowy i tworzenie kodu aż do chwili zaliczenia testu. Wówczas możemy powrócić do testu funkcjonalnego i sprawdzić, czy udało nam się posunąć nieco do przodu. Znów tworzymy minimalną ilość kodu aplikacji, opierając się na kolejnych testach jednostkowych, itd.

Mogłbyś zapytać: co z refaktoryzacją w kontekście testów funkcjonalnych? Test funkcjonalny jest używany do sprawdzenia, czy funkcjonalność aplikacji została zachowana. Możemy jednak zmienić, a także dodać lub usunąć testy jednostkowe oraz wykorzystać cykl testu jednostkowego w celu faktycznej zmiany implementacji.

Testy funkcjonalne to ostateczny środek pozwalający na ocenę działania aplikacji. Z kolei testy jednostkowe to narzędzie pomagające w pracach nad aplikacją.

Taki sposób prowadzenia pracy jest czasami określany mianem „podwójnej pętli TDD”. Jedna z wybitnych recenzentek technicznych, Emily Bache, opublikowała post⁷ poświęcony wymienionemu zagadniению. Zachęcam Cię do jego lektury, ponieważ pozwala spojrzeć na problem z innej perspektywy.

W kolejnych rozdziałach dokładniej przeanalizujemy poszczególne etapy prowadzenia pracy w wymieniony powyżej sposób.

⁷ <http://coding-is-like-cooking.info/2013/04/outside-in-development-with-double-loop-tdd/>

Jak „sprawdzić” kod lub pominąć sprawdzanie (jeśli musisz)

Wszystkie przykładowe fragmenty kodu omawiane i prezentowane w książce znajdują się w *moim repozytorium*⁸ w serwisie GitHub. Jeżeli kiedykolwiek będziesz chciał porównać swój kod z moim, wystarczy zajrzeć do wymienionego repozytorium.

Każdy rozdział ma własną gałąź o nazwie utworzonej według konwencji chapter_XX:

- Rozdział 3. — https://github.com/hjwp/book-example/tree/chapter_03
- Rozdział 4. — https://github.com/hjwp/book-example/tree/chapter_04
- Rozdział 5. — https://github.com/hjwp/book-example/tree/chapter_05
- itd.

Musisz mieć świadomość, że każda gałąź zawiera cały kod przedstawiony w rozdziale, a więc jej stan odzwierciedla kod na *koncu* rozdziału.

Użycie systemu Git do sprawdzania postępu prac

Jeżeli chcialbyś poszerzyć swoje umiejętności w zakresie stosowania systemu kontroli wersji Git, wówczas moje repozytorium możesz wykorzystać jako *zdalne*:

```
git remote add harry https://github.com/hjwp/book-example.git  
git fetch harry
```

Następnie sprawdź różnice względem kodu na *koncu* rozdziału 4.:

```
git diff harry/chapter_04
```

Git potrafi obsługiwać wiele zdalnych repozytoriów. Powyższe rozwiązanie możesz zastosować, nawet jeśli korzystasz już z kodu pobieranego z GitHub lub Bitbucket.

Pamiętaj, że kolejność poszczególnych metod w klasie może być różna u mnie i u Ciebie. To niewątpliwie utrudni przeglądanie różnic.

Pobieranie archiwum ZIP zawierającego kod dla całego rozdziału

Jeżeli z jakiegokolwiek powodu będziesz chciał rozpoczęć pracę „zupełnie od początku” w danym rozdziale, pominąć etapy przejściowe⁹ lub po prostu nie czujesz się pewnie podczas pracy z systemem Git, wówczas opracowany przez mnie kod możesz pobrać w postaci archiwum ZIP z adresu URL stosującego przedstawiony wzorzec:

```
https://github.com/hjwp/book-example/archive/chapter\_05.zip  
https://github.com/hjwp/book-example/archive/chapter\_06.zip
```

Nie rób z siebie ofiary

Spróbuj nie podglądać odpowiedzi, dopóki naprawdę nie zapędzisz się w kozi róg. Jak wspomniałem na początku poprzedniego rozdziału, samodzielne usuwanie błędów niesie w sobie dużą wartość. W trakcie pracy nad rzeczywistymi projektami nie będziesz miał dostępu do „repozytorium Harry’ego”, aby porównać swój kod oraz znaleźć odpowiedzi na ewentualne pytania.

⁸ <https://github.com/hjwp/book-example/>

⁹ Nie zalecam pomijania etapów przejściowych. Rozdziały tej książki nie zostały napisane jako zupełnie oddzielne partie materiału, a każdy kolejny opiera się na materiale przedstawionym w poprzednim.

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 programistie 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 funkcyjonalnych.

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 żdaniami POST, znaczniki szablonów Django itd. Wymienione koncepcje będę mógł przedstawić *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.

Odpowiedź na pytanie: 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 nieoczekiwanyem niepowodzeniem, wówczas możemy podjąć wiele różnych działań, aby odszukać źródło błędu:

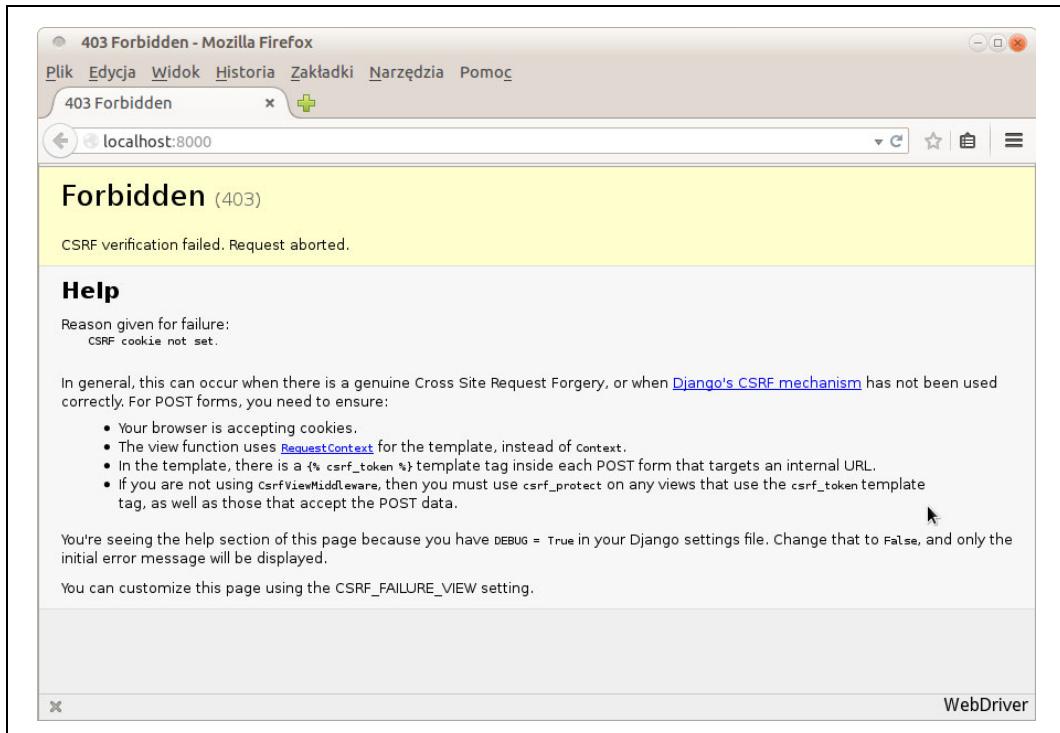
- Dodanie poleceń print w celu wyświetlania na przykład tekstu bieżącej strony.
- Usprawnienie komunikatu błędu, aby wyświetlał więcej informacji o bieżącym stanie.
- Ręczne odwiedzenie problematycznej witryny.
- Użycie wywołania time.sleep() w celu zrobienia przerwy podczas wykonywania testu.

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

```
# Po naciśnięciu klawisza Enter strona została uaktualniona i wyświetlona
# "I: 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 rozwiązań 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 takie 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 myśliców 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 oczekiwany 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*:

```
# "I: 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 zatrzymać się na *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 *innny* test. Udało nam się osiągnąć zaliczenie testu, nad którym pracowaliśmy. Natomiast wybrane testy jednostkowe spowodowały powstanie nieoczekiwanych konsekwencji w postaci regresji — uszkodziłyśmy 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 rozwiązywanie 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 kopij i wklej, mój przyjacielu) o sprawdzenie, czy drugi element listy został dodany do tabeli, to przekonamy się, że nasze rozwiązywanie 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 wyświetlzoną swoją listę rzeczy do zrobienia.
```

Zgodnie z oczekiwaniemi 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 reguła 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 zakłoci przebieg testu. Dlatego też wykorzystamy metodę pomocniczą — pamiętaj, że tylko metody o nazwach rozpoczętymi 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. Umieścmy 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 przyjęty" (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 # Przejrzanie 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. Srowadza 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 najbliższego z możliwych zapytań `.all()`, które pobiera wszystkie rekordy ze wskazanej tabeli. Wynikiem jest przypominający listę obiekt o nazwie `QuerySet`, z którego można wyodrębnić 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 doskonale 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. Zaczniemy od możliwości zimportowania 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 zakończyło 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 komplikacji. 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'

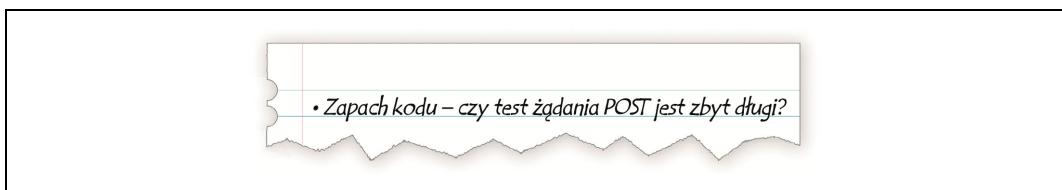
    response = home_page(request)

    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 oczekiwanej 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.

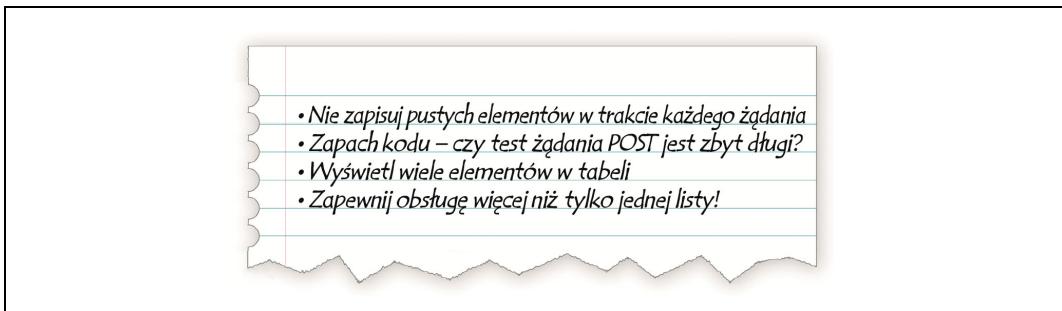
To oczywiście nie oznacza, że „w rzeczywistych projektach” zawsze powinniśmy ignorować tak rażące problemy. Kiedy problem zostanie wcześniej dostrzeżony, wtedy trzeba rozważyć, czy przerwać dotychczas wykonywane zadanie i zacząć raz jeszcze, czy jednak można odłożyć rozwiązanie problemu na później. Czasami 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

Zaczniemy 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 oczekiwany 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
    })
```

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

❶ 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 niesatyfakcjonujące. 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 przeglądarkę 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 wykrywanie 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'], '/')
```

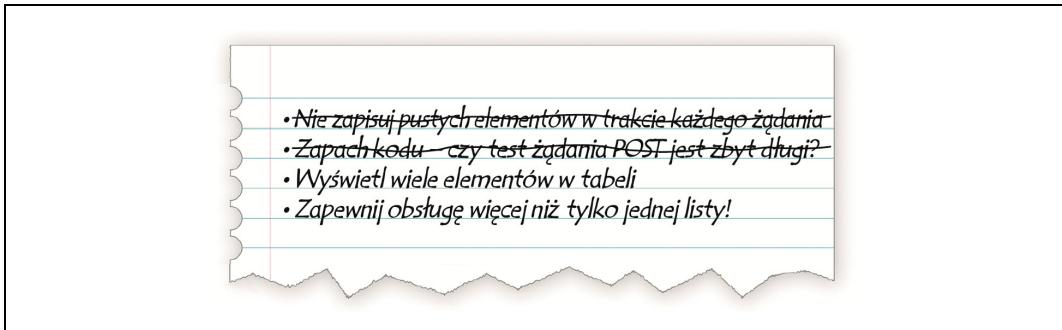
Teraz mamy zaliczonych sześć testów zamiast pięciu:

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 liczących 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 nadziej się 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('/')

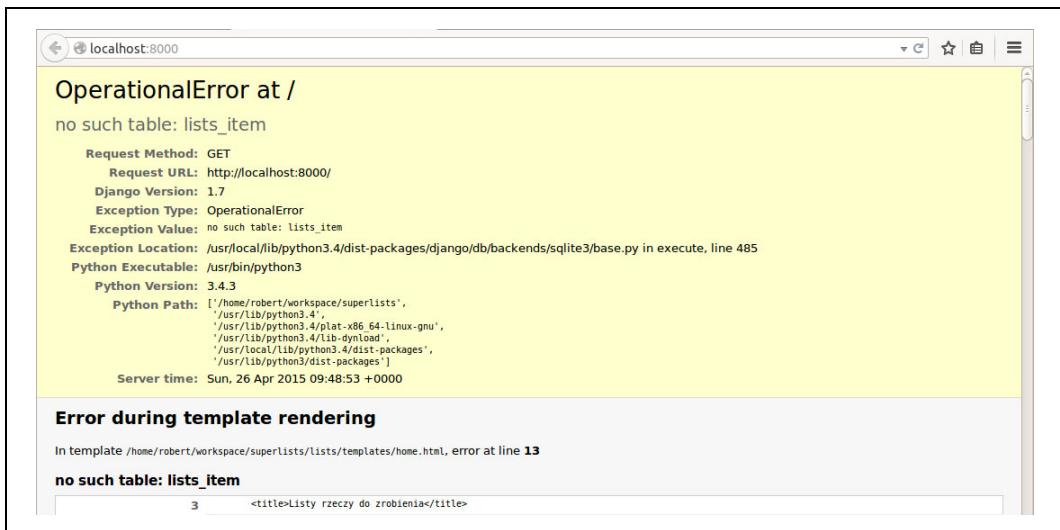
    items = Item.objects.all()
    return render(request, 'home.html', {'items': items})
```

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 użytkczy komunikat wyświetlony w trakcie procesu usuwania błędów

Utworzenie produkcyjnej bazy danych za pomocą polecenia migrate

Otrzymaliśmy kolejny użytkczy 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 otrzymało 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 wyświetcone 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 funkcjonalne⁹:

```
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 zauważyc 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 rozwiązywanie 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.

Rozpoczniemy od wydania polecień `git status` i `git diff`, a zobaczymy 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-rozdzialu-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 poczeganie 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.

Przygotowanie minimalnej działającej wersji witryny

W tym rozdziale zajmiemy się problemami odkrytymi na końcu poprzedniego rozdziału. Najpierw przystąpimy do rozwiązyania problemu związanego z usuwaniem danych wygenerowanych w trakcie wykonywania testu funkcjonalnego. W dalszej części rozdziału rozwiążemy ogólniejszy problem, jaki niewątpliwie stanowi przyjęte podejście pozwalające na obsługę tylko jednej globalnej listy rzeczy do zrobienia. Zademonstruję wówczas technikę TDD o znaczeniu krytycznym, czyli adaptację istniejącego kodu za pomocą przyrostowego procesu krok po kroku. Dzięki wspomnianemu procesowi przejdziesz od etapu działającego kodu do kolejnego etapu działającego kodu. Mam na myśli postępowanie w stylu Testing Goat, a nie kota refaktoryzacji.

Gwarancja izolacji testu w testach funkcjonalnych

Poprzedni rozdział zakończyliśmy z klasycznym problemem pojawiającym się w trakcie testowania — jak zapewnić *izolację* poszczególnych testów. Każde wykonanie testów funkcjonalnych pozostawia w bazie danych pewne dane, które mogą zakłócać wyniki kolejnych przeprowadzanych testów.

Kiedy wykonujemy testy *jednostkowe*, oferowany przez Django silnik testów automatycznie tworzy zupełnie nową bazę danych przeznaczoną dla testów (to zupełnie inna baza danych niż używana przez aplikację). Tę testową bazę danych można bezpiecznie zerować przed wykonaniem poszczególnych testów, a nawet pozbyć się jej na końcu testów. Jednak obecnie przygotowane testy funkcjonalne są wykonywane z użyciem „rzeczywistej” bazy danych o nazwie `db.sqlite3`.

Jedną z możliwości jest opracowanie własnego rozwiązania i dodanie do pliku `functional_test.py` pewnego kodu odpowiedzialnego za usuwanie niepotrzebnych danych testowych. Metody `setUp()` i `tearDown()` idealnie nadają się do tego typu zadań.

Począwszy od wydania Django 1.4, istnieje nowa klasa o nazwie `LiveServerTestCase` gotowa do wykonania wymienionych wcześniej zadań. Automatycznie utworzy testową bazę danych (podobnie jak ma to miejsce podczas wykonywania testu jednostkowego) i uruchomi serwer pozwalający na wykonanie testów funkcjonalnych. Wprawdzie omawiane narzędzie ma pewne ograniczenia, którymi zajmiemy się nieco później, ale na obecnym etapie prac jest niezwykle użyteczne. Zobaczmy więc, jak możemy je wykorzystać.

Klasa `LiveServerTestCase` jest przeznaczona do użycia przez silnik testów Django za pomocą `manage.py`. Począwszy od wydania Django 1.6, silnik testów wyszukuje wszystkie pliki o nazwach rozpoczynających się od `test`. W celu zachowania przejrzystości i porządku tworzymy katalog przeznaczony dla testów funkcjonalnych, aby całość bardziej przypominała aplikację. Django wymaga, aby ten katalog był prawidłowym modelem Pythona (wraz z plikiem `__init__.py`):

```
$ mkdir functional_tests  
$ touch functional_tests/__init__.py
```

Następnie przenosimy testy funkcjonalne z oddzielnego pliku o nazwie `functional_tests.py` do pliku `tests.py` aplikacji `functional_tests`. Wydajemy polecenie `git mv`; Git „zauważył” przeniesienie pliku:

```
$ git mv functional_tests.py functional_tests/tests.py  
$ git status # Polecamie pokazuje zmianę nazwy na functional_tests/tests.py and __init__.py.
```

Na tym etapie struktura katalogów powinna przedstawiać się następująco:

```
 .  
   db.sqlite3  
   functional_tests  
     __init__.py  
     tests.py  
   lists  
     admin.py  
     __init__.py  
     migrations  
       0001_initial.py  
       0002_item_text.py  
       __init__.py  
       __pycache__  
     models.py  
     __pycache__  
     templates  
       home.html  
     tests.py  
     views.py  
   manage.py  
   superlists  
     __init__.py  
     __pycache__  
     settings.py  
     urls.py  
     wsgi.py
```

Plik `functional_tests.py` już nie istnieje i został zastąpiony przez `functional_tests/tests.py`. Jeżeli teraz chcesz wykonać opracowane tutaj testy funkcjonalne, wówczas zamiast polecenia `python3 functional_tests.py` wydaj polecenie `python3 manage.py test functional_tests`.



Istnieje możliwość łączenia testów funkcjonalnych z testami aplikacji `lists`. Osobiście wolę je rozdzielać, ponieważ testy funkcjonalne zwykle mają różne zadania wykonywane w poszczególnych aplikacjach. Testy funkcjonalne są przeznaczone do sprawdzania aplikacji z punktu widzenia użytkownika. Użytkownicy Twojej aplikacji naprawdę nie są zainteresowani tym, że podzieliłeś kod na dwie aplikacje.

Przystępujemy do edycji pliku `functional_tests/tests.py` w celu zmiany klasy `NewVisitorTest`, aby wykorzystywała wspomnianą wcześniej klasę `LiveServerTestCase`.

Plik `functional_tests/tests.py` (ch06l001):

```
from django.test import LiveServerTestCase
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class NewVisitorTest(LiveServerTestCase):

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

Następnie¹ zamiast na stałe definiować przejście do portu 8000 w komputerze lokalnym, klasa `LiveServerTestCase` udostępnia atrybut o nazwie `live_server_url`.

Plik `functional_tests/tests.py` (ch06l002):

```
def test_can_start_a_list_and_retrieve_it_later(self):
    # Edyta dowiedziała się o nowej, wspaniałej aplikacji w postaci listy rzeczy do zrobienia.
    # Postanowiła więc przejść na stronę główną tej aplikacji.
    self.browser.get(self.live_server_url)
```

Z końca kodu można usunąć także wiersz `if __name__ == '__main__'`, ponieważ do wykonania testów funkcjonalnych będziemy używać silnika testów Django.

Na tym etapie jesteśmy gotowi do wykonania testów funkcjonalnych za pomocą silnika testów Django. Odbiera się to przez nakazanie wykonania testów naszej nowej aplikacji `functional_tests`:

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
F
=====
FAIL: test_can_start_a_list_and_retrieve_it_later
(functional_tests.tests.NewVisitorTest)

-----
Traceback (most recent call last):
  File "/workspace/superlists/functional_tests/tests.py", line 61, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Zakończenie testu!')
AssertionError: Zakończenie testu!

-----
Ran 1 test in 6.378s
FAILED (failures=1)
Destroying test database for alias 'default'...
```

Testy funkcjonalne są wykonywane aż do wywołania `self.fail()`, czyli podobnie jak przed refaktoryzacją. Zwróć uwagę, że po wykonaniu testów po raz drugi wyniki nie zawierają elementów listy dodanych w trakcie poprzedniego testu — dane testowe są z powodzeniem usuwane po zakończeniu testu. Sukces! Powinniśmy przekazać teraz pliki do repozytorium, ponieważ mamy wprowadzoną niepodzieloną zmianę:

¹ Czy nie jesteś w stanie przejść dalej, ponieważ zastanawiasz się, co oznaczają zapisy `ch06l0xx`, które pojawiają się w niektórych fragmentach kodu? To są odwołania do konkretnych operacji przekazanych kodu do repozytorium (https://github.com/hjwp/book-example/commits/chapter_06) zawierającego kod omawiany w książce i są powiązane ze stosowanymi przeze mnie testami poprawności książki. Można powiedzieć, że to testy dla testów przedstawionych w książce poświęcone testowaniu. Nawiąsem mówiąc, testy mają własne testy.

```
$ git status # Zmiana nazwy pliku functional_tests.py i jego modyfikacja, nowy plik __init__.py.  
$ git add functional_tests  
$ git diff --staged -M  
$ git commit # Komunikat, na przykład "Utworzenie aplikacji functional_tests i użycie klasy LiveServerTestCase."
```

Opcja `-M` polecenia `git diff` jest szczególnie użyteczna. Oznacza „wykryj operacje przeniesienia”, a więc „zauważ”, że `functional_tests.py` i `functional_tests/tests.py` to ten sam plik. Skutkiem będzie czytelniejsze wyświetlenie różnic (spróbuj wydać polecenie `git diff` bez omówionej opcji).

Wykonanie tylko testów jednostkowych

Jeżeli teraz wydasz polecenie `manage.py test`, to Django wykona testy zarówno funkcjonalne, jak i jednostkowe:

```
$ python3 manage.py test  
Creating test database for alias 'default'...  
.....F  
=====  
FAIL: test_can_start_a_list_and_retrieve_it_later  
[...]  
AssertionError: Zakończenie testu!  
  
-----  
Ran 8 tests in 3.132s  
  
FAILED (failures=1)  
Destroying test database for alias 'default'...
```

W celu wykonania tylko testów jednostkowych konieczne jest wskazanie, że chcemy przeprowadzić jedynie testy aplikacji `lists`:

```
$ python3 manage.py test lists  
Creating test database for alias 'default'...  
.....  
-----  
Ran 7 tests in 0.009s  
  
OK  
Destroying test database for alias 'default'...
```

Uaktualnienie dotyczące użytecznych poleceń

Wykonanie testów funkcjonalnych

`python3 manage.py test functional_tests`

Wykonanie testów jednostkowych

`python3 manage.py test lists`

Co masz zrobić, gdy w tekście znajdzie się polecenie „wykonaj testy”, a nie jesteś pewien, które testy mam na myśli? Spójrz ponownie na wykres przedstawiony na końcu rozdziału 4. i spróbuj ustalić, w którym miejscu się znajdujemy. Regułą jest, że testy funkcjonalne są wykonywane zwykle po zaliczeniu wszystkich testów jednostkowych. Jeżeli masz jakiekolwiek wątpliwości, warto wykonać oba rodzaje testów!

Teraz musimy się zastanowić, jak zaimplementować w aplikacji obsługę wielu list. Aktualnie testy funkcjonalne mają przedstawioną poniżej postać.

Plik `functional_tests/tests.py`:

```
# 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 wyświetlzoną swoją listę rzeczy do zrobienia.

# Usatysfakcjonowana kładzie się spać
```

Jednak chcemy rozbudować obecną postać, aby poszczególni użytkownicy aplikacji widzieli jedynie zdefiniowane przez siebie elementy listy. Ponadto będą otrzymywać własny adres URL, za pomocą którego zyskają możliwość wyświetlenia zapisanych wcześniej list. Zastanówmy się nad tym przez chwilę.

Stawiaj na małe projekty

Programowanie sterowane testami jest blisko związane z ruchem programowania zwinnego, które stanowi przeciwnieństwo stosowanej podczas tworzenia oprogramowania tradycyjnej praktyki BDUF (ang. *Big Design Up Front*). W przypadku BDUF mamy długi okres określania wymagań, a następnie równie długą fazę planowania oprogramowania na papierze. Filozofia metod zwinnych polega na przejściu do rozwiązywania problemów w praktyce, a nie w teorii, zwłaszcza po jak najszybszym udostępnieniu aplikacji prawdziwym użytkownikom. Zamiast długo planować, próbujemy jak najwcześniej przygotować *minimalną działającą wersję aplikacji*, a następnie powoli usprawniać projekt na podstawie informacji pochodzących od używających jej użytkowników.

Takie podejście nie oznacza jednak, że w ogóle należy przestać zastanawiać się nad projektem aplikacji. W poprzednim rozdziale zobaczyłeś, że poruszanie się do przodu może odbywać się nieudolnie, choć ostatecznie udaje się uzyskać *odpowiedni wynik*. Chwila zastanowienia nad projektem często może skrócić tę drogę. Zastanówmy się, jak powinna wyglądać minimalna działająca wersja aplikacji w postaci listy rzeczy do zrobienia oraz jaki zastosujemy projekt podczas jej tworzenia.

- Chcemy, aby każdy użytkownik mógł zachować własną listę rzeczy do zrobienia — na razie przynajmniej jedną.
- Lista może składać się z wielu elementów, których podstawowym atrybutem jest opisowy tekst.
- Listy powinny być przechowywane między kolejnymi odwiedzinami użytkownika. Na razie każdemu użytkownikowi możemy nadać unikatowy adres URL prowadzący do listy. Później można zastosować inne rozwiązanie, na przykład w postaci automatycznego rozpoznawania użytkownika i wyświetlania mu jego listy.

W celu przygotowania elementów oznaczonych jako „na razie” listę i jej elementy będziemy przechowywać w bazie danych. Każda lista będzie miała unikatowy adres URL, a poszczególne elementy będą stanowiły tekst powiązany z określoną listą.

YAGNI!

Kiedy zaczynasz zastanawiać się nad projektem, zatrzymanie się może być trudne. Do głowy napływają różnego rodzaju pomysły — lista może otrzymać nazwę lub tytuł, rozpoznawanie użytkownika ma odbywać się za pomocą nazwy użytkownika i hasła, do listy dodamy pola krótsze oraz dłuższe przeznaczone na notatki, wprowadzimy możliwość stosowania kolejności, sortowania itd. One stanowią jednak złamanie innej zasad programowania zwanego: YAGNI (ang. *you aint gonna need it*, czyli nie potrzebujesz tego). Programista oprogramowania czerpie radość z tworzenia i czasami trudno mu się oprzeć pokusie implementacji pewnej funkcji tylko dlatego, że pojawił się dany pomysł i ktoś kiedyś *może* jej potrzebować. Problem polega na tym, że najczęściej niezależnie od tego, jak świetny to jest pomysł, zaimplementowana funkcja *nie* jest używana. Zamiast tego mamy zbędny kod w aplikacji oraz większy poziom jej skomplikowania. YAGNI to mantra, której powinniśmy używać w celu opierania się hurraoptimistycznemu dążeniu do tworzenia funkcjonalności.

REST

Mamy opracowany pomysł na projekt struktury danych, czyli modelu we wzorcu MVC. Móglbyś w tym miejscu zapytać: co z widokiem i kontrolerem? Jak za pomocą przeglądarki internetowej użytkownik powinien używać listy i jej elementów?

REST (ang. *representational state transfer*) to podejście najczęściej stosowane podczas projektowania API sieciowego. Gdy projektuje się witrynę internetową przeznaczoną dla użytkowników, nie ma możliwości *ścisłego* stosowania się do reguł REST, ale one nadal mogą być źródłem inspiracji.

Podejście REST sugeruje przygotowanie struktury adresu URL dopasowanej do struktury danych, którą w omawianym przykładzie jest lista i jej elementy. Każda lista może mieć własny adres URL:

```
/lists/<identyfikator listy>/
```

W ten sposób zostanie spełnione wymaganie określone w teście funkcjonalnym. W celu wyświetlenia listy należy użyć żądania GET (w zwykłej przeglądarce internetowej do wyświetlenia strony).

Do utworzenia zupełnie nowej listy mamy specjalny adres URL akceptujący żądania POST:

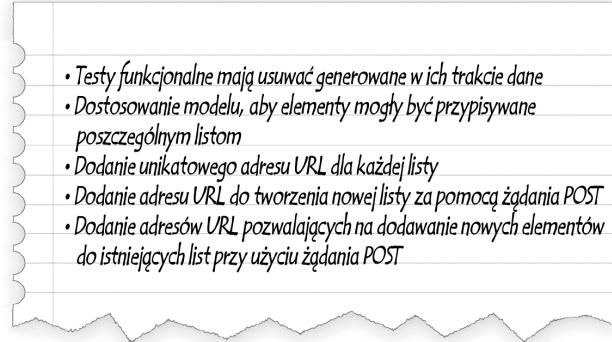
```
/lists/new
```

Natomiast w celu dodania nowego elementu do istniejącej listy używamy oddzielnego adresu URL, do którego mogą być wykonywane żądania POST:

```
/lists/<identyfikator listy>/add_item
```

(Warto przypomnieć ponownie, że nie próbuję tutaj ściśle stosować reguł podejścia REST, co wymagałoby użycia żądań PUT. Podejście REST wykorzystujemy w charakterze inspiracji).

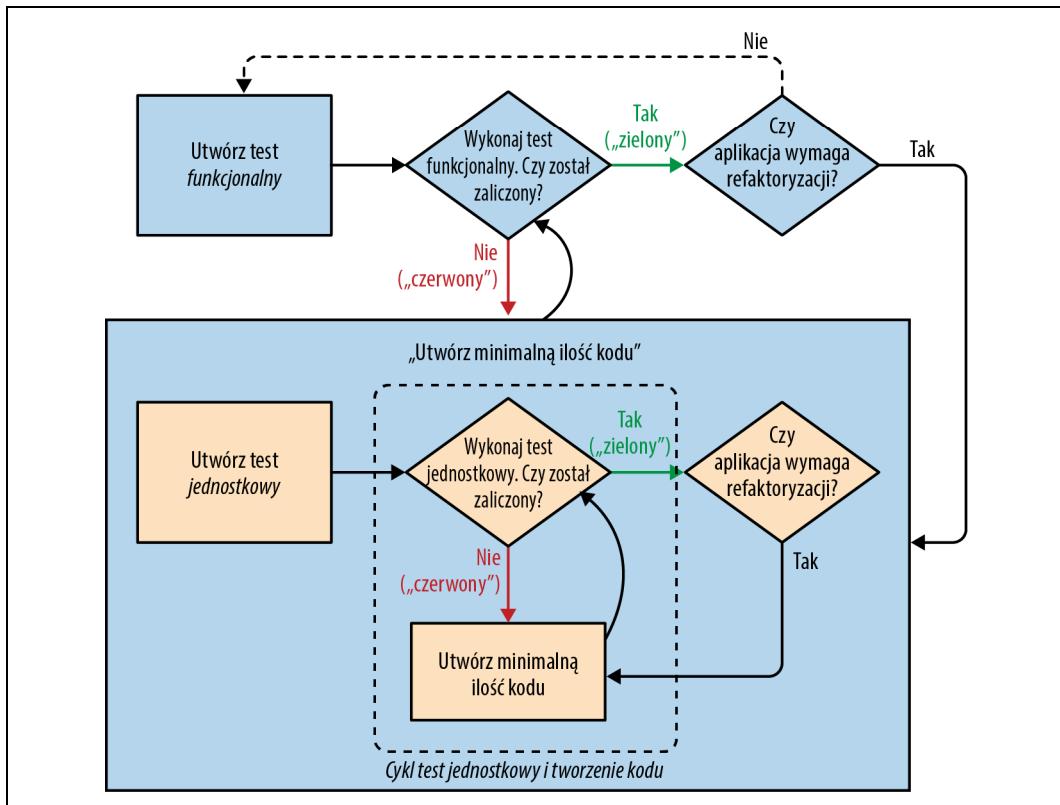
Podsumowując, nasza osobista lista rzeczy do zrobienia ma w tym rozdziale postać pokazaną na rysunku 6.1.



Rysunek 6.1. Nasza osobista lista rzeczy do zrobienia

Implementacja nowego projektu za pomocą TDD

W jaki sposób możemy użyć techniki TDD do implementacji nowego projektu? Spójrz raz jeszcze na pokazany na rysunku 6.2 wykres przedstawiający proces TDD.



Rysunek 6.2. Proces TDD wraz z testami funkcjonalnymi i jednostkowymi

Na najwyższym poziomie używamy następującego połączenia: dodanie nowej funkcjonalności (przez rozbudowę testów funkcjonalnych i utworzenie nowego kodu aplikacji) i refakto-ryzacja aplikacji — na przykład przez ponowne przygotowanie istniejącej implementacji, aby dostarczała użytkownikom tę samą funkcjonalność, ale przy użyciu aspektów nowego pro- jektu. Na poziomie testów jednostkowych dodajemy nowe testy lub modyfikujemy istniejące w celu przetestowania wprowadzonych zmian. Niezmodyfikowane testy jednostkowe wyko- rzystamy do upewnienia się, że w trakcie procesu zmiany projektu nie nastąpiło uszkodzenie żadnej funkcjonalności.

Przekształćmy teraz naszą papierową listę rzeczy do zrobienia na test funkcjonalny. Gdy tylko Edyta wprowadzi pierwszy element listy, ma nastąpić utworzenie nowej listy, umieszczenie na niej nowo wprowadzonego elementu oraz podanie Edycie adresu URL prowadzącego do jej listy. Odszukaj w kodzie polecenie `inputbox.send_keys('Kupić pawie pióra')`, a następnie wprowadź modyfikacje w kolejnym bloku kodu.

Plik `functional_tests/tests.py`:

```
inputbox.send_keys('Kupić pawie pióra')

# Po naciśnięciu klawisza Enter strona została aktualniona i wyświetla
# "1: Kupić pawie pióra" jako element listy rzeczy do zrobienia.
inputbox.send_keys(Keys.ENTER)
edith_list_url = self.browser.current_url
self.assertRegex(edith_list_url, '/lists/.+') #❶
self.check_for_row_in_list_table('1: Kupić pawie pióra')

# Na stronie nadal znajduje się pole tekstowe zachęcające do podania kolejnego zadania.
[...]
```

- ❶ `assertRegex()` to funkcja pomocnicza z modułu `unittest` odpowiedzialna za sprawdzenie, czy ciąg tekstowy został dopasowany do wyrażenia regularnego. W omawianym przykła- dzie wykorzystujemy ją do sprawdzenia, czy został zaimplementowany nasz nowy projekt oparty na podejściu REST.

Zmieńmy nieco końcówkę testu i przyjmijmy założenie, że pojawił się nowy użytkownik. Naszym celem jest sprawdzenie, czy po przejściu na stronę główną będzie mógł zobaczyć wprowadzone przez Edytę elementy listy oraz czy każdy użytkownik otrzymuje unikatowy adres URL dla tworzonych list.

Usuń wszystko od komentarzy tuż przed poleceniem `self.fail` (wspomniany komentarz ma treść „Edyta była ciekawa, czy witryna zapamięta jej listę...”) i zastąp nowym kodem oma- wianego testu funkcjonalnego.

Plik `functional_tests/tests.py`:

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

# Teraz nowy użytkownik Franek zaczyna korzystać z witryny.

## Używamy nowej sesji przeglądarki internetowej, aby mieć pewność, że żadne
## informacje dotyczące Edyty nie zostaną ujawnione, na przykład przez cookies. #❶
self.browser.quit()
self.browser = webdriver.Firefox()

# Franek odwiedza stronę główną.
```

```

# Nie znajduje żadnych śladów listy Edty.
self.browser.get(self.live_server_url)
page_text = self.browser.find_element_by_tag_name('body').text
self.assertNotIn('Kupić pawie pióra', page_text)
self.assertNotIn('zrobienia przymęty', page_text)

# Franek tworzy nową listę, wprowadzając nowy element.
# Jego lista jest mniej interesująca niż Edty...
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Kupić mleko')
inputbox.send_keys(Keys.ENTER)

# Franek otrzymuje unikatowy adres URL prowadzący do listy.
francis_list_url = self.browser.current_url
self.assertRegex(francis_list_url, '/lists/.+')
self.assertNotEqual(francis_list_url, edith_list_url)

# Ponownie nie ma żadnego śladu po liście Edty.
page_text = self.browser.find_element_by_tag_name('body').text
self.assertNotIn('Kupić pawie pióra', page_text)
self.assertIn('Kupić mleko', page_text)

# Usatysfakcjonowani, oboje kładą się spać.

```

- ❶ Stosuję konwencję dwóch znaków hash (##) w celu wskazania „metakomentarzy”, czyli komentarzy wskazujących na sposób i powód działania testu. Dzięki temu można je odróżnić od zwykłych komentarzy testu funkcjonalnego, wyjaśniających informacje otrzymane od użytkownika. Metakomentarze są przeznaczone dla nas. Bez nich w przyszłości mógłbyś się zastanawiać, dlaczego w kodzie ponownie uruchamiamy przeglądarkę internetową...

Pozostałe wprowadzone zmiany nie są trudne do zrozumienia. Zobaczmy, jaki będzie wynik wykonania teraz testu funkcjonalnego:

```
AssertionError: Regex didn't match: '/lists/.+' not found in
'http://localhost:8081/'
```

Wynik jest zgodny z oczekiwaniemi. Przekazujemy pliki do repozytorium, a następnie przystępujemy do tworzenia nowych modeli i widoków:

```
$ git commit -a
```



Gdy dzisiaj spróbowałem wykonać omawiane testy funkcjonalne, okazało się to niemożliwe. Konieczne było uaktualnienie narzędzia Selenium za pomocą polecenia pip3 install --upgrade selenium. Ze wstępu powinieneś pamiętać, jak ważne jest posiadanie zainstalowanej najnowszej wersji Selenium. Od ostatniej aktualizacji minęło zaledwie kilka miesięcy, a numer wersji zmienił się o sześć. Jeżeli w trakcie wykonywania testów zauważysz coś dziwnego, spróbuj uaktualnić Selenium.

Iteracja w kierunku nowego projektu

Będąc podekscytowany nowym projektem, czułem nieodpartą chęć przystąpienia natychmiast do pracy i rozpoczęcia modyfikacji pliku *models.py*. Niewątpliwym skutkiem takiego pośpiechu byłoby zepsucie połowy testów jednostkowych, a następnie zmiana prawie każdego wiersza kodu, wszysktko za jednym podejściem. To naturalne pragnienie, a programowanie sterowane testami jako dyscyplina stanowi nieustanną z nim walkę. Postępuj w stylu Testing Goat, a nie kota refaktoryzacji. Nie ma absolutnie żadnego powodu, aby nowy projekt

zaimplementować w jednym podejściu. Lepiej poruszać się małymi krokami, co pozwoli na przejście od stanu działającej aplikacji do kolejnego stanu działającej aplikacji, a projekt będzie powoli usprawniany na każdym etapie.

Nasza lista rzeczy do zrobienia ma cztery elementy. Test funkcjonalny wraz z komunikatem o braku dopasowania wyrażenia regularnego wskazuje, że element drugi — nadanie listom unikatowego adresu URL i identyfikatora — jest tym, którym powinniśmy się teraz zająć. Zróbcmy więc to i tylko to.

Adres URL pochodzi z przekierowania po wykonaniu żądania POST. W pliku *lists/tests.py* znajdź metodę `test_home_page_redirects_after_POST()` i zmień oczekiwane położenie przekierowania.

Plik *lists/tests.py*:

```
self.assertEqual(response.status_code, 302)
self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')
```

Czy to nie wydaje się nieco dziwne? Nie ulega wątpliwości, że */lists/the-only-list-in-the-world* nie jest adresem URL, który będzie stosowany w ostatecznym projekcie aplikacji. Jednak jednak jednorazowo mamy zmieniać tylko jedną rzecz. Skoro budowana aplikacja obsługuje wyłącznie jedną listę, to będzie tylko jeden adres URL, który ma sens. Mimo wszystko nadal posuwamy się do przodu, ponieważ adresy listy i strony głównej są teraz odmienne. To niewątpliwie jest krok na drodze do zastosowania podejścia w stylu REST. Gdy na dalszym etapie prac zaimplementujemy obsługę wielu list, zmiana będzie łatwa do wprowadzenia.



Innym sposobem myślenia jest zastosowanie techniki rozwiązywania problemu: nowy projekt adresów URL nie jest jeszcze zaimplementowany, a więc sprawdza się dla zera elementów. Ostatecznie to ma być rozwiązanie dla n elementów, ale rozwiązanie dla jednego elementu również jest dobrym krokiem na drodze prowadzącej do ostatecznego rozwiązania.

Po ponownym wykonaniu testów jednostkowych wynikiem będzie oczekiwane niepowodzenie:

```
$ python3 manage.py test lists
[...]
AssertionError: '/' != '/lists/the-only-list-in-the-world/'
```

Teraz można zmodyfikować widok `home_page` w pliku *lists/views.py*.

Plik *lists/views.py*:

```
def home_page(request):
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/lists/the-only-list-in-the-world/')

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

Oczywiście w ten sposób całkowicie zepsujemy testy funkcjonalne, ponieważ w budowanej witrynie nie ma jeszcze podanego adresu URL. Dlatego też po wykonaniu testów funkcjonalnych zakończą się one niepowodzeniem po próbie wysłania pierwszego elementu. Komunikat błędu poinformuje o braku tabeli dla listy. To prawda, przecież adres URL */lists/the-only-list-in-the-world* jeszcze nie istnieje.

```
    self.check_for_row_in_list_table('1: Kupić pawie pióra')
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_list_table"}' ; Stacktrace:
```

Przystępujemy więc do utworzenia specjalnego adresu URL przeznaczonego dla naszej jedynej listy rzeczy do zrobienia.

Testowanie widoków, szablonów i adresów URL za pomocą testu klienta Django

W poprzednich rozdziałach testy jednostkowe wykorzystaliśmy do sprawdzenia sposobu ustalania adresów URL, tego, czy funkcje widoku faktycznie je wywołują, oraz czy widoki prawidłowo generują szablony. Tak naprawdę Django oferuje niewielkie narzędzie, które może jednocześnie wykonać trzy wymienione wcześniej zadania. Teraz zobaczysz przykład jego użycia.

Najpierw chciałem pokazać, jak przygotować „własne rozwiązanie” w tym zakresie, ponieważ to pozwolioby na lepsze poznanie sposobu działania Django. Skoro wymienione techniki są przenośne, nie zawsze będziesz używać Django, choć niemal zawsze masz funkcje widoku, szablony i mapowanie adresów URL. Dlatego też powinieneś wiedzieć, jak to wszystko przetestować.

Nowa klasa testowa

Wykorzystamy teraz klienta Django. Otwórz plik *lists/tests.py* i dodaj nową klasę testu o nazwie *ListViewTest*. Następnie do nowej klasy skopiuj metodę *test_home_page_displays_all_list_items()* z klasy *HomePageTest*, zmień jej nazwę i nieco ją zmodyfikuj.

Plik *lists/tests.py* (ch09l009):

```
class ListViewTest(TestCase):

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

        response = self.client.get('/lists/the-only-list-in-the-world/') #❶

        self.assertContains(response, 'itemey 1') #❷
        self.assertContains(response, 'itemey 2') #❸
```

- ❶ Zamiast bezpośredniego wywołania funkcji widoku używamy klienta testów Django, który jest atrybutem `self.client` klasy Django o nazwie `TestCase`. Za pomocą `get()` nakazujemy mu pobranie testowanego adresu URL — w rzeczywistości to API jest niezwykle podobne do używanego przez Selenium.
- ❷❸ Zamiast nieco irytującej kombinacji `assertIn/response.content.decode()` Django oferuje metodę `assertContains()`, która potrafi pracować z odpowiedziami i sprawdzać ich zawartość.



Niektórzy naprawdę nie lubią klienta testów Django. Uważają, że w jego działaniu występuje zbyt wiele „magii”, a ponadto angażuje zbyt dużą część stosu w prawdziwym teście „jednostkowym” — efektem jest powstanie testów, które powinny raczej nazywać się testami zintegrowanymi. Powodem narzekan jest również stosunkowo wolne wykonywanie testów (pomiary są przeprowadzane w milisekundach). Do tego argumentu powrócimy jeszcze w dalszej części rozdziału. Teraz użyjemy klienta testów Django, ponieważ to niezwykle wygodne rozwiązanie!

Spróbujmy teraz wykonać test:

```
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
```

Nowy adres URL

Adres URL dla listy jeszcze nie istnieje. Musimy przeprowadzić modyfikację pliku *superlists/urls.py*.



Zwróć uwagę na ukośniki znajdujące się na końcu adresów URL, zarówno w testach, jak i w pliku *urls.py*. Wspomniane ukośniki bardzo często są źródłem błędów.

Plik *superlists/urls.py*:

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/the-only-list-in-the-world/$', 'lists.views.view_list',
        name='view_list'
    ),
    # url(r'^admin/', include(admin.site.urls)),
)
```

Po ponownym wykonaniu testów otrzymujemy następujące dane wyjściowe:

```
AttributeError: 'module' object has no attribute 'view_list'
[...]
django.core.exceptions.ViewDoesNotExist: Could not import
lists.views.view_list. View does not exist in module lists.views.
```

Nowa funkcja widoku

Jak sądzę, nie wymaga ona dodatkowych wyjaśnień. W pliku *lists/views.py* tworzymy prostą funkcję widoku.

Plik *lists/views.py*:

```
def view_list(request):
    pass
```

Po wykonaniu testów otrzymujemy następujący komunikat:

```
ValueError: The view lists.views.view_list didn't return an HttpResponseRedirect
object. It returned None instead.
```

Kopijujemy dwa ostatnie wiersze kodu z widoku *home_view* i sprawdzamy, czy to rozwiązuje problem.

Plik *lists/views.py*:

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

Po ponownym uruchomieniu testów powinniśmy zobaczyć, że zostały zaliczone:

```
Ran 8 tests in 0.016s
OK
```

W przypadku testów funkcjonalnych również posunęliśmy się nieco do przodu:

```
AssertionError: '2: Użyć pawich piór do zrobienia przynęty' not found in ['1: Buy
peacock feathers']
```

Zielony? Refaktoryzacja

Czas na drobne porządkи w kodzie.

W cyklu czerwony, zielony i refaktoryzacja docieramy do etapu zielony i warto sprawdzić, czy istnieje potrzeba przeprowadzenia refaktoryzacji. Mamy teraz dwa widoki, jeden przeznaczony dla strony głównej i drugi dla listy rzeczy do zrobienia. Aktualnie wykorzystujemy ten sam szablon i przekazujemy mu wszystkie elementy listy zapisane w bazie danych. Jeżeli dokładnie przyjrzyisz się metodom testu jednostkowego, zauważysz pewne fragmenty kodu, które prawdopodobnie chciałbyś zmodyfikować:

```
$ grep -E "class|def" lists/tests.py
class HomePageTest(TestCase):
    def test_root_url_resolves_to_home_page_view(self):
    def test_home_page_returns_correct_html(self):
    def test_home_page_displays_all_list_items(self):
    def test_home_page_can_save_a_POST_request(self):
    def test_home_page_redirects_after_POST(self):
    def test_home_page_only_saves_items_when_necessary(self):
class ListViewTest(TestCase):
    def test_displays_all_items(self):
class ItemModelTest(TestCase):
    def test_saving_and_retrieving_items(self):
```

Zdecydowanie można usunąć metodę `test_home_page_displays_all_list_items()`, ponieważ nie jest dłużej potrzebna. Po wydaniu polecenia `manage.py test lists` zobaczysz, że wykonanych zostało siedem testów zamiast ośmiu:

```
Ran 7 tests in 0.016s
OK
```

Nie ma konieczności, aby strona główna wyświetlała wszystkie elementy listy. Powinna wyświetlać jedynie pole tekstowe zachęcające użytkownika do utworzenia nowej listy.

Oddzielny szablon do wyświetlania list

Ponieważ widoki strony głównej i listy to zupełnie odmienne strony, powinny więc być oparte na oddzielnich szablonach HTML. W szablonie *home.html* może znajdować się pojęcie pole dla danych wejściowych, podczas gdy nowy szablon *list.html* będzie odpowiedzialny za wyświetlenie tabeli wraz z istniejącymi elementami listy.

Dodamy teraz nowy test pozwalający na sprawdzenie, czy używany jest inny szablon.

Plik *lists/tests.py*:

```
class ListViewTest(TestCase):

    def test_uses_list_template(self):
        response = self.client.get('/lists/the-only-list-in-the-world/')
        self.assertTemplateUsed(response, 'list.html')

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

Jedną z użyteczniejszych funkcji oferowanych przez klienta testów Django jest `assertTemplateUsed()`. Zobaczmy, jak przedstawia się wynik wykonania testów:

```
AssertionError: False is not true : Template 'list.html' was not a template
used to render the response. Actual template(s) used: home.html
```

Doskonale! Zmieniamy teraz widok.

Plik *lists/tests.py*:

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

Oczywiście wskazany szablon jeszcze nie istnieje. Po wykonaniu testów jednostkowych otrzymamy następujący komunikat:

```
django.template.base.TemplateDoesNotExist: list.html
```

Utwórzmy więc nowy plik *lists/templates/list.html*:

```
$ touch lists/templates/list.html
```

Poniżej pusty szablon powodujący wygenerowanie błędu — dobrze wiedzieć, że testy są gwarancją uzupełnienia szablonu:

```
AssertionError: False is not true : Couldn't find 'itemey 1' in response
```

Szablon przeznaczony do wyświetlenia listy wykorzysta dużą ilość kodu istniejącego obecnie w pliku *home.html*. Dlatego też pracę można zacząć od po prostu utworzenia kopii wymienionego pliku:

```
$ cp lists/templates/home.html lists/templates/list.html
```

W ten sposób docieramy do etapu zaliczenia testu (zielony). Możemy przystąpić do uporządkowania kodu (refaktoryzacja). Wcześniej stwierdziłem, że strona główna nie musi wyświetlać elementów listy, a jedynie pole tekstowe dla danych wejściowych. Z pliku *list/templates/home.html* można usunąć pewne wiersze kodu i nieco zmodyfikować znacznik `<h1>`, aby wyświetlał komunikat *Utwórz nową listę rzeczy do zrobienia*.

Plik *list/templates/home.html*:

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

Ponownie wykonujemy testy jednostkowe, sprawdzając, czy nie uszkodziliśmy jakiejś funkcjonalności. Wszystko jest dobrze.

Tak naprawdę w widoku `home_page` nie ma konieczności przekazania wszystkich elementów do szablonu `home.html`, co pozwala na jego uproszczenie.

Plik `lists/views.py`:

```
def home_page(request):
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/lists/the-only-list-in-the-world/')
    return render(request, 'home.html')
```

Ponownie wykonujemy testy jednostkowe i nadal są zaliczane. Przechodzimy więc do testów funkcjonalnych:

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

Nadal mamy problem z drugim elementem. Co się tutaj dzieje? Cóż, problem z nowym formularzem przeznaczonym do wpisywania elementów polega na brakującym atrybutem `action=`. Oznacza to, że domyślnie formularz jest przekazywany do tego samego adresu URL, pod którym został wygenerowany. Takie rozwiązanie sprawdza się w przypadku strony głównej, ponieważ to jedyna, która obecnie potrafi obsługiwać żądania POST. To jednak nie działa z funkcją `view_list()`, stąd zignorowanie żądań POST.

Odpowiednią zmianę trzeba wprowadzić w pliku `lists/templates/list.html`.

Plik `lists/templates/list.html` (ch06l019):

```
<form method="POST" action="/">
```

Teraz ponownie wykonujemy testy funkcjonalne:

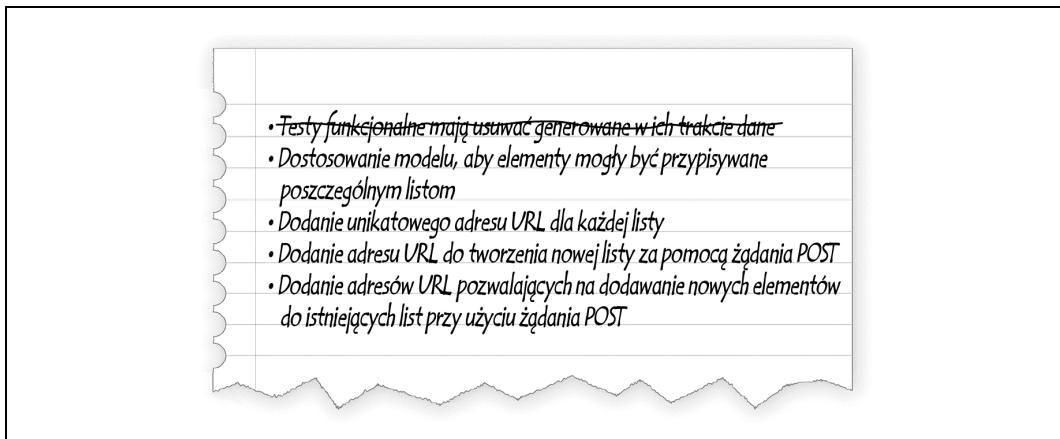
```
self.assertEqual(francis_list_url, edith_list_url)
AssertionError: 'http://localhost:8081/lists/the-only-list-in-the-world/' ==
'http://localhost:8081/lists/the-only-list-in-the-world/'
```

Hura! Otrzymaliśmy dokładnie taki sam wynik jak wcześniej, czyli refaktoryzacja zakończyła się powodzeniem. Udało się przygotować unikatowe adresy URL dla poszczególnych list. Był może nie masz wrażenia, że posunęliśmy się do przodu, ponieważ witryna działa w dokładnie taki sam sposób, jak działała w postaci przedstawionej na początku rozdziału. Jednak naprawdę osiągnęliśmy postęp. Zaczęliśmy marsz ku nowemu projektowi i zaimplementowaliśmy kilka kamieni milowych *bez jakiegokolwiek uszkodzenia dotychczasowej funkcjonalności aplikacji*. Warto więc przekazać pliki do repozytorium:

```
$ git status # Polecenie pokaż cztery zmienione pliki i jeden nowy — list.html.
$ git add lists/templates/list.html
$ git diff # Powinniśmy zobaczyć uproszczony plik home.html, a także
           # test przeniesiony do nowej klasy w pliku lists/tests.py, dodany nowy widok
           # w pliku views.py, uproszczony widok home_page oraz zmiany wprowadzone
           # w pliku urls.py.
$ git commit -a # Dodanie komunikatu podsumowującego powyższe zmiany, być może w postaci:
               # "Nowy adres URL, widok i szablon przeznaczone do wyświetlania list."
```

Kolejny adres URL i widok pozwalający na dodanie elementów listy

Co nam pozostało na naszej osobistej liście rzeczy do zrobienia (patrz rysunek 6.3)?



Rysunek 6.3. Nasza osobista lista rzeczy do zrobienia

W gruncie rzeczy poczyniliśmy postęp dotyczący trzeciego elementu listy, nawet jeśli na świecie miałaby istnieć tylko jedna lista rzeczy do zrobienia. Pozycja nr 2 wygląda nieco przerażająco. Czy możemy zrobić cokolwiek dotyczącego pozycji 4. lub 5.?

Przygotujmy nowy adres URL przeznaczony do dodania nowych elementów listy. To przy najmniej uprości widok strony głównej.

Klasa testowa dla operacji tworzenia nowej listy

Otwórz plik `lists/tests.py`, a następnie przenieś do niego metody `test_home_page_can_save_a_POST_request()` i `test_home_page_redirects_after_POST()` oraz zmień ich nazwy.

Plik `lists/tests.py` (ch06l021-1):

```
class NewListTest(TestCase):

    def test_saving_a_POST_request(self):
        request = HttpRequest()
        request.method = 'POST'
        [...]

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

Teraz użyjemy klienta testów Django.

Plik `lists/tests.py` (ch06l021-2):

```
class NewListTest(TestCase):

    def test_saving_a_POST_request(self):
        self.client.post(
            '/lists/new',
```

```

        data={'item_text': 'Nowy element listy'}
    )
self.assertEqual(Item.objects.count(), 1)
new_item = Item.objects.first()
self.assertEqual(new_item.text, 'Nowy element listy')

def test_redirects_after_POST(self):
    response = self.client.post(
        '/lists/new',
        data={'item_text': 'Nowy element listy'}
    )
    self.assertEqual(response.status_code, 302)
    self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')

```

To jest kolejne miejsce, gdzie trzeba zwrócić uwagę na ukośniki znajdujące się na końcu adresów URL. Zwróć uwagę na fakt, że mamy adres URL `/new` bez ukośnika na końcu. Stosuję konwencję, w której adresy URL bez ukośników na końcu są „akcjami” odpowiedzialnymi za modyfikację bazy danych.

Próbowiemy wykonać testy:

```

self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
[...]
self.assertEqual(response.status_code, 302)
AssertionError: 404 != 302

```

Pierwszy komunikat niepowodzenia wskazuje, że nowy element nie został zapisany w bazie danych. Natomiast drugi informuje, że zamiast przekierowania (kod stanu 302) wynikiem jest błąd na skutek nieistniejącego zasobu (kod stanu 404). Po prostu nie utworzyliśmy jeszcze adresu URL dla `/lists/new`, a więc wywołanie `client.post` otrzymuje odpowiedź wraz z kodem stanu 404.



Czy pamiętasz, jak w poprzednim rozdziale podzieliśmy test na dwa? Jeżeli mielibyśmy tylko jeden test sprawdzający zapis elementu i wykonanie przekierowania, wówczas otrzymalibyśmy niepowodzenie `0 != 1`. Tego rodzaju niejednoznaczny komunikat utrudniłby usunięcie błędu. Jak sądzisz, skąd o tym wiem?

Adres URL i widok przeznaczony do tworzenia nowej listy

Przystępujemy teraz do przygotowania nowego adresu URL.

Plik `superlists/urls.py`:

```

urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/the-only-list-in-the-world/$', 'lists.views.view_list',
        name='view_list'
    ),
    url(r'^lists/new$', 'lists.views.new_list', name='new_list'),
    # url(r'^admin/', include(admin.site.urls)),
)

```

Otrzymujemy komunikat `ViewDoesNotExist`, a więc musimy wprowadzić zmianę w pliku `lists/views.py`.

Plik `lists/views.py`:

```

def new_list(request):
    pass

```

Po powyższej zmianie otrzymujemy komunikat informujący, że widok `lists.views.new_list` nie zwraca obiektu `HttpResponse` (to brzmi całkiem znajomo). Możemy zwrócić zwykły obiekt `HttpResponse`, ale ponieważ wiemy, że potrzebujemy wykonać przekierowanie, więc odpowiedni wiersz kodu pożyczamy z widoku `home_page`.

Plik `lists/views.py`:

```
def new_list(request):
    return redirect('/lists/the-only-list-in-the-world/')
```

Po wykonaniu testów otrzymujemy komunikat:

```
self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
[...]
AssertionError: 'http://testserver/lists/the-only-list-in-the-world/' !=
'/lists/the-only-list-in-the-world/'
```

Zaczniemy od pierwszego niepowodzenia, ponieważ usunięcie jego przyczyny jest względnie proste. Z widoku `home_page` pożyczamy kolejny wiersz kodu:

```
def new_list(request):
    Item.objects.create(text=request.POST['item_text'])
    return redirect('/lists/the-only-list-in-the-world/')
```

W ten sposób docieramy do drugiego, nieoczekiwanej niepowodzenia:

```
self.assertEqual(response['location'],
 '/lists/the-only-list-in-the-world/')
AssertionError: 'http://testserver/lists/the-only-list-in-the-world/' !=
'/lists/the-only-list-in-the-world/'
```

Niepowodzenie powstaje, ponieważ klient testów Django zachowuje się nieco odmiennie niż zwykła funkcja widoku. Używa pełnego stosu Django, co oznacza dodanie nazwy domeny do względnego adresu URL. Wykorzystamy teraz inną funkcję pomocniczą klienta testów Django zamiast naszego dwustopniowego sprawdzenia, czy zostało wykonane przekierowanie.

Plik `lists/tests.py`:

```
def test_redirects_after_POST(self):
    response = self.client.post(
        '/lists/new',
        data={'item_text': 'Nowy element listy'}
    )
    self.assertRedirects(response, '/lists/the-only-list-in-the-world/')
```

Testy zostają zaliczone:

```
Ran 8 tests in 0.030s
OK
```

Usunięcie zbędnego kodu i dalsze testy

Wszystko wygląda dobrze. Ponieważ nasze widoki wykonują teraz większość zadań, które wcześniej realizował widok `home_page`, zyskujemy możliwość jego znacznego uproszczenia. Czy możemy na przykład usunąć cały blok `request.method == 'POST'`?

Plik `lists/views.py`:

```
def home_page(request):
    return render(request, 'home.html')
```

Tak!

OK

Przy okazji możemy usunąć zbędną już metodę o nazwie `test_home_page_only_saves_items_when_necessary()`.

Czy tak nie jest lepiej? Funkcje widoku są teraz znacznie prostsze. Ponownie wykonujemy testy, aby mieć pewność...

Ran 7 tests in 0.016s

OK

Wskazanie formularzy w nowym adresie URL

Na koniec trzeba jeszcze zmodyfikować oba formularze, aby używały nowego adresu URL. Zmianę trzeba wprowadzić zarówno w pliku `home.html`, jak i `lists.html`.

Pliki `lists/templates/home.html` i `lists/templates/list.html`:

```
<form method="POST" action="/lists/new">
```

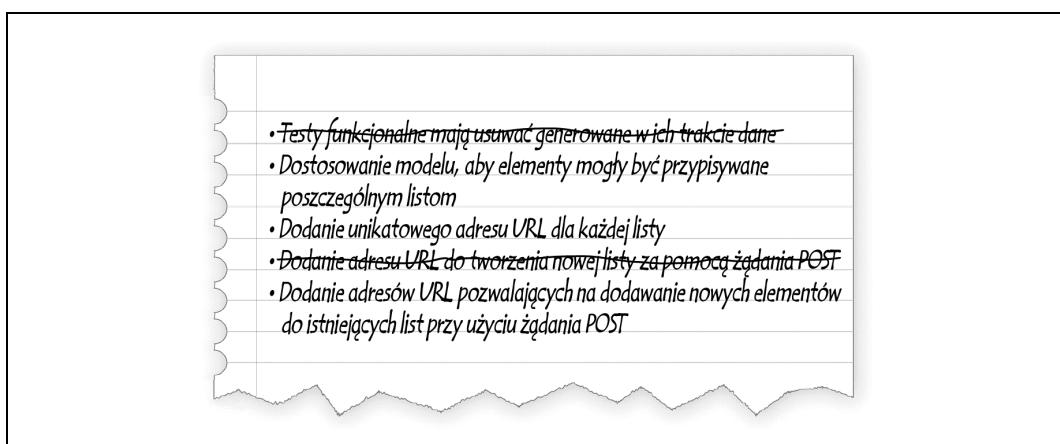
Ponownie wykonujemy testy funkcjonalne, aby upewnić się, że wszystko nadal działa lub przynajmniej działa nie gorzej niż wcześniej...

```
AssertionError: 'http://localhost:8081/lists/the-only-list-in-the-world/' ==  
'http://localhost:8081/lists/the-only-list-in-the-world/'
```

Tak, jesteśmy dokładnie w tym samym punkcie co wcześniej. To będzie elegancka, samodzielna operacja przekazania plików do repozytorium. Wprowadziliśmy wiele zmian w adresach URL, kod w pliku `views.py` jest znacznie bardziej elegancki i uporządkowany, a ponadto mamy pewność, że aplikacja działa jak wcześniej. Poradziliśmy sobie całkiem dobrze z tą refaktoryzacją!

```
$ git status # Mamy pięć zmienionych plików.  
$ git diff # Adresy URL dla dwóch formularzy, przeniesiony kod w widokach i testach, nowy adres URL.  
$ git commit -a
```

Teraz możemy skreślić punkt na naszej osobistej liście rzeczy do zrobienia (patrz rysunek 6.4).



Rysunek 6.4. Nasza osobista lista rzeczy do zrobienia

Dostosowanie modeli

Wystarczy porządków związanych z adresami URL. Pora chwycić byka za rogi i przystąpić do zmiany modeli. Zmodyfikujemy teraz test jednostkowy modelu. Dla odmiany konieczne do wprowadzenia modyfikacje przedstawię w postaci wyniku działania polecenia diff.

Plik *lists/tests.py*:

```
@@ -3,7 +3,7 @@ from django.http import HttpRequest
 from django.template.loader import render_to_string
 from django.test import TestCase

-from lists.models import Item
+from lists.models import Item, List
 from lists.views import home_page

 class HomePageTest(TestCase):
@@ -60,22 +60,32 @@ class ListViewTest(TestCase):

-class ItemModelTest(TestCase):
+class ListAndItemModelsTest(TestCase):

     def test_saving_and_retrieving_items(self):
+
         list_ = List()
+
         list_.save()

+
         first_item = Item()
         first_item.text = 'Absolutnie pierwszy element listy'
+
         first_item.list = list_
         first_item.save()

+
         second_item = Item()
         second_item.text = 'Element drugi'
+
         second_item.list = list_
         second_item.save()

+
         saved_list = List.objects.first()
+
         self.assertEqual(saved_list, list_)

+
         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(first_saved_item.list, list_)
         self.assertEqual(second_saved_item.text, 'Element drugi')
+
         self.assertEqual(second_saved_item.list, list_)
```

Tworzymy nowy obiekt `List`, a następnie przypisujemy mu poszczególne elementy za pomocą jego właściwości `list`. Upewniamy się o prawidłowym zapisaniu listy, a następnie sprawdzamy, czy dla tych dwóch elementów zapisano także informacje o związkach zachodzących między nimi. Zwróć uwagę na możliwość bezpośredniego porównywania obiektów listy (`saved_list()` i `list()`) — w tle operacja porównania jest przeprowadzana przez sprawdzenie, czy ich klucze podstawowe (atrribut `id`) są takie same.



W przykładzie używam zmiennej o nazwie `list_`, aby uniknąć nałożenia z wbudowaną funkcją Pythona `list()`. Wprawdzie to nieeleganckie rozwiązanie, ale inne rozważane przeze mnie opcje okazały się równie kiepskie lub jeszcze gorsze (`my_list, the_list, list1, listey...`).

Nadeszła pora na kolejny cykl test jednostkowy i tworzenie kodu.

W przypadku kilku pierwszych iteracji zamiast wyraźnie przedstawiać kod do wprowadzenia między kolejnymi operacjami wykonania testów, po prostu zaprezentuję oczekiwane komunikaty błędów wygenerowane po testach. Tobie pozostawiam decyzję, jaka jest minimalna ilość kodu, który należy zmodyfikować.

Pierwszy komunikat błędu powinien mieć postać:

```
ImportError: cannot import name 'List'
```

Po usunięciu problemu powinniśmy otrzymać następujący komunikat:

```
AttributeError: 'List' object has no attribute 'save'
```

Następny komunikat błędu to:

```
django.db.utils.OperationalError: no such table: lists_list
```

Wydajemy polecenie `makemigrations`:

```
$ python3 manage.py makemigrations
Migrations for 'lists':
  0003_list.py:
    - Create model List
```

Otrzymujemy wówczas następujące dane wyjściowe:

```
self.assertEqual(first_saved_item.list, list_)
AttributeError: 'Item' object has no attribute 'list'
```

Związek klucza zewnętrznego

W jaki sposób obiektowi `Item` można przekazać atrybut listy? Spróbujmy naiwnie zastosować podejście jak w przypadku atrybutu `text`.

Plik `lists/models.py`:

```
from django.db import models

class List(models.Model):
    pass

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

Jak zwykle testy wskazują na konieczność przeprowadzenia migracji:

```
$ python3 manage.py test lists
[...]
django.db.utils.OperationalError: table lists_item has no column named list

$ python3 manage.py makemigrations
Migrations for 'lists':
  0004_item_list.py:
    - Add field list to item
```

Zobaczmy, dokąd nas to zaprowadzi:

```
AssertionError: 'List object' != <List: List object>
```

To nie całkiem miejsce, do którego spodziewaliśmy się dotrzeć. Spójrz uważnie na obie strony operatora !=. Framework Django zapisał jedynie ciąg tekstowy przedstawiający obiekt List. W celu zachowania w obiekcie informacji o związkach konieczne jest użycie ForeignKey do poinformowania Django o związkach między dwoma klasami.

Plik *lists/models.py*:

```
from django.db import models

class List(models.Model):
    pass

class Item(models.Model):
    text = models.TextField(default='')
    list = models.ForeignKey(List, default=None)
```

Znów wymagana jest migracja. Ponieważ poprzednia była tylko rozwiązaniem tymczasowym, usuwamy ją i zastępujemy nową:

```
$ rm lists/migrations/0004_item_list.py
$ python3 manage.py makemigrations
Migrations for 'lists':
0004_item_list.py:
- Add field list to item
```



Usuwanie migracji jest niebezpieczne. Jeżeli usuniesz migrację zastosowaną w bazie danych, Django może pogubić się w bieżącym stanie bazy danych i nie potrafić przeprowadzić migracji w przyszłości. Dlatego też migrację powinieneś usuwać tylko wtedy, gdy masz absolutną pewność, że nie została ona nigdzie użyta. Dobrą regułą jest postrzeganie się od usuwania migracji po przekazaniu plików do repozytorium VCS.

Dostosowanie reszty świata do naszych nowych modeli

Powracamy do naszych testów. Co się teraz stanie?

```
$ python3 manage.py test lists
[...]
ERROR: test_displays_all_items (lists.tests.ListViewTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_saving_a_POST_request (lists.tests.NewListTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
Ran 7 tests in 0.021s
FAILED (errors=3)
```

Jejku!

Bez wątpienia mamy pewne dobre wiadomości. Wprawdzie trudno to dostrzec, ale testy modelu są zaliczone. Jednak trzy testy widoku w paskudny sposób kończą się niepowodzeniem.

Powodem jest wprowadzenie nowego związku między obiektami Items i Lists, co wymaga, aby każda lista miała element nadzędny, na istnienie którego nasze stare testy nie są przygotowane.

Jednak od tego mamy przecież testy. Wykorzystamy je teraz, aby ponownie testy były zaliczane. Najłatwiejszy sposób polega na użyciu klasy `ListViewTest` — po prostu utworzymy listę nadziedną dla dwóch elementów testowych.

Plik `lists/tests.py` (ch06l031):

```
class ListViewTest(TestCase):

    def test_displays_all_items(self):
        list_ = List.objects.create()
        Item.objects.create(text='itemey 1', list=list_)
        Item.objects.create(text='itemey 2', list=list_)
```

Tym samym liczba niezaliczonych testów spada do dwóch, oba próbują wykonać żądanego POST do nowego widoku `new_list`. Analizując stos wywołań za pomocą standardowej techniki przeglądania danych, począwszy od komunikatu błędu do wiersza kodu testowego, jesteśmy w stanie zlokalizować kod odpowiedzialny za niepowodzenie:

```
File "/workspace/superlists/lists/views.py", line 14, in new_list
    Item.objects.create(text=request.POST['item_text'])
```

Błąd pojawia się w trakcie próby utworzenia elementu pozbawionego listy nadziednej. W widoku wprowadzamy zmianę podobną do wcześniejszej.

Plik `lists/views.py`:

```
from lists.models import Item, List
[...]
def new_list(request):
    list_ = List.objects.create()
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect('/lists/the-only-list-in-the-world/')
```

Teraz testy ponownie są zaliczane:

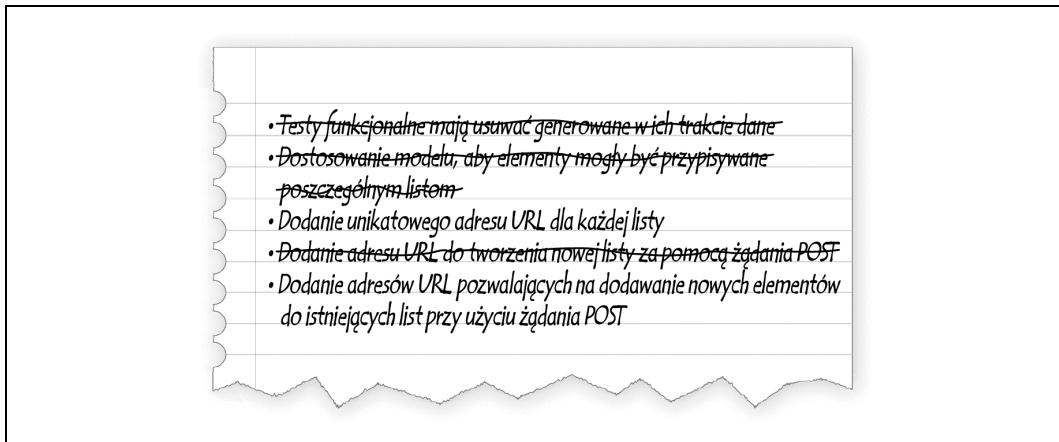
OK

Czy w tym momencie czujesz się zażenowany? *Brr! To z gruntu niewłaściwe, aby tworzyć nową listę dla każdego nowego elementu. Ponadto nadal wyświetlamy wszystkie elementy, jakby należały do tej samej listy!* Wiem i czuję dokładnie to samo. Podejście krok po kroku, w którym przechodzimy od jednego działającego kodu do innego, jest sprzeczne z intuicją. Zawsze mam wrażenie, że próbuję poprawić wszystko w jednym podejściu, zamiast przechodzić od jednego stanu częściowo przygotowanej aplikacji do kolejnego takiego stanu. Jednak pamiętaj o podejściu Testing Goat! Kiedy wspinasz się na szczyt, musisz dokładnie uważać, gdzie stawiasz stopy, i wykonywać pojedyncze kroki, za każdym razem upewniając się, że nie spadniesz w dół.

Należy się więc upewnić o prawidłowym działaniu aplikacji, ponownie wykonując testy funkcjonalne. Oczywiście znajdujemy się w tym samym miejscu, w którym byliśmy wcześniej. Nie uszkodziliśmy żadnej funkcjonalności i wprowadziliśmy zmianę w bazie danych. Możemy być dumni z dokonanego postępu. Na obecnym etapie warto przekazać pliki do repozytorium:

```
$ git status # Mamy trzy zmienione pliki plus dwie migracje.
$ git add lists
$ git diff --staged
$ git commit
```

Teraz możemy skreślić kolejny punkt na naszej osobistej liście rzeczy do zrobienia (patrz rysunek 6.5).



Rysunek 6.5. Nasza osobista lista rzeczy do zrobienia

Każda lista powinna mieć własny adres URL

Jakiej techniki powinniśmy użyć do identyfikacji poszczególnych list rzeczy do zrobienia? Teraz prawdopodobnie najłatwiejszą będzie użycie automatycznie wygenerowanej wartości kolumny id w bazie danych. Zmienimy więc klasę ListViewTest, aby dwa testy wskazywały nowe adresy URL.

Zmienimy także istniejącą metodę `test_displays_all_items()` i nadamy jej nową nazwę `test_displays_only_items_for_that_list()`. Zadaniem zmodyfikowanej metody będzie sprawdzanie jedynie tych elementów, które należą do wyświetlonej listy.

Plik `lists/tests.py` (ch06l033-1):

```
class ListViewTest(TestCase):

    def test_uses_list_template(self):
        list_ = List.objects.create()
        response = self.client.get('/lists/%d/' % (list_.id,))
        self.assertTemplateUsed(response, 'list.html')

    def test_displays_only_items_for_that_list(self):
        correct_list = List.objects.create()
        Item.objects.create(text='itemey 1', list=correct_list)
        Item.objects.create(text='itemey 2', list=correct_list)
        other_list = List.objects.create()
        Item.objects.create(text='Element pierwszy innej listy', list=other_list)
        Item.objects.create(text='Element drugi innej listy', list=other_list)

        response = self.client.get('/lists/%d/' % (correct_list.id,))

        self.assertContains(response, 'itemey 1')
        self.assertContains(response, 'itemey 2')
        self.assertNotContains(response, 'Element pierwszy innej listy')
        self.assertNotContains(response, 'Element drugi innej listy')
```



Jeżeli dokładnie nie rozumiesz sposobu działania zastępowania ciągów tekstowych w Pythonie lub funkcji `printf()` w języku C, to zapis `%d` może wydawać się nieco mylący. W książce *Dive Into Python*² znajdziesz dobre omówienie wspomnianego tematu, jeśli chcesz go szybko opanować. W dalszej części książki przedstawiono także alternatywną składnię zastępowania ciągów tekstowych.

Po wykonaniu testów jednostkowych otrzymamy oczekiwany błąd 404 oraz inny, pokrewny:

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
(expected 200)
[...]
FAIL: test_uses_list_template (lists.tests.ListViewTest)
AssertionError: No templates used to render the response
```

Przechwytywanie parametrów z adresów URL

Teraz się dowiesz, jak do widoku można przekazać parametry pobrane z adresów URL.

Plik `superlists/urls.py`:

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/(.+)$', 'lists.views.view_list', name='view_list'),
    url(r'^lists/new$', 'lists.views.new_list', name='new_list'),
    # url(r'^admin/', include(admin.site.urls)),
)
```

Wyrażenie regularne odpowiedzialne za dopasowanie adresu URL modyfikujemy w taki sposób, aby zawierało tak zwaną *grupę przechwytywania* `(.+)`, która powoduje dopasowanie dowolnych znaków, aż do następnego wystąpienia ukośnika `/`. Przechwytyony tekst zostanie przekazany widokowi jako argument.

Innymi słowy, po przejściu do adresu URL w postaci `/lists/1/` widok `view_list` otrzyma drugi argument (ciąg tekstowy "1") po standardowym `request`. Dlatego też przejście do adresu `/lists/foo` oznacza wywołanie `view_list(request, "foo")`.

Jednak nasz widok nie oczekuje jeszcze drugiego argumentu! To niewątpliwie prowadzi do problemów:

```
ERROR: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
ERROR: test_uses_list_template (lists.tests.ListViewTest)
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
[...]
TypeError: view_list() takes 1 positional argument but 2 were given
```

Łatwym rozwiązaniem może być użycie fikcyjnego parametru w pliku `views.py`.

Plik `lists/views.py`:

```
def view_list(request, list_id):
    [...]
```

W ten sposób pozostajemy jedynie przy oczekiwany niepowodzeniu:

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
AssertionError: 1 != 0 : Response should not contain 'Element pierwszy innej listy'
```

² <http://www.diveintopython.net/>

Określamy teraz, które elementy listy będą przekazywane do szablonu.

Plik *lists/views.py*:

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    items = Item.objects.filter(list=list_)
    return render(request, 'list.html', {'items': items})
```

Dostosowanie new_list do nowego świata

Na obecnym etapie błędy pojawiły się w innym teście:

```
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
ValueError: invalid literal for int() with base 10:
'the-only-list-in-the-world'
```

Spójrzmy na kod testu wskazanego jako źródło błędu.

Plik *lists/tests.py*:

```
class NewListTest(TestCase):
    [...]
    def test_redirects_after_POST(self):
        response = self.client.post(
            '/lists/new',
            data={'item_text': 'Nowy element listy'}
        )
        self.assertRedirects(response, '/lists/the-only-list-in-the-world/')
```

Wygląda na to, że powyższy test nie został dostosowany do nowego świata elementów Lists i Items. Wynikiem testu powinien być komunikat informujący, że widok powoduje przekierowanie do adresu URL nowo utworzonej listy.

Plik *lists/tests.py* (ch06l036-1):

```
def test_redirects_after_POST(self):
    response = self.client.post(
        '/lists/new',
        data={'item_text': 'Nowy element listy'}
    )
    new_list = List.objects.first()
    self.assertRedirects(response, '/lists/%d/' % (new_list.id,))
```

Nadal otrzymujemy komunikat o *nieprawidłowym literale*. Musimy raz jeszcze spojrzeć na widok i zmienić jego kod w taki sposób, aby przekierowanie odbywało się we właściwym kierunku.

Plik *lists/views.py* (ch06l036-2):

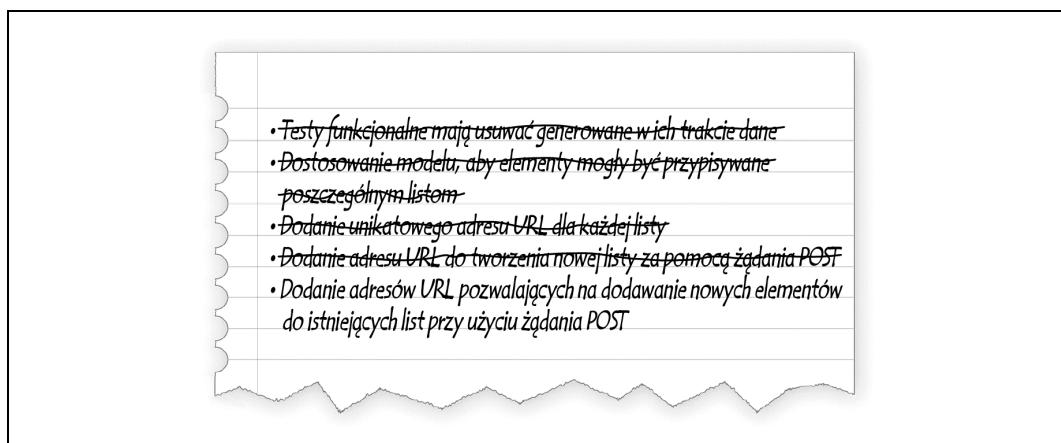
```
def new_list(request):
    list_ = List.objects.create()
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect('/lists/%d/' % (list_.id,))
```

Znów mamy zaliczone testy jednostkowe. Jak wygląda sytuacja z testami funkcjonalnymi? Jesteśmy już prawie u celu.

```
AssertionError: '2: Użyc pawich piór do zrobienia przynęty' not found in ['1: Use peacock feathers to make a fly']
```

Komunikat wygenerowany przez testy funkcjonalne ostrzega o regresji w aplikacji. Ponieważ nowa lista jest tworzona dla każdego żądania POST, nastąpiło uszkodzenie funkcjonalności i dodanie wielu elementów do listy. Dokładnie do tego celu są przeznaczone testy funkcjonalne!

Ponadto wyświetlony komunikat elegancko pasuje do ostatniego elementu na naszej osobistej liście rzeczy do zrobienia (patrz rysunek 6.6).



Rysunek 6.6. Nasza osobista lista rzeczy do zrobienia

Jeszcze jeden widok pozwalający na dodanie elementu do istniejącej listy



Obsługa operacji dodania nowego elementu do istniejącej listy wymaga adresu URL oraz widoku (`/lists/<identyfikator_listy>/add_item`). Jesteśmy na dobrej drodze i szybko wprowadzamy odpowiednie zmiany.

Plik `lists/tests.py`:

```
class NewItemTest(TestCase):
    def test_can_save_a_POST_request_to_an_existing_list(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        self.client.post(
            '/lists/%d/add_item' % (correct_list.id,),
            data={'item_text': 'Nowy element dla istniejącej listy'}
        )

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

    def test_redirects_to_list_view(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        response = self.client.post(
            '/lists/%d/add_item' % (correct_list.id,),
            data={'item_text': 'Nowy element dla istniejącej listy'}
        )
        self.assertRedirects(response, '/lists/%d/' % (correct_list.id,))
```

Otrzymujemy następujący komunikat:

```
AssertionError: 0 != 1
[...]
AssertionError: 301 != 302 : Response didn't redirect as expected: Response
code was 301 (expected 302)
```

Uwaga na żarłoczne wyrażenia regularne!

To jest nieco dziwne. Nie podaliśmy jeszcze adresu URL dla `/lists/a/add_item`, a więc oczekiwane niepowodzenie powinno mieć postać `404 != 302`. Dlaczego więc otrzymaliśmy kod stanu `301`?

To wielka zagadka, ponieważ w adresie URL użyliśmy niezwykle „żarłocznego” wyrażenia regularnego:

```
url(r'^lists/(.+)$', 'lists.views.view_list', name='view_list'),
```

Django posiada pewien wewnętrzny kod przeznaczony do wykonania trwałego przekierowania (`301`), gdy pojawi się żądanie adresu URL *niemal* prawidłowego, ale bez ukośnika na końcu. W omawianym przykładzie adres URL w postaci `/lists/1/add_item/` będzie dopasowany przez wyrażenie `lists/(.+)`, natomiast `(.+)` przechwytuje jedynie `1/add_item`. Dlatego Django „użytecznie” odgaduje, że żąдаliśmy adresu URL z ukośnikiem na końcu.

Rozwiązańiem może być modyfikacja wzorca URL, aby dopasowane zostały jedynie liczby. W tym celu należy użyć wyrażenia regularnego `\d`.

Plik `superlists/urls.py`:

```
url(r'^lists/(\d+)$', 'lists.views.view_list', name='view_list'),
```

Wprowadzona zmiana powoduje, że otrzymujemy następujący komunikat błędu:

```
AssertionError: 0 != 1
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

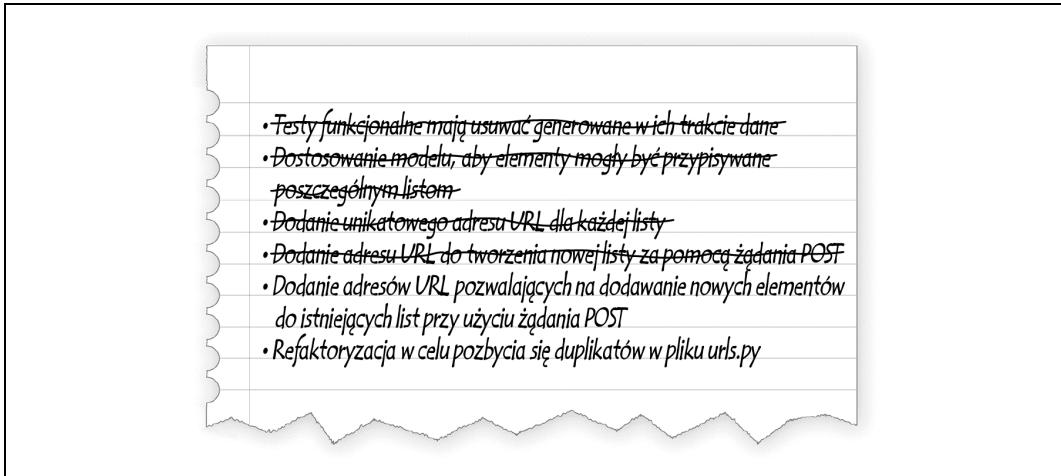
Ostatni nowy adres URL

Skoro otrzymaliśmy już oczekiwany błąd o kodzie stanu `404`, możemy przystąpić do zdefiniowania nowego adresu URL przeznaczonego do dodawania nowych elementów do istniejących list.

Plik `superlists/urls.py`:

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/(\d+)$', 'lists.views.view_list', name='view_list'),
    url(r'^lists/(\d+)/add_item$', 'lists.views.add_item', name='add_item'),
    url(r'^lists/new$', 'lists.views.new_list', name='new_list'),
    # url(r'^admin/', include(admin.site.urls)),
)
```

Powyżej mamy trzy podobne do siebie adresy URL. Warto więc umieścić nowy punkt na naszej osobistej liście rzeczy do zrobienia (patrz rysunek 6.7), ponieważ wspomniane adresy URL są dobrymi kandydatami do refaktoryzacji.



Rysunek 6.7. Nasza osobista lista rzeczy do zrobienia

Powracamy do testów i otrzymujemy następujący komunikat:

```
django.core.exceptions.ViewDoesNotExist: Could not import lists.views.add_item.  
View does not exist in module lists.views.
```

Ostatni nowy widok

Spróbujmy.

Plik *lists/views.py*:

```
def add_item(request):  
    pass
```

Aha.

Plik *lists/views.py*:

```
TypeError: add_item() takes 1 positional argument but 2 were given  
  
def add_item(request, list_id):  
    pass
```

Następnie otrzymujemy komunikat:

```
ValueError: The view lists.views.add_item didn't return an HttpResponseRedirect object.  
It returned None instead.
```

Kopiujemy wywołanie `redirect()` z widoku `new_list` oraz wywołanie `List.objects.get` z widoku `view_list`.

Plik *lists/views.py*:

```
def add_item(request, list_id):  
    list_ = List.objects.get(id=list_id)  
    return redirect('/lists/%d/' % (list_.id,))
```

To nas doprowadza do komunikatu:

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

Na końcu zapisujemy nowy element listy.

Plik *lists/views.py*:

```
def add_item(request, list_id):
    list_ = List.objects.get(id=list_id)
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect('/lists/%d/' % (list_.id,))
```

W ten sposób wszystkie testy znów są zaliczone.

```
Ran 9 tests in 0.050s
```

```
OK
```

Jak można użyć adresu URL w formularzu?

Przygotowanego adresu URL trzeba użyć w szablonie *list.html*. Otwórz wymieniony plik i zmodyfikuj znacznik `<form>`.

Plik *lists/templates/list.html*:

```
<form method="POST" action="co należy tutaj wpisać?">
```

Ach tak. W celu uzyskania adresu URL pozwalającego na dodanie elementu do bieżącej listy szablon musi „wiedzieć”, która lista jest wyświetlana oraz jakie elementy się na niej znajdują. Musimy mieć możliwość wykonania operacji podobnej do przedstawionej poniżej.

Plik *lists/templates/list.html*:

```
<form method="POST" action="/lists/{{ list.id }}/add_item">
```

Aby takie rozwiązanie zadziałało, widok musi przekazać listę szablonowi. W klasie *ListViewTest* tworzymy nowy test jednostkowy.

Plik *lists/tests.py* (ch06l041):

```
def test_passes_correct_list_to_template(self):
    other_list = List.objects.create()
    correct_list = List.objects.create()
    response = self.client.get('/lists/%d/' % (correct_list.id,))
    self.assertEqual(response.context['list'], correct_list)
```

Element `response.context` przedstawia kontekst przekazywany funkcji generującej — klient testów Django umieszcza kontekst w obiekcie `response`, co ułatwia testy. Wynikiem jest otrzymanie następującego komunikatu:

```
KeyError: 'list'
```

ponieważ nie przekazaliśmy egzemplarza `list` szablonowi. Zyskujemy więc okazję do przeprowadzenia pewnego uproszczenia.

Plik *lists/views.py*:

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    return render(request, 'list.html', {'list': list_})
```

Zmiana spowoduje, że testy nie zostaną zaliczone, ponieważ szablon oczekuje obiektu `items`:

```
AssertionError: False is not true : Couldn't find 'itemem 1' in response
```

Problem możemy jednak usunąć w pliku *list.html*, a także dostosowując atrybut `action` znacznika `<form>`.

Plik `lists/templates/list.html` (ch06l043):

```
<form method="POST" action="/lists/{{ list.id }}/add_item">  
[...]  
    {%- for item in list.item_set.all %}  
        <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>  
    {%- endfor %}
```

Wywołanie `item_set` nosi nazwę „odwrotnego wyszukiwania” — to jest jedna z niezwykle użytecznych cech mechanizmu ORM w Django i pozwala na wyszukiwanie w różnych tabelach elementów powiązanych obiektów.

Na obecnym etapie wszystkie testy jednostkowe powinny zostać zaliczone:

```
Ran 10 tests in 0.060s
```

```
OK
```

A co z testami funkcjonalnymi:

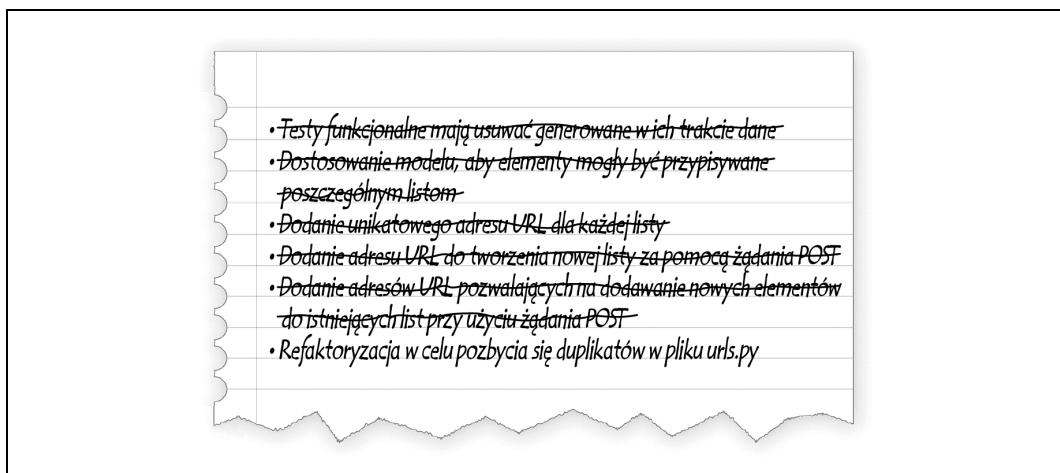
```
$ python3 manage.py test functional_tests  
Creating test database for alias 'default'...
```

```
-----  
Ran 1 test in 5.824s
```

```
OK
```

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

Tak! Rzucamy jeszcze okiem na naszą osobistą listę rzeczy do zrobienia (patrz rysunek 6.8).



Rysunek 6.8. Nasza osobista lista rzeczy do zrobienia

Wprawdzie to jest irytujące, ale w podejściu opartym na Testing Goat pedantyzm odgrywa ważną rolę i dlatego musimy zrobić jeszcze jedno.

Zanim przystąpimy do pracy, warto przekazać pliki do repozytorium. Zanim przystąpisz do refaktoryzacji, zawsze upewnij się o przekazaniu do repozytorium działającej wersji aplikacji.

```
$ git diff  
$ git commit -am "Nowy adres URL + widok pozwalający na dodawanie elementów do istniejących list.  
→ Testy funkcjonalne są zaliczone :-)"
```

Ostatnia refaktoryzacja za pomocą polecenia include

Plik *superlists/urls.py* jest przeznaczony do obsługi adresów URL mających zastosowanie w całej witrynie. W przypadku adresów URL stosowanych jedynie w aplikacji *lists* Django zachęca do użycia pliku *lists/urls.py*, aby aplikacja była bardziej samodzielna. Najprostsze rozwiązywanie polega na utworzeniu kopii istniejącego pliku *urls.py*:

```
$ cp superlists/urls.py lists/
```

Następnie trzy wiersze w pliku *superlists/urls.py* zastępujemy poleceniem `include`. Zauważ, że polecenie `include` może pobierać prefiks w postaci wyrażenia regularnego dla adresu URL i będzie ono zastosowane we wszystkich dodłączanych adresach URL. (To jest odpowiednie miejsce na pozbicie się powieleń, co pomoże w zapewnieniu lepszej struktury kodu).

Plik *superlists/urls.py*:

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/', include('lists.urls')),
    # url(r'^admin/', include(admin.site.urls)),
)
```

Zawartość pliku *lists/urls.py* można skrócić do jedynie zawierającej dalszą część trzech wspomnianych adresów URL. Nie trzeba w nim umieszczać zawartości pochodzącej z nadzawanego pliku *urls.py*.

Plik *lists/urls.py* (ch06l045):

```
from django.conf.urls import patterns, url

urlpatterns = patterns('',
    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^(\d+)/add_item$', 'lists.views.add_item', name='add_item'),
    url(r'^new$', 'lists.views.new_list', name='new_list'),
)
```

Ponownie wykonujemy testy jednostkowe i sprawdzamy, czy wszystko działa. Kiedy to zrobilem, nie mogłem wręcz uwierzyć, że wszystko udało się zrobić prawidłowo za pierwszym razem. Z reguły warto zachować odrobinę sceptyczmu wobec własnych możliwości, więc celowo zmieniłem jeden z adresów URL, aby się przekonać, czy to spowoduje niezaliczenie testu. Jesteśmy zabezpieczeni na wiele różnych sposobów.

Wypróbuj to sam. Pamiętaj, aby później wycofać wspomnianą zmianę, ponownie upewnić się o zaliczeniu wszystkich testów, a następnie przekazać pliki do repozytorium:

```
$ git status
$ git add lists/urls.py
$ git add superlists/urls.py
$ git diff --staged
$ git commit
```

Uf! Ten rozdział można nazwać prawdziwym maratonem. Jednak omówiłem w nim wiele ważnych tematów, począwszy od izolacji testu, aż po rozważania dotyczące projektu. Wspomniałem także o pewnych regułach, takich jak „YAGNI” oraz „do trzech razy sztuka, a później refaktoryzacja”. Co najważniejsze, zobaczyłeś krok po kroku, jak można zaadaptować istniejącą witrynę, przechodząc od stanu działającej aplikacji do innego stanu działającej aplikacji na drodze prowadzącej do nowego projektu.

Mogę stwierdzić, że jesteśmy już bardzo blisko ukończenia tej witryny internetowej, a pierwsza wersja beta już wkrótce zostanie udostępniona użytkownikom. Prawdopodobnie wymaga jedynie niewielkiego upiększenia... Zobaczmy, co trzeba będzie jeszcze zrobić, aby ją wdrożyć w kilku kolejnych rozdziałach.

Użyteczne koncepcje TDD i reguły

Izolacja testu i stan globalny

Poszczególne testy nie powinny na siebie wzajemnie wpływać. Oznacza to konieczność wyzerowania stanu na końcu każdego testu. Silnik testów Django pomaga w tym, tworząc testową bazę danych, której zawartość jest zerowana między poszczególnymi testami — patrz również rozdział 19.

Przejście od stanu działającej aplikacji do kolejnego stanu działającej aplikacji (inaczej podejście Testing Goat kontra kot refaktoryzacji)

Naszym naturalnym pragnieniem jest zagłębianie się w pracę i poprawienie wszystkiego za jednym razem... Jeżeli jednak nie zachowamy ostrożności, skończymy jak kot refaktoryzacji, w sytuacji, gdy wprowadzono wiele zmian w kodzie, a nic nie działa, jak powinno. Podejście Testing Goat zachęca do wykonywania pojedynczych kroków i przechodzenia od jednego stanu działającej aplikacji do kolejnego stanu działającej aplikacji.

YAGNI

To skrót oznaczający „nie będziesz tego potrzebować”. Unikaj pokusy tworzenia kodu, o którym sądzisz, że może być użyteczny, ponieważ masz takie odczucie w danym momencie. Istnieje znaczne prawdopodobieństwo, że w ogóle nie wykorzystasz danego kodu lub błędnie określasz przyszłe wymagania. W rozdziale 18. znajdziesz omówienie metodologii pomagającej w usunięciu tej pułapki.

Programowanie sieciowe

Prawdziwi programiści wydają aplikacje.

— Jeff Atwood

Jeżeli to byłby po prostu podręcznik dotyczący stosowania technik TDD w zwykłym obszarze programowania, na tym etapie moglibyśmy sobie pogratulować. Zdobyliśmy solidne podstawy z zakresu programowania sterowanego testami i okiełznaliśmy Django. Teraz musimy przystąpić do utworzenia witryny internetowej.

Jednak prawdziwi programiści wydają aplikacje. Aby móc wydać naszą aplikację, musimy zahaczyć o kilka trudniejszych, choć nieuniknionych aspektów programowania sieciowego, takich jak pliki statyczne, weryfikacja danych formularza, budzący lęk kod JavaScript oraz temat sprawiający największą trudność — wdrożenie aplikacji w serwerze produkcyjnym.

Na każdym etapie techniki TDD mogą pomóc we właściwym wykonaniu zadań.

W tej części książki próbuję stosunkowo powoli wprowadzać nowy materiał, ale niewątpliwie poznasz wiele nowych koncepcji i technologii. Objetość książki pozwala jedynie na ich ogólne omówienie. Mam nadzieję, że przedstawię wystarczające wprowadzenie, aby pomóc Ci w rozpoczęciu stosowania wspomnianych koncepcji i technologii we własnych projektach. Jednak kiedy zaczniesz je stosować w „rzeczywistości”, prawdopodobnie będziesz musiał samodzielnie uzupełnić wiedzę.

Na przykład jeżeli przed rozpoczęciem lektury niniejszej książki nie spotkałeś się z frameworkm Django, być może będziesz musiał poświęcić nieco czasu na zapoznanie się z oficjalnym samouczkiem Django. Przedstawiony w nim materiał będzie doskonałym uzupełnieniem zaprezentowanego dotąd w książce, a ponadto pomoże Ci w zrozumieniu zagadnień omawianych w kilku kolejnych rozdziałach. Dzięki temu będziesz mógł się skoncentrować na najważniejszych koncepcjach.

Przed nami jeszcze wiele interesujących tematów. Bądź cierpliwy!

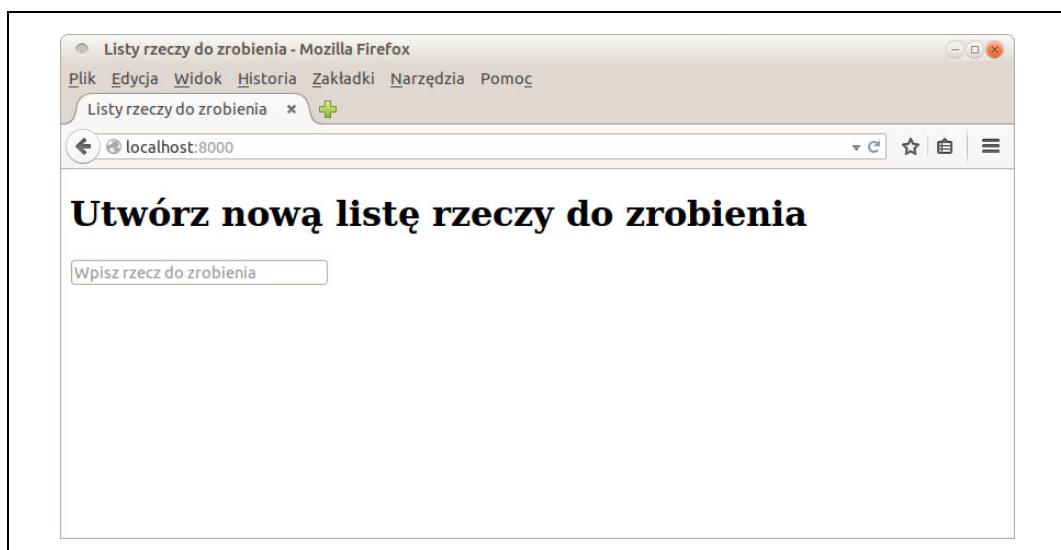


Upiększanie — jak przetestować układ i style?

Zaczynamy zastanawiać się nad wydaniem pierwszej wersji budowanej witryny internetowej, ale jesteśmy nieco zakłopotani jej brzydkim wyglądem na obecnym etapie. W tym rozdziale omówimy temat stosowania podstawowych stylów oraz integracji z frameworkm HTML/CSS o nazwie Bootstrap. Dowiesz się, jak wygląda obsługa plików statycznych w Django oraz co trzeba zrobić, aby je przetestować.

Jaką funkcjonalność należy testować w przypadku układu i stylów?

Na obecnym etapie nasza witryna niewątpliwie istnieje, ale pozostaje nieatrakcyjna (patrz rysunek 7.1).



Rysunek 7.1. Strona główna omawianej witryny wygląda po prostu brzydko...



Jeżeli uruchomisz serwer za pomocą polecenia `manage.py runserver`, możesz otrzymać komunikat błędu informujący, że tabela `lists_item` nie ma kolumny o nazwie `list_id`. W takim przypadku musisz uaktualnić lokalną bazę danych, aby odzwielić zmiany wprowadzone w pliku `models.py`. Wydaj polecenie `manage.py migrate`.

Ponieważ nie chcemy przyłączać się do głosów uznających Pythona za język *brzydkiego*¹, więc musimy wprowadzić kilka usprawnień w wyglądzie aplikacji. Oto kilka kwestii, które warto wziąć pod uwagę:

- Ładne, duże pole tekstowe pozwalające na tworzenie nowych i modyfikację istniejących list.
- Umieszczenie wspomnianego pola tekstowego w dużym i przyciągającym uwagę obszarze na środku okna przeglądarki internetowej.

Jak zastosować techniki TDD podczas wykonywania powyższych zadań? Większość osób jest zdania, że nie należy testować estetyki. Mają tutaj rację, ponieważ to przypomina testowanie stałej, a tego rodzaju testy zwykle nie przynoszą żadnej wartości dodanej.

Możemy jednak przetestować implementację wspomnianej estetyki — to będzie wystarczające do upewnienia się, że rozwiązanie działa zgodnie z oczekiwaniami. Na przykład styl aplikacji nadamy za pomocą kaskadowych arkuszy stylów (CSS), które są wczytywane jako pliki statyczne. Wspomniane pliki statyczne mogą sprawiać nieco trudności w trakcie konfiguracji (zwłaszcza — jak się wkrótce przekonasz — podczas przenoszenia plików między komputerem lokalnym a serwerem WWW). Dlatego też przyda się pewnego rodzaju prosty test potwierdzający wczytanie plików CSS. Nie będziemy testować czcionek i kolorów każdego piksela, ale możemy sprawdzić, czy główne pole tekstowe zostało umieszczone na stronach zgodnie z oczekiwaniami. W ten sposób zyskamy pewność, że pozostałe style dla danej strony również zostały wczytane.

Pracę rozpoczynamy od umieszczenia w teście funkcjonalnym nowej metody testowej.

Plik `functional_tests/tests.py` (ch07l001):

```
class NewVisitorTest(LiveServerTestCase):
    [...]

    def test_layout_and_styling(self):
        # Edyta przeszła na stronę główną.
        self.browser.get(self.live_server_url)
        self.browser.set_window_size(1024, 768)

        # Zauważa, że pole tekstowe zostało elegancko wyśrodkowane.
        inputbox = self.browser.find_element_by_id('id_new_item')
        self.assertAlmostEqual(
            inputbox.location['x'] + inputbox.size['width'] / 2,
            512,
            delta=5
        )
```

Zwróć uwagę na kilka szczegółów. Rozpoczynamy od zdefiniowania wymiarów okna. Następnie znajdujemy element `<input>`, sprawdzamy jego wielkość i położenie, a także wykonujemy pewne obliczenia matematyczne w celu upewnienia się, że element został umieszczony na środku strony. Asercja `assertAlmostEqual()` pomaga w przypadku wystąpienia błędów zaokrąglenia, ponieważ pozwala na określenie dozwolonego odchylenia w operacjach matematycznych. W omawianym przykładzie to 5 pikseli.

¹ <http://grokcode.com/746/dear-python-why-are-you-so-ugly/>

Po wykonaniu testów funkcjonalnych otrzymamy następujący wynik:

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
.F
=====
FAIL: test_layout_and_styling (functional_tests.tests.NewVisitorTest)
-----
Traceback (most recent call last):
  File "/workspace/superlists/functional_tests/tests.py", line 104, in
    test_layout_and_styling
      delta=5
AssertionError: 110.0 != 512 within 5 delta
-----
Ran 2 tests in 9.188s

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

Mamy więc oczekiwane niepowodzenie. Ten rodzaj testu funkcjonalnego może łatwo przynieść nieprawidłowe wyniki i dlatego użyjemy jeszcze szybkiego „oszustwa” w celu sprawdzenia, czy test funkcjonalny zostanie zaliczony, gdy pole tekstowe będzie wyśrodkowane. Poniższy kod usuniemy niemalże natychmiast po przeprowadzeniu testu funkcjonalnego.

Plik *lists/templates/home.html* (ch07l002):

```
<form method="POST" action="/lists/new">
  <p style="text-align: center;">
    <input name="item_text" id="id_new_item" placeholder="Wpisz rzecz do zrobienia" />
  </p>
  {%- csrf_token %}
</form>
```

Test zostaje zaliczony, co oznacza, że testy funkcjonalne działają prawidłowo. Rozbudujemy je teraz, aby sprawdzić, czy pole tekstowe będzie wyśrodkowane na stronie przeznaczonej dla nowej listy.

Plik *functional_tests/tests.py* (ch07l003):

```
# Edyta utworzyła nową listę i zobaczyła,
# że pole tekstowe nadal jest wyśrodkowane.
inputbox.send_keys('testing\n')
inputbox = self.browser.find_element_by_id('id_new_item')
self.assertAlmostEqual(
    inputbox.location['x'] + inputbox.size['width'] / 2,
    512,
    delta=5
)
```

Otrzymujemy kolejny komunikat o niepowodzeniu testu:

```
File "/workspace/superlists/functional_tests/tests.py", line 114, in
  test_layout_and_styling
    delta=5
AssertionError: 110.0 != 512 within 5 delta
```

Przekazujemy do repozytorium pliki ze zmodyfikowanym testem funkcjonalnym:

```
$ git add functional_tests/tests.py
$ git commit -m "Pierwsze kroki w teście funkcjonalnym dla układu i stylów"
```

Teraz prawdopodobnie czujesz się usprawiedliwiony wyszukiwaniem „właściwego” rozwiązania dla zapewnienia odpowiedniejszego stylu dla witryny. Możemy więc powrócić do wcześniejszej sztuczki w postaci polecenia <p style="text-align: center">:

```
$ git reset --hard
```



Polecenie `git reset --hard` stanowi podejście w stylu „wystartuj i pozbądź się witryny z orbity”, więc zachowaj ostrożność podczas jego użycia. Wymienione polecenie spowoduje pozbycie się wszystkich zmian nieprzekazanych do repozytorium. W przeciwieństwie do praktycznie wszystkich pozostałych operacji Git nie ma możliwości cofnięcia skutków wykonania powyższego.

Upiększanie za pomocą frameworka CSS

Projektowanie to trudna sztuka i bez wątpienia trzeba będzie zmierzyć się z różnymi jej aspektami, na przykład przygotowaniem układu dla smartfonów, tabletów itd. Dlatego też wielu programistów, zwłaszcza tak leniwych jak ja, sięga po frameworki CSS, które za nich rozwiązuje wspomniane problemy. Wprawdzie dostępnych jest wiele frameworków, ale jednym z pierwszych i najpopularniejszych pozostaje Bootstrap. Użyjmy go więc w budowanej aplikacji.

Framework Bootstrap znajdziesz w witrynie <http://getbootstrap.com/>.

Pobierzemy Bootstrap i umieścimy w nowym katalogu o nazwie `static` w aplikacji `lists`²:

```
$ wget -O bootstrap.zip https://github.com/twbs/bootstrap/releases/download/v3.1.0/bootstrap-3.1.0-dist.zip
$ unzip bootstrap.zip
$ mkdir lists/static
$ mv dist lists/static/bootstrap
$ rm bootstrap.zip
```

Framework Bootstrap jest dostarczany w postaci zwykłej, niezmodyfikowanej instalacji umieszczonej w katalogu `dist`. Wykorzystamy go teraz podczas pracy, ale tak naprawdę nigdy nie powinieneś tego robić w przypadku rzeczywistej witryny. Niezmodyfikowany framework Bootstrap zostaje niemal natychmiast zidentyfikowany i stanowi czytelny sygnał dla każdego, że nie przejmujesz się nadaniem stylu danej witrynie internetowej. Naucz się używać LESS i przynajmniej zmień czcionkę! Wiele użytecznych informacji o Bootstrap znajdziesz w dokumentacji, a także w doskonałym *artykule*³ opublikowanym przez „Smashing Magazine”.

Obecnie struktura katalogów aplikacji `lists` przedstawia się następująco:

```
$ tree lists
lists
├── __init__.py
└── __pycache__
    └── [...]
├── admin.py
└── models.py
static
└── bootstrap
```

² W systemie Windows możesz nie mieć poleceń `wget` i `unzip`, ale jestem przekonany, że wiesz, jak pobrać Bootstrap, rozpakować i umieścić zawartość katalogu `dist` w katalogu `lists/static/bootstrap`.

³ <http://www.smashingmagazine.com/2013/03/12/customizing-bootstrap/>

```

css
    bootstrap.css
    bootstrap.css.map
    bootstrap.min.css
    bootstrap-theme.css
    bootstrap-theme.css.map
    bootstrap-theme.min.css
fonts
    glyphicon-halflings-regular.eot
    glyphicon-halflings-regular.svg
    glyphicon-halflings-regular.ttf
    glyphicon-halflings-regular.woff
js
    bootstrap.js
    bootstrap.min.js
templates
    home.html
    list.html
tests.py
urls.py
views.py

```

Jeżeli spojrzesz na sekcję *Getting Started*⁴ w dokumentacji Bootstrap, to zauważysz, że szablon HTML powinien mieć postać podobną do poniższej:

```

<!DOCTYPE html>
<html>
    <head>
        <title>Bootstrap 101 Template</title>
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <!-- Bootstrap -->
        <link href="css/bootstrap.min.css" rel="stylesheet" media="screen">
    </head>
    <body>
        <h1>Witaj, świeciel!</h1>
        <script src="http://code.jquery.com/jquery.js"></script>
        <script src="js/bootstrap.min.js"></script>
    </body>
</html>

```

Na obecnym etapie prac mamy już dwa szablony HTML. Nie chcemy dodać całego powyższego kodu do obu szablonów, a więc nadeszła odpowiednia chwila na zastosowanie zasady „nie powtarzaj się” i połączenie ze sobą pewnych fragmentów kodu. Na szczęście język szablonów Django niezwykle ułatwia użycie funkcji określanej mianem dziedziczenia szablonu.

Dziedziczenie szablonu w Django

Przejrzymy różnice, jakie dzielą pliki *home.html* i *list.html*:

```

$ diff lists/templates/home.html lists/templates/list.html
7,8c7,8
<         <h1>Utwórz nową listę rzeczy do zrobienia</h1>
<         <form method="POST" action="/lists/new">
---
>         <h1>Twoja lista rzeczy do zrobienia</h1>
>         <form method="POST" action="/lists/{{ list.id }}/add_item">
11a12,18
>

```

⁴ <http://getbootstrap.com/getting-started/>

```

>     <table id="id_list_table">
>         {% for item in list.item_set.all %}
>             <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
>         {% endfor %}
>     </table>
>

```

Wymienione pliki różnią się tekstem nagłówka, a zdefiniowane w nich formularze używają innych adresów URL. Ponadto plik *list.html* zawiera dodatkowy element `<table>`.

Skoro już wiemy, który kod jest używany w obu plikach, możemy zmodyfikować te szablony, aby dziedziczyły po jednym „superszablonie”. Rozpoczynamy od utworzenia kopii pliku *home.html*:

```
$ cp lists/templates/home.html lists/templates/base.html
```

Na podstawie wymienionego pliku przygotujemy szablon bazowy zawierający najczęściej używany kod i oznaczmy „bloki”, czyli miejsca, w których szablony potomne mogą wprowadzać pewne modyfikacje.

Plik *lists/templates/base.html*:

```

<html>
<head>
    <title>Listy rzeczy do zrobienia</title>
</head>
<body>
    <h1>{% block header_text %}{% endblock %}</h1>
    <form method="POST" action="{% block form_action %}{% endblock %}">
        <input name="item_text" id="id_new_item" placeholder="Wpisz rzecz do zrobienia" />
        {% csrf_token %}
    </form>
    {% block table %}
    {% endblock %}
</body>
</html>

```

Szablon bazowy definiuje serię obszarów nazywanych „blokami”, będących punktami zaцепienia dla innych szablonów, które w tych miejscach mogą dodawać własną zawartość. Zobaczmy, jak takie rozwiązanie sprawdza się w praktyce. Zmieniamy plik *home.html* w taki sposób, aby „dziedziczył po” pliku *base.html*.

Plik *lists/templates/home.html*:

```

{% extends 'base.html' %}

{% block header_text %}Utwórz nową listę rzeczy do zrobienia!{% endblock %}

{% block form_action %}/lists/new{% endblock %}

```

Jak możesz zobaczyć, duża ilość kodu HTML tworzącego szkielet strony zniknęła i koncentrujemy się na fragmentach, których zawartość będzie modyfikowana. Takie samo podejście stosujemy w pliku *list.html*.

Plik *lists/templates/list.html*:

```

{% extends 'base.html' %}

{% block header_text %}Twoja lista rzeczy do zrobienia!{% endblock %}

{% block form_action %}/lists/{{ list.id }}/add_item{% endblock %}

```

```

{%
    block table %
        <table id="id_list_table">
            {% for item in list.item_set.all %}
                <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
            {% endfor %}
        </table>
    {% endblock %}

```

To jest refaktoryzacja sposobu działania szablonów. Ponownie wykonujemy testy funkcjonalne, aby upewnić się, że nie uszkodziliśmy żadnej funkcjonalności...

```
AssertionError: 110.0 != 512 within 5 delta
```

Otrzymujemy dokładnie taki sam wynik jak przed refaktoryzacją. Na obecnym etapie warto więc przekazać pliki do repozytorium:

```

$ git diff -b
# Opcja -b oznacza ignorowanie znaków odstępu, jest tutaj użyteczna, ponieważ zmieniliśmy wcięcia kodu HTML.
$ git status
$ git add lists/templates # Na razie pomijamy katalog static.
$ git commit -m"Refaktoryzacja szablonów, aby używały szablonu bazowego."

```

Integracja z frameworkiem Bootstrap

Integracja kodu tworzącego strukturę kodu z frameworkiem Bootstrap będzie teraz łatwiejsza. Nie musimy dodawać kodu JavaScript, wystarczą same arkusze stylów.

Plik *lists/templates/base.html* (ch07l006):

```

<!DOCTYPE html>
<html lang="pl">

<head>
    <title>Listy rzeczy do zrobienia</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="css/bootstrap.min.css" rel="stylesheet" media="screen">
</head>
[...]

```

Wiersze i kolumny

Wreszcie możemy zająć się pewną magią frameworka Bootstrap. Powinieneś samodzielnie zapoznać się z dokumentacją, ale podpowiadam, że osiągnięcie interesującego nas efektu wymaga użycia systemu siatki i klasy `text-center`.

Plik *lists/templates/base.html* (ch07l007):

```

<body>
<div class="container">

    <div class="row">
        <div class="col-md-6 col-md-offset-3">
            <div class="text-center">
                <h1>{{ block header_text }}</h1>
                <form method="POST" action="{{ block form_action }}>
                    <input name="item_text" id="id_new_item"
                           placeholder="Wpisz rzecznik do zrobienia"
                    />
                    {{ csrf_token }}
                </form>
            </div>
        </div>
    </div>
</div>

```

```
</div>
</div>
</div>

<div class="row">
    <div class="col-md-6 col-md-offset-3">
        {% block table %}
        {% endblock %}
    </div>
</div>

</div>
</body>
```

(Jeżeli nigdy wcześniej nie spotkałeś się z podziałem znacznika HTML na kilka wierszy, wówczas postać znacznika `<input>` może wydawać się nieco szokująca. Kod tego znacznika na pewno jest prawidłowy, ale nie musisz go używać w takiej postaci.



Poświęć nieco czasu i przejrzyj *dokumentację Bootstrap*⁵, jeśli jeszcze tego nie zrobileś. Ten framework przypomina wózek na zakupy wypełniony użytecznymi narzędziami, które możesz wykorzystać w swojej witrynie.

Czy aplikacja działa?

```
AssertionError: 110.0 != 512 within 5 delta
```

Hm, nie. Dlaczego arkusze stylów CSS nie są wczytywane?

Pliki statyczne w Django

Podobnie jak każdy serwer WWW, także Django musi otrzymać pewne informacje, aby móc obsługiwać pliki statyczne:

1. Jak określić, kiedy adres URL żądania wskazuje plik statyczny, a nie prowadzi do kodu HTML generowanego przez funkcję widoku?
2. Gdzie znajdują się pliki statyczne żądane przez użytkownika?

Innymi słowy, pliki statyczne są mapowaniem adresów URL na pliki znajdujące się na dysku.

W przypadku punktu 1. Django pozwala na zdefiniowanie „prefiksu” adresu URL wskazującego, że każdy adres URL rozpoczynający się od danego prefiksu powinien być traktowany jako żądanie pliku statycznego. Domyslnie wspomnianym prefiksem jest `/static/` i został on zdefiniowany w pliku `settings.py`.

Plik `settings.py`:

```
[...]
```

```
# Pliki statyczne (CSS, JavaScript, obrazy).
# https://docs.djangoproject.com/en/1.7/howto/static-files/
STATIC_URL = '/static/'
```

Pozostała część tej sekcji jest powiązana z punktem 2., czyli wyszukiwaniem rzeczywistych plików na dysku.

⁵ <http://getbootstrap.com/>

Używając serwera programistycznego Django (`manage.py runserver`), można zdać się na Django w zakresie magicznego odnajdywania plików statycznych. Framework będzie po prostu przeszukiwał podkatalogi jednego z katalogów `static` w aplikacji.

Teraz już znasz powód umieszczenia wszystkich plików statycznych Bootstrap w katalogu `lists/static`. Powstaje więc pytanie, dlaczego w tym momencie nie są one wczytywane? Odpowiedź jest prosta: ponieważ nie używamy prefiku `/static/` dla adresów URL. Spójrz ponownie na umieszczone w pliku `base.html` łącze do arkusza stylów CSS.

Plik `lists/templates/base.html`:

```
<link href="css/bootstrap.min.css" rel="stylesheet" media="screen">
```

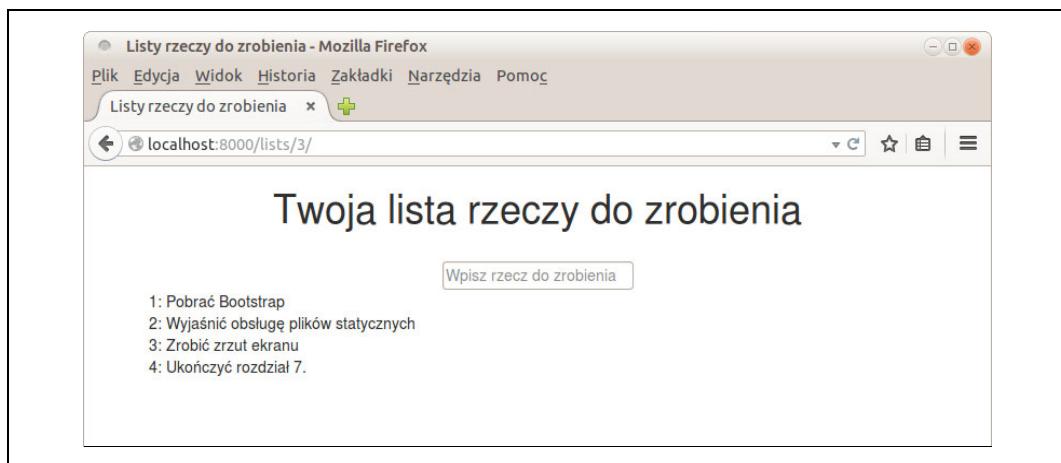
Aby wspomniany plik statyczny został wczytany, łącze należy zmienić na pokazane poniżej.

Plik `lists/templates/base.html`:

```
<link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet" media="screen">
```

Kiedy serwer programistyczny otrzymuje powyższe żądanie, to na podstawie prefiku `/static/`, „wie”, że dotyczy ono pliku statycznego. Próbuje więc znaleźć plik o nazwie `bootstrap/css/bootstrap.min.css`, szukając w tym celu podkatalogów `static` w aplikacji. Ostatecznie zostanie znaleziony plik `lists/static/bootstrap/css/bootstrap.min.css`.

Jeżeli w przeglądarce internetowej ręcznie sprawdzisz aplikację, to przekonasz się, że działa (patrz rysunek 7.2).



Rysunek 7.2. Budowana witryna internetowa zaczyna wyglądać coraz lepiej...

Zaczynamy używać klasy StaticLiveServerCase

Jednak po wykonaniu testów funkcjonalnych zobaczysz, że nie są one zaliczane:

```
AssertionError: 110.0 != 512 within 5 delta
```

Wprawdzie serwer programistyczny (`runserver`) bez problemów odszukuje pliki statyczne, ale `LiveServerTestCase` już nie. Nie powinieneś się jednak obawiać, ponieważ twórcy Django opracowali jeszcze bardziej magiczną klasę testową o nazwie `StaticLiveServerCase` (zapoznaj się z dokumentacją⁶).

⁶ <https://docs.djangoproject.com/en/1.7/howto/static-files/#staticfiles-testing-support>

Zastosujmy więc wymienioną klasę.

Plik *functional_tests/tests.py*:

```
@@ -1,8 +1,8 @@
-from django.test import LiveServerTestCase
+from django.contrib.staticfiles.testing import StaticLiveServerCase
 from selenium import webdriver
 from selenium.webdriver.common.keys import Keys

-class NewVisitorTest(LiveServerTestCase):
+class NewVisitorTest(StaticLiveServerCase):
```

Po wprowadzeniu powyższej zmiany pliki CSS są znajdywane, co powoduje zaliczenie testu:

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
..
-----
Ran 2 tests in 9.764s
```



Na tym etapie użytkownicy Windows mogątrzymać pewne (nieszkodliwe, choć irytujące) komunikaty błędów `socket.error` informujące o zamknięciu połączenia przez zdalny komputer. Aby się pozbyć tych komunikatów, wystarczy w metodzie `tearDown()` dodać wywołanie `self.browser.refresh()` przed wywołaniem `self.browser.quit()`. Ten błąd można monitorować, ponieważ został zgłoszony⁷ twórcom Django.

Hura!

Użycie komponentów Bootstrap do poprawy wyglądu witryny

Przekonajmy się, co można jeszcze poprawić za pomocą oferowanych narzędzi frameworka Bootstrap.

Jumbotron

Framework Bootstrap oferuje klasę `jumbotron` przeznaczoną dla elementów, które mają być szczególnie wyróżnione na stronie. Wykorzystamy więc ją do powiększenia nagłówka strony głównej oraz formularza danych wejściowych.

Plik *lists/templates/base.html* (ch07l009):

```
<div class="col-md-6 col-md-offset-3 jumbotron">
    <div class="text-center">
        <h1>{% block header_text %}{% endblock %}</h1>
        <form method="POST" action="{% block form_action %}{% endblock %}">
            [...]
```



W trakcie modyfikacji układu graficznego dobrze jest pozostawić otwarte okno przeglądarki internetowej, aby móc często odświeżać stronę. Za pomocą polecenia `python3 manage.py runserver` uruchom serwer programistyczny, wpisz w przeglądarce adres `http://localhost:8000/`, a będziesz mógł na bieżąco obserwować postępy podczas pracy.

⁷ <https://code.djangoproject.com/ticket/21227>

Ogromne pola danych wejściowych

Klasa jumbotron jest dobra na początek, ale obecnie pole tekstowe dla danych wprowadzanych przez użytkownika jest niezwykle małe w porównaniu z pozostałymi komponentami. Na szczęście klasy Bootstrap przeznaczone do kontroli formularza pozwalają na zwiększenie wspomnianego pola tekstowego.

Plik *lists/templates/base.html* (ch07l010):

```
<input name="item_text" id="id_new_item"
       class="form-control input-lg"
       placeholder="Wpisz rzecz do zrobienia"
/>
```

Nadanie stylu tabeli

Tekst tabeli wydaje się zbyt mały w porównaniu z pozostałą częścią strony. Dodanie klasy Bootstrap o nazwie `table` znacznie poprawia tę sytuację.

Plik *lists/templates/list.html* (ch07l011):

```
<table id="id_list_table" class="table">
```

Użycie własnych arkuszy stylów CSS

Na koniec chcemy nieco odsunąć pole tekstowe danych wejściowych od tekstu nagłówka. Ponieważ nie ma odpowiedniego stylu we framework'u Bootstrap, musimy przygotować własny. To będzie wymagało zdefiniowania naszego pliku CSS.

Plik *lists/templates/base.html*:

```
<head>
  <title>Listy rzeczy do zrobienia</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet" media="screen">
  <link href="/static/base.css" rel="stylesheet" media="screen">
</head>
```

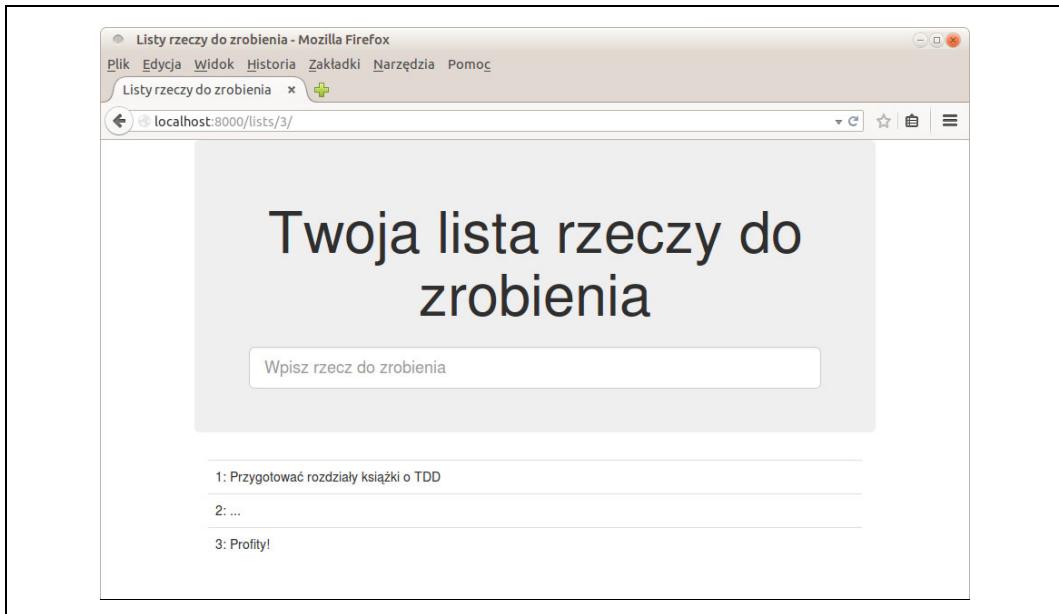
W katalogu *lists/static* tworzymy plik o nazwie `base.css`, w którym umieścimy regułę CSS. W celu odszukania elementu `<input>` i nadania mu odpowiedniego stylu wykorzystamy atrybut `id` o wartości `id_new_item`.

```
#id_new_item {
  margin-top: 2ex;
}
```

Cała operacja wymagała wykonania kilku kroków, ale jestem całkowicie zadowolony z otrzymanego efektu końcowego (patrz rysunek 7.3).

Jeżeli chcesz pójść dalej w zakresie dostosowania framework'a Bootstrap do własnych potrzeb, będziesz musiał przeprowadzić komplikację LESS. Gorąco zachęcam Cię do poświęcenia czasu na przeprowadzenie takiej operacji pewnego dnia. LESS oraz inne technologie pseudo-CSS, na przykład SCSS, stanowią ogromne usprawnienie względem starych, dobrych arkuszy stylów CSS. To również użyteczne narzędzia, nawet jeśli nie korzystasz z framework'a Bootstrap. Wprawdzie tematu dostosowania Bootstrap do własnych potrzeb nie poruszę w tej książce, ale w internecie znajdziesz odpowiednie zasoby, na przykład tutaj⁸.

⁸ <http://www.smashingmagazine.com/2013/03/12/customizing-bootstrap/>



Rysunek 7.3. Strona listy zawierająca duże elementy...

Raz jeszcze wykonujemy testy funkcjonalne, aby sprawdzić, czy wszystko na pewno działa prawidłowo.

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
..
-----
Ran 2 tests in 10.084s

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

O to chodziło! Warto więc przekazać pliki do repozytorium:

```
$ git status # Polecenie pokazuje zmiany w plikach tests.py, base.html, list.html plus niemonitorowany katalog lists/static.
$ git add .
$ git status # Polecenie pokaże wszystkie modyfikacje związane z Bootstrap.
$ git commit -m"Użycie Bootstrap do poprawy układu graficznego."
```

Co zostało zatuszowane — polecenie collectstatic i inne katalogi statyczne

Wcześniej dowiedziałeś się, że serwer programistyczny Django niemal w magiczny sposób potrafi odszukać pliki statyczne znajdujące się w katalogach aplikacji i ich użyć. Takie rozwiązanie jest wystarczające podczas budowy aplikacji, ale po jej umieszczeniu w prawdziwym serwerze WWW nie chcemy, aby Django udostępniało zawartość statyczną. Użycie Pythona do udostępniania niezmodyfikowanych plików jest wolne i nieefektywne, a serwery WWW takie jak Apache lub Nginx doskonale sobie radzą z tym zadaniem. Możesz się nawet zdecydować na umieszczenie wszystkich plików statycznych w serwerach CDN zamiast we własnym serwerze WWW.

Z wymienionych powodów dobrze jest mieć możliwość zebrania wszystkich plików statycznych z różnych katalogów aplikacji i skopiowania ich do pojedynczego miejsca, gotowego do wdrożenia aplikacji. Do tego celu służy polecenie `collectstatic`.

Miejsce docelowe przeznaczone na zebrane pliki statyczne jest zdefiniowane w pliku `settings.py` jako `STATIC_ROOT`. W kolejnym rozdziale zajmiemy się wdrożeniem, więc teraz możemy nieco eksperymentować. Jako wartość wymienionej stałej podamy katalog znajdujący się poza repozytorium. To będzie katalog równorzędny dla katalogu głównego zawierającego kod źródłowy budowanej aplikacji:

```
workspace
    superlists
        lists
            models.py

        manage.py
        superlists

    static
        base.css
        etc...
```

Idea polega na tym, aby katalog plików statycznych nie był częścią repozytorium. Nie chcemy umieszczać go w systemie kontroli wersji, ponieważ stanowi powielenie wszystkich plików znajdujących się w katalogu `lists/static`.

Oto elegancki sposób wskazania katalogu i określenia jego położenia w pliku `settings.py`.

Plik `superlists/settings.py` (ch07l018):

```
# Pliki statyczne (CSS, JavaScript, obrazy).
# https://docs.djangoproject.com/en/1.7/howto/static-files/
STATIC_URL = '/static/'
STATIC_ROOT = os.path.abspath(os.path.join(BASE_DIR, '../static'))
```

Spójrz na początek pliku ustawień, a zobacysz, że zmienna `BASE_DIR` została już wcześniej zdefiniowana. Spróbujmy więc wydać polecenie `collectstatic`:

```
$ python3 manage.py collectstatic

You have requested to collect static files at the destination
location as specified in your settings:

/workspace/static

This will overwrite existing files!
Are you sure you want to do this?

Type 'yes' to continue, or 'no' to cancel:
yes

[...]
Copying '/workspace/superlists/lists/static/bootstrap/fonts/glyphicons-halfling
s-regular.svg'
74 static files copied to '/workspace/static'.
```

Gdy zajrzesz do katalogu `../static`, znajdziesz w nim wszystkie pliki CSS:

```
$ tree ../static/
../static/
    admin
    css
```

```
base.css
[...]
urlify.js
base.css
bootstrap
css
bootstrap.css
bootstrap.min.css
bootstrap-theme.css
bootstrap-theme.min.css
fonts
glyphicon-halflings-regular.eot
glyphicon-halflings-regular.svg
glyphicon-halflings-regular.ttf
glyphicon-halflings-regular.woff
js
bootstrap.js
bootstrap.min.js
```

10 directories, 74 files

Polecenie `collectstatic` zebrało także wszystkie pliki CSS dla witryny administracyjnej. To jedna z funkcji Django o potężnych możliwościach i na pewno ją omówimy, ale jeszcze nie teraz. Dlatego też w tym momencie wyłączamy tę funkcję.

Plik `superlists/settings.py`:

```
INSTALLED_APPS = (
    #django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
)
```

Próbowejmy ponownie:

```
$ rm -rf ../static/
$ python3 manage.py collectstatic --noinput
Copying '/workspace/superlists/lists/static/base.css'
Copying '/workspace/superlists/lists/static/bootstrap/js/bootstrap.min.js'
Copying '/workspace/superlists/lists/static/bootstrap/js/bootstrap.js'
[...]
13 static files copied to '/workspace/static'.
```

Teraz jest już znacznie lepiej.

W ten sposób dowiedziałeś się, jak zebrać w jednym katalogu wszystkie pliki statyczne, co znacznie ułatwi serwerowi WWW ich wyszukiwanie. W następnym rozdziale dowiesz się wszystkiego na temat wdrożenia i jego przetestowania.

Teraz po prostu zapisujemy zmiany wprowadzone w pliku `settings.py`:

```
$ git diff # Polecenie powinno pokazać zmiany wprowadzone w pliku settings.py.
$ git commit -am"Ustawienie STATIC_ROOT w ustawieniach i wyłączenie funkcji admin."
```

Kilka tematów, które nie zostały omówione

Nie ulega wątpliwości, że przedstawiłem tutaj jedynie krótkie wprowadzenie do nadawania stylu oraz do arkuszy stylów CSS. Istnieje znacznie więcej tematów, które chciałbym dokładniej omówić, ale zabrakło dla nich miejsca w rozdziale. Oto kilka zagadnień, którymi warto się zainteresować:

- Dostosowanie Bootstrap za pomocą LESS.
- Znacznik szablonu `{% static %}`, który pomaga unikać powtarzania się i przygotowywać znacznie elastyczniejsze adresy URL.
- Narzędzia pakowania po stronie klienta, na przykład bower.

Przypomnienie — testowanie układu graficznego

Krótką wersję brzmi: nie powinieneś tworzyć testów przeznaczonych do testowania układu graficznego. Tego rodzaju testy oznaczają testowanie zbyt wielu stałych, a ponadto prawdopodobnie będą łatwe do uszkodzenia.

Z uwagi na powyższe stwierdzenie *implementacja* układu graficznego może być nieco trudniejsza i wymaga użycia arkuszy stylów CSS oraz plików statycznych. Dlatego też dobrze jest przygotować pewnego rodzaju minimalny test pozwalający na sprawdzenie, czy pliki statyczne i CSS działają. Jak zobaczyssz w następnym rozdziale, wspomniany test może pomóc w wychwyceniu problemów podczas wdrażania kodu w środowisku produkcyjnym.

Przygotowanie pewnych testów zdecydowanie okaże się niezbędne, jeżeli określony rodzaj stylów wymaga dużej ilości kodu JavaScript działającego po stronie klienta. (Przykładem może być tutaj styl dynamicznej zmiany wielkości, któremu poświęciłem wiele uwagi).

Musisz mieć świadomość, że to jest niebezpieczny obszar. Przygotowanie i wykonanie minimalnych testów daje pewność, że opracowany układ graficzny działa, bez konieczności sprawdzenia, jak faktycznie przedstawia się ten układ. Spróbuj postawić się w sytuacji osoby, która może wprowadzać dowolne zmiany w układzie graficznym bez konieczności ciągłego powracania do testów i ich poprawiania.

TDD na przykładzie witryny prowizorycznej

To jest przyjemność i zabawa, dopóki nie musisz umieścić aplikacji w środowisku produkcyjnym.

— *Devops Borat*¹

Nadszedł moment wdrożenia pierwszej wersji budowanej witryny internetowej i jej publicznego udostępnienia. Można się spotkać z opinią, że jeśli programista będzie czekał z wydaniem aplikacji, dopóki nie uzna jej za gotową do wydania, to opóźnienie będzie za duże.

Czy budowana witryna jest możliwa do użycia? Czy jest lepsza niż nic? Czy pozwala na tworzenie list? Tak, tak, tak.

Nie, nie można się jeszcze zalogować. Nie, nie można oznaczyć zadania jako wykonane. Ale czy naprawdę potrzebujemy wymienionych funkcji? Nie całkiem — nigdy nie będziesz miał pewności, w jaki sposób użytkownicy będą tak naprawdę używać witryny, gdy zostanie im udostępniona. Przewidujemy, że użytkownicy będą jej używać do tworzenia listy rzeczy do zrobienia, ale być może wykorzystają ją do definiowania list dziesięciu najlepszych przynęt. W takim przypadku nie ma potrzeby istnienia funkcji typu „zadanie wykonane”. Nie będziemy o tym wiedzieli, dopóki nie udostępnimy aplikacji publicznie.

W tym rozdziale omówimy temat rzeczywistego wdrożenia budowanej witryny internetowej w prawdziwym serwerze dostępnym w internecie.

Będzie Cię kusiło, aby pominąć ten rozdział, ponieważ zawiera wiele nużących zagadnień, z którymi nie chcesz się teraz zapoznawać. Jednak gorąco zachęcam Cię do lektury tego rozdziału. To jest jeden z tych rozdziałów, z których jestem najbardziej zadowolony. Jednocześnie od wielu czytelników otrzymałem sygnały, że są niezwykle zadowoleni z przedstawionego w nim materiału.

Jeżeli nigdy wcześniej nie zajmowałeś się wdrażaniem aplikacji w serwerze, tutaj dokładnie poznasz to zagadnienie. Nie ma nic przyjemniejszego, niż ujrzeć budowaną przez siebie witrynę w internecie. Nadaj jej przyciągającą nazwę, na przykład „DevOps”, jeśli ma Cię to przekonać, że warto.

¹ https://twitter.com/DEVOPS_BORAT/status/192271992253190144



Po umieszczeniu witryny w internecie możesz mi przysłać jej adres URL. To zawsze sprawia mi przyjemność... obeythetestinggoat@gmail.com.

Techniki TDD i niebezpieczeństw związane z wdrożeniem

Wdrożenie witryny w prawdziwym serwerze WWW może być trudnym tematem. Często spotykałem się z rozpacznym płaczem — „ale to przecież działa w moim komputerze”.

Pewne niebezpieczne obszary wdrożenia obejmują:

Pliki statyczne (CSS, JavaScript, obrazy)

Serwery WWW zwykle wymagają specjalnej konfiguracji, aby móc prawidłowo udostępniać wspomniane typy plików.

Baza danych

Mogą pojawić się problemy związane z uprawnieniami i ścieżkami dostępu. Konieczne jest zachowanie ostrożności, aby uniknąć utraty danych między poszczególnymi wdrożeniami.

Zależności

Trzeba się upewnić, że pakiety wykorzystywane przez nasze oprogramowanie zostały w odpowiednich wersjach zainstalowane w serwerze.

Mamy jednak rozwiązania dla wszystkich wymienionych problemów. Po kolei:

- Użycie *witryny prowizorycznej* o takiej samej strukturze jak produkcyjna może pomóc w przetestowaniu wdrożenia i usunięciu wszystkich problemów, zanim przejdziemy do „rzeczywistej” witryny.
- Istnieje również możliwość *wykonania testów funkcjonalnych względem witryny prowizorycznej*. W ten sposób upewnisz się o posiadaniu w serwerze prawidłowego kodu i pakietów. Ponieważ mamy test sprawdzający układ graficzny witryny, to wiemy, że arkusze CSS są wczytywane prawidłowo.
- *Virtualenv* to użyteczne narzędzie do zarządzania pakietami i zależnościami w komputerze, w którym będzie uruchomionych więcej niż tylko jedna aplikacja Pythona.
- Wreszcie *automatyzacja, automatyzacja i automatyzacja*. Dzięki wykorzystaniu zautomatyzowanego skryptu przeznaczonego do wdrażania nowych wersji oraz użyciu tego samego skryptu do wdrożenia wersji prowizorycznej oraz produkcyjnej zyskujemy pewność, że wersja prowizoryczna będzie niemal taka sama jak ostateczna².

Na kolejnych kilku stronach przedstawię właściwą procedurę wdrożenia. To na pewno nie będzie doskonałą procedurą wdrożenia, więc proszę, nie traktuj jej jako rodzaju najlepszych praktyk lub rekomendacji. Procedura ma spełniać zadanie ilustracji i pokazać rodzaje problemów, jakie pojawiają się w trakcie wdrożenia, a także wskazać, jak można sobie z nimi radzić za pomocą testów.

² To, co ja nazywam serwerem prowizorycznym, inni określają mianem serwera programistycznego, a jeszcze inni wolą użyć określenia „serwer przedprodukcyjny”. Niezależnie od użytej nazwy chodzi o miejsce, w którym można wypróbować kod w środowisku jak najbardziej przypominającym rzeczywisty serwer produkcyjny.

Ogólne omówienie rozdziału

W tym rozdziale znajdziesz omówienie wielu kwestii. Dlatego też w tym miejscu je wymieniam, aby ułatwić Ci orientację.

1. Pracę rozpoczęliśmy od adaptacji naszych testów funkcjonalnych, aby mogły być wykonywane względem serwera prowizorycznego.
2. Następnie przejdziemy do uruchomienia serwera, instalacji wymaganego oprogramowania oraz wskazania mu domen prowizorycznej i rzeczywistej.
3. W kolejnym kroku przekażemy kod do serwera za pomocą Git.
4. Teraz nadszedł moment wyprowadzenia wersji tymczasowej witryny internetowej uruchomionej w prowizorycznej domenie za pomocą serwera programistycznego Django.
5. Następnie poznasz sposoby użycia narzędzia virtualenv do zarządzania w serwerze zależnościami projektu Pythona.
6. Na poszczególnych etapach pracy będą wykonywane testy funkcjonalne w celu sprawdzenia, co działa, a co nie działa.
7. Teraz przechodzimy z wersji tymczasowej do konfiguracji gotowej do umieszczenia w środowisku produkcyjnym. Wykorzystamy tutaj Gunicorn, Upstart i gniazda domeny.
8. Po uzyskaniu działającej konfiguracji przygotujemy skrypt przeznaczony do automatyzacji procesu, który właśnie wykonaliśmy ręcznie. Ten skrypt pozwoli na automatyczne wdrażanie witryny w przyszłości.
9. Na końcu użyjemy przygotowanego skryptu do wdrożenia produkcyjnej wersji witryny internetowej w jej rzeczywistej domenie.

Jak zwykle zaczynamy od testu

Nieco zmodyfikujemy testy funkcjonalne, aby można było je wykonywać względem witryny prowizorycznej. To wymaga małej modyfikacji argumentu standardowo używanego do zmiany adresu, pod którym działa serwer tymczasowy.

Plik *functional_test/tests.py* (ch08l001):

```
import sys
[...]

class NewVisitorTest(StaticLiveServerCase):

    @classmethod
    def setUpClass(cls): #❶
        for arg in sys.argv: #❷
            if 'liveserver' in arg: #❸
                cls.server_url = 'http://' + arg.split('=')[1] #❹
                return #❺
        super().setUpClass()
        cls.server_url = cls.live_server_url

    @classmethod
    def tearDownClass(cls):
        if cls.server_url == cls.live_server_url:
            super().tearDownClass()

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

Dobrze, mówiąc o niewielkich zmianach, tak naprawdę myślałem o całkiem poważnych. Czy pamiętasz, jak wspomniałem, że klasa `LiveServerTestCase` ma pewne ograniczenia? Jednym z nich jest zawsze przyjmowane założenie, że ma być używana w jej własnym serwerze testowym. Wprawdzie czasami chcę korzystać ze wspomnianego serwera, ale jednocześnie chcę mieć możliwość wskazania klasie `LiveServerTestCase`, że powinien zostać użyty prawdziwy serwer WWW.

❶ Metoda `setUpClass()` jest podobna do `setUp()` — również jest dostarczana przez moduł `unittest` i przeznaczona do przygotowania konfiguracji testowej dla całej klasy. Oznacza to, że będzie wykonana tylko raz, a nie przed każdą metodą testową. W omawianej metodzie klasa `LiveServerTestCase` lub `StaticLiveServerCase` zwykle uruchamia własny serwer testowy.

❷❸ W wierszu poleceń szukamy argumentu `liveserver` (zostanie znaleziony w `sys.argv`).

❹❺ Jeżeli wymieniony argument zostanie znaleziony, wtedy nakazujemy klasie testowej poaniuć się metody `setUpClass()` i jedynie przechowywanie w zmiennej o nazwie `server_url` adresu URL serwera prowizorycznego.

Oznacza to konieczność wprowadzenia zmian w trzech miejscach, w których użyto `self.live_server_url`.

Plik `functional_test/tests.py` (ch08l002):

```
def test_can_start_a_list_and_retrieve_it_later(self):
    # Edyta dowiedziała się o nowej, wspaniałej aplikacji w postaci listy rzeczy do zrobienia.
    # Postanowiła więc przejść na stronę główną tej aplikacji.
    self.browser.get(self.server_url)
    [...]
    # Franek odwiedza stronę główną.
    # Nie znajduje żadnych śladów listy Edyty.
    self.browser.get(self.server_url)
    [...]

def test_layout_and_styling(self):
    # Edyta przechodzi na stronę główną.
    self.browser.get(self.server_url)
```

Sprawdzamy, czy wprowadzone zmiany nie uszkodziły funkcjonalności aplikacji. W „zwykły” sposób wykonujemy więc testy funkcjonalne:

```
$ python3 manage.py test functional_tests
[...]
Ran 2 tests in 8.544s
```

OK

Teraz możemy je wykonać względem adresu URL serwera prowizorycznego. W omawianym przykładzie serwer prowizoryczny to `superlists-staging.ott.eu`:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ott.eu
Creating test database for alias 'default'...
FE
=====
FAIL: test_can_start_a_list_and_retrieve_it_later
(functional_tests.tests.NewVisitorTest)

-----
Traceback (most recent call last):
  File "/workspace/superlists/functional_tests/tests.py", line 42, in
test_can_start_a_list_and_retrieve_it_later
    self.assertIn('To-Do', self.browser.title)
```

```
AssertionError: 'To-Do' not found in 'Domain name registration | Domain names
| Web Hosting | 123-reg'

=====
FAIL: test_layout_and_styling (functional_tests.tests.NewVisitorTest)

-----
Traceback (most recent call last):
  File
"/workspace/superlists/functional_tests/tests.py", line 114, in
test_layout_and_styling
    inputbox = self.browser.find_element_by_id('id_new_item')
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_new_item"}' ; Stacktrace:
-----
Ran 2 tests in 16.480s

FAILED (failures=2)
Destroying test database for alias 'default'...
```

Jak możesz zobaczyć, zgodnie z oczekiwaniami oba testy kończą się niepowodzeniem, ponieważ na tym etapie jeszcze nie zdefiniowaliśmy witryny prowizorycznej. Na podstawie danych wyjściowych pierwszego testu możesz zobaczyć, że tak naprawdę kończy się on na stronie głównej firmy rejestrującej domeny.

Wydaje się, że testy funkcjonalne sprawdzają odpowiednie rzeczy, a więc warto przekazać pliki do repozytorium:

```
$ git diff # Polecenie powinno pokazać zmiany wprowadzone w pliku functional_tests.py.
$ git commit -am "Modyfikacja testów funkcjonalnych, aby móc sprawdzać wersję prowizoryczną witryny."
```

Pobranie nazwy domeny

Na tym etapie będziemy potrzebowali kilku domen, to mogą być subdomeny pojedynczej domeny. W omawianym przykładzie wykorzystam *superlists.ottg.eu* oraz *superlists-staging.ottg.eu*. Jeżeli nie masz własnej domeny, najwyższa pora na jej zarejestrowanie! Warto w tym miejscu dodać, że chciałbym, abyś *faktycznie* zarejestrował domenę. Jeżeli nigdy nie rejestrowałeś domeny, wybierz dostawcę i zarejestruj tanią domenę lub zdecyduj się na użycie bezpłatnych domen. Zapewniam Cię, że zobaczenie własnej witryny w internecie będzie ekscytujące.

Ręczne przygotowanie serwera do hostingu naszej witryny

Operację „wdrożenia” można podzielić na dwa zadania:

- *przygotowanie* nowego serwera do hostingu kodu;
- *wdrożenie* nowej wersji kodu w istniejącym serwerze.

Niektórzy lubią używać zupełnie nowego serwera dla poszczególnych wdrożeń — takie rozwiązanie jest stosowane w PythonAnywhere. Tego rodzaju podejście jest niezbędne tylko w przypadku ogromnych, skomplikowanych witryn internetowych lub też po wprowadzeniu poważnych zmian w istniejącej witrynie. Dla prostych witryn, takich jak budowana w książce,

sensowne będzie podzielenie operacji na dwa oddzielne zadania. Wprawdzie ostatecznie oba zadania zostaną w pełni zautomatyzowane, ale na obecnym etapie w zupełności wystarczy nam możliwość ręcznego przygotowania systemu.

W trakcie lektury rozdziału powinieneś pamiętać, że etap przygotowania nie zawsze wygląda tak samo. Dlatego też istnieje kilka uniwersalnych, najlepszych praktyk w zakresie wdrożenia. Zamiast próbować zapamiętać podawane tutaj szczegóły wykonywanych zadań, postaraj się raczej poznać powody podejmowania poszczególnych kroków, aby ten sam tok myślenia móc zastosować w konkretnych sytuacjach, w których się znajdziesz w przyszłości.

Wybór hostingu dla witryny

Obecnie istnieje wiele różnych rozwiązań w zakresie hostingu, ale ogólnie rzecz biorąc, zaliczają się one do dwóch kategorii:

- przygotowanie własnego (prawdopodobnie wirtualnego) serwera;
- użycie usługi PaaS (ang. *Python as a service*), takiej jak Heroku, DoCloud, OpenShift, PythonAnywhere itd.

Usługa typu PaaS oferuje wiele zalet, szczególnie w przypadku małych witryn internetowych. Dlatego też zalecam podążenie właśnie w tym kierunku. Jednak z kilku powodów w książce nie wykorzystamy usługi PaaS. Przede wszystkim mamy tutaj konflikt interesów. Uważam PythonAnywhere za najlepszą tego typu usługę, ale muszę w tym miejscu przypomnieć, że pracuję dla PythonAnywhere. Ponadto poszczególne usługi PaaS dość mocno się różnią między sobą, podobnie jak stosowane w nich procedury wdrożeń. Poznanie jednej niekoniecznie dostarczy Ci informacji wystarczających do pracy z inną. Poza tym gdy czytasz tę książkę, sam proces wdrożenia może być już zupełnie inny bądź też firma mogła zniknąć z rynku.

Dlatego też zdecydowałem się na omówienie starej metody administracji serwerem, co obejmuje między innymi pracę z SSH oraz konfigurację serwera WWW. Istnieje niewielkie prawdopodobieństwo zniknięcia z rynku wszystkich firm oferujących hosting, a odrobinę wiedzy dotyczącej sposobu ich działania pomoże Ci w zdobyciu szacunku wśród rozsianych tu i ówdzie dinozaurów.

Moim celem jest konfiguracja serwera w sposób tworzący środowisko przypominające to, które uzyskasz za pomocą usługi typu PaaS. Zdobyte tutaj umiejętności powinieneś móc później wykorzystać niezależnie od wybranego rozwiązania w zakresie wdrożenia aplikacji.

Uruchomienie serwera

Nie zamierzam Ci tutaj dyktować, jak należy to zrobić. Niezależnie od tego, co wybierzesz — Amazon AWS, Rackspace, Digital Ocean, własny serwer w centrum danych lub Raspberry Pi w szafce pod schodami — każde z tych rozwiązań będzie dobre, o ile spełnisz kilka wymagań:

- serwer działa pod kontrolą systemu Ubuntu (13.04 lub nowszego);
- masz dostęp do konta użytkownika root;
- serwer może udostępniać dane w internecie;
- masz możliwość połączenia z serwerem za pomocą SSH.

Zalecam użycie dystrybucji Ubuntu, ponieważ zawiera ona Pythona 3.4 oraz pewne specyficzne sposoby konfiguracji Nginx, o czym dowiesz się już wkrótce. Jeżeli wiesz, co robisz, to prawdopodobnie możesz zastosować inne rozwiązania, ale na własną odpowiedzialność.



Niektórzy czytelnicy mogą odczuwać pokusę pominięcia fragmentu dotyczącego domeny i przejścia od razu do fragmentu poświęconego „prawdziwemu serwerowi” lub po prostu użyć maszyny wirtualnej w komputerze. Nie postępuj w taki sposób. To *nie* jest to samo i możesz napotkać większe trudności w wykonywaniu poleceń, które i tak są już wystarczająco skomplikowane. Jeżeli masz obawy dotyczące kosztów, poszukaj trochę w internecie, a znajdziesz firmy oferujące bezpłatne domeny i hosting. Napisz do mnie, jeśli mimo wszystko potrzebujesz kolejnych wskazówek — zawsze chętnie pomagam.

Konto użytkownika, SSH i uprawnienia

W przedstawionej poniżej procedurze przyjęłem założenie, że masz konto użytkownika inne niż root skonfigurowane wraz z uprawnieniami sudo. Kiedy operacja wymaga uprawnień użytkownika root, wtedy będziemy używać polecenia sudo. Konieczność użycia sudo jest wyraźnie zaznaczona w różnych przedstawionych dalej poleceniach. Jeżeli chcesz utworzyć konto użytkownika inne niż root, poniżej podano szczegółową procedurę:

```
# Poniższe polecenia muszą być wykonane przez użytkownika root.  
root@serwer:$ useradd -m -s /bin/bash edyta # Dodanie użytkownika edyta.  
# Opcja -m tworzy katalog domowy, natomiast -s wskazuje bash jako powłokę domyślną.  
root@serwer:$ usermod -a -G sudo edyta # Dodanie użytkownika edyta do grupy sudoers.  
root@serwer:$ passwd edyta # Ustawienie hasła dla użytkownika edyta.  
root@serwer:$ su - edyta # Przelączenie użytkownika na edyta!  
edyta@serwer:$
```

Tworzonymu tutaj użytkownikowi możesz nadać dowolną nazwę. W celu pracy z SSH zachęcam do poznania sposobu użycia prywatnego klucza uwierzytelniania zamiast haseł. Konfiguracja nie jest trudna i sprowadza się do pobrania klucza publicznego z komputera lokalnego i umieszczenia go w pliku `~/.ssh/authorized_keys` w katalogu użytkownika w serwerze. Podobną procedurę prawdopodobnie już przeprowadziłeś w trakcie konfiguracji BitBucket lub GitHub.

Przydatne informacje na temat konfiguracji znajdziesz na przykład *tutaj*³ (zwróć uwagę, że narzędzie ssh-keygen jest instalowane wraz z GitBash w Windows).



W listingach przedstawionych w rozdziale wyszukaj ciąg tekstowy `edyta@serwer`. Wskazuje on polecenia, które muszą być wykonane w serwerze, a nie w komputerze lokalnym.

Instalacja Nginx

Musimy zainstalować jeszcze serwer WWW, a wszyscy sprytni programiści używają obecnie Nginx. Dlatego decydujemy się na ten właśnie serwer. Mając za sobą wieloletnie zmagania z serwerem Apache, mogę stwierdzić, że przejście na Nginx jest niezwykłą ulgą, między innymi pod względem czytelności jego plików konfiguracyjnych.

³ <https://www.linode.com/docs/networking/ssh/use-public-key-authentication-with-ssh/>

Instalacja Nginx w serwerze sprowadza się do wydania polecenia apt-get, a następnie zobaczyś domyślny ekran typu „Witaj, świecie!” wygenerowany przez Nginx:

```
edyta@serwer:$ sudo apt-get install nginx  
edyta@serwer:$ sudo service nginx start
```

(Prawdopodobnie będziesz musiał jeszcze wydać polecenie apt-get update i/lub apt-get upgrade).

Teraz po przejściu do adresu IP serwera powinieneś zobaczyć domyślną stronę typu „Witaj, świecie!” wyświetlzoną przez Nginx (patrz rysunek 8.1).



Rysunek 8.1. Serwer Nginx działa!

Jeżeli nie widzisz wspomnianej strony, to być może zapora sieciowa nie zezwala na otworzenie portu 80 dla świata zewnętrznego. Na przykład w usłudze AWS może wystąpić konieczność konfiguracji „grupy zabezpieczeń” serwera i otwarcia portu 80.

Skoro mamy dostęp z uprawnieniami użytkownika root, warto upewnić się, że serwer posiada oprogramowanie niezbędne na poziomie systemu, czyli Python, Git, pip i virtualenv.

```
edyta@serwer:$ sudo apt-get install git python3 python3-pip  
edyta@serwer:$ sudo pip3 install virtualenv
```

Konfiguracja domen dla witryny prowizorycznej i rzeczywistej

Nie chcemy ciągle korzystać z adresów IP i dlatego powinniśmy powiązać serwer z domenami dla witryny prowizorycznej i rzeczywistej. Na rysunku 8.2 pokazano ekran konfiguracyjny, jaki udostępnia firma, w której rejestruję domeny.

W systemie DNS wskazanie domeny pod określonym adresem IP to „Rekord A”. Poszczególne firmy mogą używać różnych ekranów konfiguracyjnych, ale przeglądając dostępne ustawienia, wreszcie trafisz na odpowiednie.

DNS ENTRY	TYPE	PRIORITY TTL	DESTINATION/TARGET	
*	A		81.21.76.62	
@	A		81.21.76.62	
@	MX	10	mx0.123-reg.co.uk.	
@	MX	20	mx1.123-reg.co.uk.	
dev	CNAME		harry.pythonanywhere...	
www	CNAME		harry.pythonanywhere...	
book-example	A		82.196.1.70	
book-example-staging	A		82.196.1.70	
Hostname	Type		Destination IPv4 address	
<input type="text" value="new"/>	<input type="button" value="A"/>		<input type="text" value="81.21.76.62"/>	<input type="button" value="Add"/>
			Time: 04.12.2018 12:01	

Rysunek 8.2. Konfiguracja domeny

Użycie testów funkcjonalnych do potwierdzenia działania domeny i serwera Nginx

Aby potwierdzić poprawność wprowadzonych zmian, możemy ponownie wykonać testy funkcjonalne i zobaczyć, czy ich komunikaty niepowodzenia uległy nieco zmianie. Jeden z nich powinien zawierać wyraźną wzmiankę o Nginx:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
[...]
selenium.common.exceptions NoSuchElementException: Message: 'Unable to locate
element: {"method":"id","selector":"id_new_item"}' ; Stacktrace:
[...]
AssertionError: 'Listy' not found in 'Welcome to nginx!'
```

Postęp!

Ręczne wdrożenie kodu

Kolejnym krokiem jest przygotowanie działającej kopii witryny prowizorycznej, aby sprawdzić, czy istnieje możliwość prowadzenia komunikacji między Nginx i Django. Kiedy to zrobimy, wtedy zaczniemy przechodzić do etapu wdrożenia, zamiast pozostać na etapie konfiguracji. Warto po drodze zastanowić się nad sposobem automatyzacji procesu.



Jedną z różnic między konfiguracją i wdrożeniem jest to, że podczas konfiguracji zwykle wymagane są uprawnienia użytkownika root, natomiast w trakcie wdrażania już nie.

Potrzebujemy katalogu na kod źródłowy. Przyjmujemy założenie, że istnieje katalog domowy dla użytkownika innego niż root. W omawianym przykładzie to będzie `/home/edyta` (prawdopodobnie nie będzie innej możliwości we współdzielonych systemach hostingu, ale pamiętaj, aby aplikacje sieciowe zawsze uruchamiać z poziomu użytkownika innego niż root). Konfiguracja witryn przedstawia się następująco:

```

/home/edyta
sites
    www.live.my-website.com
        database
            db.sqlite3
        source
            manage.py
            superlists
            etc...
        static
            base.css
            etc...
    virtualenv
        lib
        etc...
www.staging.my-website.com
    database
    etc...

```

Poszczególne witryny (prowizoryczna, rzeczywista i każda inna) są umieszczone w oddzielnych katalogach. Wewnątrz każdego z nich znajdują się podkatalogi przeznaczone na kod źródłowy, bazę danych i pliki statyczne. Logika została oparta na tym, że choć kod źródłowy może ulegać zmianie w poszczególnych wersjach witryny, to baza danych prawdopodobnie pozostanie taka sama. Katalog *static* ma to samo względne położenie (*./static*), które zostało skonfigurowane na końcu poprzedniego rozdziału. Oczywiście narzędzie *virtualenv* również otrzymuje własny podkatalog. Już słyszałeś, jak pytasz, co to jest *virtualenv*. Wkrótce się dowiesz.

Dostosowanie położenia bazy danych

Najpierw w pliku *settings.py* zmienimy położenie bazy danych, a następnie upewnijmy się o możliwości uzyskania do niej dostępu z poziomu komputera lokalnego. Użycie *os.path.abspath* pozwala uniknąć wszelkich niejasności dotyczących aktualnego katalogu roboczego.

Plik *superlists/settings.py* (ch08l003):

```

# W projekcie ścieżki dostępu są tworzone następująco: os.path.join(BASE_DIR, ...)
import os
BASE_DIR = os.path.abspath(os.path.dirname(os.path.dirname(__file__)))
[...]
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, '../database/db.sqlite3'),
    }
}
[...]
STATIC_ROOT = os.path.join(BASE_DIR, '../static')

```

Teraz wypróbujemy je lokalnie:

```

$ mkdir ../database
$ python3 manage.py migrate --noinput
Creating tables ...
[...]
$ ls ../database/
db.sqlite3

```

Wydaje się, że wszystko działa. Przekazujemy pliki do repozytorium:

```
$ git diff # Polecenie powinno pokazać zmiany wprowadzone w pliku settings.py.  
$ git commit -am "Przeniesienie bazy danych sqlite poza gałąź głównego kodu źródłowego."
```

W celu umieszczenia kodu w serwerze wykorzystamy Git oraz jedną z witryn umożliwiających dzielenie się kodem. Jeżeli jeszcze tego nie zrobiłeś, umieść kod w serwisie GitHub, BitBucket lub podobnym. Oba wymienione oferują doskonale informacje dla początkujących, wskazujące, jak można to zrobić.

Poniżej przedstawiono powłoki bash pozwalające na przeprowadzenie całej konfiguracji. Warto w tym miejscu dodać, że polecenie export powoduje ustawienie „zmiennej lokalnej” w powłoce bash:

```
edyta@serwer:$ export SITENAME=superlists-staging.ottg.eu  
edyta@serwer:$ mkdir -p ~/sites/$SITENAME/database  
edyta@serwer:$ mkdir -p ~/sites/$SITENAME/static  
edyta@serwer:$ mkdir -p ~/sites/$SITENAME/virtualenv  
# Adres URL w poniższym wierszu zastąp adresem URL wskazującym Twoje repozytorium.  
edyta@serwer:$ git clone https://github.com/hjwp/book-example.git \  
~/sites/$SITENAME/source  
Resolving deltas: 100% [...]
```



Zadeklarowane za pomocą polecenia export zmienne powłoki bash pozostają dostępne jedynie w bieżącej sesji konsoli. Po wylogowaniu i ponownym zalogowaniu się do serwera konieczne będzie ich ponowne skonfigurowanie. Trzeba o tym koniecznie pamiętać, ponieważ powłoka bash nie wyświetla komunikatu błędu, a po prostu zastąpi nieistniejącą zmienną pustym ciągiem tekstowym, co doprowadzi do dziwnych wyników... Jeżeli masz wątpliwości, zawsze możesz wydać polecenie echo \$SITENAME.

Po zainstalowaniu witryny możemy spróbować uruchomić serwer programistyczny. To jest rodzaj sprawdzenia, czy wszystkie ruchome elementy są ze sobą połączone:

```
edyta@serwer:$ cd ~/sites/$SITENAME/source  
$ python3 manage.py runserver  
Traceback (most recent call last):  
  File "manage.py", line 8, in <module>  
    from django.core.management import execute_from_command_line  
ImportError: No module named django.core.management
```

Ach. Django nie zostało zainstalowane w serwerze.

Utworzenie virtualenv

W prawdziwe moglibyśmy teraz zainstalować Django, ale to pozostawi nas z pewnym problemem. Mianowicie jeżeli kiedykolwiek zechcesz uaktualnić Django do nowej wersji, wówczas niemożliwe okaże się przetestowanie witryny prowizorycznej opartej na innej wersji Django niż rzeczywista witryna. A jeżeli z serwera korzystają także inni użytkownicy, wtedy wszyscy są zmuszeni do użycia tej samej wersji Django.

Rozwiązaniem jest użycie virtualenv, czyli elegancki sposób posiadania różnych wersji pakietu Python zainstalowanych w różnych miejscach, w ich „środowiskach wirtualnych”.

Tego rodzaju rozwiązanie wypróbowujemy najpierw lokalnie:

```
$ pip3 install virtualenv # W systemach Linux i OS X trzeba użyć polecenia sudo.
```

Przygotowujemy tę samą strukturę katalogów, jaka ma zostać użyta w serwerze:

```
$ virtualenv --python=python3 ..virtualenv  
$ ls ..virtualenv/  
bin include lib
```

Powyższe polecenia spowodują utworzenie katalogu w `..virtualenv` zawierającego własną kopię Pythona, narzędzia pip oraz miejsce na instalację pakietów Pythona. Otrzymujemy rodzaj samodzielnego, „wirtualnego” środowiska Pythona. Aby można było go używać, trzeba wykonać skrypt o nazwie `activate` zmieniający ścieżki dostępu systemową i Pythona w taki sposób, aby stosowane były pliki i pakiety pochodzące z danego środowiska wirtualnego:

```
$ which python3  
/usr/bin/python3  
$ source ..virtualenv/bin/activate  
$ which python # Przejście do Pythona w środowisku wirtualnym.  
/workspace/virtualenv/bin/python  
(virtualenv)$ python3 manage.py test lists  
[...]  
ImportError: No module named 'django'
```



To nie jest wymagane, ale możesz zapoznać się z narzędziem o nazwie `virtualenvwrapper` przeznaczonym do zarządzania środowiskami wirtualnymi w komputerze.

Środowisko wirtualne w Windows

W systemie Windows sytuacja przedstawia się nieco inaczej. Trzeba zwrócić uwagę na dwie ważne kwestie:

- Katalog `virtualenv/bin` nosi nazwę `virtualenv/Scripts`, a więc powinieneś pamiętać o użyciu odpowiedniej nazwy.
- Kiedy używasz GitBash, nie próbuj nawet uruchamiać `activate.bat` — wymieniony skrypt został przygotowany pod kątem DOS-u. Zamiast tego wydaj polecenie `source ..\virtualenv\Scripts\activate`, przy czym słowo kluczowe `source` jest niezwykle ważne.

Otrzymamy komunikat `ImportError`, ponieważ Django nie zostało jeszcze zainstalowane w środowisku wirtualnym. Możemy więc zainstalować framework Django i sprawdzić, że został umieszczony w katalogu `site-packages` środowiska wirtualnego:

```
(virtualenv)$ pip install django==1.7  
[...]  
Successfully installed django  
Cleaning up...  
(virtualenv)$ python3 manage.py test lists  
[...]  
OK  
$ ls ..virtualenv/lib/python3.4/site-packages/  
django          pip           setuptools  
Django-1.7-py3.4.egg-info  pip-1.4.1-py3.4.egg-info  setuptools-0.9.8-py3.4.egg-info  
easy_install.py    pkg_resources.py  
_markerlib        __pycache__
```

Za pomocą polecenia pip freeze tworzymy plik o nazwie *requirements.txt* zawierający listę pakietów potrzebnych w środowisku wirtualnym. Dzięki wymienionemu plikowi później będzie można łatwo odtworzyć środowisko wirtualne. Nowy plik przekazujemy do repozytorium:

```
(virtualenv)$ pip freeze > requirements.txt  
(virtualenv)$ deactivate  
$ cat requirements.txt  
Django==1.7  
$ git add requirements.txt  
$ git commit -m"Dodanie pliku requirements.txt opisującego środowisko wirtualne."
```



Wprawdzie w trakcie pisania książki framework Django w wersji 1.7 nie był jeszcze w pełni gotowy, ale zawsze możesz wydać polecenie pip install <https://github.com/django/django/archive/stable/1.7.x.zip> i tym samym do pliku *requirements.txt* dodać adres URL zamiast po prostu Django==1.7. Narzędzie pip doskonale sobie z tym poradzi. Przeprowadź test, wydając lokalnie polecenie pip install -r requirements.txt, a przekonasz się, że narzędzie pip uwzględnioło całe zainstalowane oprogramowanie.

Teraz wystarczy wydać polecenie git push, aby modyfikacje zostały wprowadzone w witrynie przeznaczonej do dzielenia się kodem:

```
$ git push
```

Kolejnym krokiem jest przekazanie do serwera wprowadzonych powyżej zmian, utworzenie środowiska wirtualnego w serwerze oraz użycie pliku *requirements.txt* wraz z pip install -r, aby serwer wirtualny odzwierciedlał lokalny:

```
edyta@serwer:$ git pull # Możesz zostać poproszony o przeprowadzenie pewnej konfiguracji Git.  
edyta@serwer:$ virtualenv --python=python3 ..../virtualenv/  
edyta@serwer:$ ..../virtualenv/bin/pip install -r requirements.txt  
Downloading/unpacking Django==1.7 (from -r requirements.txt (line 1))  
[...]  
Successfully installed Django  
Cleaning up...  
edyta@serwer:$ ..../virtualenv/bin/python3 manage.py runserver  
Validating models...  
0 errors found  
[...]
```

Wygląda na to, że proces przebiegł pomyślnie. Możesz nacisnąć klawisze *Ctrl+C*.

Zwróć uwagę na brak konieczności wydania polecenia activate w celu użycia środowiska wirtualnego. Działa również bezpośrednie wskazanie ścieżki dostępu do poleceń python i pip zainstalowanych we wspomnianym środowisku wirtualnym. W serwerze wykorzystamy te bezpośrednie ścieżki dostępu.



Większość osób tworzy środowisko wirtualne dla projektu tuż po rozpoczęciu prac nad nim. W tej książce zwlekałem z tym, ponieważ chciałem, aby materiał przedstawiony w pierwszych rozdziałach pozostał jak najprostszy.

Prosta konfiguracja Nginx

Kolejnym krokiem jest przygotowanie pliku konfiguracyjnego Nginx i określenie, aby żądania kierowane do witryny prowizorycznej były obsługiwane przez Django. Poniżej przedstawiono minimalną wersję pliku konfiguracyjnego.

Plik `/etc/nginx/sites-available/superlists-staging.ottg.eu` w serwerze:

```
server {  
    listen 80;  
    server_name superlists-staging.ottg.eu;  
  
    location / {  
        proxy_pass http://localhost:8000;  
    }  
}
```

Powyższa konfiguracja jest przeznaczona jedynie dla witryny prowizorycznej i stanowi „proxy” dla wszystkich żądań kierowanych do portu lokalnego 8000, na którym oczekiwane jest znalezienie framework'a Django gotowego do udzielania odpowiedzi na żądania.

Konfigurację zapisałem⁴ w pliku o nazwie `superlists-staging.ottg.eu` umieszczonym w katalogu `/etc/nginx/sites-available`. Następnie dodałem plik do katalogu witryn dostępnych dla serwera, posługując się w tym celu dowiązaniem symbolicznym:

```
edyta@serwer:$ echo $SITENAME # Sprawdzamy naszą witrynę.  
superlists-staging.ottg.eu  
edyta@serwer:$ sudo ln -s ..sites-available/$SITENAME \  
/etc/nginx/sites-enabled/$SITENAME  
edyta@serwer:$ ls -l /etc/nginx/sites-enabled # Sprawdzamy istnienie dowiązania symbolicznego.
```

To jest preferowany w systemach Debian i Ubuntu sposób zapisywania konfiguracji serwera Nginx. Rzeczywisty plik konfiguracyjny znajduje się w katalogu `sites-available`, a w katalogu `sites-enabled` tworzymy dowiązanie symboliczne wskazujące plik konfiguracyjny. Takie rozwiązanie ma ułatwić włączanie i wyłączanie witryn internetowych.

Istnieje również możliwość usunięcia domyślnej, powitalnej konfiguracji Nginx, aby tym samym uniknąć wszelkich niejasności:

```
edyta@serwer:$ sudo rm /etc/nginx/sites-enabled/default
```

Teraz możemy przetestować wprowadzone zmiany:

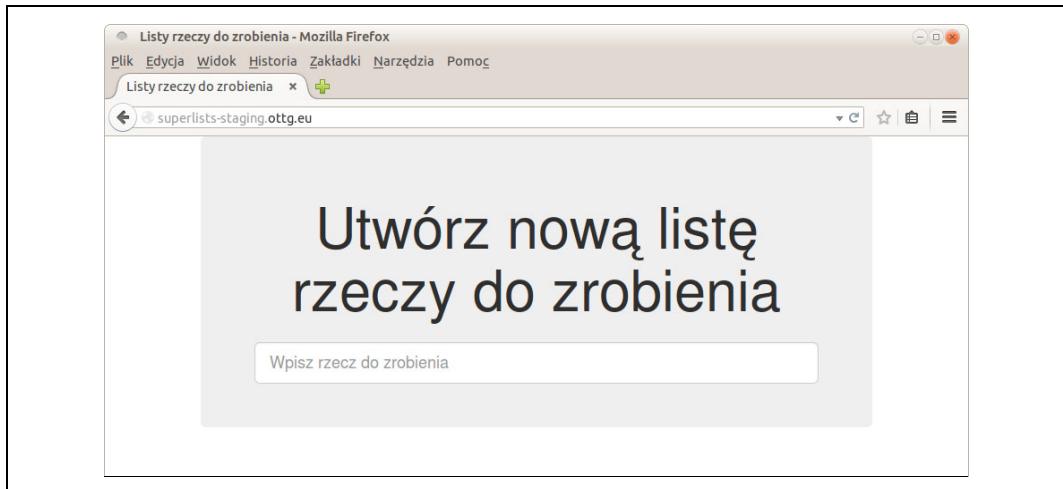
```
edyta@serwer:$ sudo service nginx reload  
edyta@serwer:$ ../virtualenv/bin/python3 manage.py runserver
```



Konieczna jest również edycja pliku `/etc/nginx/nginx.config` i usunięcie znaku komentara na początku wiersza `server_names_hash_bucket_size 64;`. Ta zmiana ma na celu umożliwienie użycia długich nazw domen. Możesz nie mieć tego problemu, ponieważ Nginx wyświetli ostrzeżenie, gdy po otrzymaniu polecenia `reload` napotka jakiekolwiek problemy związane z jego plikami konfiguracyjnymi.

Po spojrzeniu na okno przeglądarki internetowej otrzymujemy potwierdzenie działania witryny (patrz rysunek 8.3).

⁴ Nie wiesz, jak edytować plik w serwerze? Zawsze będziesz miał dostępny edytor vi i dlatego nieustannie zechęcam do jego poznawania. Ewentualnie możesz wypróbować przyjazny dla początkujących edytor nano. Zwrót uwagę na konieczność użycia polecenia `sudo`, ponieważ omawiany plik konfiguracyjny znajduje się w katalogu systemowym.

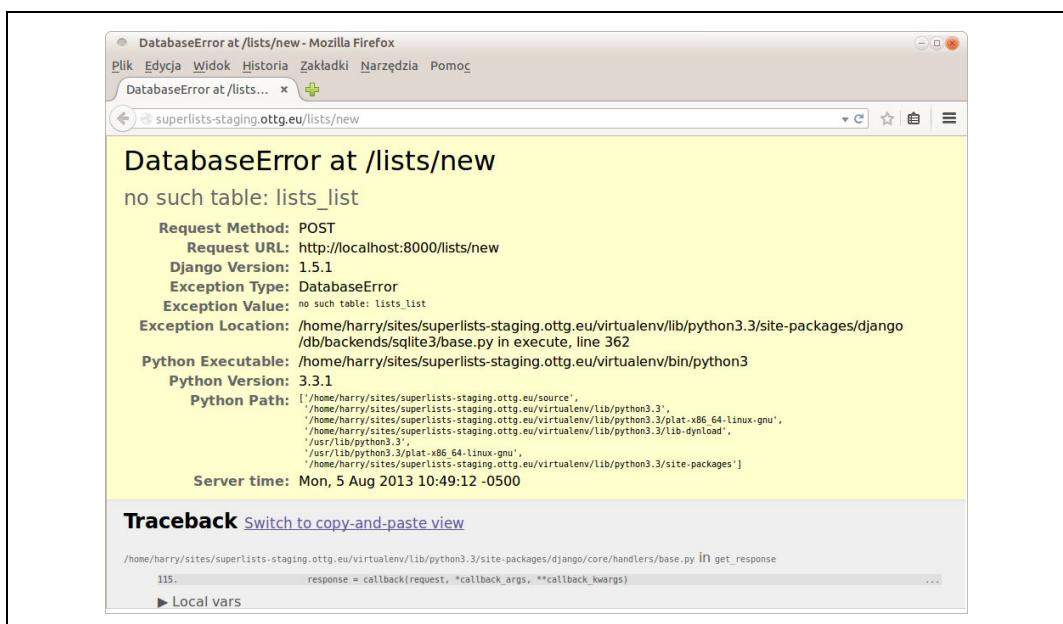


Rysunek 8.3. Witryna prowizoryczna działa!

Zobaczmy, jaki będzie wynik wykonania testów funkcjonalnych:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
[...]
AssertionError: 0.0 != 512 within 3 delta
```

Testy funkcjonalne kończą się niepowodzeniem tuż po wprowadzeniu nowego elementu listy, ponieważ jeszcze nie skonfigurowaliśmy bazy danych. Prawdopodobnie zauważysz już żółtą stronę Django (patrz rysunek 8.4) wyświetlającą informacje o przeprowadzonych testach.



Rysunek 8.4. Baza danych nie została przygotowana



Dzięki testom unikamy tutaj możliwego zakłopotania. Witryna internetowa *wyglądała* dobrze po wczytaniu strony głównej. W pośpiechu można uznać pracę za wykonaną, jednak użytkownicy odkryliby niemalą stronę błędu Django. Oczywiście nieco tutaj przesadzam i *być może* sprawdziłbyś działanie całej witryny. Zastanów się jednak, jak to wygląda, gdy witryna rozrasta się i jest coraz bardziej skomplikowana? Ręcznie nie jesteś w stanie sprawdzić wszystkiego. Natomiast testy to umożliwiają.

Utworzenie bazy danych za pomocą polecenia migrate

Wydajemy polecenie `migrate` wraz z opcją `--noinput`, aby zawiesić wyświetlanie dwóch pytań typu „czy jesteś pewien”:

```
edyta@serwer:$ ../virtualenv/bin/python3 manage.py migrate --noinput
Creating tables ...
[...]
edyta@serwer:$ ls ..database/
db.sqlite3
edyta@serwer:$ ../virtualenv/bin/python3 manage.py runserver
```

Ponownie wykonujemy testy funkcjonalne:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
Creating test database for alias 'default'...
..
-----
Ran 2 tests in 10.718s

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

Doskonale jest zobaczyć działającą witrynę! Przed przejściem do kolejnego podrozdziału możesz zrobić sobie zasłużoną przerwę na herbatę...



Jeżeli otrzymasz błąd o kodzie 502, prawdopodobnie wynika to z faktu, że zapomniałeś ponownie uruchomić serwer programistyczny, wydając po poleceniu `migrate` polecenie `manage.py runserver`.

Wdrożenie w środowisku produkcyjnym

Mamy pewność, że przygotowana witryna działa, ale tak naprawdę w środowisku produkcyjnym nie możemy używać serwera programistycznego. Ponadto nie możemy się opierać na jego ręcznym uruchamianiu za pomocą polecenia `runserver`.

Użycie Gunicorn

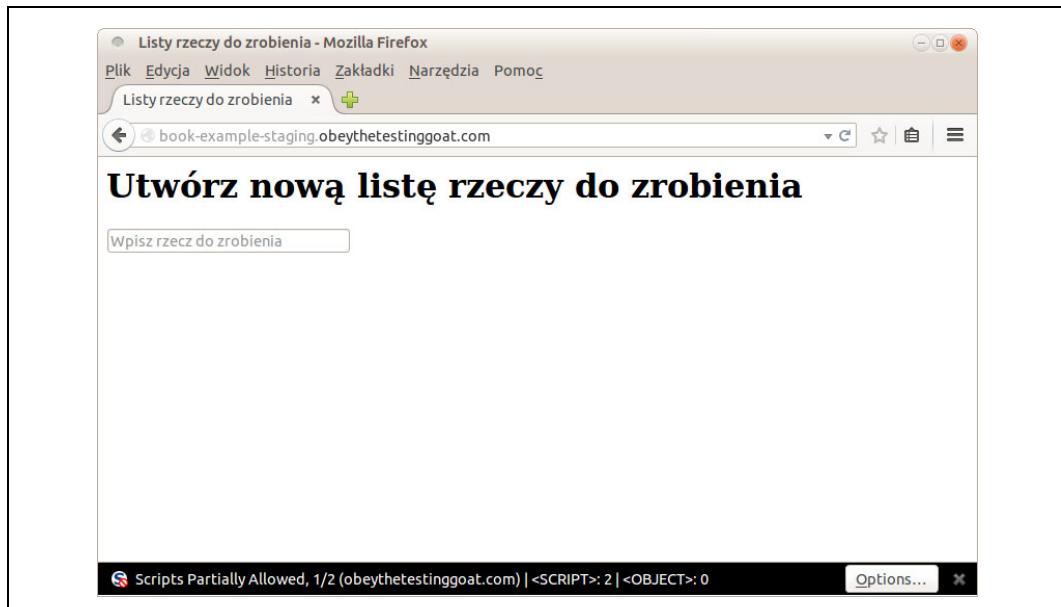
Czy wiesz, dlaczego maskotką Django jest kucyk? Framework Django jest dostarczany wraz z wieloma potrzebnymi komponentami, takimi jak mapowanie obiektowo-relacyjne, wszelkiej maści dostawcy, sekcja administracyjna... Czego jeszcze potrzebujesz, kucyka? Cóż, nazwa Gunicorn oznacza „Green Unicorn”, czyli „zielony jednorożec”, który — jak sądzę — jest kolejną pozycją na liście życzeń właścicieli kucyków...

```
edyta@serwer:$ ../virtualenv/bin/pip install gunicorn
```

Narzędzie Gunicorn musi mieć podaną ścieżkę dostępu do serwera WSGI, który najczęściej jest funkcją o nazwie `application()`. Django dostarcza ją w pliku `superlists/wsgi.py`:

```
edyta@serwer:$ ../virtualenv/bin/gunicorn superlists.wsgi:application
2013-05-27 16:22:01 [10592] [INFO] Starting gunicorn 0.18.0
2013-05-27 16:22:01 [10592] [INFO] Listening at: http://127.0.0.1:8000 (10592)
[...]
```

Jeżeli teraz przejdziesz do naszej witryny, przekonasz się, że arkusze stylów CSS nie zostały wczytane (patrz rysunek 8.5).



Rysunek 8.5. Arkusze stylów CSS nie zostały wczytane

Wynik wykonania testów funkcyjnych potwierdza istnienie nieprawidłowości. Test dotyczący dodawania elementów listy jest zaliczany, natomiast test dotyczący układu graficznego kończy się niepowodzeniem. Dobra robota, testy!

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
[...]
AssertionError: 125.0 != 512 within 3 delta
FAILED (failures=1)
```

Powodem niewczytywania plików CSS jest fakt, że choć serwer programistyczny Django automatycznie zajmuje się obsługą plików statycznych, to Gunicorn już nie. Nadszedł czas na wykorzystanie do tego celu serwera Nginx.

Użycie Nginx do obsługi plików statycznych

Przede wszystkim trzeba wydać polecenie `collectstatic`, aby wszystkie pliki statyczne skopiować do katalogu, w którym znajdzie je serwer Nginx:

```
edyta@serwer:$ ../virtualenv/bin/python3 manage.py collectstatic --noinput
edyta@serwer:$ ls ..../static/
base.css  bootstrap
```

Zwrć uwagę, że zamiast użyć polecenia activate środowiska wirtualnego, posługujemy się bezpośrednią ścieżką dostępu do Pythona zainstalowanego w wymienionym środowisku.

Teraz można nakazać serwerowi Nginx obsługę plików statycznych.

Plik `/etc/nginx/sites-available/superlists-staging.ottg.eu` w serwerze:

```
server {  
    listen 80;  
    server_name superlists-staging.ottg.eu;  
  
    location /static {  
        alias /home/edyta/sites/superlists-staging.ottg.eu/static;  
    }  
  
    location / {  
        proxy_pass http://localhost:8000;  
    }  
}
```

Ponownie wczytujemy konfigurację Nginx i ponownie uruchamiamy Gunicorn...

```
edyta@serwer:$ sudo service nginx reload  
edyta@serwer:$ ../virtualenv/bin/gunicorn superlists.wsgi:application
```

Po raz kolejny spójrzmy na witrynę. Sprawy wyglądają już znacznie lepiej. Możemy więc ponownie wykonać testy funkcjonalne:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu  
Creating test database for alias 'default'...  
--  
Ran 2 tests in 10.718s  
  
OK  
Destroying test database for alias 'default'...
```

Użycie gniazd systemu Unix

Kiedy mamy prowizoryczną i rzeczywistą wersję witryny, oba serwery nie mogą używać tego samego portu 8000. Wprawdzie można zdecydować się na wykorzystanie różnych numerów portów, ale to niewygodne rozwiązanie. Ponadto niebezpiecznie łatwo można się pomylić i uruchomić serwer prowizoryczny na porcie rzeczywistego i na odwrót.

Lepszym rozwiązaniem będzie użycie gniazd domeny systemu Unix. To są pliki na dysku, które mogą być wykorzystywane przez Nginx i Gunicorn do wzajemnej komunikacji. Wspomniane gniazda umieścimy w katalogu `/tmp`. Przystępujemy do zmiany ustawień proxy w Nginx.

Plik `/etc/nginx/sites-available/superlists-staging.ottg.eu` w serwerze:

```
[...]  
    location / {  
        proxy_set_header Host $host;  
        proxy_pass http://unix:/tmp/superlists-staging.ottg.eu.socket;  
    }  
}
```

Dyrektywa `proxy_set_header` jest używana w celu zagwarantowania, że Gunicorn i Django znają domenę, w której zostały uruchomione. To jest konieczne do działania funkcji zabezpieczeń o nazwie `ALLOWED_HOSTS`, którą zamierzamy włączyć.

Teraz możemy ponownie uruchomić Gunicorn, ale tym razem nakazujemy nasłuchiwanie gniazda zamiast portu domyślnego:

```
edyta@serwer:$ sudo service nginx reload
edyta@serwer:$ ../virtualenv/bin/gunicorn --bind \
    unix:/tmp/superlists-staging.ottg.eu.socket superlists.wsgi:application
```

Ponownie wykonujemy testy funkcjonalne, aby się upewnić, że nadal są zaliczane:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
OK
```

Udało nam się przejść kilka dodatkowych kroków naprzód!

Przypisanie opcji DEBUG wartości False i ustawienie ALLOWED_HOSTS

Tryb debugowania w Django jest niezwykle użyteczny podczas pracy z własnym serwerem, ale udostępnienie w rzeczywistej witrynie w internecie generowanych przez niego stron *jest niebezpieczne*⁵.

Opcję `DEBUG` znajdziesz na początku pliku `settings.py`. Przypisujemy jej wartość `False`, a ponadto konfigurujemy opcję o nazwie `ALLOWED_HOSTS`. Wymieniona opcja została w Django 1.5 dodana jako funkcja zabezpieczeń⁶. Niestety w domyślnym pliku `settings.py` nie umieszczone zostało poświęconego jej użytkownika komentarza, ale możemy to zrobić samodzielnie. Poniższe zmiany wprowadź w serwerze.

Plik `superlists/settings.py` w serwerze:

```
# OSTRZEŻENIE BEZPIECZEŃSTWA: nie pozostawiaj włączonej opcji DEBUG w środowisku produkcyjnym!
DEBUG = False

TEMPLATE_DEBUG = DEBUG

# Opcja wymagana w przypadku użycia DEBUG=False.
ALLOWED_HOSTS = ['superlists-staging.ottg.eu']
[...]
```

Raz jeszcze przeprowadzamy ponowne uruchomienie Gunicorn i ponownie wykonujemy testy funkcjonalne, aby sprawdzić, czy wszystko działa.



Nie przekazuj do repozytorium zmian wprowadzonych w serwerze. Na obecnym etapie to jedynie sztuczka pozwalająca na działanie aplikacji, a nie zmiana, którą chcemy zachować w repozytorium. Ogólnie rzecz biorąc, w celu zachowania prostoty pliki do repozytorium Git przekazuję jedynie z poziomu komputera lokalnego, a następnie używam poleceń `git push` i `git pull`, gdy chcę je zsynchronizować z serwerem.

⁵ <https://docs.djangoproject.com/en/1.7/ref/settings/#debug>

⁶ https://docs.djangoproject.com/en/1.7/ref/settings/#std:setting-ALLOWED_HOSTS

Użycie Upstart do uruchamiania Gunicorn wraz z systemem

Ostatnim krokiem jest zagwarantowanie automatycznego uruchamiania Gunicorn wraz z systemem oraz jego ponownego uruchamiania po ewentualnej awarii. W systemie Ubuntu wykorzystujemy Upstart.

Plik `/etc/init/gunicorn-superlists-staging.ottg.eu.conf` w serwerze:

```
description "Serwer Gunicorn dla superlists-staging.ottg.eu"

start on net-device-up #❶
stop on shutdown

respawn #❷

setuid edyta #❸
chdir /home/edyta/sites/superlists-staging.ottg.eu/source #❹

exec ../virtualenv/bin/gunicorn \
--bind unix:/tmp/superlists-staging.ottg.eu.socket \
superlists.wsgi:application
```

Przeprowadzanie konfiguracji Upstart jest przyjemnością (zwłaszcza jeśli kiedykolwiek miałeś wątpliwą przyjemność tworzenia skryptu `init.d`), a znaczenie poszczególnych wierszy łatwe do określenia:

- ❶ Polecenie `start on net-device-up` gwarantuje, że uruchomienie Gunicorn nastąpi tylko wtedy, gdy cały serwer nawiąże połączenie z internetem.
- ❷ Polecenie `respawn` powoduje automatyczne ponowne uruchomienie procesu, jeśli ulegnie on awarii.
- ❸ Polecenie `setuid` powoduje, że proces jest uruchomiony w ramach konta użytkownika `edyta`.
- ❹ Polecenie `chdir` powoduje ustawienie katalogu roboczego.
- ❺ Polecenie `exec` wskazuje rzeczywisty proces do uruchomienia.

Skrypty Upstart znajdują się w katalogu `/etc/init`, a ich nazwy muszą kończyć się rozszerzeniem `.conf`.

Teraz można uruchomić serwer Gunicorn za pomocą polecenia `start`:

```
edyta@serwer:$ sudo start gunicorn-superlists-staging.ottg.eu
```

Po ponownym wykonaniu testów funkcjonalnych przekonasz się, że wszystko nadal działa. Możesz nawet sprawdzić, czy witryna internetowa będzie dostępna po ponownym uruchomieniu serwera!

Zachowanie wprowadzonych zmian — dodanie Gunicorn do pliku requirements.txt

Powracamy do *lokalnej* kopii repozytorium i dodajemy Gunicorn do listy pakietów niezbędnych w środowisku wirtualnym:

```
$ source ../virtualenv/bin/activate # O ile konieczne.
(virtualenv)$ pip install gunicorn
(virtualenv)$ pip freeze > requirements.txt
(virtualenv)$ deactivate
$ git commit -am "Dodanie gunicorn do wymagań virtualenv."
$ git push
```



W takcie pisania książki instalacja Gunicorn w systemie Windows przez narzędzie pip kończy się powodzeniem, ale serwer faktycznie nie działa. Na szczęście z Gunicorn korzystamy jedynie w serwerze, więc to nie stanowi problemu. Obsługa Gunicorn w Windows nadal pozostaje przedmiotem dyskusji⁷ ...

Automatyzacja

Przypomnę teraz procedury przygotowania i wdrożenia aplikacji.

Przygotowanie

1. Upewniamy się o posiadaniu konta użytkownika i katalogu domowego.
2. Wydajemy polecenie `apt-get nginx git python-pip`.
3. Wydajemy polecenie `pip install virtualenv`.
4. Dodajemy konfigurację Nginx dla komputera wirtualnego.
5. Dodajemy zadanie Upstart dla Gunicorn.

Wdrożenie

1. Tworzymy strukturę katalogów w `~/sites`.
2. Pobieramy kod źródłowy i umieszczamy w katalogu o nazwie `source`.
3. Uruchamiamy `virtualenv` w katalogu `./virtualenv`.
4. Wydajemy polecenie `pip install -r requirements.txt`.
5. Wydajemy polecenie `manage.py migrate` w celu przygotowania bazy danych.
6. Wydajemy polecenie `collectstatic` w celu zebrania plików statycznych.
7. W pliku `settings.py` ustawiamy opcje `DEBUG = False` i `ALLOWED_HOSTS`.
8. Ponownie uruchamiamy zadanie Gunicorn.
9. Wykonujemy testy funkcjonalne, aby sprawdzić, czy wszystko działa.

Przy założeniu, że nie jesteśmy jeszcze gotowi na pełną automatyzację procedury przygotowania, rodzi się pytanie o sposób, w jaki powinniśmy zapisać osiągnięte dotąd wyniki. Pliki konfiguracyjne Nginx i Upstart są zapisane w pewnych miejscach, co ułatwia ich późniejsze ponowne wykorzystanie. Umieśćmy je jednak także w nowym podkatalogu naszego repozytorium:

```
$ mkdir deploy_tools
```

Plik `deploy_tools/nginx.template.conf`:

```
server {
    listen 80;
    server_name SITENAME;

    location /static {
        alias /home/edyta/sites/SITENAME/static;
    }
}
```

⁷ <http://stackoverflow.com/questions/11087682/does-gunicorn-run-on-windows>

```
        location / {
            proxy_set_header Host $host;
            proxy_pass http://unix:/tmp/SITENAME.socket;
        }
    }
```

Plik *deploy_tools/gunicorn-upstart.template.conf*:

```
description "Serwer Gunicorn dla SITENAME"

start on net-device-up
stop on shutdown

respawn

setuid edyta
chdir /home/edyta/sites/SITENAME/source

exec ../virtualenv/bin/gunicorn \
--bind unix:/tmp/SITENAME.socket \
superlists.wsgi:application
```

Wygenerowanie nowej witryny za pomocą dwóch powyższych plików jest łatwym zadaniem, wystarczy znaleźć i zastąpić SITENAME.

Na potrzeby testu dodamy kilka uwag, które również warto umieścić w repozytorium.

Plik *deploy_tools/provisioning_notes.md*:

```
Przygotowanie nowej witryny
=====
```

```
## Wymagane pakiety:
* nginx
* Python 3
* Git
* pip
* virtualenv
```

Na przykład w systemie Ubuntu należy wydać polecenia:
sudo apt-get install nginx git python3 python3-pip
sudo pip3 install virtualenv

```
## Konfiguracja wirtualnych hostów w Nginx
```

* Zobacz plik *nginx.template.conf*.
* SITENAME należy zastąpić odpowiednią nazwą, na przykład *staging.my-domain.com*.

```
## Zadanie Upstart
```

* Zobacz plik *gunicorn-upstart.template.conf*.
* SITENAME należy zastąpić odpowiednią nazwą, na przykład *staging.my-domain.com*.

```
## Struktura katalogów:
```

Przymajemy założenie o istnieniu konta użytkownika w postaci */home/użytkownik*.

```
/home/użytkownik
sites
    SITENAME
        database
        source
        static
        virtualenv
```

Teraz pliki możemy przekazać do repozytorium:

```
$ git add deploy_tools  
$ git status # Polecenie powinno wyświetlić trzy nowe pliki.  
$ git commit -m "Notatki i szablonowe pliki konfiguracyjne dla procedury przygotowania."
```

Struktura drzewa katalogów powinna na obecnym etapie przedstawiać się następująco:

```
$ tree -I __pycache__  
.  
    deploy_tools  
        gunicorn-upstart.template.conf  
        nginx.template.conf  
        provisioning_notes.md  
    functional_tests  
        __init__.py  
        [...]  
    lists  
        __init__.py  
        models.py  
        static  
            base.css  
            [...]  
        templates  
            base.html  
            [...]  
    manage.py  
    requirements.txt  
    superlists  
        [...]
```

Sterowana testami konfiguracja serwera i wdrożenie

Testy nieco zmniejszają poziom niepewności co do wyniku wdrożenia

Dla programistów administracja serwerem zawsze była „zabawna”, co oznacza, że ten proces jest pełen niepewności i niespodzianek. W tym rozdziale chciałem pokazać, jak zestaw testów funkcjonalnych może wyeliminować pewien poziom niepewności z procesu wdrażania.

Typowe obszary sprawiające problemy — baza danych, pliki statyczne, zależności i ustawienia niestandardowe

Aspekty, na które warto mieć oko podczas praktycznej każdej procedury wdrożenia, to między innymi konfiguracja bazy danych, pliki statyczne, zależności oprogramowania oraz niestandardowe ustawienia różniące się w środowiskach programistycznym oraz produkcyjnym. Dokonując wdrożenia, powinieneś zwracać baczną uwagę na wymienione kwestie.

Testy pozwalają na eksperymenty

Po wprowadzeniu zmiany w konfiguracji serwera można ponownie wykonać zestaw testów i upewnić się, że wszystko działa dokładnie tak samo jak wcześniej. Testy pozwalają więc na przeprowadzanie bez obaw eksperymentów z konfiguracją.

Zachowanie informacji o postępie

Możliwość wykonania testów funkcjonalnych względem prowizorycznego serwera może być niezwykle uspokajająca. Jednak w większości przypadków nie chcesz wykonywać testów funkcjonalnych względem „rzeczywistego” serwera. Aby zachować informacje o postępie i mieć pewność, że serwer produkcyjny będzie działał równie dobrze jak rzeczywisty, konieczne jest zapewnienie powtarzalności procesu wdrożenia.

Odpowiedzią na wspomniane zapotrzebowanie będzie automatyzacja, która jest tematem kolejnego rozdziału.

Zautomatyzowane wdrożenie za pomocą Fabric

Automatyzacja, automatyzacja, automatyzacja.

— Cay Horstman

Automatyzacja wdrożenia ma znaczenie krytyczne, aby testy witryny prowizorycznej miały jakiekolwiek znaczenie. Zapewniając powtarzalność procedury wdrożenia, zyskujemy gwarancję, że wszystko przebiegnie dobrze, gdy aplikacja będzie wdrażana w środowisku produkcyjnym.

Fabric to narzędzie pozwalające na automatyzację poleceń, które mają być wykonywane w serwerach. Wymienione narzędzie możesz zainstalować jako dostępne w całym systemie, nie stanowi fragmentu budowanej przez nas witryny. Nie trzeba więc go umieszczać w środowisku wirtualnym i wymieniać w pliku *requirements.txt*. Dlatego też w komputerze lokalnym wydaj poniższe polecenie:

```
$ pip2 install fabric
```



W trakcie pisania książki narzędzie Fabric nie zostało jeszcze przystosowane do obsługi Pythona 3 i dlatego użyjemy wersji dla Pythona 2. Na szczęście kod Fabric jest zupełnie oddzielny od pozostałej części kodu naszego projektu i wspomniana niedogodność nie stanowi żadnego problemu.

Standardowa konfiguracja opiera się na pliku *fabfile.py* zawierającym jedną lub więcej funkcji, które następnie można wywoływać z poziomu narzędzia wiersza poleceń o nazwie fab, na przykład w następujący sposób:

```
fab nazwa_funkcji,host=ADRES_SERWERA
```

Powyższe polecenie spowoduje wywołanie funkcji *nazwa_funkcji()* i przekazanie połączenia z serwerem *ADRES_SERWERA*. Istnieje wiele innych opcji przeznaczonych do wskazania nazwy użytkownika i hasła. Dostępne opcje zostaną wyświetcone po wydaniu polecenia *fab --help*.

Instalacja Fabric w Windows

Fabric zależy od pycrypto, który jest pakietem wymagającym komplikacji. Kompilacja w systemie Windows to proces najeżony trudnościami. Znacznie lepszym rozwiązaniem jest wykorzystanie prekompilowanych plików binarnych udostępnionych przez dobrą duszę. W omarwianym przypadku Michael Foord¹ udostępnił² pliki binarne pycrypto dla systemu Windows. (Nie zapomnij w tym miejscu zachichotać, wspominając absurdalne przepisy USA dotyczące kontroli eksportu technologii szyfrujących).

Poniżej przedstawiono procedurę dla systemu Windows:

1. Z podanego wcześniej adresu URL pobierz pycrypto, a następnie zainstaluj bibliotekę.
2. Wydaj polecenie `pip install fabric`.

Inne doskonałe źródło prekompilowanych dla Windows pakietów Pythona jest oferowane przez *Christopha Gohlke*³.

Analiza skryptu Fabric dla naszego wdrożenia

Najlepszym sposobem poznania sposobu działania narzędzia Fabric jest użycie przykładu. *Tużaj*⁴ znajdziesz przygotowany przeze mnie wcześniej przykład automatyzacji wszystkich omówionych kroków procedury wdrożenia. Funkcja główna nosi nazwę `deploy()` i jest wywoływana z poziomu powłoki. Jej działanie opiera się na kilku funkcjach pomocniczych. Zmienna `env.host` przechowuje adres używanego serwera.

Plik `deploy_tools/fabfile.py`:

```
from fabric.contrib.files import append, exists, sed
from fabric.api import env, local, run
import random

REPO_URL = 'https://github.com/hjwp/book-example.git' #❶

def deploy():
    site_folder = '/home/%s/sites/%s' % (env.user, env.host) #❷❸
    source_folder = site_folder + '/source'
    _create_directory_structure_if_necessary(site_folder)
    _get_latest_source(source_folder)
    _update_settings(source_folder, env.host)
    _update_virtualenv(source_folder)
    _update_static_files(source_folder)
    _update_database(source_folder)
```

- ❶ W zmiennej `REPO_URL` umieść adres URL Twojego repozytorium Git, które założyłeś w witrynie pozwalającej na dzielenie się kodem.
- ❷ Zmienna `env.host` zawiera adres serwera podanego w powłoce, na przykład `superlists.ottg.eu`.
- ❸ Zmienna `env.user` zawiera nazwę użytkownika, którego konto jest wykorzystywane w celu zalogowania się do serwera.

¹ Autor biblioteki Mock oraz osoba odpowiedzialna za moduł `unittest`. Jeżeli świat testów w Pythonie mógłby mieć gwiazdę, Michael na pewno bynią był.

² <http://www.voidspace.org.uk/python/modules.shtml#pycrypto>

³ <http://www.lfd.uci.edu/~gohlke/pythonlibs/>

⁴ <http://www.bbc.co.uk/cult/classic/blue peter/valpete john/trivia.shtml>

Na szczęście nazwy wspomnianych funkcji pomocniczych jasno wskazują ich przeznaczenie. Ponieważ teoretycznie dowolna funkcja zdefiniowana w pliku *fabfile.py* może być wywołana z powłoki, zastosowałem konwencję poprzedzenia znakiem podkreślenia nazw tych funkcji, które nie mają być częścią „publicznego API” wymienionego pliku. Tutaj funkcje zostały podane w kolejności chronologicznej.

Poniższa funkcja jest przeznaczona do utworzenia struktury katalogów, ale w taki sposób, aby jej działanie nie zakończyło się niepowodzeniem, gdy dany katalog już istnieje.

Plik *deploy_tools/fabfile.py*:

```
def _create_directory_structure_if_necessary(site_folder):
    for subfolder in ('database', 'static', 'virtualenv', 'source'):
        run('mkdir -p %s/%s' % (site_folder, subfolder)) #❶❷
```

- ❶ Najczęściej używanym poleceniem Fabric jest `run`. Oznacza „wykonaj w serwerze to polecenie powłoki”.
- ❷ Polecenie `mkdir -p` to użytkczna odmiana polecenia `mkdir`, choć lepsza pod dwoma względami. Po pierwsze, pozwala na utworzenie katalogów na wielu poziomach zagnieżdżenia. Po drugie, katalogi są tworzone tylko wtedy, gdy jeszcze nie istnieją. Dlatego też polecenie `mkdir -p /tmp/foo/bar` powoduje utworzenie nie tylko katalogu *bar*, ale również jego katalogu nadzawanego *foo*, o ile zachodzi potrzeba. Nie otrzymasz komunikatu informującego o istnieniu katalogu *bar*⁵.

Kolejnym krokiem jest pobranie kodu źródłowego.

Plik *deploy_tools/fabfile.py*:

```
def _get_latest_source(source_folder):
    if exists(source_folder + '/.git'): #❶
        run('cd %s && git fetch' % (source_folder,)) #❷❸
    else:
        run('git clone %s %s' % (REPO_URL, source_folder)) #❹
    current_commit = local("git log -n 1 --format=%H", capture=True) #❺
    run('cd %s && git reset --hard %s' % (source_folder, current_commit)) #❻
```

- ❶ Polecenie `exists` sprawdza, czy katalog lub plik już istnieje w serwerze. Sprawdzane jest także istnienie ukrytego katalogu *.git*, ponieważ jego obecność wskazuje na klonowanie repozytorium w danym katalogu.
- ❷ Wiele poleceń rozpoczyna się od `cd` w celu zmiany bieżącego katalogu roboczego. Fabric nie posiada żadnego stanu, a więc nie pamięta odwiedzanych przez Ciebie katalogów⁶.
- ❸ Wydanie polecenia `git fetch` w istniejącym repozytorium powoduje pobranie z sieci ostatnio przekazanych plików do repozytorium.
- ❹ Alternatywne rozwiążanie polega na użyciu `git clone` wraz z adresem URL repozytorium i tym samym pobranie świeżego drzewa kodu źródłowego.
- ❺ Polecenie `local` narzędzia Fabric powoduje wykonanie polecenia `git log` w komputerze lokalnym. Tak naprawdę polecenie `local` jest rodzajem wygodnego opakowania dla wywołania `subprocess.Popen()`. W omawianym przykładzie przechwytyujemy dane wyjściowe

⁵ Jeżeli zastanawiasz się, dlaczego ścieżki dostępu są tworzone ręcznie za pomocą `%s` zamiast przedstawionego wcześniej polecenia `os.path.join`, to wyjaśniam, że `path.join` zastosuje lewe ukośniki po uruchomieniu skryptu w Windows. Na pewno nie chcemy, aby w serwerze stosowane były lewe ukośniki.

⁶ Istnieje jeszcze polecenie `cd` dodane przez Fabric, ale uznałem je za jedną z wielu rzeczy, których nie zdąział omówić w rozdziale.

wywołania git log w celu pobrania wartości hash bieżącej operacji przekazania plików do repozytorium w lokalnym drzewie katalogów. Oznacza to, że serwer będzie zawierał ten sam kod, który znajduje się w Twoim komputerze (o ile oczywiście przekazałeś kod do serwera).

- ❶ Polecenie reset --hard powoduje usunięcie wszystkich bieżących zmian w znajdująącym się w serwerze katalogu kodu.



Aby przedstawiony skrypt działał, konieczne jest wydanie polecenia git push w komputerze lokalnym, co pozwoli serwerowi na pobranie kodu i wykonanie zerowania. Jeżeli otrzymasz komunikat błędu informujący o braku możliwości przetworzenia obiektu, spróbuj wydać polecenie git push.

Następnie uaktualniamy plik ustawień, konfigurujemy opcje ALLOWED_HOSTS i DEBUG oraz tworzymy nowy klucz prywatny.

Plik `deploy_tools/fabfile.py`:

```
def _update_settings(source_folder, site_name):
    settings_path = source_folder + '/superlists/settings.py'
    sed(settings_path, "DEBUG = True", "DEBUG = False") #❶
    sed(settings_path,
        'ALLOWED_HOSTS = .+$',
        'ALLOWED_HOSTS = [%s]"' % (site_name,) #❷
    )
    secret_key_file = source_folder + '/superlists/secret_key.py'
    if not exists(secret_key_file): #❸
        chars = 'abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*(-_=+)'
        key = ''.join(random.SystemRandom().choice(chars) for _ in range(50))
        append(secret_key_file, "SECRET_KEY = '%s'" % (key,))
        append(settings_path, '\nfrom .secret_key import SECRET_KEY') #❹❺
```

- ❶ Polecenie sed w Fabric przeprowadza operację zastępowania ciągu tekstowego w pliku. Tutaj jest odpowiedzialne za zmianę wartości opcji DEBUG z True na False.
- ❷ W tym wierszu modyfikujemy opcję ALLOWED_HOSTS, dopasowując odpowiedni wiersz za pomocą wyrażenia regularnego.
- ❸ Do pewnych zadań wymagających szyfrowania, na przykład obsługi plików cookies oraz zabezpieczania CSRF, Django używa SECRET_KEY. Dobrą praktyką jest upewnienie się, że klucz prywatny w serwerze pozostaje inny niż (prawdopodobnie publiczny) w repozytorium kodu źródłowego. Ten kod spowoduje wygenerowanie nowego klucza przeznaczonego do zainportowania w ustawieniach, o ile jeszcze nie posiadasz klucza. (Gdy masz już klucz, powiniens on pozostać taki sam w poszczególnych wdrożeniach). Więcej informacji na ten temat znajdziesz w *dokumentacji Django*⁷.
- ❹ Wywołanie append() dodaje wiersz na końcu pliku. (Nie ma sensu przejmować się, jeśli pusty wiersz znajduje się już na końcu pliku. Natomiast niezbyt rozsądne będzie pominięcie jego dodania, jeśli na końcu pliku brakuje pustego wiersza. Stąd ten kod).
- ❺ W przykładzie użyto *względnego importu* (from .secret_key zamiast from secret_key), aby mieć absolutną pewność, że importowany jest moduł lokalny, a nie znajdujący się gdzieś w położeniu wskazanym przez sys.path. W kolejnym rozdziale dowieš się nieco więcej o względnych importach.

⁷ <https://docs.djangoproject.com/en/1.7/topics/signing/>



Niektórzy, na przykład autorzy doskonałej książki *Two Scoops of Django*, sugerują użycie zmiennych środowiskowych podczas konfiguracji elementów takich jak klucze prywatne. Powinieneś stosować takie rozwiązanie, które jest najbezpieczniejsze w Twoim środowisku.

Kolejnym krokiem jest utworzenie lub uaktualnienie środowiska wirtualnego.

Plik `deploy_tools/fabfile.py`:

```
def _update_virtualenv(source_folder):
    virtualenv_folder = source_folder + '/../virtualenv'
    if not exists(virtualenv_folder + '/bin/pip'): #❶
        run('virtualenv --python=python3 %s' % (virtualenv_folder,))
    run('%s/bin/pip install -r %s/requirements.txt' % ( #❷
        virtualenv_folder, source_folder
    ))
```

- ❶ Zagładamy do katalogu `virtualenv` w poszukiwaniu plików wykonywalnych dla pip. W ten sposób sprawdzamy, czy faktycznie istnieją.
- ❷ Następnie jak wcześniej używamy polecenia `pip install -r`.

Uaktualnienie plików statycznych sprowadza się do pojedynczego polecenia.

Plik `deploy_tools/fabfile.py`:

```
def _update_static_files(source_folder):
    run('cd %s && ../virtualenv/bin/python3 manage.py collectstatic --noinput' % ( #❶
        source_folder,
    ))
```

- ❶ Katalog plików binarnych środowiska wirtualnego jest używany, gdy zachodzi potrzeba wykonania polecenia Django `manage.py`. Gwarantujemy tym samym użycie wersji Django w środowisku wirtualnym, a nie w systemie komputera.

Na końcu uaktualniamy bazę danych za pomocą polecenia `manage.py migrate`.

Plik `deploy_tools/fabfile.py`:

```
def _update_database(source_folder):
    run('cd %s && ../virtualenv/bin/python3 manage.py migrate --noinput' % (
        source_folder,
    ))
```

Wypróbowanie rozwiązania

Przygotowane rozwiązanie możemy wypróbować na istniejącej witrynie prowizorycznej — skrypt powinien działać dla witryny zarówno istniejącej, jak i nowej. Jeżeli lubisz słowa z łacińskimi korzeniami, skrypt możesz określić mianem idempotentnego, co oznacza, że nic się nie stanie po jego dwukrotnym uruchomieniu...

```
$ cd deploy_tools
$ fab deploy:host=edyta@superlists-staging.ottg.eu

[superlists-staging.ottg.eu] Executing task 'deploy'
[superlists-staging.ottg.eu] run: mkdir -p /home/edyta/sites/superlists-staging
```

```
[superlists-staging.ottg.eu] run: cd /home/edyta/sites/superlists-staging.ottg
[localhost] local: git log -n 1 --format=%H
[superlists-staging.ottg.eu] run: cd /home/edyta/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out: HEAD is now at 85a6c87 Add a fabfile for autom
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: sed -i.bak -r -e 's/DEBUG = True/DEBUG = False
[superlists-staging.ottg.eu] run: echo 'ALLOWED_HOSTS = ["superlists-staging.ott
[superlists-staging.ottg.eu] run: echo 'SECRET_KEY = '\\''4p2u8fi6)b1tep(6nd_3tt
[superlists-staging.ottg.eu] run: echo 'from .secret_key import SECRET_KEY' >> "
[superlists-staging.ottg.eu] run: /home/edyta/sites/superlists-staging.ottg.eu
[superlists-staging.ottg.eu] out: Requirement already satisfied (use --upgrade t
[superlists-staging.ottg.eu] out: Requirement already satisfied (use --upgrade t
[superlists-staging.ottg.eu] out: Cleaning up...
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: cd /home/edyta/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out:
[superlists-staging.ottg.eu] out: 0 static files copied, 11 unmodified.
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: cd /home/edyta/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out: Creating tables ...
[superlists-staging.ottg.eu] out: Installing custom SQL ...
[superlists-staging.ottg.eu] out: Installing indexes ...
[superlists-staging.ottg.eu] out: Installed 0 object(s) from 0 fixture(s)
[superlists-staging.ottg.eu] out:
Done.
Disconnecting from superlists-staging.ottg.eu... done.
```

Wspaniale. Uwielbiam, gdy komputery generują tak dużą ilość danych wyjściowych. Jeżeli dokładnie przeanalizujesz przedstawione dane wyjściowe, to zauważysz wykonywane operacje. Polecenie `mkdir -p` tworzy katalogi i działa doskonale, nawet jeśli katalogi istnieją. Następnie polecenie `git pull` pobiera kod z kilku operacji przekazania kodu do repozytorium. Z kolei polecenia `sed` i `echo >>` modyfikują plik `settings.py`. Dalej mamy zakończone powodzeniem wywołanie `pip3 install -r requirements.txt`, komunikat informujący, że istniejące środowisko wirtualne zawiera wszystkie niezbędne pakiety. Polecenie `collectstatic` wskazuje, że pliki statyczne są gotowe, a całość kończy wywołanie `migrate`.

Konfiguracja Fabric

Jeżeli do zalogowania używasz klucza SSH i przechowujesz go w położeniu domyślnym, a także używasz w serwerze takiej samej nazwy użytkownika jak w komputerze lokalnym, wówczas narzędzie Fabric powinno „po prostu działać”. W przeciwnym razie mamy kilka możliwości zmuszenia polecenia `fab` do działania zgodnie z oczekiwaniami. Obracają się one wokół nazwy użytkownika, położenia używanego klucza SSH oraz hasła.

Wspomniane dane możesz przekazać narzędziu Fabric za pomocą powłoki. Sprawdź to:

```
$ fab --help
```

Więcej informacji znajdziesz w dokumentacji⁸ Fabric.

⁸ <http://docs.fabfile.org/en/1.10/>

Wdrożenie w środowisku produkcyjnym

Omówiony skrypt wypróbowujemy teraz w środowisku produkcyjnym!

```
$ fab deploy:host=edyta@superlists.ottg.eu
$ fab deploy --host=superlists.ottg.eu

[superlists.ottg.eu] Executing task 'deploy'
[superlists.ottg.eu] run: mkdir -p /home/edyta/sites/superlists.ottg.eu
[superlists.ottg.eu] run: mkdir -p /home/edyta/sites/superlists.ottg.eu/database
[superlists.ottg.eu] run: mkdir -p /home/edyta/sites/superlists.ottg.eu/static
[superlists.ottg.eu] run: mkdir -p /home/edyta/sites/superlists.ottg.eu/virtualenv
[superlists.ottg.eu] run: mkdir -p /home/edyta/sites/superlists.ottg.eu/source
[superlists.ottg.eu] run: git clone https://github.com/hjwp/book-example.git /home/edyta/sites/superlists.ottg.eu/source
[superlists.ottg.eu] out: Cloning into '/home/edyta/sites/superlists.ottg.eu/source'...
[superlists.ottg.eu] out: remote: Counting objects: 3128, done.
[superlists.ottg.eu] out: Receiving objects:   0% (0/3128)
[superlists.ottg.eu] out: ...
[superlists.ottg.eu] out: Receiving objects: 100% (3128/3128), 2.60 MiB | 829 KiB/s
[superlists.ottg.eu] out: Resolving deltas: 100% (1545/1545), done.
[superlists.ottg.eu] out:

[localhost] local: git log -n 1 --format=%H
[superlists.ottg.eu] run: cd /home/edyta/sites/superlists.ottg.eu/source && git pull
[superlists.ottg.eu] out: HEAD is now at 6c8615b use a secret key file
[superlists.ottg.eu] out:

[superlists.ottg.eu] run: sed -i.bak -r -e 's/DEBUG = True/DEBUG = False/g' "$(_env_file)"
[superlists.ottg.eu] run: echo 'ALLOWED_HOSTS = ["superlists.ottg.eu"]' >> "$(_env_file)"
[superlists.ottg.eu] run: echo 'SECRET_KEY = '\\"mqu(ffd5vleol%ke^jil*x1mkj-4' >> "$(_env_file)"
[superlists.ottg.eu] run: echo 'from .secret_key import SECRET_KEY' >> "$(_env_file)"
[superlists.ottg.eu] run: virtualenv --python=python3 /home/edyta/sites/superlists.ottg.eu
[superlists.ottg.eu] out: Already using interpreter /usr/bin/python3
[superlists.ottg.eu] out: Using base prefix '/usr'
[superlists.ottg.eu] out: New python executable in /home/edyta/sites/superlists.ottg.eu/virtualenv
[superlists.ottg.eu] out: Also creating executable in /home/edyta/sites/superlists.ottg.eu/virtualenv/bin/python
[superlists.ottg.eu] out: Installing Setuptools.....done.
[superlists.ottg.eu] out: Installing Pip.....done.
[superlists.ottg.eu] out:

[superlists.ottg.eu] run: /home/edyta/sites/superlists.ottg.eu/source/../.virtualenv/bin/pip install -t /home/edyta/sites/superlists.ottg.eu/source/../.virtualenv/src -r /home/edyta/sites/superlists.ottg.eu/source/requirements.txt
[superlists.ottg.eu] out: Downloading/unpacking Django==1.7 (from -r /home/edyta/sites/superlists.ottg.eu/source/requirements.txt)
[superlists.ottg.eu] out:   Downloading Django-1.7.tar.gz (8.0MB):
[superlists.ottg.eu] out: ...
[superlists.ottg.eu] out:   Downloading Django-1.7.tar.gz (8.0MB): 100% 8.0MB
[superlists.ottg.eu] out:   Running setup.py egg_info for package Django
[superlists.ottg.eu] out:
[superlists.ottg.eu] out:     warning: no previously-included files matching '.*' found under directory 'docs/_build'
[superlists.ottg.eu] out:     warning: no previously-included files matching '.*' found under directory 'tests/_test'
[superlists.ottg.eu] out:   Downloading/unpacking gunicorn==17.5 (from -r /home/edyta/sites/superlists.ottg.eu/source/requirements.txt)
[superlists.ottg.eu] out:   Downloading gunicorn-17.5.tar.gz (367kB): 100% 367kB
[superlists.ottg.eu] out: ...
[superlists.ottg.eu] out:   Downloading gunicorn-17.5.tar.gz (367kB): 367kB down
[superlists.ottg.eu] out:   Running setup.py egg_info for package gunicorn
[superlists.ottg.eu] out:
[superlists.ottg.eu] out:     Installing collected packages: Django, gunicorn
[superlists.ottg.eu] out:     Running setup.py install for Django
[superlists.ottg.eu] out:       changing mode of build/scripts-3.3/django-admin.py
[superlists.ottg.eu] out:
[superlists.ottg.eu] out:     warning: no previously-included files matching '.*'
```

```
[superlists.ottg.eu] out:      warning: no previously-included files matching '*'.
[superlists.ottg.eu] out:      changing mode of /home/edyta/sites/superlists.ot
[superlists.ottg.eu] out:      Running setup.py install for gunicorn
[superlists.ottg.eu] out:
[superlists.ottg.eu] out:      Installing gunicorn_paster script to /home/edyta
[superlists.ottg.eu] out:      Installing gunicorn script to /home/edyta/sites/
[superlists.ottg.eu] out:      Installing gunicorn_django script to /home/edyta
[superlists.ottg.eu] out: Successfully installed Django gunicorn
[superlists.ottg.eu] out: Cleaning up...
[superlists.ottg.eu] out:

[superlists.ottg.eu] run: cd /home/edyta/sites/superlists.ottg.eu/source && ..
[superlists.ottg.eu] out: Copying '/home/edyta/sites/superlists.ottg.eu/source'
[superlists.ottg.eu] out: Copying '/home/edyta/sites/superlists.ottg.eu/source'
[...]
[superlists.ottg.eu] out: Copying '/home/edyta/sites/superlists.ottg.eu/source'
[superlists.ottg.eu] out:
[superlists.ottg.eu] out: 11 static files copied.
[superlists.ottg.eu] out:

[superlists.ottg.eu] run: cd /home/edyta/sites/superlists.ottg.eu/source && ..
[superlists.ottg.eu] out: Creating tables ...
[superlists.ottg.eu] out: Creating table auth_permission
[...]
[superlists.ottg.eu] out: Creating table lists_item
[superlists.ottg.eu] out: Installing custom SQL ...
[superlists.ottg.eu] out: Installing indexes ...
[superlists.ottg.eu] out: Installed 0 object(s) from 0 fixture(s)
[superlists.ottg.eu] out:
Done.
Disconnecting from superlists.ottg.eu... done.
```

Jak możesz zobaczyć, skrypt działa nieco inaczej. Zamiast git pull wykonywane jest polecenie git clone w celu pobrania zupełnie nowego repozytorium. Konieczne jest również skonfigurowanie całkiem od początku nowego środowiska wirtualnego, łącznie ze świeżą instalacją pip i Django. Polecenie collectstatic tym razem tworzy nowe pliki, polecenie migrate również działa.

Pliki konfiguracyjne Nginx i Gunicorn odtworzone za pomocą sed

Co jeszcze musimy zrobić, aby nasza rzeczywista witryna znalazła się w środowisku produkcyjnym? Powracamy do wcześniej przygotowanych notatek, z których wynika, że możemy użyć przygotowanych wcześniej plików szablonów do utworzenia wirtualnych hostów Nginx oraz skryptu Upstart. Wykorzystamy więc odrobinę magii powłoki systemu Unix:

```
edyta@serwer:~$ sed "s/SITENAME/superlists.ottg.eu/g" \
    deploy_tools/nginx.template.conf | sudo tee \
    /etc/nginx/sites-available/superlists.ottg.eu
```

Polecenie sed („stream editor”, czyli edytor strumienia) pobiera strumień tekstu i edytuje go. To nie jest przypadek, że polecenie służące do zastępowania ciągów tekstowych w Fabric ma taką samą nazwę. W omawianym przykładzie polecenie sed zastępuje ciąg tekstowy SITENAME adresem naszej witryny internetowej — do tego celu została użyta składnia s/tekst_do_zastąpienia/tekst_zastępujący/g. Dzięki zastosowaniu potoku (|) dane wyjściowe są przekazywane do procesu użytkownika root (sudo) używającego polecenia tee do wyświetlenia danych umieszczanych w pliku. Tutaj to będzie konfiguracja wirtualnego hosta dla Nginx.

Teraz możemy już aktywować utworzony plik:

```
edyta@serwer:$ sudo ln -s ../sites-available/superlists.ottg.eu \
/etc/nginx/sites-enabled/superlists.ottg.eu
```

Następnie zapisujemy skrypt Upstart:

```
edyta@serwer: sed "s/SITENAME/superlists.ottg.eu/g" \
deploy_tools/gunicorn-upstart.template.conf | sudo tee \
/etc/init/gunicorn-superlists.ottg.eu.conf
```

Na końcu uruchamiamy obie usługi:

```
edyta@serwer:$ sudo service nginx reload
edyta@serwer:$ sudo start gunicorn-superlists.ottg.eu
```

Sprawdzamy naszą witrynę. Działa, hura!

Plik *fabfile.py* możemy więc przekazać do repozytorium:

```
$ git add deploy_tools/fabfile.py
$ git commit -m "Dodanie pliku fabfile pozwalającego na zautomatyzowane wdrożenie."
```

Użycie polecenia git tag do oznaczenia wydania

Do wykonania pozostało jeszcze ostatnie zadanie administracyjne. W celu przygotowania oznaczeń dla wydań historycznych używamy polecenia `git tag` do oznaczenia stanu bazy kodu odzwierciedlającego aktualny kod w serwerze produkcyjnym:

```
$ git tag LIVE
$ export TAG=date +DEPLOYED-%F/%H%M` # Polecenie generuje znacznik czasu.
$ echo $TAG # Polecenie powinno wyświetlić "DEPLOYED-", a następnie znacznik czasu.
$ git tag $TAG
$ git push origin LIVE $TAG # Przekazanie znaczników.
```

Teraz w dowolnej chwili będzie można bardzo łatwo sprawdzić różnice między bieżącą bazą kodu a kodem znajdującym się w serwerze produkcyjnym. Taka możliwość stanie się niezwykle przydatna w dalszych rozdziałach, gdy będziemy omawiać migracje baz danych. Spójrz na znacznik w historii:

```
$ git log --graph --oneline --decorate
```

W tym momencie masz już wdrożoną witrynę internetową. Poinformuj o niej przyjaciół. Powiedz mamie, jeśli nikt inny nie jest zainteresowany. W kolejnym rozdziale powracamy do tworzenia kodu.

Dalsza lektura

Nie istnieje coś takiego jak jeden właściwy sposób wdrożenia aplikacji, a ja na pewno nie załączam się do ekspertów w tym zakresie. Postarałem się przedstawić rozsądne rozwiązanie, ale istnieje wiele rzeczy, które możesz zrobić inaczej, a przy okazji naprawdę dużo się nauczyć. Oto kilka źródeł, które uznaję za inspirujące:

- *Solid Python Deployments for Everybody*⁹ — Hynek Schlawack.
- *Git-based fabric deployments are awesome*¹⁰ — Dan Bravender.
- Rozdział poświęcony wdrożeniom, który znajdziesz w książce *Two Scoops of Django* napisanej przez Dana Greenfelda i Audrey Roy.
- *The 12-factor App*¹¹ — zespół Heroku.

W dodatku C znajdziesz pewne wskazówki dotyczące automatyzacji procesu przygotowania witryny, a także dotyczące rozwiązania alternatywnego dla Fabric, o nazwie Ansible.

Zautomatyzowane wdrożenia

Fabric

Fabric pozwala na wykonywanie z poziomu skryptu Pythona poleceń w serwerze. To jest doskonałe narzędzie do automatyzacji zadań administracyjnych.

Idempotencja

Jeżeli skrypt ma zostać użyty do wdrożenia witryny w istniejących serwerach, wówczas trzeba go przygotować w taki sposób, aby działał z czystą instalacją oraz w już skonfigurowanym serwerze.

Umieszczenie plików konfiguracyjnych w systemie kontroli kodu źródłowego

Upewnij się, że plik konfiguracyjny w serwerze to jedyna jego kopia! To ma znaczenie krytyczne dla aplikacji i podobnie jak wszystko inne, tego rodzaju pliki powinny być w systemie kontroli wersji.

Automatyzacja procedury przygotowania

Ostatecznie wszystko powinno być zautomatyzowane, co obejmuje przygotowanie zupełnie nowych serwerów i zagwarantowanie, że będą miały zainstalowane odpowiednie oprogramowanie. Automatyzacja będzie musiała współpracować z API oferowanym przez dostawcę hostingu.

Narzędzia zarządzania konfiguracją

Fabric to bardzo elastyczne narzędzie, ale jego logika nadal opiera się na skryptach. Bardziej zaawansowane narzędzia stosują bardziej „deklaracyjne” podejście i jeszcze bardziej mogą ułatwić Ci pracę. Ansible i Vagrant to dwa rozwiązania warte sprawdzenia (zajrzyj do dodatku C), choć istnieje jeszcze wiele innych (Chef, Puppet, Salt, Juju...).

⁹ <https://hynek.me/talks/python-deployments/>

¹⁰ http://dan.bravender.net/2012/5/11/git-based_fabric_deploys_are_awesome.html

¹¹ <http://12factor.net/>

Weryfikacja danych wejściowych i organizacja testu

W kilku kolejnych rozdziałach będziemy zajmować się testowaniem i implementacją weryfikacji danych wejściowych użytkownika. Będziemy mieli także możliwość przeprowadzenia drobnych porządków w kodzie zarówno aplikacji, jak i testów.

Testy funkcjonalne weryfikacji danych — ochrona przed pustymi elementami

Gdy pierwsi użytkownicy zaczęli korzystać z witryny, zauważyleś, że czasami popełniają błędy wprowadzające bałagan w ich listach. Przykładem może być przypadkowe tworzenie pustych elementów lub umieszczenie na liście dwóch takich samych elementów. Komputery mają pomagać nam w uniknięciu popełniania tego rodzaju pomyłek, zobaczymy więc, jak można usprawnić naszą witrynę internetową.

Poniżej przedstawiono ogólny zarys testów funkcjonalnych.

Plik `functional_tests/tests.py` (ch10l001):

```
def test_cannot_add_empty_list_items(self):
    # Edyta przeszła na stronę główną i przypadkowo spróbowała utworzyć
    # pusty element na liście. Naciągnęła klawisz Enter w pustym polu tekstowym.

    # Po odświeżeniu strony głównej zobaczyła komunikat błędu
    # informujący o niemożliwości utworzenia pustego elementu na liście.

    # Spróbowała ponownie, wpisując dowolny tekst, i wszystko zadziałało.

    # Przekornie po raz drugi spróbowała utworzyć pusty element na liście.

    # Na stronie listy otrzymała ostrzeżenie podobne do wcześniejszego.

    # Element mogła poprawić, wpisując w nim dowolny tekst.
    self.fail('Napisz mnie!')
```

To wszystko brzmi nieźle, ale zanim przejdziemy dalej, zwróć uwagę na jedno — plik testów funkcjonalnych staje się coraz większy. Podzielimy go więc na kilka mniejszych plików, z których każdy będzie zawierał pojedynczą metodę testową.

Pamiętaj, że testy funkcjonalne są ściśle związane z informacjami od użytkownika. Jeżeli korzystasz z jakiegoś narzędzia przeznaczonego do zarządzania projektami, na przykład narzędzia do śledzenia błędów, wówczas upewnij się, że każdy plik odpowiada jednemu błędowi, a jego nazwa zawiera identyfikator zgłoszonego błędu. Jeżeli wolisz myśleć w kategoriach „funkcji”, gdzie jedna funkcja może zawierać wiele informacji pochodzących od użytkownika, wówczas możesz przygotować po jednym pliku i klasie dla funkcji oraz metody dla poszczególnych informacji pobranych od użytkownika.

Utworzymy także jedną bazową klasę testów, po której inne będą dziedziczyć. Poniżej znajdziesz omówioną krok po kroku procedurę pozwalającą na implementację wymienionego powyżej rozwiązania.

Pominięcie testu

Podczas refaktoryzacji dobrze jest, aby wszystkie testy w pakiecie zostały zaliczone. Dopiero przygotowaliśmy test, który zgodnie z oczekiwaniemi ma zakończyć się niepowodzeniem. Wyłączmy go tymczasowo za pomocą dekoratora o nazwie `skip` oferowanego przez moduł `unittest`.

Plik `functional_tests/tests.py` (ch10l001-1):

```
from unittest import skip
[...]
@skip
def test_cannot_add_empty_list_items(self):
```

Ten test nakazuje silnikowi testów, aby go zignorował. Możesz się o tym przekonać, uruchamiając testy — komunikat powinien informować o zaliczeniu testu:

```
$ python3 manage.py test functional_tests
[...]
Ran 3 tests in 11.577s
OK
```



Pomijanie testów jest niebezpieczne — musisz pamiętać o ich ponownym włączeniu przed przekazaniem zmian do repozytorium. To jeden z powodów, dla których dobrym pomysłem jest przeglądanie zmian wiersz po wierszu podczas przekazywania plików do repozytorium.

Nie zapominaj o etapie refaktoryzacji w cyklu czerwony, zielony, refaktoryzacja

Częstym powodem krytyki technik TDD jest opinia, że ich stosowanie prowadzi do utworzenia nieprawidłowo opracowanego kodu, ponieważ programista koncentruje się na zaliczeniu testów, zamiast zastanowić się nad tym, jak powinien być zaprojektowany cały system. Ten powód uważam za niesprawiedliwy.

Programowanie sterowane testami na pewno nie jest złotym środkiem na wszystko. Nadal musisz poświęcić nieco czasu i zastanowić się nad dobrym projektem. Jednak bardzo często programiści zapominają o etapie refaktoryzacji w cyklu czerwony, zielony, refaktoryzacja. Metodologia pozwala na wykorzystanie starego kodu w celu zaliczenia testów, ale jednocześnie zachęca do poświęcenia nieco czasu na refaktoryzację, aby poprawić projekt.

Bardzo często najlepsze pomysły dotyczące sposobu refaktoryzacji kodu nie pojawiają się od razu. Możesz na nie wpaść po upływie dni, tygodni, a nawet miesięcy od chwili utworzenia danego fragmentu kodu, gdy pracujesz nad czymś zupełnie innym i ponownie spojrzysz na stary kod świeżym okiem. Jeżeli jednak jesteś w trakcie wykonywania innego zadania, to czy powinieneś się zatrzymać i zająć refaktoryzacją starego kodu?

Odpowiedź brzmi: to zależy. W tym rozdziale nawet nie zaczeliśmy tworzyć nowego kodu. Wiemy, że aplikacja znajduje się w działającym stanie, więc możemy ją na chwilę odłożyć i przejść do nowych testów funkcjonalnych (aby znów były w pełni zaliczane) i od razu przeprowadzić pewną refaktoryzację.

W dalszej części rozdziału zauważysz kolejne fragmenty kodu, które będziesz chciał zmodyfikować. W takich przypadkach zamiast podjąć ryzyko refaktoryzacji aplikacji, która może nie znajdować się w stanie „działającej”, warto poczynić odpowiednie notatki dotyczące ewentualnych zmian i wstrzymać się z ich wprowadzeniem do chwili, gdy wszystkie testy będą zaliczane.

Podział testów funkcjonalnych na wiele plików

Zaczynamy od umieszczenia poszczególnych testów we własnych klasach, nadal w tym samym pliku.

Plik `functional_tests/tests.py` (ch10l002):

```
class FunctionalTest(StaticLiveServerCase):

    @classmethod
    def setUpClass(cls):
        [...]
    @classmethod
    def tearDownClass(cls):
        [...]
    def setUp(self):
        [...]
    def tearDown(self):
        [...]
    def check_for_row_in_list_table(self, row_text):
        [...]

class NewVisitorTest(FunctionalTest):

    def test_can_start_a_list_and_retrieve_it_later(self):
        [...]

class LayoutAndStylingTest(FunctionalTest):

    def test_layout_and_styling(self):
        [...]

class ItemValidationTest(FunctionalTest):

    @skip
    def test_cannot_add_empty_list_items(self):
        [...]
```

Na tym etapie możemy ponownie wykonać testy funkcjonalne i sprawdzić, czy nadal są zaliczane:

Ran 3 tests in 11.577s

OK

To było pracochnonne i prawdopodobnie zadanie można było wykonać za pomocą mniejszej liczby kroków, ale jak nieustannie powtarzam, stosowanie metody krok po kroku w łatwych przypadkach powoduje, że gdy zetkniesz się z trudniejszymi przypadkami, zadanie stanie się łatwiejsze.

Przechodzimy teraz z pojedynczego pliku testów do oddzielnych plików dla poszczególnych klas oraz jednego pliku „bazowego” zawierającego klasę bazową, po której będą dziedziczyć wszystkie testy. Tworzymy więc cztery kopie pliku *tests.py*, nadajemy im odpowiednie nazwy, a następnie z każdego usuwamy niepotrzebne w nim fragmenty:

```
$ git mv functional_tests/tests.py functional_tests/base.py  
$ cp functional_tests/base.py functional_tests/test_simple_list_creation.py  
$ cp functional_tests/base.py functional_tests/test_layout_and_styling.py  
$ cp functional_tests/base.py functional_tests/test_list_item_validation.py
```

Zawartość pliku *base.py* można ograniczyć do klasy `FunctionalTest`. W klasie bazowej pozostawiamy metody pomocnicze, ponieważ przypuszczamy, że będziemy ich ponownie używać w nowym teście funkcyjnym.

Plik *functional_tests/base.py* (ch10l003):

```
from django.contrib.staticfiles.testing import StaticLiveServerCase  
from selenium import webdriver  
import sys  
  
class FunctionalTest(StaticLiveServerCase):  
  
    @classmethod  
    def setUpClass(cls):  
        [...]  
    def tearDownClass(cls):  
        [...]  
    def setUp(self):  
        [...]  
    def tearDown(self):  
        [...]  
    def check_for_row_in_list_table(self, row_text):  
        [...]
```



Pozostawienie metod pomocniczych w klasie bazowej `FunctionalTest` to dobry sposób na uniknięcie powielania kodu w testach funkcyjnych. W dalszej części książki (rozdział 21.) użyjemy wzorca strony, który działa podobnie, ale preferuje kompozycję zamiast dziedziczenia.

Nasz pierwszy test funkcyjny znajduje się w oddzielnym pliku i powinien składać się z pojedynczej klasy zawierającej jedną metodę testową.

Plik *functional_tests/test_simple_list_creation.py* (ch10l004):

```
from .base import FunctionalTest  
from selenium import webdriver  
from selenium.webdriver.common.keys import Keys  
  
class NewVisitorTest(FunctionalTest):  
  
    def test_can_start_a_list_and_retrieve_it_later(self):  
        [...]
```

Wykorzystałem tutaj względny import (`from .base`). Niektórzy dość często korzystają z tej możliwości podczas tworzenia kodu Django, na przykład widoki mogą importować modele za pomocą polecenia `from .models import List` zamiast `from list.models`. To jednak kwestia

upodobań. Osobiście preferuję użycie względnego importu jedynie wtedy, gdy mam absolutną pewność, że położenie importowanego komponentu nie ulegnie zmianie. Tak się dzieje w omawianym przykładzie, ponieważ wiem na pewno, że wszystkie testy będą względne dla pliku `base.py`, po którym dziedziczą.

Test odpowiedzialny za sprawdzenie układu i stylu znajduje się w pojedynczym pliku i obejmuje jedną klasę.

Plik `functional_tests/test_layout_and_styling.py` (ch10l005):

```
from .base import FunctionalTest

class LayoutAndStylingTest(FunctionalTest):
    [...]
```

Test przeprowadzający weryfikację również znalazł się w oddzielnym pliku.

Plik `functional_tests/test_list_item_validation.py` (ch10l006):

```
from unittest import skip
from .base import FunctionalTest

class ItemValidationTest(FunctionalTest):

    @skip
    def test_cannot_add_empty_list_items(self):
        [...]
```

Upewniamy się o prawidłowym działaniu całości, ponownie wydając polecenie `manage.py test functional_test`, i raz jeszcze sprawdzamy, czy wszystkie testy zostały zaliczone:

```
Ran 3 tests in 11.577s
```

```
OK
```

Teraz możemy usunąć wiersz powodujący pominięcie testu.

Plik `functional_tests/test_list_item_validation.py` (ch10l007):

```
class ItemValidationTest(FunctionalTest):

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

Wykonanie pojedynczego pliku testu

Miłym efektem wprowadzonych zmian jest możliwość wykonania pojedynczego pliku testu, na przykład następująco:

```
$ python3 manage.py test functional_tests.test_list_item_validation
[...]
AssertionError: write me!
```

Doskonale! Nie trzeba dłużej czekać na wykonanie wszystkich testów funkcjonalnych, gdy jesteśmy zainteresowani tylko jednym. Jednak od teraz trzeba pamiętać o wykonywaniu ich wszystkich, aby upewnić się, że nie wprowadzono regresji. W dalszej części książki zobaczysz, jak to zadanie zlecić zautomatyzowanej pętli nieustannej integracji (ang. *continuous integration*). Teraz warto przekazać pliki do repozytorium:

```
$ git status
$ git add functional_tests
$ git commit -m"Przeniesienie testów funkcjonalnych do oddzielnych plików."
```

Podparcie testów funkcjonalnych

Przystępujemy teraz do implementacji testu, a przynajmniej jego początku.

Plik `functional_tests/test_list_item_validation.py` (ch10l008):

```
def test_cannot_add_empty_list_items(self):
    # Edyta przeszła na stronę główną i przypadkowo spróbowała utworzyć
    # pusty element na liście. Nacisnęła klawisz Enter w pustym polu tekstowym.
    self.browser.get(self.server_url)
    self.browser.find_element_by_id('id_new_item').send_keys('\n')

    # Po odświeżeniu strony głównej zobaczyła komunikat błędu
    # informujący o niemożliwości utworzenia pustego elementu na liście.
    error = self.browser.find_element_by_css_selector('.has-error') #❶
    self.assertEqual(error.text, "Element nie może być pusty")

    # Spróbowała ponownie, wpisując dowolny tekst, i wszystko zadziałało.
    self.browser.find_element_by_id('id_new_item').send_keys('Kupić mleko\n')
    self.check_for_row_in_list_table('1: Kupić mleko') #❷

    # Przekorna po raz drugi spróbowała utworzyć pusty element na liście.
    self.browser.find_element_by_id('id_new_item').send_keys('\n')

    # Na stronie listy otrzymała ostrzeżenie podobne do wcześniejszego.
    self.check_for_row_in_list_table('1: Kupić mleko')
    error = self.browser.find_element_by_css_selector('.has-error')
    self.assertEqual(error.text, "Element nie może być pusty")

    # Element mogła poprawić, wpisując w nim dowolny tekst.
    self.browser.find_element_by_id('id_new_item').send_keys('Zrobić herbatę\n')
    self.check_for_row_in_list_table('1: Kupić mleko')
    self.check_for_row_in_list_table('2: Zrobić herbatę')
```

Kilka uwag dotyczących powyższego testu.

- ❶ Do nadania stylu tekstowi komunikatu błędu używamy klasy CSS frameworka Bootstrap o nazwie `.has-error`. Jak możesz zobaczyć, Bootstrap zawiera użyteczne style do tego rodzaju zadań.
- ❷ Zgodnie z przewidywaniami ponownie używamy metody pomocniczej `check_for_row_in_list_table()` w celu potwierdzenia, że dodanie elementu na liście *nie* działa.

Technika polegająca na umieszczeniu metod pomocniczych w klasie nadzędnej pozwala na uniknięcie powielania kodu w testach funkcjonalnych. Jeżeli pewnego dnia zdecydujesz się na zmianę implementacji tabeli wyświetlającej listę, wówczas kod testu funkcjonalnego będziesz musiał zmodyfikować tylko w jednym miejscu zamiast w dziesiątkach plików zawierających testy funkcjonalne...

Posuwamy się naprzód!

```
selenium.common.exceptions NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":".has-error"}' ; Stacktrace:
```

Pora na przekazanie zmian do repozytorium.

Sprawdzenie warstwy modelu

Istnieją dwa poziomy, na których w Django można przeprowadzić operacje sprawdzenia. Pierwszy to poziom modelu, natomiast drugi (wyższy) to poziom formularzy. Gdy tylko istnieje możliwość, wolę przeprowadzać operacje sprawdzania na niższym poziomie, po części z powodu dużego wyczulenia na zachowanie spójności bazy danych, a po części dlatego, że takie rozwiązanie jest po prostu bezpieczniejsze. Czasami możesz zapomnieć, który formularz jest używany do sprawdzenia danych, ale zawsze korzystasz z tej samej bazy danych.

Refaktoryzacja testów jednostkowych na oddzielne pliki

Dodamy kolejny test do modelu, ale wcześniej warto uporządkować testy jednostkowe w podobny sposób, jak to wcześniej zrobiliśmy z testami funkcjonalnymi. Różnica polega na tym, że aplikacja lists zawiera rzeczywisty kod aplikacji oraz testy, więc testy zostaną umieszczone w oddzielnym katalogu:

```
$ mkdir lists/tests  
$ touch lists/tests/__init__.py  
$ git mv lists/tests.py lists/tests/test_all.py  
$ git status  
$ git add lists/tests  
$ python3 manage.py test lists  
[...]  
Ran 10 tests in 0.034s  
  
OK  
$ git commit -m"Przeniesienie testów jednostkowych do oddzielnego katalogu z pojedynczym plikiem."
```

Jeżeli otrzymasz komunikat o wykonaniu zera testów, będzie to oznaczało, że prawdopodobnie zapomniałeś dodać dunderinit¹ — ten plik jest niezbędny, ponieważ w przeciwnym razie katalog *test* nie będzie poprawnym modelem Pythona...

Teraz możemy podzielić plik *test_all.py* na dwa: pierwszy, o nazwie *test_views.py*, zawierający jedynie testy widoku, i drugi, *test_models.py*, przeznaczony na testy modelu.

```
$ git mv lists/tests/test_all.py lists/tests/test_views.py  
$ cp lists/tests/test_views.py lists/tests/test_models.py
```

Skracamy plik *test_models.py* do zaledwie pojedynczego testu — to oznacza potrzebę użycia mniejszej liczby poleceń import.

Plik *lists/tests/test_models.py* (ch10l009):

```
from django.test import TestCase  
from lists.models import Item, List  
  
class ListAndItemModelsTest(TestCase):  
    [...]
```

Z kolei plik *test_views.py* traci zaledwie jedną klasę.

Plik *lists/tests/test_views.py* (ch10l010):

```
--- a/lists/tests/test_views.py  
+++ b/lists/tests/test_views.py  
@@ -103,34 +104,3 @@ class ListViewTest(TestCase):
```

¹ Słowo *dunder* to skrót oznaczający dwa znaki podkreślenia, więc *dunderinit* oznacza plik *__init__.py*.

```
    self.assertNotContains(response, 'other list item 1')
    self.assertNotContains(response, 'other list item 2')
-
-
-class ListAndItemModelsTest(TestCase):
-
-    def test_saving_and_retrieving_items(self):
[...]
```

Ponownie wykonujemy testy, aby sprawdzić, czy wszystko działa zgodnie z oczekiwaniami:

```
$ python3 manage.py test lists
[...]
Ran 10 tests in 0.040s
```

OK

Doskonale!

```
$ git add lists/tests
$ git commit -m "Umieszczenie testów jednostkowych w dwóch oddzielnych plikach."
```



Niektórzy umieszczają testy jednostkowe w katalogu tuż po rozpoczęciu pracy nad projektem oraz dodają plik *test_forms.py*. To jest perfekcyjne rozwiązanie. Wstrzymałem się z tym już w pierwszym rozdziale aż do chwili, gdy zwiększyła się liczba koniecznych do przeprowadzenia operacji porządkowych.

Testy jednostkowe sprawdzania modelu oraz menedżer kontekstu `self.assertRaises()`

W klasie `ListAndItemModelTest` umieszczamy nową metodę testową, która próbuje utworzyć nowy, pusty element listy.

Plik *lists/tests/test_models.py* (ch10l012-1):

```
from django.core.exceptions import ValidationError
[...]
class ListAndItemModelsTest(TestCase):
[...]
    def test_cannot_save_empty_list_items(self):
        list_ = List.objects.create()
        item = Item(list=list_, text='')
        with self.assertRaises(ValidationError):
            item.save()
```



Jeżeli dopiero zaczynasz programowanie w Pythonie, to mogłeś się wcześniej nie spotkać z poleceniem `with`. Jest stosowane wraz z tak zwany „menedżerami kontekstu” opakowującymi blok kodu, najczęściej pewnego rodzaju konfiguracji, operacje porządkowe lub kod odpowiedzialny za obsługę błędów. Więcej informacji na ten temat znajdziesz w dokumentacji² Pythona.

² <https://docs.python.org/release/2.5/whatsnew/pep-343.html>

To jest nowa technika przeprowadzania testu jednostkowego. Kiedy chcesz sprawdzić, czy wykonanie danego zadania spowoduje zgłoszenie błędu, wówczas używasz menedżera kontekstu `self.assertRaises()`. Możesz użyć konstrukcji podobnej do przedstawionej poniżej:

```
try:  
    item.save()  
    self.fail('Operacja zapisu powinna spowodować zgłoszenie wyjątku.')  
except ValidationError:  
    pass
```

Jednak formuła oparta na poleceniu `with` jest bardziej elegancka. Teraz spróbuj wykonać test i zobacz, że kończy się niepowodzeniem:

```
item.save()  
AssertionError: ValidationError not raised
```

Dziwactwo Django — zapis modelu nie wywołuje operacji sprawdzenia poprawności

Teraz odkryjesz jedno z niewielkich dziwactw Django. *Ten test powinien być zaliczony.* Jeżeli zajrzesz do dokumentacji³ Django dotyczącej modeli, to zauważysz, że wartością domyślną `TextField` jest `blank=False`. Oznacza to, że puste wartości powinny być niedozwolone.

Dlaczego więc test nie kończy się niepowodzeniem? Cóż, z powodów historycznych⁴ modele Django nie wywołują operacji pełnego sprawdzenia poprawności w trakcie zapisu. Jak się później przekonasz, wszelkie ograniczenia faktyczne zaimplementowane w bazie danych spowodują zgłoszenie błędów podczas zapisu. Jednak SQLite nie obsługuje egzekwowania ograniczenia dotyczącego pustych wartości w kolumnach tekstowych i dlatego metoda odpowiedzialna za zapis bez żadnego powiadomienia przepuszcza nieprawidłową wartość.

Istnieje możliwość sprawdzenia, czy ograniczenie następuje na poziomie bazy danych, czy nie. Jeżeli ograniczenie występuje na poziomie bazy danych, konieczne będzie przeprowadzenie migracji, aby zastosować ograniczenie. Jednak Django „wie”, że SQLite nie obsługuje tego typu ograniczeń, więc próba wydania polecenia `makemigrations` wyświetli komunikat informujący o braku zadań do wykonania:

```
$ python3 manage.py makemigrations  
No changes detected
```

Django zawiera metodę o nazwie `full_clean()` przeznaczoną do ręcznego wykonania pełnej operacji sprawdzenia. Zobaczmy, w jaki sposób działa:

```
with self.assertRaises(ValidationError):  
    item.save()  
    item.full_clean()
```

Testy zostają zaliczone:

```
OK
```

W ten sposób dowiedziałeś się nieco o sprawdzaniu poprawności w Django, a test ma Cię ostrzec, jeśli kiedykolwiek zapomnisz o wymogu i przypiszesz wartość `blank=True` dla pola tekstowego (spróbuj sam!).

³ <https://docs.djangoproject.com/en/1.7/ref/models/fields/#blank>

⁴ <https://groups.google.com/forum/#!topic/django-developers/ulhzSwWHj4c>

Wyświetlanie w widoku błędów z weryfikacji modelu

Spróbujemy teraz wyegzekwować w warstwie modelu ograniczenia związane ze sprawdzaniem poprawności modelu. Przeniesiemy je do przygotowanych szablonów, aby użytkownik mógł zobaczyć skutek ich działania. Poniżej pokazano, jak opcjonalnie można wyświetlić komunikat błędu w kodzie HTML. Kod sprawdza, czy szablon otrzymał zmienną błędu. Jeśli tak, odpowiedni komunikat zostaje wyświetlony obok formularza.

Plik *lists/templates/base.html* (ch10l013):

```
<form method="POST" action="{% block form_action %}{% endblock %}>
    <input name="item_text" id="id_new_item"
        class="form-control input-lg"
        placeholder="Wpisz rzecznik do zrobienia"
    />
    {% csrf_token %}
    {% if error %}
        <div class="form-group has-error">
            <span class="help-block">{{ error }}</span>
        </div>
    {% endif %}
</form>
```

Więcej informacji dotyczących kontrolek formularza znajdziesz w *dokumentacji*⁵ Bootstrap.

Przekazanie błędu szablonowi to zadanie dla funkcji widoku. Spójrz na testy jednostkowe w klasie *NewListTest*. W omawianym przykładzie zamierzam użyć dwóch nieco odmiennych podejść w zakresie obsługi błędów.

W pierwszym przypadku adres URL i widok dla nowych list opcjonalnie wygenerują ten sam szablon, który jest używany przez stronę główną, ale wraz z dodatkiem w postaci komunikatu błędu. Poniżej przedstawiono test jednostkowy dla wymienionej funkcjonalności.

Plik *lists/tests/test_views.py* (ch10l014):

```
class NewListTest(TestCase):
    [...]
    def test_validation_errors_are_sent_back_to_home_page_template(self):
        response = self.client.post('/lists/new', data={'item_text': ''})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')
        expected_error = "Element nie może być pusty"
        self.assertContains(response, expected_error)
```

W trakcie tworzenia tego testu możesz być nieco zdziwiony adresem URL */lists/new*, który ręcznie wprowadzamy jako ciąg tekstowy. W naszych testach, widokach i szablonach mamy wiele adresów URL zdefiniowanych na stałe, co stanowi złamanie zasad „nie powtarzaj się”. Nie przejmuję się aż tak bardzo powielaniem kodu w testach, ale zdecydowanie należy wyszukać w widokach i szablonach na stałe zdefiniowane adresy URL, a następnie zapisać je jako przeznaczone do refaktoryzacji. Jednak nie pozbędziemy się ich od razu, ponieważ w tym momencie aplikacja znajduje się w stanie „niedziałającej”. Dlatego też najpierw trzeba przywrócić aplikację do stanu działania.

⁵ <http://getbootstrap.com/css/>

Powracamy do testu, który kończy się niepowodzeniem, ponieważ widok zwraca błąd o kodzie stanu 302 (przekierowanie) zamiast „normalnej” odpowiedzi o kodzie 200:

```
AssertionError: 302 != 200
```

Spróbujmy wywołać funkcję `full_clean()` w widoku:

```
def new_list(request):
    list_ = List.objects.create()
    item = Item.objects.create(text=request.POST['item_text'], list=list_)
    item.full_clean()
    return redirect('/lists/%d/' % (list_.id,))
```

Analizując kod widoku, znajdziemy pewne na stałe zdefiniowane adresy URL, które są dobrymi kandydatami do pozbycia się ich. Na naszej osobistej liście rzeczy do zrobienia dodajemy wpis pokazany na rysunku 10.1.



Rysunek 10.1. Nasza osobista lista rzeczy do zrobienia

Teraz operacja sprawdzenia poprawności spowoduje zgłoszenie wyjątku, który pojawi się w widoku:

```
[...]
File "/workspace/superlists/lists/views.py", line 11, in new_list
    item.full_clean()
[...]
django.core.exceptions.ValidationError: {'text': ['This field cannot be
blank.']}
```

Próbowiemy więc zastosować pierwsze podejście polegające na użyciu konstrukcji `try-except` do wykrywania błędów. Wsłuchując się w Testing Goat, pracę rozpoczynamy po prostu od wymienionej konstrukcji i niczego więcej. Testy powinny wskazać, czym trzeba się zająć w dalszej kolejności.

Plik `lists/views.py` (ch10l015):

```
from django.core.exceptions import ValidationError
[...]

def new_list(request):
    list_ = List.objects.create()
    item = Item.objects.create(text=request.POST['item_text'], list=list_)
    try:
        item.full_clean()
    except ValidationError:
        pass
    return redirect('/lists/%d/' % (list_.id,))
```

To nas prowadzi z powrotem do asercji `302 != 200`:

```
AssertionError: 302 != 200
```

Kod zwraca wygenerowany szablon i również zajmuje się jego sprawdzeniem.

Plik *lists/views.py* (ch10l016):

```
except ValidationError:  
    return render(request, 'home.html')
```

W tym momencie testy powinny przekazywać szablonowi komunikaty błędów:

```
AssertionError: False is not true : Couldn't find 'Element nie może być pusty' in response
```

W tym celu wykorzystujemy nową zmienną szablonu.

Plik *lists/views.py* (ch10l017):

```
except ValidationError:  
    error = "Element nie może być pusty"  
    return render(request, 'home.html', {"error": error})
```

W przypadku komunikatów zawierających znaki inne niż alfanumeryczne rozwiązanie jednak nie działa zgodnie z oczekiwaniami:

```
AssertionError: False is not true : Couldn't find 'Pójść do restauracji McDonald's' in response
```

Stosujemy więc procedurę usuwania błędów opartą na wywołaniu `print()`.

Plik *lists/tests/test_views.py*:

```
expected_error = "Pójść do restauracji McDonald's"  
print(response.content.decode())  
self.assertContains(response, expected_error)
```

I mamy już przyczynę problemu: framework Django zamienił⁶ apostrof na encję HTML:

```
[...]  
<span class="help-block">Pójść do restauracji McDonald's</span>
```

Wprawdzie można zastosować pewną sztuczkę w teście:

```
expected_error = "Pójść do restauracji McDonald's"
```

ale lepszym rozwiązaniem jest użycie oferowanej przez Django funkcji pomocniczej.

Plik *lists/tests/test_views.py* (ch10l019):

```
from django.utils.html import escape  
[...]  
  
expected_error = escape("Pójść do restauracji McDonald's")  
self.assertContains(response, expected_error)
```

Testy zostają zaliczone!

```
Ran 12 tests in 0.047s
```

```
OK
```

Upewnienie się, że nieprawidłowe dane nie zostaną zapisane w bazie danych

Zanim przejdziemy dalej, pozostała jeszcze jedna kwestia. Czy zauważłeś niewielki błąd logiczny w przygotowanej implementacji? Obecnie można utworzyć obiekt, nawet jeśli operacja sprawdzenia poprawności zakończy się niepowodzeniem.

⁶ <https://docs.djangoproject.com/en/1.7/topics/templates/#automatic-html-escaping>

Plik *lists/views.py*:

```
item = Item.objects.create(text=request.POST['item_text'], list=list_)
try:
    item.full_clean()
except ValidationError:
    [...]
```

Dodamy teraz nowy test jednostkowy, aby się upewnić, że pusty element listy nie zostanie zapisany.

Plik *lists/tests/test_views.py* (ch10l020-1):

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

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        [...]

    def test_invalid_list_items_arent_saved(self):
        self.client.post('/lists/new', data={'item_text': ''})
        self.assertEqual(List.objects.count(), 0)
        self.assertEqual(Item.objects.count(), 0)
```

Otrzymujemy następujące dane wyjściowe:

```
[...]
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests/test_views.py", line 57, in
test_invalid_list_items_arent_saved
    self.assertEqual(List.objects.count(), 0)
AssertionError: 1 != 0
```

Usuwamy problem, modyfikując kod w pokazany poniżej sposób.

Plik *lists/tests/test_views.py* (ch10l020-2):

```
def new_list(request):
    list_ = List.objects.create()
    item = Item(text=request.POST['item_text'], list=list_)
    try:
        item.full_clean()
        item.save()
    except ValidationError:
        list_.delete()
        error = "Element nie może być pusty"
        return render(request, 'home.html', {"error": error})
    return redirect('/lists/%d/' % (list_.id,))
```

Czy testy funkcjonalne zostaną zaliczone?

```
$ python3 manage.py test functional_tests.test_list_item_validation
[...]
  File "/workspace/superlists/functional_tests/test_list_item_validation.py",
line 26, in test_cannot_add_empty_list_items
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":".has-error"}' ; Stacktrace:
```

Udało nam się posunąć odrobinę do przodu. Sprawdzamy wiersz 26. i widzimy, że pierwsza część testu zostaje zaliczona. Jesteśmy teraz na drugiej, czyli wysłanie pustego elementu powoduje wyświetlenie błędu.

Ponieważ mamy pewną ilość działającego kodu, przekazujemy go do repozytorium:

```
$ git commit -am"Dostosowanie nowego widoku listy do weryfikacji modelu."
```

Wzorzec Django — przetwarzanie żądań POST w widoku generującym formularz

Tym razem zastosujemy nieco odmienne podejście, które w rzeczywistości jest dość często stosowanym wzorcem w Django — użycie tego samego widoku zarówno do przetworzenia żądań POST, jak i wyświetlania formularza, z którego one pochodzą. Wprawdzie to nie całkiem pasuje do modelu typu RESTful, ale ma ważną zaletę w postaci tego samego adresu URL przeznaczonego do wyświetlenia formularza oraz ewentualnych błędów, które wystąpiły podczas przetwarzania danych wejściowych użytkownika.

Na obecnym etapie mamy jeden widok i adres URL przeznaczony do wyświetlenia listy oraz drugi widok i adres URL przeznaczony do przetwarzania nowych elementów dodawanych do listy. Dlatego też połączymy je w jeden. Formularz zdefiniowany w pliku *list.html* otrzymuje inny adres docelowy.

Plik *lists/templates/list.html* (ch10l020):

```
{% block form_action %}/lists/{{ list.id }}/{% endblock %}
```

Niestety skutkiem jest kolejny adres URL na stałe zdefiniowany w kodzie. Przechodzimy więc do naszej osobistej listy rzeczy do zrobienia i pamiętając o powyższym, dodajemy także wzmiankę o pliku *home.html* (patrz rysunek 10.2).



Rysunek 10.2. Nasza osobista lista rzeczy do zrobienia

Wprowadzona zmiana natychmiast powoduje uszkodzenie początkowego testu funkcjonalnego, ponieważ strona `view_list` nie potrafi jeszcze przetwarzać żądań POST:

```
$ python3 manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":".has-error"}' ; Stacktrace:
[...]
AssertionError: '2: Użyć pawich piór do zrobienia przynęty' not found in ['1: Kupić pawie pióra']
```



W tym podrozdziale przeprowadzamy refaktoryzację na poziomie aplikacji. Odbywa się to przez zmianę lub dodanie testów jednostkowych, a następnie dostosowanie kodu. Testy funkcjonalne wykorzystamy do poinformowania o zakończeniu refaktoryzacji i przywróceniu aplikacji ponownie do stanu działania. Jeżeli potrzebujesz dodatkowych wyjaśnień, spójrz ponownie na wykres przedstawiony na końcu rozdziału 4.

Refaktoryzacja

— przekształcenie funkcjonalności new_item na view_list

Stare testy z klasy NewItemTests odpowiedzialne za sprawdzenie operacji zapisu żądań POST w istniejących listach przeniesiemy teraz do klasy ListViewTest. Przy okazji zmodyfikujemy je w taki sposób, aby wskazywały adres URL bazowy zamiast `./new_item`.

Plik `lists/tests/test_views.py` (ch10l021):

```
class ListViewTest(TestCase):

    def test_uses_list_template(self):
        [...]

    def test_passes_correct_list_to_template(self):
        [...]

    def test_displays_only_items_for_that_list(self):
        [...]

    def test_can_save_a_POST_request_to_an_existing_list(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        self.client.post(
            '/lists/%d/' % (correct_list.id,),
            data={'item_text': 'Nowy element dla istniejącej listy'}
        )

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

    def test_POST_redirects_to_list_view(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        response = self.client.post(
            '/lists/%d/' % (correct_list.id,),
            data={'item_text': 'Nowy element dla istniejącej listy'}
        )
        self.assertRedirects(response, '/lists/%d/' % (correct_list.id,))
```

Zwróć uwagę na fakt, że klasa `NewItemTest` zniknęła całkowicie. Ponadto zmieniliśmy nazwę testu przekierowania, aby wyraźnie wskazywała na zastosowanie jedynie do żądań POST.

Otrzymujemy następujące dane wyjściowe:

```
FAIL: test_POST_redirects_to_list_view (lists.tests.test_views.ListViewTest)
AssertionError: 200 != 302 : Response didn't redirect as expected: Response
code was 200 (expected 302)
[...]
FAIL: test_can_save_a_POST_request_to_an_existing_list
(lists.tests.test_views.ListViewTest)
AssertionError: 0 != 1
```

Modyfikujemy funkcję `view_list()` i przystosowujemy ją do obsługi dwóch typów żądań.

Plik *lists/views.py* (ch10l022-1):

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'], list=list_)
        return redirect('/lists/%d/' % (list_.id,))
    return render(request, 'list.html', {'list': list_})
```

Po zmianach wszystkie testy są zaliczane:

```
Ran 13 tests in 0.047s
```

```
OK
```

W tym momencie możemy już usunąć widok `add_item`, ponieważ nie jest dłużej potrzebny...

Ups, pojawia się kilka nieoczekiwanych niepowodzeń:

```
[...]
django.core.exceptions.ViewDoesNotExist: Could not import lists.views.add_item.
View does not exist in module lists.views.
[...]
FAILED (errors=4)
```

Powyzsze niepowodzenia są skutkiem usunięcia widoku, podczas gdy w pliku *urls.py* nadal znajdują się odwołania do wspomnianego widoku. Dlatego je również należy usunąć.

Plik *lists/urls.py* (ch10l023):

```
urlpatterns = patterns('',
    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^new$', 'lists.views.new_list', name='new_list'),
)
```

Teraz wszystkie testy zostają zaliczone. Sprawdzamy, jak to wygląda w przypadku testów funkcjonalnych:

```
$ python3 manage.py test functional_tests
[...]
```

```
Ran 3 tests in 15.276s
```

```
FAILED (errors=1)
```

Mamy jedno niepowodzenie w nowym teście funkcjonalnym. Refaktoryzacja funkcjonalności `add_item` jest zakończona. Warto przekazać pliki do repozytorium:

```
$ git commit -am"Refaktoryzacja widoku listy, aby obsługiwał żądania POST dotyczące nowych elementów listy."
```



Wygląda na to, że złamałem zasadę mówiącą o tym, aby nigdy nie przeprowadzać refaktoryzacji dla testów kończących się niepowodzeniem. W omawianym przypadku to jest dozwolone, ponieważ refaktoryzacja była wymagana do zapewnienia działania nowej funkcjonalności. Zdecydowanie nigdy nie powinieneś decydować się na refaktoryzację, gdy test *jednostkowy* kończy się niepowodzeniem. W niniejszej książce jest to dopuszczalne dla testu funkcjonalnego dotyczącego aktualnych informacji od użytkownika, nad którymi pracujemy. Jeżeli preferujesz czyste wykonywanie testów, możesz dodać dekorator `@skip` do bieżącego testu funkcjonalnego.

Egzekwowanie w widoku view_list weryfikacji modelu

Nadal chcemy, aby elementy dodawane do istniejących list podlegały regułom weryfikacji modelu. Do tego celu przygotujemy nowy test jednostkowy, bardzo podobny do przeznaczonego dla strony głównej, choć wprowadzimy w nim kilka zmian.

Plik *lists/tests/test_views.py* (ch10l024):

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

    def test_validation_errors_end_up_on_lists_page(self):
        list_ = List.objects.create()
        response = self.client.post(
            '/lists/%d/' % (list_.id,),
            data={'item_text': ''}
        )
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'list.html')
        expected_error = escape("Element nie może być pusty")
        self.assertContains(response, expected_error)
```

Test powinien zakończyć się niepowodzeniem, ponieważ widok obecnie nie przeprowadza żadnej weryfikacji danych, a jedynie stosuje przekierowania dla wszystkich żądań POST:

```
self.assertEqual(response.status_code, 200)
AssertionError: 302 != 200
```

Oto niezbędna implementacja.

Plik *lists/views.py* (ch10l025):

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    error = None

    if request.method == 'POST':
        try:
            item = Item(text=request.POST['item_text'], list=list_)
            item.full_clean()
            item.save()
            return redirect('/lists/%d/' % (list_.id,))
        except ValidationError:
            error = "Element nie może być pusty"

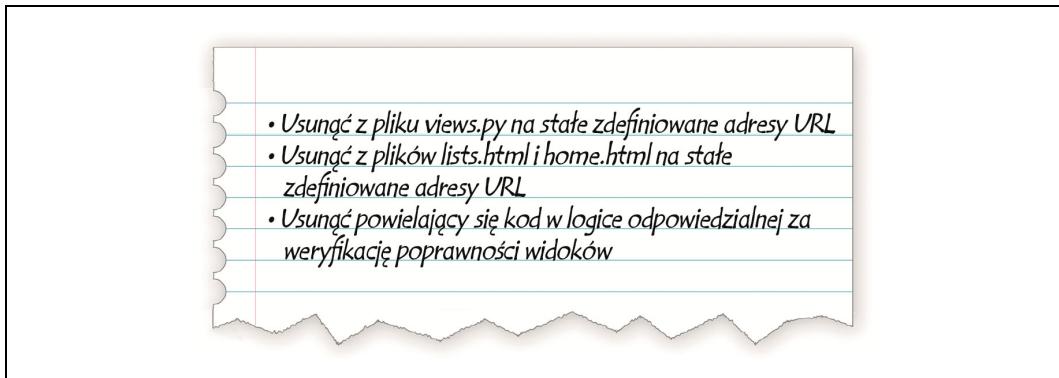
    return render(request, 'list.html', {'list': list_, 'error': error})
```

Nie jesteś zbyt usatysfakcjonowany rozwiązaniem? Na pewno mamy tutaj do czynienia z pewną ilością powielonego kodu — w pliku *views.py* dwukrotnie pojawia się konstrukcja *try-except*, a rozwiązanie ogólnie pozostaje słabe.

```
Ran 14 tests in 0.047s
```

```
OK
```

Mimo wszystko musimy się nieco wstrzymać z refaktoryzacją, ponieważ wiemy już, że zastosujemy nieco inny kod odpowiedzialny za weryfikację powielonych elementów. Problem zapisujemy jednak na naszej osobistej liście rzeczy do zrobienia (patrz rysunek 10.3).



Rysunek 10.3. Nasza osobista lista rzeczy do zrobienia



Jednym z powodów istnienia zasady „do trzech razy sztuka, a później refaktoryzacja” jest to, że jeśli poczekasz aż do wystąpienia trzech przypadków użycia, każdy z nich może być nieco inny i pomóc w uzyskaniu świeżego spojrzenia na łączącą je funkcjonalność. Jeżeli zbyt wcześnie zdecydujesz się na refaktoryzację, wówczas może się okazać, że trzeci przypadek użycia niezbyt pasuje do zrefaktoryzowanego kodu.

Jednak przynajmniej testy funkcjonalne znów są zaliczane:

```
$ python3 manage.py test functional_tests  
[...]  
OK
```

Aplikacja ponownie znajduje się w stanie „działającej”, a więc możemy zająć się niektórymi punktami w naszej osobistej liście rzeczy do zrobienia. Teraz nadszedł dobry moment na przekazanie plików do repozytorium i być może na zrobienie sobie przerwy na herbatę.

```
$ git commit -am"Egzekwowanie w widoku listy weryfikacji modelu."
```

Refaktoryzacja — usunięcie na stałe zdefiniowanych adresów URL

Czy pamiętasz parametry `name=` w pliku `urls.py`? Skopiowaliśmy je po prostu z domyślnego przykładu dostarczanego przez Django i nadaliśmy im pewne opisowe nazwy. Teraz dowieś się, do czego służą wspomniane parametry.

Plik `lists/urls.py`:

```
url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),  
url(r'^new$', 'lists.views.new_list', name='new_list'),
```

Znacznik szablonu { % url % }

Zdefiniowany na stałe adres URL w pliku `home.html` można zastąpić znacznikiem szablonu Django, który odwołuje się do nazwy adresu URL.

Plik `lists/templates/home.html` (ch10l026-1):

```
{% block form_action %}{% url 'new_list' %}{% endblock %}
```

Sprawdzamy, czy testy jednostkowe nadal są zaliczane:

```
$ python3 manage.py test lists  
OK
```

Podobną zmianę wprowadzamy w drugim szablonie. Ten jest znacznie bardziej interesujący, ponieważ otrzymuje parametr.

Plik *lists/templates/list.html* (ch10l026-2):

```
{% block form_action %}{% url 'view_list' list.id %}{% endblock %}
```

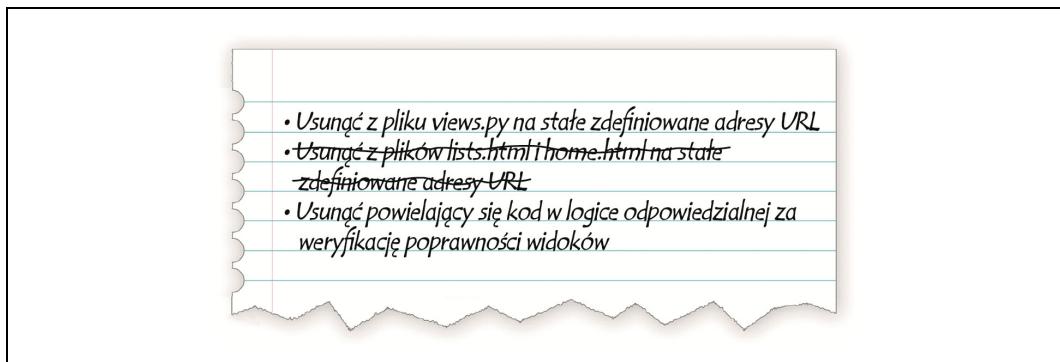
Więcej informacji znajdziesz w dokumentacji Django poświęconej określaniu nazw adresów URL⁷.

Ponownie wykonujemy testy i sprawdzamy, czy zostają zaliczone:

```
$ python3 manage.py test lists  
OK  
$ python3 manage.py test functional_tests  
OK
```

Doskonale (patrz rysunek 10.4):

```
$ git commit -am"Refactor hard-coded URLs out of templates"
```



Rysunek 10.4. Nasza osobista lista rzeczy do zrobienia

Użycie `get_absolute_url` w przekierowaniach

Bierzemy się teraz za plik *views.py*. Jedno z możliwych rozwiązań jest podobne do zastosowanego w szablonie — chodzi o przekazanie nazwy adresu URL i argumentu pozycyjnego.

Plik *lists/views.py* (ch10l026-3):

```
def new_list(request):  
    [...]  
    return redirect('view_list', list_.id)
```

Wprowadzona zmiana spowoduje zaliczenie testów jednostkowych i funkcjonalnych, ale możliwości funkcji `redirect()` są znacznie większe! Ponieważ obiekty modelu są często powiązane z określonymi adresami URL, więc w Django można zdefiniować funkcję specjalną o nazwie `get_absolute_url()`, wskazującą stronę wyświetlającą element. To jest użyteczne w omawianym przykładzie, a także podczas przeprowadzania zadań administracyjnych w Django (nie zostaną przedstawione w tej książce, ale szybko je odkryjesz). Wymieniona funkcja pozwala na przejście

⁷ <https://docs.djangoproject.com/en/1.7/topics/http/urls/#reverse-resolution-of-urls>

od analizy danego obiektu w widoku administracyjnym do jego analizy w działającej witrynie. Zawsze zalecam zdefiniowanie funkcji `get_absolute_url()` dla modelu, o ile ma to sens; takie zadanie nie wymaga poświęcenia zbyt dużej ilości czasu.

Potrzebny będzie jedynie niezwykle prosty test jednostkowy w pliku `test_models.py`.

Plik `lists/tests/test_models.py` (ch10l026-4):

```
def test_get_absolute_url(self):
    list_ = List.objects.create()
    self.assertEqual(list_.get_absolute_url(), '/lists/%d/' % (list_.id,))
```

Wykonanie testu powoduje wygenerowanie następujących danych wyjściowych:

```
AttributeError: 'List' object has no attribute 'get_absolute_url'
```

Poniżej przedstawiono implementację obejmującą funkcję `reverse()` w Django, której działanie jest praktycznie przeciwnieństwem standardowych działań podejmowanych przez Django dla pliku `urls.py` (patrz *dokumentacja*⁸).

Plik `lists/models.py` (ch10l026-5):

```
from django.core.urlresolvers import reverse

class List(models.Model):

    def get_absolute_url(self):
        return reverse('view_list', args=[self.id])
```

Teraz możemy użyć funkcji `get_absolute_url()` w widoku. Funkcja `redirect()` pobiera obiekt do przekierowania, a następnie w tle automatycznie używa funkcji `get_absolute_url()`.

Plik `lists/views.py` (ch10l026-6):

```
def new_list(request):
    [...]
    return redirect(list_)
```

Więcej informacji na ten temat znajdziesz w *dokumentacji Django*⁹. Sprawdzamy jeszcze, czy testy jednostkowe są nadal zaliczane:

```
OK
```

Następnie to samo rozwiążanie stosujemy w `view_list`.

Plik `lists/views.py` (ch10l026-7):

```
def view_list(request, list_id):
    [...]

        item.save()
        return redirect(list_)
    except ValidationError:
        error = "Element nie może być pusty"
```

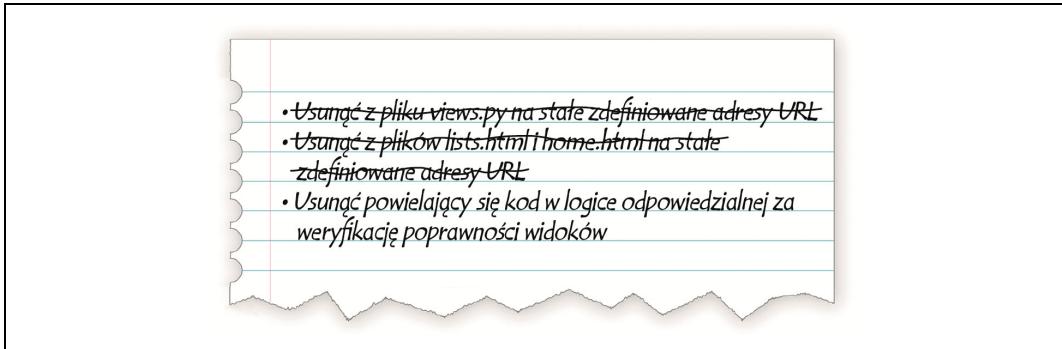
Na koniec wykonujemy w pełni testy jednostkowe i funkcjonalne, aby mieć gwarancję, że wszystko działa zgodnie z oczekiwaniemi:

```
$ python3 manage.py test lists
OK
$ python3 manage.py test functional_tests
OK
```

Pozostało już tylko skreślić pozycję z naszej osobistej listy rzeczy do zrobienia (patrz rysunek 10.5).

⁸ <https://docs.djangoproject.com/en/1.7/topics/http/urls/#reverse-resolution-of-urls>

⁹ <https://docs.djangoproject.com/en/1.7/topics/http/shortcuts/#redirect>



Rysunek 10.5. Nasza osobista lista rzeczy do zrobienia

Warto teraz przekazać pliki do repozytorium:

```
$ git commit -am"Use get_absolute_url on List model to DRY urls in views"
```

Ostatni element naszej listy będzie tematem kolejnego rozdziału...

Podpowiedzi dotyczące organizacji testów i refaktoryzacji

Użycie katalogu `tests`

Podobnie jak kod aplikacji znajduje się w wielu plikach, tak samo testy powinieneś podzielić i umieścić w oddzielnych plikach.

- Użyj katalogu o nazwie `tests`, dodaj plik `__init__.py` odpowiedzialny za import wszystkich klas testów.
- W przypadku testów funkcjonalnych pogrupuj je na testy dla poszczególnych funkcji lub informacji pochodzących od użytkownika.
- W przypadku testów jednostkowych powinieneś stosować oddzielny plik testu dla każdego testowanego pliku kodu źródłowego. Jeżeli używasz framework'a Django, zwykle oznacza to konieczność utworzenia plików `test_models.py`, `test_views.py` i `test_forms.py`.
- Przynajmniej przygotuj miejsce zarezerwowane na test dla każdej funkcji i klasy.

Nie zapomnij o etapie refaktoryzacji w cyklu czerwony, zielony, refaktoryzacja

Celem przygotowywania testów jest możliwość przeprowadzenia refaktoryzacji kodu! Dlatego używaj ich i postaraj się zapewnić maksymalną przejrzystość kodu.

Nie przeprowadzaj refaktoryzacji, gdy testy kończą się niepowodzeniem

- W ogóle!
- Nie liczą się testy funkcjonalne, nad którymi obecnie pracujesz.
- Czasami możesz pominąć test przeznaczony do sprawdzenia funkcjonalności, której jeszcze nie zaimplementowałeś.
- Znacznie częściej stosuje się następujące podejście: utworzenie notatki o komponentie przeznaczonym do refaktoryzacji, zakończenie obecnego zadania i odłożenie refaktoryzacji na później, gdy aplikacja znów będzie w stanie „działająca”.
- Nie zapomnij o usunięciu dekoratorów `@skip` przed przekazaniem kodu do repozytorium. Zawsze powinieneś wiersz po wierszu przeglądać wprowadzone zmiany, aby wychwycić tego rodzaju polecenia.

Prosty formularz

Na końcu poprzedniego rozdziału dowiedziałeś się o istnieniu zbyt dużej ilości powielonego kodu w logice odpowiedzialnej za sprawdzanie poprawności widoków. Framework Django zachęca do użycia klas formularza podczas weryfikacji danych wejściowych użytkownika oraz do dokonania wyboru wyświetlanych komunikatów błędów. Zobaczmy, jak takie rozwiązanie sprawdza się w praktyce.

W tym rozdziale poświęcimy także nieco czasu na uporządkowanie testów jednostkowych i upewnienie się, że każdy z nich w danej chwili sprawdza tylko jedną rzecz.

Przeniesienie do formularza logiki odpowiedzialnej za sprawdzanie poprawności danych



W Django skomplikowane formularze mają zapach kodu. Powstaje więc pytanie o możliwość przeniesienia tej logiki do formularza, własnych metod w klasie modelu lub też niestandardowego modułu Django przedstawiającego logikę biznesową.

Formularze w Django mają kilka potężnych możliwości:

- Mogą przetwarzać dane wejściowe użytkownika i sprawdzać je pod kątem błędów.
- Mogą być użyte w szablonach do generowania elementów HTML dla danych wejściowych oraz komunikatów błędów.
- Jak się później przekonasz, niektóre mogą również zapisywać dane w bazie danych.

Nie musisz wykorzystywać wszystkich powyższych możliwości w każdym formularzu sieciowym. Być może preferujesz przygotowanie własnego kodu HTML lub też procedury zapisu danych. Jednak formularz to doskonałe miejsce na umieszczenie logiki odpowiedzialnej za sprawdzenie poprawności danych.

Użycie testu jednostkowego do analizy API formularzy

Przeprowadzimy teraz mały eksperyment z formularzami, używając do tego testu jednostkowego. Moim celem jest osiągnięcie pełnego rozwiązania i wprowadzenie Cię powoli do tematu formularzy, aby wszystko miało sens, jeśli nigdy wcześniej nie spotkałeś się z formularzami.

Przede wszystkim dodajemy nowy plik dla testów jednostkowych formularza i rozpoczynamy od testu, który po prostu sprawdza kod HTML formularza.

Plik `lists/tests/test_forms.py`:

```
from django.test import TestCase

from lists.forms import ItemForm

class ItemFormTest(TestCase):

    def test_form_renders_item_text_input(self):
        form = ItemForm()
        self.fail(form.as_p())
```

Metoda `form.as_p()` powoduje wygenerowanie formularza jako kodu HTML. Test jednostkowy używa wywołania `self.fail()` do przeprowadzenia pewnych operacji sprawdzenia kodu. Równie łatwo mógłbyś użyć sesji `manage.py shell`, choć wówczas każda zmiana wymagałaby odświeżania kodu.

Przygotowujemy minimalny formularz sieciowy. Dziedziczy po klasie bazowej `Forms` i posiada pojedyncze pole o nazwie `item_text`.

Plik `lists/forms.py`:

```
from django import forms

class ItemForm(forms.Form):
    item_text = forms.CharField()
```

Otrzymujemy teraz komunikat błędu pokazujący, jak będzie przedstawał się automatycznie wygenerowany formularz HTML:

```
self.fail(form.as_p())
AssertionError: <p><label for="id_item_text">Tekst elementu:</label> <input
id="id_item_text" name="item_text" type="text" /></p>
```

Tak naprawdę kod jest niezwykle podobny do zdefiniowanego w pliku `base.html`. Brakuje atrybutu `placeholder` oraz klas frameworka Bootstrap CSS. Tworzymy teraz test jednostkowy dla formularza.

Plik `lists/tests/test_forms.py`:

```
class ItemFormTest(TestCase):

    def test_form_item_input_has_placeholder_and_css_classes(self):
        form = ItemForm()
        self.assertIn('placeholder="Wpisz rzecz do zrobienia"', form.as_p())
        self.assertIn('class="form-control input-lg"', form.as_p())
```

Wynikiem jest niepowodzenie testu, co stanowi uzasadnienie potrzeby utworzenia pewnego kodu. W jaki sposób możemy dostosować do własnych potrzeb pole danych wejściowych w formularzu? Za pomocą „widżetu”. Poniżej przedstawiono jedynie wyświetlenie odpowiedzi w polu.

Plik *lists/forms.py*:

```
class ItemForm(forms.Form):
    item_text = forms.CharField(
        widget=forms.fields.TextInput(attrs={
            'placeholder': 'Wpisz rzecz do zrobienia',
        })),
)
```

Otrzymujemy następujące dane wyjściowe:

```
AssertionError: 'class="form-control input-lg"' not found in '<p><label
for="id_item_text">Tekst elementu:</label> <input id="id_item_text" name="item_text"
placeholder="Wpisz rzecz do zrobienia" type="text" /></p>'
```

A następnie wprowadzamy drobną modyfikację widżetu.

Plik *lists/forms.py*:

```
widget=forms.fields.TextInput(attrs={
    'placeholder': 'Wpisz rzecz do zrobienia',
    'class': 'form-control input-lg',
}),
```



Przeprowadzanie modyfikacji widżetu w przedstawiony powyżej sposób bardzo szybko stanie się żmudne w przypadku ogromnych i skomplikowanych formularzy sieciowych. Zobacz, jaką pomoc oferuje w tym zakresie Django za pomocą *django-crispy-forms*¹ i *django-floppyforms*².

Programowanie sterowane testami — testy jednostkowe usprawiedliwiające tworzenie testowego kodu

Czy to nie brzmi trochę jak programowanie sterowane testami? To w porządku, teraz i później.

Kiedy poznajesz nowe API, oczywiście przez jakiś czas możesz przeprowadzać na nim pewne eksperymenty, zanim powrócisz do rygoru narzuconego przez TDD. Dozwolone jest użycie interaktywnej konsoli lub utworzenie testowego kodu (musisz jednak obiecać Testing Goat, że wspomniany kod testowy później wyrzucisz i poprawnie utworzysz go od początku).

W tym miejscu tworzymy test jednostkowy będący formą eksperymentów z API formularzy. To naprawdę jest całkiem dobre podejście pozwalające na poznanie sposobu działania pewnych rozwiązań.

¹ <https://django-crispy-forms.readthedocs.org/en/latest/>

² <http://django-floppyforms.readthedocs.org/en/latest/>

Przejście do Django ModelForm

Co dalej? Chcemy, aby formularz ponownie wykorzystywał zdefiniowany już w modelu kod odpowiedzialny za sprawdzanie danych. Django oferuje klasę specjalną o nazwie `ModelForm`, która automatycznie generuje formularz dla modelu. Jak będziesz mógł zobaczyć, wspomniana klasa jest konfigurowana za pomocą atrybutu specjalnego `Meta`.

Plik `lists/forms.py`:

```
from django import forms

from lists.models import Item

class ItemForm(forms.ModelForm):

    class Meta:
        model = Item
        fields = ('text',)
```

W atrybucie `Meta` podajemy model, dla którego jest przeznaczony formularz, oraz wskazujemy pola przeznaczone do użycia.

Klasa `ModelForm` wykonuje wszelkiego rodzaju zadania, takie jak przypisywanie odpowiednich typów elementu HTML `<input>` poszczególnym polom, a także stosuje domyślne sprawdzanie poprawności danych. Więcej informacji na ten temat znajdziesz w *dokumentacji*³.

Teraz otrzymujemy nieco inaczej wyglądający formularz HTML:

```
AssertionError: 'placeholder="Wpisz rzecz do zrobienia"' not found in '<p><label
for="id_text">Tekst:</label> <textarea cols="40" id="id_text" name="text"
rows="10">\r\n</textarea></p>'
```

Zniknęła podpowiedź wyświetlaną przez element oraz klasy CSS. Możesz również zauważyc użycie `name="text"` zamiast `name="item_text"`. To prawdopodobnie będzie zbyt dużym problemem. Jednak formularz został zbudowany w oparciu o element `<textarea>` zamiast zwykłego `<input>`, a to nie jest interfejs użytkownika, jaki chcemy zastosować w aplikacji. Na szczęście istnieje możliwość nadpisania widżetów dla pól `ModelForm`, podobnie jak to zrobiliśmy wcześniej w zwykłym formularzu.

Plik `lists/forms.py`:

```
class ItemForm(forms.ModelForm):

    class Meta:
        model = Item
        fields = ('text',)
        widgets = {
            'text': forms.fields.TextInput(attrs={
                'placeholder': 'Wpisz rzecz do zrobienia',
                'class': 'form-control input-lg',
            }),
        }
```

Teraz test zostaje zaliczony.

³ <https://docs.djangoproject.com/en/1.7/topics/forms/modelforms/>

Testowanie i dostosowanie do własnych potrzeb logiki weryfikacji formularza

Zobaczmy teraz, czy klasa ModelForm stosuje te same reguły weryfikacji, które zostały zdefiniowane w modelu. Dowiesz się także, jak przekazać dane do formularza, tak jakby zostały wprowadzone przez użytkownika.

Plik `lists/tests/test_forms.py` (ch11l008):

```
def test_form_validation_for_blank_items(self):
    form = ItemForm(data={'text': ''})
    form.save()
```

Wynikiem jest następujący błąd:

```
ValueError: The Item could not be created because the data didn't validate.
```

Dobrze. Formularz nie pozwala na zachowanie danych, jeśli pole tekstowe pozostaje puste.

Teraz zobaczymy, czy można uzyskać określony komunikat błędu. API przeznaczone do sprawdzenia poprawności formularza *przed* podjęciem próby zapisu danych to funkcja o nazwie `is_valid()`.

Plik `lists/tests/test_forms.py` (ch11l009):

```
def test_form_validation_for_blank_items(self):
    form = ItemForm(data={'text': ''})
    self.assertFalse(form.is_valid())
    self.assertEqual(
        form.errors['text'],
        ["Element listy nie może być pusty"]
    )
```

Wynikiem wywołania `form.is_valid()` jest wartość `True` lub `False`, choć efektem ubocznym będzie weryfikacja danych wejściowych pola oraz wypełnienie atrybutu `errors`. Wymieniony atrybut to słownik mapowań nazw pól na listę błędów dla tych pól (istnieje możliwość, że pole będzie miało więcej niż tylko jeden błąd).

Otrzymujemy następujące dane wyjściowe:

```
AssertionError: ['This field is required.'] != ["Element listy nie może być pusty"]
```

Django zawiera domyślny komunikat błędu wyświetlany użytkownikowi — możesz się już z nim spotkać, jeśli w pośpiechu tworzyłeś aplikację sieciową. Tutaj jednak zachowujemy ostrożność i starannie przygotowujemy komunikat błędu. Jego dostosowanie oznacza konieczność modyfikacji `error_messages`, czyli kolejnej zmiennej klasy Meta.

Plik `lists/forms.py` (ch11l010):

```
class Meta:
    model = Item
    fields = ('text',)
    widgets = {
        'text': forms.fields.TextInput(attrs={
            'placeholder': 'Wpisz rzecz do zrobienia',
            'class': 'form-control input-lg',
        }),
    }
    error_messages = {
        'text': {'required': "Element listy nie może być pusty"}
    }
```

Testy są zaliczone:

OK

Czy wiesz, jak jeszcze lepiej można poradzić sobie z ciągami tekstowymi komunikatów błędów? Za pomocą stałej.

Plik *lists/forms.py* (ch11l011):

```
EMPTY_LIST_ERROR = "Element listy nie może być pusty"
[...]
error_messages = {
    'text': {'required': EMPTY_LIST_ERROR}
}
```

Ponownie wykonujemy testy i widzimy, że są zaliczone. Dobrze, teraz możemy zmienić test.

Plik *lists/tests/test_forms.py* (ch11l012):

```
from lists.forms import EMPTY_LIST_ERROR, ItemForm
[...]
def test_form_validation_for_blank_items(self):
    form = ItemForm(data={'text': ''})
    self.assertFalse(form.is_valid())
    self.assertEqual(form.errors['text'], [EMPTY_LIST_ERROR])
```

Testy nadal są zaliczane:

OK

Doskonale. Warto teraz przekazać pliki do repozytorium:

```
$ git status # Polecenie powinno wyświetlić pliki lists/forms.py i tests/test_forms.py.
$ git add lists
$ git commit -m "Nowy formularz dla elementów listy."
```

Użycie formularza w widokach

Początkowo chciałem rozbudować przedstawiony formularz, aby zaimplementować obsługę unikalności elementów, jak również sprawdzenie pod kątem pustego elementu. Jednak istnieje pewne następstwo zastosowania metody „wdrażaj aplikację jak najwcześniej” nazywane „łącz kod jak najwcześniej”. Innymi słowy, wykorzystując ten sposób tworzenia kodu formularzy, można przez całe wieki bardzo łatwo dodawać do niego kolejne funkcje. Doskonale o tym wiem, ponieważ właśnie takie rozwiązywanie zastosowałem podczas przygotowywania szkicu niniejszego rozdziału. Gdy dodałem wszelkie funkcje i wszystko było już dopięte na ostatni guzik, uszwiadomiłem sobie, że takie rozwiązanie nie sprawdza się w najprostszych sytuacjach.

Dlatego też staraj się jak najwcześniej używać nowo tworzonego kodu. W ten sposób będziesz miał pewność, że w aplikacji nigdy nie znajdą się nieużywane fragmenty kodu. Zyskasz możliwość wcześniego testowania kodu w rzeczywistym środowisku.

Mamy klasę formularza pozwalającą na wygenerowanie pewnego kodu HTML oraz prowadzenie weryfikacji pod kątem przynajmniej jednego rodzaju błędu, więc zacznijmy jej używać! Wspomnianej klasy możemy użyć w szablonie *base.html*, a więc we wszystkich widokach.

Użycie formularza w widoku za pomocą żądania GET

Pracę rozpoczynamy w testach jednostkowych dla widoku strony głównej. Opracowane w starym stylu metody `test_home_page_returns_correct_html()` i `test_root_url_resolves_to_home_page_view()` zastępujemy zestawem testów opartym na kliencie testów Django. Na początek pozostawiamy stare testy, aby sprawdzić, czy nowe są ich odpowiednikami.

Plik `lists/tests/test_views.py` (ch11l013):

```
from lists.forms import ItemForm

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        [...]

    def test_home_page_returns_correct_html(self):
        request = HttpRequest()
        [...]

    def test_home_page_renders_home_template(self):
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'home.html') #❶

    def test_home_page_uses_item_form(self):
        response = self.client.get('/')
        self.assertIsInstance(response.context['form'], ItemForm) #❷
```

- ❶ Metody pomocnicze `assertTemplateUsed()` używamy w celu zastąpienia starego testu ręcznego szablonem.
- ❷ Metody `isinstance()` używamy do sprawdzenia, czy widoki stosują odpowiedniego rodzaju formularz.

Otrzymujemy następujący wynik:

```
KeyError: 'form'
```

Możemy więc użyć formularza w widoku strony głównej.

Plik `lists/views.py` (ch11l014):

```
[...]
from lists.forms import ItemForm
from lists.models import Item, List

def home_page(request):
    return render(request, 'home.html', {'form': ItemForm()})
```

Dobrze. Teraz spróbujemy użyć formularza w szablonie. Poprzednie elementy `<input ...>` zastępujemy elementem `{% form.text %}`.

Plik `lists/templates/base.html` (ch11l015):

```
<form method="POST" action="{% block form_action %}{% endblock %}">
    {{ form.text }}
    {% csrf_token %}
    {% if error %}
        <div class="form-group has-error">
```

Wywołanie `{% form.text %}` powoduje wygenerowanie elementu HTML `<input>` dla pola typu `text` formularza sieciowego.

W tym momencie poprzedni test jest już niepotrzebny:

```
self.assertEqual(response.content.decode(), expected_html)
AssertionError: '<!DOCTYPE html> <input class="form-control input-lg"
id="text">\n' != '<!DOCTYPE html> \n
[233 chars]>\n'
```

Komunikat błędu jest trudny do odczytania. Dlatego też warto go nieco uprościć.

Plik *lists/tests/test_views.py* (ch11l016):

```
class HomePageTest(TestCase):
    maxDiff = None #❶
    [...]
    def test_home_page_returns_correct_html(self):
        request = HttpRequest()
        response = home_page(request)
        expected_html = render_to_string('home.html')
        self.assertMultiLineEqual(response.content.decode(), expected_html) #❷
```

- ❶ Metoda `assertMultiLineEqual()` jest użyteczna podczas porównywania dużej ilości ciągów tekstowych. Generuje dane wyjściowe w stylu polecenia `diff`, ale domyślnie skraca długie bloki poszczególnych różnic...

- ❷ ... i dlatego też na początku klasy testu konieczne jest ustawienie `maxDiff = None`.

Wynikiem jest oczywiście błąd, ponieważ wywołanie `render_to_string()` nie ma żadnych informacji o formularzu:

```
[...]
<form method="POST" action="/lists/new">
-         <input class="form-control input-lg" id="id_text"
name="text" placeholder="Wpisz rzecz do zrobienia" type="text" />
+
[...]
```

To jednak można bardzo łatwo naprawić.

Plik *lists/tests/test_views.py*:

```
def test_home_page_returns_correct_html(self):
    request = HttpRequest()
    response = home_page(request)
    expected_html = render_to_string('home.html', {'form': ItemForm()})
    self.assertMultiLineEqual(response.content.decode(), expected_html)
```

I ponownie testy są zaliczane. Zyskaliśmy pewność, że sposób zachowania nie uległ zmianie i dlatego możemy już usunąć dwa stare testy. Metody `assertTemplateUsed()` i `response.text()` sprawdzają, czy nowe testy są wystarczające do przetestowania podstawowego widoku za pomocą żądania GET.

Klasa `HomePageTest` zawiera teraz jedynie dwie metody testowe.

Plik *lists/tests/test_views.py* (ch11l017):

```
class HomePageTest(TestCase):
    def test_home_page_renders_home_template(self):
        [...]
    def test_home_page_uses_item_form(self):
        [...]
```

Duża operacja znajdź i zastąp

Istotnym skutkiem przeprowadzonej modyfikacji formularza jest to, że nie używa on już tych samych atrybutów id i name. Jeżeli wykonasz testy funkcjonalne, to zobaczysz niepowodzenie natychmiast w trakcie próby odszukania elementu <input>:

```
selenium.common.exceptions NoSuchElementException: Message: 'Unable to locate  
element: {"method":"id","selector":"id_new_item"}' ; Stacktrace:
```

Ten błąd trzeba naprawić, co oznacza konieczność przeprowadzenia dużej operacji znajdź i zastąp. Zanim jednak do tego przystąpimy, warto przekazać pliki do repozytorium i oddzielić zmian nazw od zmian w logice aplikacji.

```
$ git diff # Przejrzyj zmiany wprowadzone w plikach home.html, views.py oraz ich testach.  
$ git commit -am "Użycie nowego formularza na stronie głównej, uproszczenie testów. Zmiany spowodowały  
↳niezaliczenie testów funkcjonalnych."
```

Przystępujemy do poprawienia testów funkcjonalnych. Wydanie polecenia grep wskazuje na użycie id_new_item w wielu miejscach:

```
$ grep id_new_item functional_tests/*  
functional_tests/test_layout_and_styling.py:           inputbox =  
self.browser.find_element_by_id('id_new_item')  
functional_tests/test_layout_and_styling.py:           inputbox =  
self.browser.find_element_by_id('id_new_item')  
functional_tests/test_list_item_validation.py:  
self.browser.find_element_by_id('id_new_item').send_keys('\n')  
[...]
```

To jest doskonała okazja do przeprowadzenia refaktoryzacji. W pliku base.py tworzymy nową metodę pomocniczą.

Plik functional_tests/base.py (ch11l018):

```
class FunctionalTest(StaticLiveServerCase):  
    [...]  
    def get_item_input_box(self):  
        return self.browser.find_element_by_id('id_text')
```

Tę przygotowaną metodę będziemy często wykorzystywać, trzeba więc wprowadzić trzy zmiany w pliku test_simple_list_creation.py, dwie w test_layout_and_styling.py oraz cztery w test_list_item_validation.py, na przykład jak pokazano poniżej.

Plik functional_tests/test_simple_list_creation.py:

```
# Od razu zostaje zachecona, aby wpisać rzecz do zrobienia.  
inputbox = self.get_item_input_box()
```

lub

Plik functional_tests/test_list_item_validation.py:

```
# ... pusty element na liście. Naciśnęła klawisz Enter w pustym polu tekstowym.  
self.browser.get(self.server_url)  
self.get_item_input_box().send_keys('\n')
```

Nie pokazuję tutaj wszystkich miejsc, w których trzeba wprowadzić zmiany. Jestem pewien, że doskonale sobie z tym poradzisz. Później możesz ponownie wydać polecenie grep i sprawdzić, czy na pewno wprowadziłeś wszystkie niezbędne zmiany.

Wykonaliśmy pierwszy krok, ale teraz trzeba dostosować pozostały kod aplikacji. Konieczne będzie więc wyszukanie wszystkich wystąpień `id(id_new_item)` i `name(item_text)`, a następnie zastąpienie ich przez odpowiednio `id_text` i `text`:

```
$ grep -r id_new_item lists/  
lists/static/base.css:#id_new_item {
```

Powyżej przedstawiono tylko jedną zmianę, druga jest przeprowadzana podobnie:

```
$ grep -Ir item_text lists  
lists/views.py:    item = Item(text=request.POST['item_text'], list=list_ )  
lists/views.py:        item = Item(text=request.POSt['item_text'],  
lists/tests/test_views.py:            data={'item_text': 'Nowy element listy'}  
lists/tests/test_views.py:            data={'item_text': 'Nowy element listy'}  
lists/tests/test_views.py:            response = self.client.post('/lists/new',  
data={'item_text': ''})  
[...]
```

Po wprowadzeniu zmian można ponownie wykonać testy jednostkowe i sprawdzić, czy wszystko nadal działa zgodnie z oczekiwaniemi:

```
$ python3 manage.py test lists  
Creating test database for alias 'default'...  
.....  
-----  
Ran 17 tests in 0.126s  
  
OK  
Destroying test database for alias 'default'...
```

Nie zapomnij o wykonaniu testów funkcjonalnych:

```
$ python3 manage.py test functional_tests  
[...]  
File "/workspace/superlists/functional_tests/test_simple_list_creation.py",  
line 40, in test_can_start_a_list_and_retrieve_it_later  
    return self.browser.find_element_by_id('id_text')  
File "/workspace/superlists/functional_tests/base.py", line 31, in  
get_item_input_box  
    return self.browser.find_element_by_id('id_text')  
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate  
element: {"method":"id","selector":"id_text"}' ; Stacktrace:  
[...]  
FAILED (errors=3)
```

Wynik nie jest całkiem zgodny z oczekiwaniami! Zobaczmy, co się tutaj dzieje. Po sprawdzeniu wiersza wymienionego w komunikacie okazuje się, że po każdej operacji wysłania pierwszego elementu listy element `<input>` znika ze strony.

Jeżeli przeanalizujemy plik `views.py` i widok `new_list`, wówczas odkryjemy tam przyczynę problemu. W przypadku wystąpienia błędu podczas operacji sprawdzania poprawności nie następuje przekazanie formularza do szablonu `home.html`.

Plik `lists/views.py`:

```
except ValidationError:  
    error = "Element listy nie może być pusty"  
    return render(request, 'home.html', {"error": error})
```

W wymienionym widoku także użyjemy naszego formularza. Jednak przed wprowadzeniem dodatkowych zmian warto przekazać pliki do repozytorium:

```
$ git status  
$ git commit -am"Zmiana nazw wszystkich atrybutów id i names elementów <input>. Testy funkcjonalne  
→nadal niezaliczone."
```

Użycie formularza w widoku obsługującym żądania POST

Kolejnym zadaniem jest dostosowanie testów jednostkowych do widoku `new_list`. To dotyczy zwłaszcza testu związanego z weryfikacją danych. Zróbmy to więc na początek.

Plik `lists/tests/test_views.py`:

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

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')
        expected_error = escape("Element listy nie może być pusty")
        self.assertContains(response, expected_error)
```

Adaptacja testów jednostkowych dla widoku `new_list`

Omawiany tutaj test przeprowadza sprawdzenie jednorazowo zbyt wielu rzeczy, a więc mamy możliwość wprowadzenia pewnych zmian. Test powinien zostać podzielony na dwie oddzielne asercje:

- Jeżeli w trakcie operacji sprawdzania poprawności wystąpi błąd, wtedy należy wygenerować szablon `home.html` wraz z kodem stanu 200.
- Jeżeli w trakcie operacji sprawdzania poprawności wystąpi błąd, wtedy odpowiedź powinna zawierać komunikat błędu.

Ponadto można dodać nową asercję:

- Jeżeli w trakcie operacji sprawdzania poprawności wystąpi błąd, wtedy obiekt formularza powinien zostać przekazany szablonowi.

Przy okazji zdefiniowany na stałe ciąg tekstowy komunikatu błędu możemy zastąpić stałą.

Plik `lists/tests/test_views.py` (ch11l023):

```
from lists.forms import ItemForm, EMPTY_LIST_ERROR
[...]

class NewListTest(TestCase):
    [...]

    def test_for_invalid_input_renders_home_template(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')

    def test_validation_errors_are_shown_on_home_page(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertContains(response, escape(EMPTY_LIST_ERROR))

    def test_for_invalid_input_passes_form_to_template(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertIsInstance(response.context['form'], ItemForm)
```

Teraz klasa przedstawia się znacznie lepiej. Poszczególne testy są przeznaczone do sprawdzania pojedynczych rzeczy. Przy odrobinie szczęścia tylko jeden zakończy się niepowodzeniem, a jego komunikat błędu wyraźnie wskaże przyczynę problemów:

```
$ python3 manage.py test lists
[...]
=====
ERROR: test_for_invalid_input_passes_form_to_template
(lists.tests.test_views.NewListTest)
-----
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests/test_views.py", line 55, in
    test_for_invalid_input_passes_form_to_template
    self.assertIsInstance(response.context['form'], ItemForm)
[...]
KeyError: 'form'

-----
Ran 19 tests in 0.041s

FAILED (errors=1)
```

Użycie formularza w widoku

Poniżej pokazano sposób, w jaki formularz zostanie użyty w widoku.

Plik *lists/views.py*:

```
def new_list(request):
    form = ItemForm(data=request.POST) #❶
    if form.is_valid(): #❷
        list_ = List.objects.create()
        Item.objects.create(text=request.POST['text'], list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form}) #❸
```

- ❶ Do konstruktora formularza zostają przekazane dane `request.POST`.
- ❷ Metoda `form.is_valid()` jest używana w celu sprawdzenia, czy przekazane dane są prawidłowe.
- ❸ W przypadku nieprawidłowych danych szablonowi zostaje przekazany formularz zamiast na stałe zdefiniowanego ciągu tekstowego błędu.

Ten widok teraz prezentuje się znacznie lepiej. Poza jednym wszystkie testy funkcjonalne zostają zaliczone:

```
self.assertContains(response, escape(EMPTY_LIST_ERROR))
[...]
AssertionError: False is not true : Couldn't find 'Element listy nie może być pusty' in response
```

Użycie formularza w celu wyświetlenia błędów w szablonie

Test kończy się niepowodzeniem, ponieważ w szablonie jeszcze nie użyliśmy formularza do wyświetlania błędów.

Plik *lists/templates/base.html* (ch11l026):

```
<form method="POST" action="{% block form_action %}{% endblock %}>
  {{ form.text }}
  {{ csrf_token }}
```

```

{%
    if form.errors %} #❶
        <div class="form-group has-error">
            <div class="help-block">{{ form.text.errors }}</div> #❷
        </div>
    {% endif %}
</form>

```

- ❶ W `form.errors` znajduje się lista wszystkich błędów, które odkryto w formularzu.
- ❷ W `form.text.errors` znajduje się lista błędów jedynie dla pola typu `text`.

Co musimy jeszcze zrobić z testami?

```

FAIL: test_validation_errors_end_up_on_lists_page
(lists.tests.test_views.ListViewTest)
[...]
AssertionError: False is not true : Couldn't find 'Element listy nie może być pusty' in response

```

Nadal mamy nieoczekiwane niepowodzenie — w testach dotyczących ostatniego widoku o nazwie `view_list`. Ponieważ zmianie uległ sposób wyświetlania błędów we *wszystkich* szablonach, nie będzie dłużej wyświetlany błąd ręczni przekazywany do szablonu.

To oznacza konieczność modyfikacji widoku `view_list`, aby aplikacja znów znalazła się w stanie określonym mianem „działająca”.

Użycie formularza w innym widoku

Ostatni omawiany tutaj widok obsługuje żądania zarówno `GET`, jak i `POST`. Pracę rozpoczęliśmy od sprawdzenia użycia formularza w żądaniach `GET`. Do tego celu można przygotować nowy test.

Plik `lists/tests/test_views.py`:

```

class ListViewTest(TestCase):
    [...]

    def test_displays_item_form(self):
        list_ = List.objects.create()
        response = self.client.get('/lists/%d/' % (list_.id,))
        self.assertIsInstance(response.context['form'], ItemForm)
        self.assertContains(response, 'name="text"')

```

Otrzymujemy następujące dane wyjściowe:

```
KeyError: 'form'
```

Poniżej przedstawiono minimalną implementację widoku.

Plik `lists/views.py` (ch11l028):

```

def view_list(request, list_id):
    [...]
    form = ItemForm()
    return render(request, 'list.html', {
        'list': list_, "form": form, "error": error
    })

```

Metoda pomocnicza dla wielu krótkich testów

Błędy wykryte w formularzu chcemy wykorzystać w drugim widoku. Dlatego też obecną metodę testową dla nieprawidłowych danych (`test_validation_errors_end_up_on_lists_page()`) podzielimy na kilka oddzielnych.

Plik `lists/tests/test_views.py` (ch11l030):

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

    def post_invalid_input(self):
        list_ = List.objects.create()
        return self.client.post(
            '/lists/%d/' % (list_.id,),
            data={'text': ''}
        )

    def test_for_invalid_input_nothing_saved_to_db(self):
        self.post_invalid_input()
        self.assertEqual(Item.objects.count(), 0)

    def test_for_invalid_input_renders_list_template(self):
        response = self.post_invalid_input()
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'list.html')

    def test_for_invalid_input_passes_form_to_template(self):
        response = self.post_invalid_input()
        self.assertIsInstance(response.context['form'], ItemForm)

    def test_for_invalid_input_shows_error_on_page(self):
        response = self.post_invalid_input()
        self.assertContains(response, escape EMPTY_LIST_ERROR))
```

Dzięki utworzeniu niewielkiej funkcji pomocniczej `post_invalid_input()` możemy wykonać cztery oddzielne testy bez niepotrzebnego powielania dużej liczby wierszy kodu.

Wielokrotnie spotykałem się już z tego rodzaju sytuacją. Bardziej naturalne wydaje się tworzenie testów widoku w postaci pojedynczego, monolitycznego bloku asercji — widok powinien zrobić to i to, a następnie zwrócić to. Jednak podział całości na wiele testów na pewno niesie ze sobą korzyści. Jak zobaczyłeś w poprzednich rozdziałach, podział testów pomaga w izolacji konkretnego problemu, który może później wystąpić, gdy zmodyfikujesz kod i przypadkowo wprowadzisz błąd. Metody pomocnicze to jedne z narzędzi zmniejszających barierę psychologiczną.

Na przykład teraz otrzymujemy jedno niepowodzenie i jasny komunikat dotyczący jego przyczyny:

```
FAIL: test_for_invalid_input_shows_error_on_page
(lists.tests.test_views.ListViewTest)
AssertionError: False is not true : Couldn't find 'Element listy nie może być pusty' in response
```

Zobaczmy, czy możliwe jest odpowiednie przeprojektowanie widoku, aby używał opracowanego wcześniej formularza. Poniżej przedstawiono pierwsze podejście.

Plik `lists/views.py`:

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ItemForm()
    if request.method == 'POST':
        form = ItemForm(data=request.POST)
        if form.is_valid():
            Item.objects.create(text=request.POST['text'], list=list_)
    return redirect(list_)
return render(request, 'list.html', {'list': list_, "form": form})
```

Wszystkie testy jednostkowe zostają zaliczone:

```
Ran 23 tests in 0.086s
```

```
OK
```

A jak to wygląda w przypadku testów funkcjonalnych?

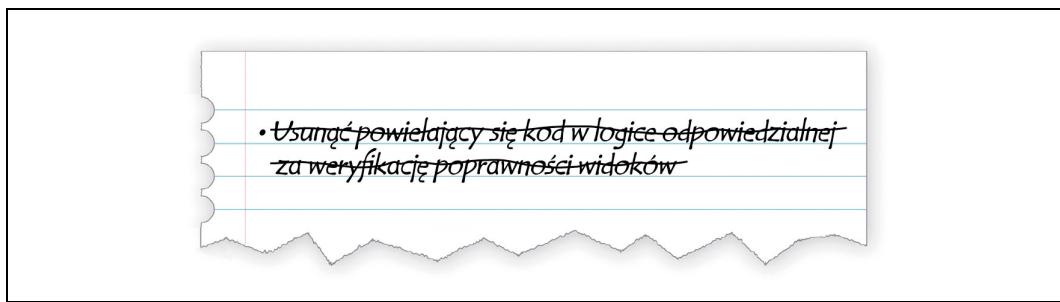
```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
...
```

```
Ran 3 tests in 12.154s
```

```
OK
```

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

Hura! Czy czujesz ulgę? Wprowadziliśmy poważną zmianę w naszej niewielkiej aplikacji — element `<input>` wraz z atrybutami `id` i `name` ma znaczenie krytyczne dla prawidłowego działania aplikacji. Podczas refaktoryzacji pracowaliśmy z siedmioma lub ośmioma różnymi plikami i przeprowadziliśmy dość dużą operację. To jest ten rodzaj operacji, której wykonanie bez użycia testów poważnie by mnie martwiło. Mógłbym nawet uznać, że nie warto kombinować z kodem działającym prawidłowo. Skoro mamy pełny zestaw testów, można się pokusić o eksperymenty z kodem i przeprowadzenie operacji porządkowych, mając świadomość, że testy wychwycią wszelkie popelnione pomyłki. Obecność testów zwiększa prawdopodobieństwo przeprowadzenia refaktoryzacji, porządkowania i utrzymania tendencji, aby kod był elegancki, przejrzysty, schludny, precyzyjny, spójny, funkcjonalny i dobry. Z naszej osobistej listy rzeczy do zrobienia usuwamy kolejną pozycję (patrz rysunek 11.1).



Rysunek 11.1. Nasza osobista lista rzeczy do zrobienia

Zdecydowanie powinniśmy przekazać pliki do repozytorium:

```
$ git diff
$ git commit -am"Użycie formularza we wszystkich widokach, aplikacja znów działa."
```

Użycie metody save() formularza

Możemy wykonać jeszcze kilka zadań, aby dalej uprościć widoki. Wcześniej wspomniałem, że formularze mają możliwość zapisu danych w bazie danych. W przypadku budowanej tutaj aplikacji tego rodzaju rozwiążanie nie działa standardowo, ponieważ element musi „wiedzieć”, na której liście powinien być zapisany. To nie jest zbyt trudna poprawka do wprowadzenia.

Pracę jak zwykle zaczynamy od przygotowania testu. W celu ilustracji problemu zobaczymy, co się stanie w przypadku próby wywołania `form.save()`.

Plik `lists/tests/test_forms.py` (ch11l032):

```
def test_form_save_handles_saving_to_a_list(self):
    form = ItemForm(data={'text': 'dowolne zadanie'})
    new_item = form.save()
```

Framework Django nie może wykonać zleconego zadania, ponieważ element musi należeć do jakieś listy:

```
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
```

Rozwiążanie polega na wskazaniu metodzie `save()` listy, w której powinien zostać zapisany element.

Plik `lists/tests/test_forms.py`:

```
from lists.models import Item, List
[...]
def test_form_save_handles_saving_to_a_list(self):
    list_ = List.objects.create()
    form = ItemForm(data={'text': 'dowolne zadanie'})
    new_item = form.save(for_list=list_)
    self.assertEqual(new_item, Item.objects.first())
    self.assertEqual(new_item.text, 'dowolne zadanie')
    self.assertEqual(new_item.list, list_)
```

Następnie trzeba się upewnić o prawidłowym zapisaniu elementu w bazie danych wraz z odpowiednimi atrybutami:

```
TypeError: save() got an unexpected keyword argument 'for_list'
```

Poniżej przedstawiono przykładowy sposób własnej implementacji metody `save()`.

Plik `lists/forms.py` (ch11l034):

```
def save(self, for_list):
    self.instance.list = for_list
    return super().save()
```

Atrybut `instance` w formularzu przedstawia modyfikowany lub tworzony obiekt bazy danych. Dowiedziałem się o tym, pisząc niniejszy rozdział! Istnieją jeszcze inne rozwiązania, na przykład ręczne utworzenie obiektu lub użycie argumentu `commit=False`, ale jak sądzę, użycie powyższego jest najbardziej eleganckie. W następnym rozdziale poznasz jeszcze inne sposoby wskazywania formularzowi listy, w której ma zostać zapisany element:

```
Ran 24 tests in 0.086s
```

```
OK
```

Wreszcie możemy przystąpić do refaktoryzacji widoków. Na pierwszy ogień idzie `new_list`.

Plik *lists/views.py*:

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List.objects.create()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

Ponownie wykonaj testy, aby się upewnić, że wszystkie nadal są zaliczane:

```
Ran 24 tests in 0.086s
```

```
OK
```

Następnie przeprowadzamy refaktoryzację widoku `view_list`.

Plik *lists/views.py*:

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ItemForm()
    if request.method == 'POST':
        form = ItemForm(data=request.POST)
        if form.is_valid():
            form.save(for_list=list_)
            return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

Wszystkie testy nadal są zaliczane:

```
Ran 24 tests in 0.111s
```

```
OK
```

i:

```
Ran 3 tests in 14.367s
```

```
OK
```

Doskonale! Dwa widoki w naszej aplikacji przypominają teraz „zwykłe” widoki Django — pobierają informacje z żądania wykonanego przez użytkownika, łącząc je z własną logiką lub informacjami pobranymi z adresu URL (`list_id`), przekazując do formularza sieciowego w celu weryfikacji i ewentualnego zapisu, a następnie przekierowując lub generując szablon.

Formularze i ich weryfikacja mają duże znaczenie w Django oraz ogólnie w programowaniu sieciowym. W następnym rozdziale zobaczymy, czy można utworzyć nieco bardziej skomplikowane rozwiązanie niż tutaj omówione.

Podpowiedzi

Niewielkie widoki

Jeżeli musisz analizować skomplikowane widoki i tworzyć dla nich testy, to najwyższa pora na zastanowienie się, czy znajdująca się w nich logikę można przenieść w inne miejsce, prawdopodobnie do formularza, jak przedstawiono w rozdziale. Inne miejsce docelowe dla wspomnianej logiki to własna metoda w klasie modelu. Kiedy poziom skomplikowania aplikacji będzie tego wymagał, logikę można przenieść z plików typowych dla Django i umieścić we własnych klasach oraz funkcjach przechwytyujących podstawową logikę biznesową aplikacji.

Każdy test powinien być poświęcony testowaniu jednej rzeczy

Jeżeli test zawiera więcej niż tylko jedną asercję, wówczas heurystyka będzie podejrzana. Czasami dwie asercje są tak blisko ze sobą powiązane, że mogą pozostać razem. Jednak czasami przygotowywane są testy przeznaczone do testowania wielu zachowań. W takich przypadkach warto podzielić test na kilka oddzielnych. Dzięki wykorzystaniu funkcji pomocniczych można uniknąć nadmiernej ilości kodu w funkcjach testowych.

Bardziej skomplikowane formularze

Teraz zajmiemy się przykładami użycia znacznie bardziej skomplikowanych formularzy. Skoro pomogliśmy użytkownikom w uniknięciu dodawania pustych elementów listy, teraz pomóżemy im w uniknięciu powielania elementów na liście.

W tym rozdziale przejdziemy do znacznie bardziej złożonych szczegółów dotyczących weryfikacji formularzy w Django. Jeżeli temat dostosowania formularzy Django do własnych potrzeb nie jest Ci obcy, ten rozdział możesz uznać za opcjonalny. Natomiast jeśli dopiero poznajesz framework Django, znajdziesz tutaj wiele przydatnych informacji. Oczywiście nadal możesz pominąć ten rozdział. Zapoznaj się tylko z ramką dotyczącą głupoty programistów, a na końcu rozdziału z omawianymi tutaj widokami.

Kolejny test funkcjonalny dotyczący powielonych elementów

Do klasy `ItemValidationTest` dodajemy drugą metodę testową.

Plik `functional_tests/test_list_item_validation.py` (ch12l001):

```
def test_cannot_add_duplicate_items(self):
    # Edyta przeszła na stronę główną i zaczęła tworzyć nową listę.
    self.browser.get(self.server_url)
    self.get_item_input_box().send_keys('Kupić kalosze\n')
    self.check_for_row_in_list_table('1: Kupić kalosze')

    # Przypadkowo spróbowała wpisać element, który już znajdował się na liście.
    self.get_item_input_box().send_keys('Kupić kalosze\n')

    # Otrzymała czytelny komunikat błędu.
    self.check_for_row_in_list_table('1: Kupić kalosze')
    error = self.browser.find_element_by_css_selector('.has-error')
    self.assertEqual(error.text, "Podany element już istnieje na liście.")
```

Dlaczego mamy dwie metody testowe zamiast rozbudowanej jednej? To jest uzasadnione rozwiązańem. Obie metody są blisko ze sobą związane, przeprowadzają weryfikację tego samego elementu `<input>`, więc umieszczenie ich w jednym pliku wydaje się być właściwym podejściem. Z drugiej strony, pod względem logicznym są na tyle różne, że ma sens umieszczenie ich w oddzielnych metodach:

```
$ python3 manage.py test functional_tests.test_list_item_validation
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":".has-error"}' ; Stacktrace:
Ran 2 tests in 9.613s
```

Dobrze. W tym momencie wiemy o zaliczeniu pierwszego z dwóch testów. Już słyszę, jak pytasz, czy istnieje możliwość wykonania jedynie testu kończącego się niepowodzeniem. Tak, istnieje taka możliwość:

```
$ python3 manage.py test functional_tests.\test_list_item_validation.ItemValidationTest.test_cannot_add_duplicate_items
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":".has-error"}' ; Stacktrace:
```

Ochrona przed duplikatami w warstwie modelu

Poniżej przedstawiono rozwiązanie, które nas najbardziej interesuje. To nowy test sprawdzający, czy pojawienie się na liście drugiego takiego samego elementu spowoduje wygenerowanie błędu.

Plik *lists/tests/test_models.py* (ch09l028):

```
def test_duplicate_items_are_invalid(self):
    list_ = List.objects.create()
    Item.objects.create(list=list_, text='bla')
    with self.assertRaises(ValidationError):
        item = Item(list=list_, text='bla')
        item.full_clean()
```

Kiedy błąd zostaje zgłoszony, przeprowadzany jest kolejny test odpowiedzialny za sprawdzenie, czy nie przesadzamy czasem ze zdefiniowanymi ograniczeniami w zakresie spójności.

Plik *lists/tests/test_models.py* (ch09l029):

```
def test_CAN_save_same_item_to_different_lists(self):
    list1 = List.objects.create()
    list2 = List.objects.create()
    Item.objects.create(list=list1, text='bla')
    item = Item(list=list2, text='bla')
    item.full_clean() # Nie powinien być zgłoszony.
```

Zawsze lubię umieszczać niewielki komentarz w przypadku testów sprawdzających, czy dana sytuacja *nie* powinna zgłosić błędu. W przeciwnym razie może być trudno określić, co tak naprawdę jest testowane.

```
AssertionError: ValidationError not raised
```

Jeżeli celowo chcesz popełnić błąd, możesz to zrobić na przykład w przedstawiony poniżej sposób.

Plik *lists/models.py* (ch09l030):

```
class Item(models.Model):
    text = models.TextField(default='', unique=True)
    list = models.ForeignKey(List, default=None)
```

To pozwala na sprawdzenie, czy drugi test faktycznie wychwytuje powstały problem:

```
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests/test_models.py", line 62, in
test_CAN_save_same_item_to_different_lists
    item.full_clean() # should not raise
    [...]
django.core.exceptions.ValidationError: {'text': ['Item with this Text already
exists.']}
```

Kiedy tworzyć test pod kątem głupoty programisty?

Jedno z uzasadnień pojawiających się podczas tworzenia testów brzmi mniej więcej tak: „sprawdzenie, czy nie zrobiłem czegoś głupiego”. Ogólnie rzecz biorąc, należy uważać na wspomniane testy.

W omawianym przykładzie przygotowaliśmy test sprawdzający, czy możliwe jest zapisanie na liście dwóch takich samych elementów. Najprostszym sposobem na zaliczenie tego testu za pomocą jak najmniejszej liczby wierszy kodu będzie uniemożliwienie zapisu *wszelkich* powielonych elementów. To uzasadni utworzenie kolejnego testu pomimo faktu, że to będzie po prostu bez sensu.

Jednak nie można przygotować testów sprawdzających każdy sposób, na jaki można nieprawidłowo utworzyć kod. Jeżeli masz funkcję obliczającą sumę dwóch liczb, możesz dla niej przygotować kilka testów:

```
assert adder(1, 1) == 2
assert adder(2, 1) == 3
```

Jednocześnie masz prawo przypuszczać, że implementacja wspomnianej funkcji nie została celowo zepsuta:

```
def adder(a, b):
    # Mało prawdopodobne, aby taki kod występował!
    if a == 3:
        return 666
    else:
        return a + b
```

Musisz więc sobie zaufać, że bezsensownych rzeczy nie robisz *celowo*, a jedynie *przypadkowo*.

Podobnie jak w przypadku klasy `ModelForm` modele mają klasę `Meta`, w której można zaimplementować ograniczenia wskazujące na konieczność zachowania unikalności elementu na liście lub też wartości jego atrybutów `text` i `list`.

Plik `lists/models.py` (ch09l031):

```
class Item(models.Model):
    text = models.TextField(default='')
    list = models.ForeignKey(List, default=None)

    class Meta:
        unique_together = ('list', 'text')
```

Na tym etapie warto zjrzeć do *dokumentacji*¹ Django, aby dowiedzieć się więcej na temat atrybutów modelu.

¹ <https://docs.djangoproject.com/en/1.7/ref/models/options/>

Mała dygresja dotycząca kolejności API Querystring i przedstawiania ciągu tekstowego

Po wykonaniu testów otrzymamy nieoczekiwane niepowodzenie:

```
=====
FAIL: test_saving_and_retrieving_items
(lists.tests.test_models.ListAndItemModelsTest)

Traceback (most recent call last):
  File "/workspace/superlists/lists/tests/test_models.py", line 31, in
test_saving_and_retrieving_items
    self.assertEqual(first_saved_item.text, 'Absolutnie pierwszy element listy')
AssertionError: 'Element drugi' != 'Absolutnie pierwszy element listy'
- Element drugi
[...]
```



W zależności od używanej platformy i jej instalacji SQLite powyższy błąd może nie wystąpić. Warto jednak przeczytać ten fragment rozdziału, ponieważ kod i testy są interesujące same w sobie.

To całkiem duża zagadka. Przechodzimy do debugowania za pomocą wywołań `print()`.

Plik `lists/tests/test_models.py`:

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

Otrzymamy następujące dane wyjściowe:

```
.....Element drugi
Absolutnie pierwszy element listy
P.....
```

Wygląda na to, że konieczność zachowania unikalności namieszała z domyślną kolejnością zapytań, takich jak `Item.objects.all()`. Wprawdzie mamy już test kończący się niepowodzeniem, ale najlepszym rozwiązaniem będzie dodanie następnego testu odpowiedzialnego za sprawdzenie kolejności.

Plik `lists/tests/test_models.py` (ch09l032):

```
def test_list_ordering(self):
    list1 = List.objects.create()
    item1 = Item.objects.create(list=list1, text='e1')
    item2 = Item.objects.create(list=list1, text='element 2')
    item3 = Item.objects.create(list=list1, text='3')
    self.assertEqual(
        Item.objects.all(),
        [item1, item2, item3]
    )
```

Mamy kolejny test kończący się niepowodzeniem, ale wygenerowany komunikat nie należy do czytelnych.

```
AssertionError: [<Item: Item object>, <Item: Item object>, <Item: Item object>]
!= [<Item: Item object>, <Item: Item object>, <Item: Item object>]
```

Potrzebujemy lepszej reprezentacji ciągów tekstowych dla naszych obiektów. Dodajemy więc kolejny test jednostkowy.



Kiedy masz test kończący się niepowodzeniem, zwykle zachowujesz wstrzemięźliwość w dodawaniu kolejnych — to znacznie utrudnia odczyt danych wyjściowych i ogólnie może spowodować większe zdenerwowanie. Rodzi się pytanie, czy kiedykolwiek uda się przywrócić aplikację do stanu działania? W omawianym przykładzie mamy jedynie bardzo proste testy, więc nie powinieneś mieć obaw.

Plik *lists/tests/test_models.py* (ch12l008):

```
def test_string_representation(self):
    item = Item(text='dowolny tekst')
    self.assertEqual(str(item), 'dowolny tekst')
```

Otrzymujemy następujące dane wyjściowe:

```
AssertionError: 'Item object' != 'dowolny tekst'
```

Ponadto mamy dwa inne testy kończące się niepowodzeniem. Warto przystąpić do usuwania błędów.

Plik *lists/models.py* (ch09l034):

```
class Item(models.Model):
    ...
    def __str__(self):
        return self.text
```



W Pythonie 2.x dla Django metodą służącą do reprezentacji ciągu tekstowego jest `__unicode__()`. Podobnie jak ogólnie obsługa ciągów tekstowych została ona uproszczona w Pythonie 3. Więcej informacji na ten temat znajdziesz w dokumentacji².

Mamy teraz dwa testy kończące się niepowodzeniem, a test dotyczący kolejności wyświetla znacznie czytelniejszy komunikat błędu:

```
AssertionError: [<Item: 3>, <Item: e1>, <Item: element 2>] != [<Item: e1>, <Item: element 2>, <Item: 3>]
```

Odpowiednią poprawkę możemy wprowadzić w klasie Meta.

Plik *lists/models.py* (ch09l035):

```
class Meta:
    ordering = ('id',)
    unique_together = ('list', 'text')
```

Czy rozwiązanie działa?

```
AssertionError: [<Item: i1>, <Item: item 2>, <Item: 3>] != [<Item: i1>, <Item: item 2>, <Item: 3>]
```

Ups. Rozwiążanie działało; możesz zobaczyć, że elementy są w tej samej kolejności, ale testy wskazują co innego. Tak naprawdę nieustannie stykam się z tym problemem — elementy queryset w Django nie są zbyt dobrze porównywane z listami. Rozwiązaniem może być konwersja w teście obiektu queryset na listę³, jak przedstawiono poniżej.

² <https://docs.djangoproject.com/en/1.7/topics/python3/#str-and-unicode-methods>

³ Można wykorzystać także metodę `assertSequenceEqual()` z modułu `unittest` oraz `assertQuerysetEqual()` z narzędzi testowych Django. Kiedy po raz ostatni przyglądałem się `assertQuerysetEqual()`, byłem niezwykle zakłopotany...

Plik *lists/tests/test_models.py* (ch09l036):

```
self.assertEqual(  
    list(Item.objects.all()),  
    [item1, item2, item3]  
)
```

Takie rozwiązanie działa i wreszcie mamy zaliczone wszystkie testy:

OK

Przepisanie testu starego modelu

Rozbudowany test modelu stanowił nieocenioną pomoc podczas wyszukiwania nieoczekiwanych błędów, ale teraz trzeba go przepisać od nowa. Wcześniej przygotowaliśmy wspomniany test w rozwlekłym stylu, aby zaprezentować wprowadzenie do ORM w Django. Skoro teraz mamy test przeznaczony do sprawdzania kolejności elementów, tę samą funkcjonalność możemy osiągnąć dzięki kilku krótszym testom. Usuń więc metodę `test_saving_and_retrieving_items()` i zastąp ją przedstawionym poniżej kodem.

Plik *lists/tests/test_models.py* (ch12l010).

```
class ListAndItemModelsTest(TestCase):  
  
    def test_default_text(self):  
        item = Item()  
        self.assertEqual(item.text, '')  
  
    def test_item_is_related_to_list(self):  
        list_ = List.objects.create()  
        item = Item()  
        item.list = list_  
        item.save()  
        self.assertIn(item, list_.item_set.all())  
    [...]
```

To naprawdę więcej niż potrzeba — sprawdzenie wartości domyślnych atrybutów w nowo zainicjowanym obiekcie modelu jest wystarczające do sprawdzenia, czy w pliku *models.py* ustawione zostały wartości. Test „element jest powiązany z listą” stanowi rzeczywisty test pozwalający upewnić się, że relacja klucza zewnętrznego działa.

Przy okazji możemy podzielić plik na testy przeznaczone dla elementów `Item` i `List` — dla tego drugiego istnieje tylko jeden test o nazwie `test_get_absolute_url()`.

Plik *lists/tests/test_models.py* (ch12l011):

```
class ItemModelTest(TestCase):  
    def test_default_text(self):  
        [...]  
  
class ListModelTest(TestCase):  
    def test_get_absolute_url(self):  
        [...]
```

To znacznie bardziej elegancka i uporządkowana postać testów:

```
$ python3 manage.py test lists  
[...]  
Ran 29 tests in 0.092s
```

OK

Pewne błędy spójności ujawniają się podczas zapisu

Zanim przejdziemy dalej, do omówienia pozostał jeszcze jeden temat. Czy pamiętasz, jak w rozdziale 10. wspomniałem o błędach spójności danych pojawiających się podczas zapisu? Wszystko zależy od tego, jakie ograniczenia dotyczące spójności są tak naprawdę egzekwowane przez bazę danych.

Spróbuj wydać polecenie `makemigrations`, a zobaczysz, że Django chce dodać ograniczenie `unique_together` do samej bazy danych, zamiast korzystać z ograniczenia zdefiniowanego w warstwie aplikacji:

```
$ python3 manage.py makemigrations
Migrations for 'lists':
  0005_auto_20140414_2038.py:
    - Alter unique_together for item (1 constraints)
```

Jeżeli teraz w teście dotyczącym powielonych elementów metodę `save()` zastąpimy metodą `full_clean()`...

Plik `lists/tests/test_models.py`:

```
def test_duplicate_items_are_invalid(self):
    list_ = List.objects.create()
    Item.objects.create(list=list_, text='bla')
    with self.assertRaises(ValidationError):
        item = Item(list=list_, text='bla')
        # item.full_clean()
        item.save()
```

... wówczas otrzymamy następujący komunikat błędu:

```
ERROR: test_duplicate_items_are_invalid (lists.tests.test_models.ItemModelTest)
[...]
    return Database.Cursor.execute(self, query, params)
sqlite3.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
[...]
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

Jak możesz zobaczyć, komunikat błędu jest przekazywany z SQLite i to zupełnie inny rodzaj błędu niż oczekiwany: `IntegrityError` zamiast `ValidationError`.

Cofamy zmiany wprowadzone w teście i sprawdzamy, czy testy znów zostaną zaliczone:

```
$ python3 manage.py test lists
[...]
Ran 29 tests in 0.092s
OK
```

To jest odpowiednia pora na przekazanie plików do repozytorium:

```
$ git status #Polecenie powinno pokazać zmiany wprowadzone w testach i modelach, a także nową migrację.
# Nowej migracji nadajemy lepszą nazwę.
$ mv lists/migrations/0005_auto* lists/migrations/0005_list_item_unique_together.py
$ git add lists
$ git diff --staged
$ git commit -am "Implementacja w warstwie modelu sprawdzenia pod kątem powielonych elementów."
```

Eksperymenty w warstwie widoku sprawdzające, czy są powielone elementy

Wykonujemy teraz testy funkcjonalne, aby zobaczyć, na jakim znajdują się etapie:

```
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate  
element: {"method":"id","selector":"id_list_table"}' ; Stacktrace:
```

Jeżeli tego nie zauważysz, to podpowiadam, że dane wyjściowe to 500⁴. Szybki test jednostkowy na poziomie widoku nie pozostawia żadnych złudzeń.

Plik *lists/tests/test_views.py* (ch12l014):

```
class ListViewTest(TestCase):  
    [...]  
  
    def test_for_invalid_input_shows_error_on_page(self):  
        [...]  
  
    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):  
        list1 = List.objects.create()  
        item1 = Item.objects.create(list=list1, text='textey')  
        response = self.client.post(  
            '/lists/%d/' % (list1.id,),  
            data={'text': 'textey'})  
        expected_error = escape("Podany element już istnieje na liście.")  
        self.assertContains(response, expected_error)  
        self.assertTemplateUsed(response, 'list.html')  
        self.assertEqual(Item.objects.all().count(), 1)
```

Otrzymujemy następujące dane wyjściowe:

```
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,  
lists_item.text
```

Chcemy uniknąć błędów związanych ze spójnością danych! W idealnej sytuacji oczekiwane jest wywołanie `is_valid()` w celu wskazania istnienia powielonego elementu jeszcze przed próbą przeprowadzenia operacji zapisu. Jednak w takim przypadku formularz musi wcześniej „wiedzieć”, która lista będzie używana.

Na razie pominiemy ten test.

Plik *lists/tests/test_views.py* (ch12l015):

```
from unittest import skip  
[...]  
  
@skip  
def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
```

⁴ Wyświetlany jest błąd serwera o kodzie stanu 500. Powinieneś poznać nieco żargonu!

Bardziej skomplikowany formularz do obsługi unikalności elementów

Formularz przeznaczony do tworzenia nowej listy musi otrzymać tylko jedno: tekst nowego elementu. Z kolei formularz sprawdzający, czy elementy listy są unikatowe, musi otrzymać zarówno listę, jak i nowy element. Podobnie jak nadpisaliśmy metodę `save()` w klasie `ItemForm`, także tym razem nadpiszemy konstruktor klasy nowego formularza, aby przekazać mu nazwę listy.

Powielimy testy z poprzedniego formularza i tylko odrobinę je zmodyfikujemy.

Plik `lists/tests/test_forms.py` (ch12l016):

```
from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_LIST_ERROR,
    ExistingListItemForm, ItemForm
)
[...]

class ExistingListItemTest(TestCase):

    def test_form_renders_item_text_input(self):
        list_ = List.objects.create()
        form = ExistingListItemForm(for_list=list_)
        self.assertIn('placeholder="Wpisz rzecz do zrobienia"', form.as_p())

    def test_form_validation_for_blank_items(self):
        list_ = List.objects.create()
        form = ExistingListItemForm(for_list=list_, data={'text': ''})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors['text'], [EMPTY_LIST_ERROR])
    def test_form_validation_for_duplicate_items(self):
        list_ = List.objects.create()
        Item.objects.create(list=list_, text='żadnych powtórzeń!')
        form = ExistingListItemForm(for_list=list_, data={'text': 'żadnych powtórzeń!'})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors['text'], [DUPLICATE_ITEM_ERROR])
```

Możemy przeprowadzić kilka cykli TDD — nie pokażę ich wszystkich, ale jestem przekonany, że zrobisz wszystko prawidłowo. Pamiętaj, Testing Goat widzi wszystko! Powtarzaj cykle do chwili, aż otrzymasz formularz wraz z niestandardowym konstruktorem, który po prostu ignoruje jego argument `for_list`.

Plik `lists/forms.py` (ch09l071):

```
DUPLICATE_ITEM_ERROR = "Podany element już istnieje na liście."
[...]
class ExistingListItemForm(forms.ModelForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

Otrzymujemy następujące dane wyjściowe:

```
ValueError: ModelForm has no model class specified.
```

Patrzmy teraz, czy dziedziczenie po naszym istniejącym formularzu okaże się choć trochę pomocne.

Plik *lists/forms.py* (ch09l072):

```
class ExistingListItemForm(ItemForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

Pozostał tylko jeden test kończący się niepowodzeniem:

```
FAIL: test_form_validation_for_duplicate_items
(lists.tests.test_forms.ExistingListItemFormTest)
    self.assertFalse(form.is_valid())
AssertionError: True is not False
```

Kolejny krok wymaga pewnej wiedzy o wewnętrznych mechanizmach Django. W dokumentacji tego frameworka znajdziesz informacje dotyczące *weryfikacji modelu*⁵ oraz *weryfikacji formularzy*⁶.

Metody o nazwie `validate_unique()` Django używa zarówno dla formularzy, jak i modeli. Możemy więc użyć i jednego, i drugiego w połączeniu z atrybutem `instance`.

Plik *lists/forms.py*:

```
from django.core.exceptions import ValidationError
[...]
class ExistingListItemForm(ItemForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.instance.list = for_list

    def validate_unique(self):
        try:
            self.instance.validate_unique()
        except ValidationError as e:
            e.error_dict = {'text': [DUPLICATE_ITEM_ERROR]}
            self._update_errors(e)
```

Powyżej mamy przykład zastosowania odrobinę magii Django. W zasadzie pobieramy błąd weryfikacji, dostosowujemy jego komunikat błędu, a następnie ponownie przekazujemy do formularza. I na tym koniec. Pora na szybkie przekazanie plików do repozytorium:

```
$ git diff
$ git commit -a
```

Użycie istniejącego formularza w widoku listy

Zobaczmy teraz, czy nasz formularz możemy wykorzystać w widoku.

Usuwamy dodaną wcześniej dyrektywę `skip`, teraz można już używać nowej stałej. Pięknie.

Plik *lists/tests/test_views.py* (ch12l049):

```
from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_LIST_ERROR,
    ExistingListItemForm, ItemForm,
)
```

⁵ <https://docs.djangoproject.com/en/1.7/ref/models/instances/#validating-objects>

⁶ <https://docs.djangoproject.com/en/1.7/ref/forms/validation/>

```
[...]
```

```
def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
    [...]
    expected_error = escape(DUPLICATE_ITEM_ERROR)
```

W ten sposób powracamy do błędu dotyczącego spójności danych:

```
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

Poprawka polega na przejściu do użycia klasy nowego formularza. Zanim przystąpimy do jej implementacji, musimy wyszukać testy sprawdzające klasę formularza i je zmodyfikować.

Plik *lists/tests/test_views.py* (ch12l050):

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

    def test_displays_item_form(self):
        list_ = List.objects.create()
        response = self.client.get('/lists/%d/' % (list_.id,))
        self.assertIsInstance(response.context['form'], ExistingListItemForm)
        self.assertContains(response, 'name="text"')

    [...]

    def test_for_invalid_input_passes_form_to_template(self):
        response = self.post_invalid_input()
        self.assertIsInstance(response.context['form'], ExistingListItemForm)
```

Otrzymujemy następujące dane wyjściowe:

```
AssertionError: <lists.forms.ItemForm object at 0x7f767e4b7f90> is not an
instance of <class 'lists.forms.ExistingListItemForm'>
```

Możemy więc dostosować widok.

Plik *lists/views.py* (ch12l051):

```
from lists.forms import ExistingListItemForm, ItemForm
[...]
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ExistingListItemForm(for_list=list_)
    if request.method == 'POST':
        form = ExistingListItemForm(for_list=list_, data=request.POST)
        if form.is_valid():
            form.save()
    [...]
```

W ten sposób naprawiamy *niemal* wszystko, z wyjątkiem nieoczekiwanej niepowodzenia:

```
TypeError: save() missing 1 required positional argument: 'for_list'
```

Nasza niestandardowa metoda `save()` z klasy nadrzednej `ItemForm` nie jest już potrzebna. Przeprowadzamy tutaj szybki test jednostkowy.

Plik *lists/tests/test_forms.py* (ch12l053):

```
def test_form_save(self):
    list_ = List.objects.create()
    form = ExistingListItemForm(for_list=list_, data={'text': 'hi'})
    new_item = form.save()
    self.assertEqual(new_item, Item.objects.all()[0])
```

Formularz możemy utworzyć, wywołując metodę `save()` z klasy znajdującej się jeszcze wyżej w hierarchii.

Plik `lists/forms.py` (ch12l054):

```
def save(self):
    return forms.models.ModelForm.save(self)
```



Pozwól sobie tutaj na osobistą opinię. Wprawdzie mógłbym użyć `super`, ale staram się tego unikać, gdy konieczne jest podanie argumentów. Odkryłem, że wywołanie `super()` w Pythonie 3 bez argumentów działa niezawodnie i pozwala na dostanie się do klasy nadrzędnej. Każde inne wywołanie `super()` jest podatne na błędy i nie działa najlepiej. Twoje odczucia i doświadczenia mogą być zupełnie odmienne.

I wreszcie osiągnęliśmy cel — wszystkie testy jednostkowe zostały zaliczone:

```
$ python3 manage.py test lists
[...]
Ran 34 tests in 0.082s
```

OK

To samo dotyczy testów funkcjonalnych:

```
$ python3 manage.py test functional_tests.test_list_item_validation
Creating test database for alias 'default'...
...
-----
Ran 2 tests in 12.048s

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

Jako ostateczne potwierdzenie ponownie wykonujemy *wszystkie* testy funkcjonalne:

```
$ python3 manage.py test functional_tests
Creating test database for alias 'default'...
...
-----
Ran 4 tests in 19.048s

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

Hura! Czas na ostateczne przekazanie plików do repozytorium i powtórzenie tego, czego nauczyliśmy się w kilku ostatnich rozdziałach o testowaniu widoków.

W następnym rozdziale postaramy się, aby weryfikacja danych była nieco bardziej przyjazna dla użytkownika, wykorzystamy więc odrobinę kodu działającego po stronie klienta. O nie, chyba wiesz, co to oznacza...

Przypomnienie — co należy testować w widokach?

Listing częściowy pokazuje wszystkie testy widoku i asercje.

```
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        response = self.client.get('/lists/%d/' % (list_.id,)) #❶
        self.assertTemplateUsed(response, 'list.html') #❷
    def test_passes_correct_list_to_template(self):
        self.assertEqual(response.context['list'], correct_list) #❸
    def test_displays_item_form(self):
        self.assertIsInstance(response.context['form'], ExistingListItemForm) #❹
        self.assertContains(response, 'name="text"')
    def test_displays_only_items_for_that_list(self):
        self.assertContains(response, 'itemey 1') #❺
        self.assertContains(response, 'itemey 2') #❻
        self.assertNotContains(response, 'iny element listy 1') #❼
    def test_can_save_a_POST_request_to_an_existing_list(self):
        self.assertEqual(Item.objects.count(), 1) #❽
        self.assertEqual(new_item.text, 'Nowy element na istniejącej liście') #❾
    def test_POST_redirects_to_list_view(self):
        self.assertRedirects(response, '/lists/%d/' % (correct_list.id,)) #❿
    def test_for_invalid_input_nothing_saved_to_db(self):
        self.assertEqual(Item.objects.count(), 0) #❿
    def test_for_invalid_input_renders_list_template(self):
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'list.html') #❿
    def test_for_invalid_input_passes_form_to_template(self):
        self.assertIsInstance(response.context['form'], ExistingListItemForm) #❿
    def test_for_invalid_input_shows_error_on_page(self):
        self.assertContains(response, escape EMPTY_LIST_ERROR) #❿
    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        self.assertContains(response, expected_error)
        self.assertTemplateUsed(response, 'list.html')
        self.assertEqual(Item.objects.all().count(), 1)
```

- ❶ Użycie klienta testów Django.
- ❷ Sprawdzenie użycia szablonu. Następnie sprawdzenie każdego elementu w kontekście szablonu.
- ❸ Sprawdzenie, czy obiekty są prawidłowe lub czy obiekty queryset mają odpowiednie elementy.
- ❹ Sprawdzenie, czy formularze są oparte na prawidłowych klasach.
- ❺❻❽❾ Sprawdzenie wszelkiej logiki szablonów; polecenia `for` i `if` powinny przeprowadzać minimalne testy.
- ❿ W przypadku widoków obsługujących żądania POST trzeba się upewnić o przetestowaniu zarówno sytuacji, gdy żądanie jest prawidłowe, jak i nieprawidłowe.
- ❿ Sprawdzenie, czy formularz został wygenerowany, a ewentualne błędy wyświetcone.

Skąd wzięły się powyższe punkty? Przejdz do dodatku B, a przekonasz się, że to w zupełności wystarczy do upewnienia się o prawidłowości widoków po przeprowadzeniu refaktoryzacji mającej na celu rozpoczęcie użycia widoków opartych na klasach.

Zagłębiamy się ostrożnie w JavaScript

Gdyby dobry Bóg naprawdę chciał, byśmy się cieszyli, nie ofiarowałby nam cennego prezentu w postaci nieustającego cierpienia.

— John Calvin (przedstawiony w *Calvin and the Chipmunks*¹)

Nasza nowa logika odpowiedzialna za weryfikację jest dobra, ale byłoby wskazane, aby komunikat błędu zniknął, gdy użytkownik rozpoczęcie usuwanie problemu. Do tego celu musimy wykorzystać odrobinę kodu JavaScript.

Ponieważ jesteśmy zepsuci możliwością codziennego programowania w tak przyjemnym języku, jakim niewątpliwie jest Python, konieczność użycia języka JavaScript możemy uznać za karę. Dlatego też bardzo ostrożnie zagłębiamy się w JavaScript.



Przyjęłem założenie, że znasz podstawy składni języka JavaScript. Jeżeli jeszczе nie przeczytałeś książek *JavaScript — mocne strony*, powinieneś to zrobić niezwłocznie. To nie jest zbyt długa lektura.

Rozpoczynamy od testów funkcjonalnych

Do klasy `ItemValidationTest` dodajemy nowy test funkcjonalny.

Plik `functional_tests/test_list_item_validation.py` (ch14l001):

```
def test_error_messages_are_cleared_on_input(self):
    # Edyta utworzyła nową listę w sposób, który spowodował powstanie błędu weryfikacji:
    self.browser.get(self.server_url)
    self.get_item_input_box().send_keys('\n')
    error = self.browser.find_element_by_css_selector('.has-error')
    self.assertTrue(error.is_displayed()) #❶

    # Rozpoczęła wpisywanie danych w elemencie <input>, aby usunąć błąd.
    self.get_item_input_box().send_keys('a')

    # Była zadowolona, widząc, że komunikat błędu zniknął.
    error = self.browser.find_element_by_css_selector('.has-error')
    self.assertFalse(error.is_displayed()) #❷
```

¹ <http://onemillionpoints.blogspot.co.uk/2008/08/calvin-and-chipmunks.html>

- ❶❷ Funkcja `is_displayed()` wskazuje, czy element jest widoczny. Nie możemy się opierać na sprawdzeniu obecności elementu w modelu DOM, ponieważ teraz zaczynamy ukrywać elementy.

Test kończy się niepowodzeniem, ale zanim przejdziemy dalej: do trzech razy sztuka, a później refaktoryzacja! Mamy kilka miejsc, w których za pomocą CSS wyszukujemy element błędu. Przenieśmy to zadanie do funkcji pomocniczej.

Plik `functional_tests/test_list_item_validation.py` (ch14l002):

```
def get_error_element(self):
    return self.browser.find_element_by_css_selector('.has-error')
```



Lubię umieszczać funkcje pomocnicze w klasie testów funkcyjnych, która będzie ich używać. Natomiast przenoszę je do klasy bazowej tylko wtedy, gdy wiem, że faktycznie będą wykorzystywane też w innych miejscach. W ten sposób unikam nadmiernego rozrastania się klasy bazowej. Pamiętaj o zasadzie YAGNI.

Pozostało przeprowadzenie pięciu operacji zastępowania w pliku `test_list_item_validation.py`, jak te przedstawione poniżej.

Plik `functional_tests/test_list_item_validation.py` (ch14l003):

```
# Była zadowolona, widząc, że komunikat błędu zniknął.
error = self.get_error_element()
self.assertFalse(error.is_displayed())
```

Wynikiem jest oczekiwane niepowodzenie:

```
$ python3 manage.py test functional_tests.test_list_item_validation
[...]
    self.assertFalse(error.is_displayed())
AssertionError: True is not false
```

Teraz możemy przekazać pliki do repozytorium.

Konfiguracja prostego silnika wykonywania testów JavaScript

Wybór narzędzi testowych w świecie Pythona i Django jest całkiem prosty. Moduł `unittest` biblioteki standardowej sprawdza się tutaj doskonale, a silnik testów Django również stanoi dobry wybór domyślny. Istnieją jeszcze inne rozwiązania, do popularnych można zaliczyć `nose`², a osobiście byłem zachwycony `pytest`³. Jednak mamy jasno zdefiniowaną opcję domyślną, która sprawdza się świetnie⁴.

Inaczej jest w świecie JavaScript. W mojej firmie korzystamy z YUI, ale postanowiłem sprawdzić, jakie mam inne możliwości w tym zakresie. Szczerze mówiąc, zostałem przygnieciony liczbą dostępnych narzędzi — jsUnit, Qunit, Mocha, Chutzpah, Karma, Testacular, Jasmine i wiele innych. Na dodatek na tym nie koniec: kiedy wreszcie wybrałem jedno z nich —

² <http://nose.readthedocs.org/en/latest/>

³ <http://pytest.org/latest/>

⁴ Kiedy zaczynasz szukać narzędzi Python BDD, sytuacja nieco się skomplikuje.

Mocha⁵ — to okazało się, że konieczny jest wybór *frameworka asercji, narzędzia do zgłoszania, prawdopodobnie biblioteki imitacji...* czy to się nigdy nie skończy?

Ostatecznie zdecydowałem się na użycie *QUnit*⁶, ponieważ jest to proste rozwiązanie i doskonale współdziała z jQuery.

W katalogu *lists/static* utwórz podkatalog *tests*, umieść w nim pliki JavaScript i CSS pakietu QUnit, usuwając z nich numery wersji, jeśli zajdzie potrzeba (tutaj używam wersji 1.12). W tym podkatalogu umieść także plik o nazwie *tests.html*:

```
$ tree lists/static/tests/
lists/static/tests/
    qunit.css
    qunit.js
    tests.html
```

Poniżej przedstawiono szkielet pliku HTML dla QUnit zawierającego podstawowy test.

Plik *lists/static/tests/tests.html*:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Javascript tests</title>
    <link rel="stylesheet" href="qunit.css">
</head>

<body>
    <div id="qunit"></div>
    <div id="qunit-fixture"></div>
    <script src="qunit.js"></script>
    <script>
        /*global $, test, equal */

        test("smoke test", function () {
            equal(1, 1, "Działa!");
        });

        </script>
    </body>
</html>
```

Podczas analizy powyższego kodu należy zwrócić uwagę na kilka ważnych szczegółów. Skrypt *qunit.js* jest wczytywany za pomocą pierwszego znacznika *<script>*, natomiast drugi wykorzystujemy do utworzenia właściwych testów.

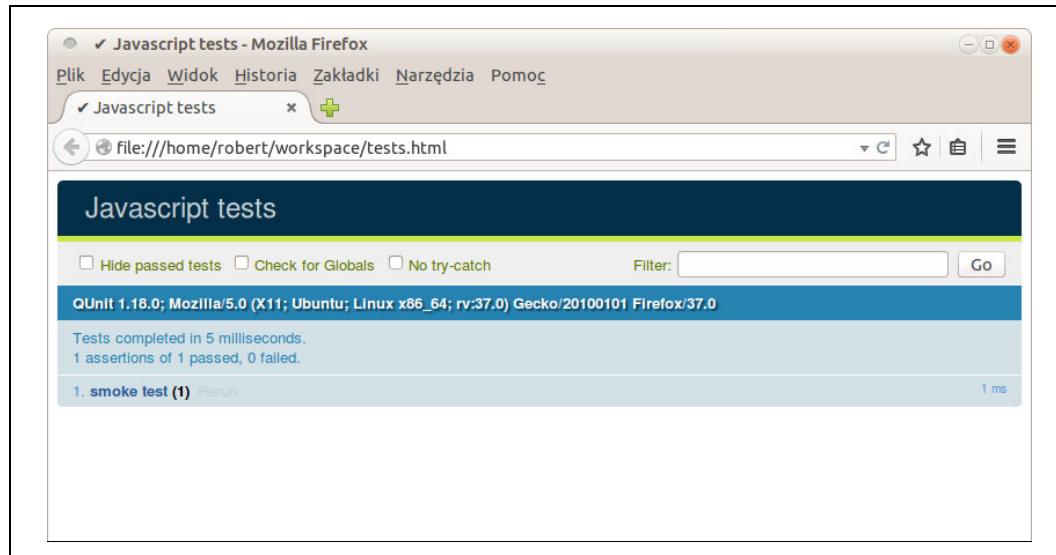


Zastanawiasz się nad komentarzem */*global?* Korzystam z narzędzia o nazwie *jshint*, które jest zintegrowane z moim edytorem i służy do sprawdzania składni JavaScript. Ten komentarz wskazuje, jakie są oczekiwane zmienne globalne. To nie jest ważne z punktu widzenia kodu, więc się tym nie przejmuj. Jednak zachęcam Cię do spojrzenia w wolnej chwili na narzędzia takie jak *jshint* i *jshint*. Mogą okazać się niezwykle użyteczne i pomóc w unikaniu „pułapek” JavaScript.

⁵ Praktycznie tylko dlatego, że zawiera silnik NyanCat.

⁶ <http://qunitjs.com/>

Jeżeli przygotowany plik HTML otworzysz w przeglądarce internetowej (nie ma potrzeby uruchamiania serwera programistycznego, wystarczy zwykłe otworzenie pliku z dysku), wówczas otrzymasz wynik podobny do pokazanego na rysunku 13.1.



Rysunek 13.1. Podstawowy ekran QUnit

Spójrz na sam test, a znajdziesz wiele podobieństw do testów Pythona, które dotąd tworzyliśmy:

```
test("smoke test", function () { #❶
    equal(1, 1, "Działa!"); #❷
});
```

- ❶ Funkcja `test()` definiuje testowany przypadek, działa podobnie do polecenia `def test_something(self)` w Pythonie. Pierwszym argumentem funkcji jest nazwa testu, natomiast drugim funkcja dla części głównej testu.
- ❷ Funkcja `equal()` jest asercją, bardzo podobną w działaniu do `assertEqual()`, i porównuje dwa argumenty. Jednak w przeciwieństwie do Pythona komunikat jest wyświetlany w przypadku zarówno powodzenia, jak i niepowodzenia asercji i powinien być traktowany pozytywnie, a nie negatywnie.

Spróbuj zmienić argumenty funkcji i zobacz, jak wygląda celowe niepowodzenie jej działania.

Użycie jQuery i stałych elementów <div>

Warto dowiedzieć się nieco więcej o możliwościach oferowanych przez framework testowy. Zaczniemy więc korzystać z biblioteki jQuery.



Jeżeli nigdy wcześniej nie spotkałeś się z jQuery, to abyś nie czuł się zupełnie zagubiony, przy pierwszym użyciu przedstawię naprawdę krótkie wprowadzenie do niej. To jednak nie jest samouczek jQuery. Na pewnym etapie lektury tego rozdziału warto poświęcić godzinę lub dwie na choć pobiczne poznanie jQuery.

Dodajemy do skryptów bibliotekę jQuery oraz kilka elementów przeznaczonych do użycia w testach.

Plik *lists/static/tests/tests.html*:

```
<div id="qunit-fixture"></div>

<form> #❶
  <input name="text" />
  <div class="has-error">Komunikat błędu</div>
</form>

<script src="http://code.jquery.com/jquery.min.js"></script>
<script src="qunit.js"></script>
<script>
/*global $, test, equal */
test("smoke test", function () {
  equal($('.has-error').is(':visible'), true); #❷❸
  $('.has-error').hide(); #❹
  equal($('.has-error').is(':visible'), false); #❺
});
</script>
```

- ❶ Element `<form>` i jego zawartość przedstawiają tutaj to, co znajdzie się na rzeczywistej stronie listy.
- ❷ W tym momencie rozpoczyna się magia jQuery. `$` to wszechstronne narzędzie jQuery. Służy do wyszukiwania elementów w modelu DOM. Pierwszym argumentem jest selektor CSS. W omawianym przykładzie nakazujemy wyszukanie wszystkich elementów mających klasę `error`. Wartością zwrotną jest obiekt przedstawiający jeden lub więcej elementów modelu DOM. Wspomniany obiekt zawiera różne użyteczne metody pozwalające na przeprowadzanie operacji na znalezionych elementach.
- ❸ Jedną z metod jest `.is()`, która wskazuje, czy element dopasował określona właściwość CSS. W omawianym przykładzie używamy `:visible` w celu sprawdzenia, czy element został wyświetlony, czy jest ukryty.
- ❹ Następnie używamy metody `.hide()` biblioteki jQuery w celu ukrycia elementu `<div>`. W tle oznacza to dynamiczne przypisanie elementowi stylu CSS `style="display: none"`.
- ❺ Za pomocą drugiej asercji `equal()` sprawdzamy, czy rozwiązanie działa zgodnie z oczekiwaniami.

Po odświeżeniu strony w przeglądarce internetowej powinieneś zobaczyć, że test został zaliczony.

Poniżej przedstawiono oczekiwane wyniki z QUnit.

```
2 assertions of 2 passed, 0 failed.
1. smoke test (0, 2, 2)
```

Nadeszła pora na sprawdzenie działania całości. Zmodyfikujemy nieco test.

Plik *lists/static/tests/tests.html*:

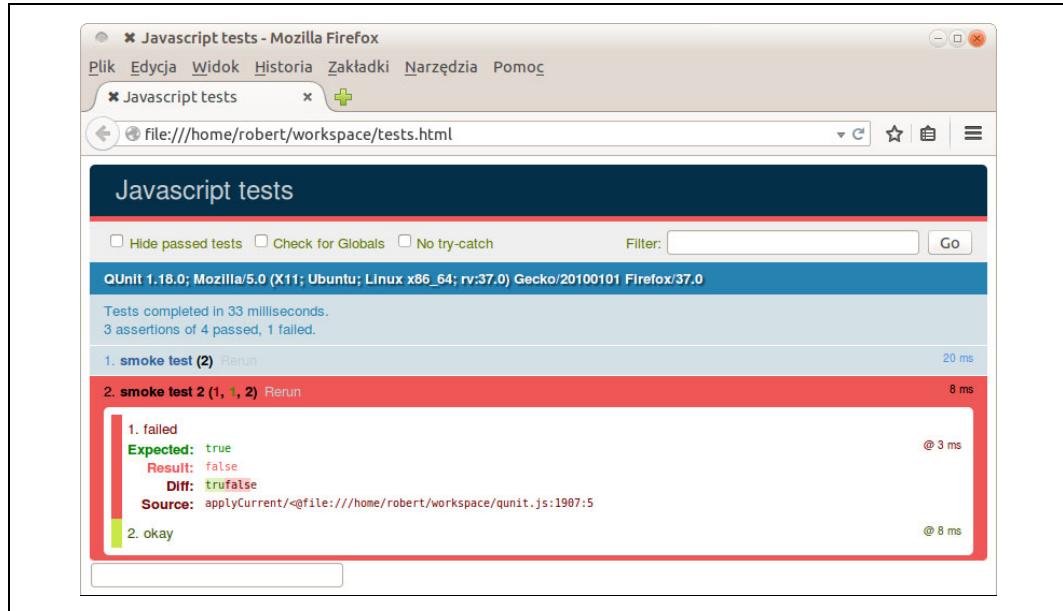
```
<script>
/*global $, test, equal */
test("smoke test", function () {
  equal($('.has-error').is(':visible'), true);
  $('.has-error').hide();
  equal($('.has-error').is(':visible'), false);
});
test("smoke test 2", function () {
```

```

equal($('.has-error').is(':visible'), true);
$('.has-error').hide();
equal($('.has-error').is(':visible'), false);
});
</script>

```

Wynik jest nieco nieoczekiwany — jak pokazano na rysunku 13.2, jeden z testów kończy się niepowodzeniem.



Rysunek 13.2. Jeden z dwóch testów kończy się niepowodzeniem

Mamy w tym miejscu do czynienia z następującą sytuacją. Pierwszy test powoduje ukrycie elementu <div>. Dlatego też po uruchomieniu drugiego wspomniany element jest niewidoczny.



Testy QUnit nie są wykonywane w możliwej do przewidzenia kolejności. Nie możesz przyjmować założenia, że test pierwszy będzie wykonany przed drugim.

Brakuje nam możliwości przeprowadzenia pewnych działań porządkowych między testami. Potrzeba czegoś takiego jak metody `setUp()` i `tearDown()` w Pythonie lub zerowanie przez silnik testów Django bazy danych między poszczególnymi testami. Element <div> o nazwie `qunit-fixture` to rozwiązanie, którego szukamy. Umieść w nim formularz.

Plik `lists/static/tests/tests.html`:

```

<div id="qunit"></div>
<div id="qunit-fixture">
<form>
    <input name="text" />
    <div class="has-error">Komunikat błędu</div>
</form>
</div>

<script src="http://code.jquery.com/jquery.min.js"></script>

```

Jak prawdopodobnie zgadłeś, przed każdym testem jQuery zeruje zawartość wymienionego elementu <div>. Wracamy więc do stanu dwóch elegancko zaliczonych testów:

```
4 assertions of 4 passed, 0 failed.  
1. smoke test (0, 2, 2)  
2. smoke test 2 (0, 2, 2)
```

Tworzenie testu jednostkowego JavaScript dla żądanej funkcjonalności

Po poznaniu narzędzi testowania w języku JavaScript możemy powrócić do tylko jednego testu i rozpocząć tworzenie faktycznie interesującego nas kodu.

Plik *lists/static/tests/tests.html*:

```
<script>  
/*global $, test, equal */  
  
test("errors should be hidden on keypress", function () {  
    $('input').trigger('keypress'); #❶  
    equal($('.has-error').is(':visible'), false);  
});  
  
</script>
```

- ❶ Oferowana przez jQuery metoda .trigger() jest używana przede wszystkim do testowania. Jej działanie można określić jako „wywołaj zdarzenie JavaScript DOM w elemencie”. W omawianym przykładzie używamy zdarzenia keypress, które jest w tle wywoływanie przez przeglądarkę internetową, gdy użytkownik rozpocznie wprowadzanie dowolnego tekstu we wskazanym elemencie <input>.



jQuery ukrywa w tle dość duży poziom skomplikowania. W witrynie *Quirksmode.org* możesz przeczytać o różnicach występujących w sposobach obsługi zdarzeń przez poszczególne przeglądarki internetowe. Jednym z powodów tak dużej popularności biblioteki jQuery jest to, że niweluje ona wspomniane różnice.

Otrzymujemy następujące dane wyjściowe:

```
0 assertions of 1 passed, 1 failed.  
1. errors should be hidden on keypress (1, 0, 1)  
   1. failed  
     Expected: false  
     Result: true
```

Przyjmujemy założenie, że kod chcemy umieścić w oddzielnym pliku JavaScript o nazwie *list.js*.

Plik *lists/static/tests/tests.html*:

```
<script src="qunit.js"></script>  
<script src="../list.js"></script>  
<script>
```

Poniżej przedstawiono minimalną ilość kodu niezbędną do zaliczenia testu.

Plik *lists/static/list.js*:

```
$('.has-error').hide();
```

Mamy tutaj do czynienia z oczywistym problemem. Lepiej będzie, jeśli dodamy kolejny test.

Plik *lists/static/tests/tests.html*:

```
test("po zdarzeniu keypress błędy powinny być ukryte", function () {
  $('input').trigger('keypress');
  equal($('.has-error').is(':visible'), false);
});

test("błąd powinny być widoczne aż do wystąpienia zdarzenia keypress", function () {
  equal($('.has-error').is(':visible'), true);
});
```

Teraz otrzymujemy oczekiwane niepowodzenie:

```
1 assertions of 2 passed, 1 failed.
1. po zdarzeniu keypress błędy powinny być ukryte (0, 1, 1)
2. błędy powinny być widoczne aż do wystąpienia zdarzenia keypress (1, 0, 1)
  1. failed
    Expected: true
    Result: false
    Diff: true false
[...]
```

Możemy więc przystąpić do utworzenia znacznie bardziej realistycznej implementacji.

Plik *lists/static/list.js*:

```
$(‘input’).on(‘keypress’, function () { #❶
  $('.has-error').hide();
});
```

- ❶ Działanie tego wiersza jest następujące: wyszukaj wszystkie elementy `<input>`, a następnie do każdego z nich dołącz obserwatora zdarzeń, który reaguje na zdarzenia keypress. Wspomniany obserwator zdarzeń to funkcja typu inline ukrywająca wszystkie elementy posiadające klasę `.has-error`.

Teraz testy jednostkowe zostają zaliczone:

```
2 assertions of 2 passed, 0 failed.
```

Świętne! Nasz skrypt i jQuery zastosujemy teraz na wszystkich stronach.

Plik *lists/templates/base.html* (ch14l014):

```
</div>
<script src="http://code.jquery.com/jquery.min.js"></script>
<script src="/static/list.js"></script>
</body>
</html>
```



Dobrą praktyką jest umieszczanie znaczników `<script>` wczytujących skrypty na końcu dokumentu HTML. Dzięki temu użytkownik nie musi czekać na wczytanie wszystkich skryptów, zanim cokolwiek zobaczy na stronie. Ponadto takie rozwiązanie pomaga w zapewnieniu, że większość elementów modelu DOM zostanie wczytana jeszcze przed uruchomieniem jakiegokolwiek skryptu.

Na koniec wykonujemy testy funkcjonalne:

```
$ python3 manage.py test functional_tests.test_list_item_validation.\
ItemValidationTest.test_error_messages_are_cleared_on_input
[...]
```

```
Ran 1 test in 3.023s
```

```
OK
```

Hura! To odpowiedni moment na przekazanie plików do repozytorium.

Testowanie JavaScript w cyklu TDD

Być może zastanawiasz się, jak te testy JavaScript wpisują się w nasz cykl TDD „podwójnej pętli”? Odpowiedź brzmi: pełnią dokładnie taką samą rolę jak testy jednostkowe Pythona.

1. Utwórz test funkcjonalny i zobacz, jak kończy się niepowodzeniem.
2. Ustal rodzaj kodu, jaki powinien zostać utworzony w następnej kolejności: Python lub JavaScript.
3. Utwórz test jednostkowy w odpowiednim języku i zobacz, jak kończy się niepowodzeniem.
4. Utwórz kod aplikacji w tym samym języku i zapewnij zaliczenie testu.
5. Powtórz ten cykl.



Potrzebujesz nieco większego doświadczenia w zakresie języka JavaScript? Zobacz, czy potrafisz zmodyfikować kod w taki sposób, aby komunikat błędu był ukrywany również po kliknięciu przez użytkownika wewnątrz elementu `<input>`, a nie tylko po rozpoczęciu wpisywania w nim dowolnego tekstu. Nie zapomnij o przeprowadzeniu testów funkcjonalnych.

Zdarzenie onload i przestrzeń nazw

Na koniec pozostała nam jeszcze jedna kwestia do omówienia. Kiedy masz skrypt JavaScript przeprowadzający operacje na elementach modelu DOM, zawsze dobrym rozwiązaniem będzie opakowanie go pewnym kodem zdarzenia `onload`. W ten sposób można zagwarantować pełne wczytanie strony przed przystąpieniem do wykonywania na niej jakichkolwiek operacji. Aktualnie przyjęte rozwiązanie sprawdza się, ponieważ znacznik `<script>` został umieszczony prawie na samym końcu strony. Nie należy jednak bezwarunkowo opierać się na tego rodzaju rozwiązaniu.

Jak pokazano poniżej, kod dla zdarzenia `onload` w jQuery jest całkiem minimalny.

Plik `lists/static/list.js`:

```
$(document).ready(function () {
    $('input').on('keypress', function () {
        $('.has-error').hide();
    });
});
```

Używamy magicznej funkcji `$` z jQuery, ale czasami inne biblioteki JavaScript również próbują użyć tej samej funkcji. To jest po prostu alias dla mniej kolidującej z innymi nazwy jQuery. Poniżej przedstawiono standardowy sposób uzyskania znacznie większej kontroli nad przestrzenią nazw.

Plik `lists/static/list.js`:

```
jQuery(document).ready(function ($) {
    $('input').on('keypress', function () {
        $('.has-error').hide();
    });
});
```

Więcej informacji na ten temat znajdziesz w dokumentacji jQuery poświęconej metodzie `.ready()`⁷.

W tym momencie jesteśmy prawie gotowi, aby przejść do części III książki. Ostatnim krokiem jest wdrożenie nowego kodu w serwerach.

Kilka rozwiązań, które się nie sprawdzają

- Selektor `$(input)` jest zbyt zachłanny i powoduje przypisanie procedury obsługi do każdego elementu `<input>` na stronie. Jako ćwiczenie spróbuj dodać procedurę obsługi kliknięć, a zobacysz, dlaczego wymieniony selektor stanowi problem. Postaraj się to naprawić.
- Na tym etapie nasz test jedynie sprawdza, czy JavaScript działa na jednej stronie. Rozwiązanie działa, ponieważ skrypt został dołączony do szablonu `base.html`. Po dołączeniu skryptu do szablonu `home.html` testy nadal będą zaliczone. Możesz czuć się usprawiedliwiony i zdecydować na utworzenie dodatkowego testu.

Uwagi dotyczące testowania w JavaScript

- Jedną z doskonałych zalet Selenium jest to, że pozwala na sprawdzenie, czy kod JavaScript faktycznie działa, podobnie jak ma to miejsce z kodem Pythona.
- Istnieje wiele bibliotek pozwalających na przeprowadzanie testów w JavaScript. Biblioteka QUnit jest ściśle związana z jQuery i to najważniejszy powód, dla którego ją wybrałem.
- QUnit oczekuje „wykonania” testów za pomocą rzeczywistej przeglądarki internetowej. Zaletą takiego podejścia jest możliwość łatwego tworzenia elementów HTML dopasowanych do faktycznie znajdujących się na stronie elementów HTML i ich przetestowania.
- Naprawdę nie wiem, co miałem na myśli, wypowiadając niepochlebne opinie o języku JavaScript. Tworzenie w nim kodu naprawdę może sprawiać radość. Jednak jak wcześniej wspomniałem, warto przeczytać książkę *JavaScript — mocne strony*.

⁷ <http://api.jquery.com/ready/>

Wdrożenie nowego kodu

Najwyższy czas, aby nasz olśniewający kod odpowiedzialny za weryfikację wdrożyć w serwerach. To będzie szansa na zobaczenie po raz drugi zautomatyzowanych skryptów wdrożenia.



W tym miejscu składam ogromne podziękowania Andrew Godwinowi i caemu zespołowi Django. Przed powstaniem wersji Django 1.7 musiałem przygotować cały podrozdział poświęcony wyłącznie migracjom. Na szczęście teraz migracje „po prostu działają”, więc mogłem pozbyć się wspomnianego podrozdziału. Dziękuję wszystkim zaangażowanym w Django za doskonałą pracę nad frameworkm.

Wdrożenie prowizoryczne

Zaczynamy od uruchomienia serwera prowizorycznego:

```
$ cd deploy_tools  
$ fab deploy:host=elsbeth@superlists-staging.ottg.eu  
Disconnecting from superlists-staging.ottg.eu... done.
```

Ponownie uruchamiamy Gunicorn:

```
edyta@serwer:$ sudo restart gunicorn-superlists-staging.ottg.eu
```

Teraz możemy wykonać testy względem wersji prowizorycznej:

```
$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu  
OK
```

Wdrożenie rzeczywiste

Przyjmując założenie, że wszystko przebiegło doskonale, możemy przeprowadzić wdrożenie w serwerze produkcyjnym:

```
$ fab deploy:host=edyta@superlists.ottg.eu
```

A jeśli wystąpi błąd bazy danych?

Ponieważ nasze migracje wprowadzają nowe ograniczenie dotyczące spójności danych, przeprowadzenie ich może się okazać niemożliwe, ponieważ istniejące dane są niezgodne z nowym ograniczeniem.

W takim przypadku masz dwie możliwości:

- Usunięcie bazy danych w serwerze i ponowne spróbowanie przeprowadzenia migracji. W końcu to tylko projekt służący rozrywce.
- Lepsze poznanie migracji danych. Zapoznaj się z dodatkiem D.

Podsumowanie — git tag i nowe wydanie

Ostatnim zadaniem jest oznaczenie w systemie VCS tagiem bieżącego wydania. To bardzo ważne, ponieważ pozwoli na łatwe sprawdzenie, co znajduje się w serwerze produkcyjnym:

```
$ git tag -f LIVE  # Konieczne jest użycie opcji -f, ponieważ zastępujemy poprzedni tag.  
$ export TAG=~date +DEPLOYED-%F%H%M~  
$ git tag $TAG  
$ git push -f origin LIVE $TAG
```



Niektórzy nie lubią używać polecenia push -f i uaktualniać istniejącego znacznika. Zamiast tego preferują zastosowanie pewnego rodzaju numeracji wersji i tak oznaczają wydania. Użyj takiego rozwiązania, które uznasz za wygodne dla siebie.

Na tym kończymy część II książki i możemy przejść dalej, do bardziej ekskrytujących tematów w części III. Nie mogę się już doczekać!

Bardziej zaawansowane zagadnienia

O rety, co takiego? Kolejna część książki? Harry, czuję się wyczerpany po lekturze ponad 200 stron książki i nie sądzę, że jestem gotowy na zupełnie nową część książki. Zwłaszcza że jej tytuł zawiera słowo „zaawansowane”... może mógłbym ją pominąć?

Nie, nie możesz! Ta część jest wypełniona naprawdę ważnymi tematami dotyczącymi technik TDD oraz programowania sieciowego. Nie ma nawet mowy, abyś mógł pominąć tę część. Przeciwnie, ta część jest wręcz *znacznie ważniejsza* niż dwie pierwsze.

Poruszony zostanie temat integracji systemów firm trzecich oraz ich testowania. Nowoczesne programowanie sieciowe jest oparte na ponownym użyciu istniejących komponentów. Będziemy omawiać tworzenie imitacji i izolację testu, co naprawdę stanowi ważną część TDD. Te techniki są potrzebne praktycznie we wszystkich projektach, poza tymi najprostszymi. Poruszone zostaną także kwestie usuwania błędów po stronie serwera, konfiguracji testów (ang. *test fixtures*), a także środowiska nieustannej integracji. W projektach żadne z wymienionych funkcji nie zaliczają się do opcjonalnych luksusów w postaci „użyj lub zapomnij”, a wręcz mają bardzo istotne znaczenie!

Poziom trudności materiału przedstawionego w tej części książki jest nieco większy. Być może niektóre fragmenty będziesz musiał przeczytać kilkukrotnie, zanim staną się jasne. Ponadto pewne rozwiązania mogą nie działać od razu i będziesz musiał samodzielnie przeprowadzić proces usuwania błędów. Bądź jednak wytrwały! Im bardziej zadanie, tym większa nagroda za jego wykonanie. Jak zwykle chętnie Ci pomogę, jeśli zapędzisz się w koziego róg — wystarczy napisać do mnie wiadomość e-mail na adres *obeythetestinggoat@gmail.com*.

Zaczynamy! Obiecuję, że najlepsze jest jeszcze przed Tobą!



Użycie JavaScript do uwierzytelniania użytkownika, integracji wtyczek i przygotowania imitacji

Nasze piękne listy rzeczy do zrobienia są dostępne od paru dni, a użytkownicy zaczynają zgłaszać propozycje i uwagi dotyczące aplikacji. „Uwielbiamy tę witrynę — tak twierdzą — ale nieustannie giną nasze listy. Ponadto konieczność zapamiętania adresu URL jest niewygodna. Byłyby świetnie, gdyby można było zapamiętać rozpoczęte przez nas listy”.

Pamiętasz Henry'ego Forda i szybsze konie? Kiedy słyszysz o potrzebie zgłoszonej przez użytkownika, ważne jest przeanalizowanie jego wypowiedzi i podjęcie próby ustalenia, jaką tak naprawdę zgłasza potrzebę. Zadaj sobie wówczas pytanie: czy propozycję użytkownika mogę przekuć na nową, fascynującą technologię, którą sam chciałbym wypróbować?

Bez wątpienia użytkownicy witryny oczekują implementacji pewnego rodzaju kont użytkowników. Dlatego też bez zbędnych ceregieli zagłębiamy się w temat uwierzytelniania.

Oczywiście nie zamierzamy sobie zaprzątać głowy koniecznością zapamiętywania haseł — to brzmi jak powrót do lat dziewięćdziesiątych ubiegłego wieku, a ponadto bezpieczne przechowywanie haseł użytkowników to koszmar, który lepiej pozostawić innym. Zamiast tego wykorzystamy ogólny system uwierzytelniania.

(Jeżeli *upierasz* się przy przechowywaniu własnych haseł, oferowany przez Django domyślny moduł auth jest gotowy i czeka na Ciebie. Użycie go należy do prostych zadań i dlatego pozwostawiam Ci przyjemność odkrywania możliwości i sposobu użycia wymienionego modułu).

W tym rozdziale dość mocno zagłębimy się w technikę testowania określana mianem „imitacji”. Potrzebowałem kilku tygodni, aby tak naprawdę zrozumieć stosowanie imitacji i przywyknąć do nich, więc nie przejmuj się, jeśli nie wszystko będzie od razu jasne. W rozdziale dość intensywnie będziemy wykorzystywać imitacje w języku JavaScript. W kolejnym rozdziale przejdziemy do imitacji w Pythonie, co może się okazać nieco łatwiejsze do opanowania. Zalecam jednoczesną lekturę obu rozdziałów, co pozwoli Ci na przesiąknięcie koncepcją imitacji. Później będziesz mógł do nich wracać i zobaczyć, czy zrozumiałeś wszystkie kroki nieco lepiej niż za pierwszym razem.



Napisz do mnie na adres obeythetestinggoat@gmail.com, jeśli uważasz, że którykolwiek fragmenty nie zostały wystarczająco wyjaśnione.

Mozilla Persona (BrowserID)

Którego ogólnego systemu uwierzytelniania użyjemy? Oauth? Openid? „Zaloguj za pomocą konta Facebook”? Och. W mojej książce wszystkie wymienione przyprawiają o gęsią skórkę: dlaczego Google lub Facebook miałyby „wiedzieć”, do jakich witryn i kiedy się logujesz? Na szczęście w świecie technologii nadal istnieją idealisci, a kochani inżynierowie Mozilli opracowali przyjazny pod względem zachowania prywatności mechanizm autoryzacji nazywany *Persona* lub czasami *BrowserID*.

W teorii przeglądarka internetowa pełni rolę firmy trzeciej znajdującej się między witryną sprawdzającą Twoją tożsamość a witryną potwierdzającą Twoją tożsamość. Drugą z wymienionych może być Google lub Facebook, ale dzięki sprytnie opracowanemu protokołowi wymienione firmy nigdy nie wiedzą, kiedy i do których witryn internetowych się logujesz.

Ostatecznie usługa Persona może nigdy nie stać się platformą uwierzytelniania, ale materiał przedstawiony w kilku kolejnych rozdziałach powinien pozostać aktualny niezależnie od tego, jaki system uwierzytelniania chcesz zintegrować.

- Nie testuj kodu lub API opracowanego przez innych.
- Jednak powinieneś przetestować, czy prawidłowo zintegrowałś je z własnym kodem.
- Sprawdź, czy wszystko działa prawidłowo z punktu widzenia użytkownika.
- Upewnij się o eleganckiej degradacji funkcjonalności systemu, gdy nastąpi awaria usług oferowanych przez firmę trzecią.

Kod eksperymentalny, czyli „Spiking”

Zanim napisałem ten rozdział, cała moja wiedza o projekcie Persona sprowadzała się do odbytej na konferencji PyCon rozmowy z Danem Callahanem, w trakcie której obiecał, że cała implementacja może się zmieścić w 30 wierszach kodu, i przedstawił demo użycia usługi Persona. Innymi słowy, w ogóle nie znałem usługi Persona.

W rozdziałach 10. i 11. zobaczyłeś, że testu jednostkowego można użyć w celu poznawania nowego API, choć czasami zachodzi potrzeba sprawdzenia czegoś bez testów, aby się po prostu przekonać, czy w ogóle działa. Na tej podstawie można zdecydować o użyciu danego rozwiązania lub jego porzuceniu. To jest jak najbardziej właściwe podejście. Kiedy poznajesz nowe narzędzie lub wypróbowujesz ewentualne rozwiązanie, często odpowiednie będzie odłożenie na bok rygorystycznego procesu TDD i przygotowanie małego prototypu bez testów lub być może z ich niewielką liczbą. Testing Goat nie będzie mieć nic przeciwko i przez chwilę „popatrzy” w innym kierunku.

Tego rodzaju aktywność w zakresie tworzenia prototypu jest często określana mianem „spike” z kilku dobrze znanych powodów¹.

Pierwszym krokiem, jaki podjąłem, było przeanalizowanie istniejącego rozwiązania o nazwie *Django-BrowserID*² pozwalającego na integrację Django z usługą Persona. Niestety nie zapewnia ono pełnej obsługi Pythona 3. Jestem pewien, że gdy czytasz tę książkę, ten problem jest już rozwiązany. Przynajmniej, że sprawia mi to ulgę, ponieważ chciałem samodzielnie opracować kod odpowiedzialny za integrację Django i usługi Persona.

Trzy godziny zajęły mi eksperymenty z wykorzystaniem połączenia kodu przedstawionego przez Dana i przykładowych fragmentów dostępnych w witrynie Persona³, a wynikiem był kod, który po prostu działa. Pokażę Ci opracowane przeze mnie rozwiązanie, a następnie zaprezentuję zmianę rozwiązania eksperimentalnego na zwykłe.

Omawiany w tym rozdziale kod powinieneś dodać do własnej witryny, poeksperimentować z nim, spróbować zalogować się za pomocą własnego adresu e-mail i przekonać się, że naprawdę działa.

Utworzenie nowej gałęzi dla Spike

Zanim przystąpimy do pracy nad kodem eksperimentalnym, dobrym pomysłem jest utworzenie nowej gałęzi, aby nadal można było korzystać z systemu VCS bez obaw, że kod eksperimentalny będzie mieszał się z kodem produkcyjnym:

```
$ git checkout -b persona-spikes
```

Łączenie kodu JavaScript i interfejsu użytkownika

Rozpoczynamy od interfejsu użytkownika. Użyłem techniki kopij i wklej dla kodu znalezionejego w witrynie usługi Persona oraz wprowadziłem minimalne modyfikacje przedstawione na slajdach przez Dana.

Plik *lists/templates/base.html* (ch15l001):

```
<script src="http://code.jquery.com/jquery.min.js"></script>
<script src="/static/list.js"></script>
<script src="https://login.persona.org/include.js"></script>
<script>
$(document).ready(function() {

    var loginLink = document.getElementById('login');
    if (loginLink) {
        loginLink.onclick = function() { navigator.id.request(); };
    }

    var logoutLink = document.getElementById('logout');
    if (logoutLink) {
        logoutLink.onclick = function() { navigator.id.logout(); };
    }

    var currentUser = '{{ user.email }}' || null;
}
```

¹ <http://stackoverflow.com/questions/249969/why-are-tdd-spikes-called-spikes>

² <https://github.com/mozilla/django-browserid>

³ <https://developer.mozilla.org/en-US/Persona>

```

var csrf_token = '{{ csrf_token }}';
console.log(currentUser);

navigator.id.watch({
  loggedInUser: currentUser,
  onlogin: function(assertion) {
    $.post('/accounts/login', {assertion: assertion, csrfmiddlewaretoken: csrf_token})
    .done(function() { window.location.reload(); })
    .fail(function() { navigator.id.logout(); });
  },
  onlogout: function() {
    $.post('/accounts/logout')
    .always(function() { window.location.reload(); });
  }
});
});
</script>

```

Biblioteka JavaScript dla usługi Persona udostępnia obiekt specjalny `navigator.id`. Jego metoda o nazwie `request()` zostaje powiązana z naszym łączem „login” (które będzie umieszczone gdzieś na początku strony). Podobnie łącze „logout” zostanie powiązane z funkcją `logout()`.

Plik `lists/templates/base.html` (ch15l002):

```

<body>
<div class="container">

  <div class="navbar">
    {% if user.email %}
      <p>Zalogowany jako {{ user.email }}</p>
      <p><a id="logout" href="{% url 'logout' %}">Wyloguj</a></p>
    {% else %}
      <a href="#" id="login">Zaloguj</a>
    {% endif %}
    <p>Użytkownik: {{user}}</p>
  </div>

  <div class="row">
  [...]

```

Protokół Browser-ID

Jeżeli użytkownik kliknie link logowania, usługa Persona wyświetli okno dialogowe pozwalające na uwierzytelnienie się. To, co się później stanie, jest pochodną sprytnie opracowanego protokołu usługi Persona: użytkownik podaje adres e-mail, a przeglądarka internetowa zajmuje się weryfikacją tego adresu e-mail u dostawcy poczty (Google, Yahoo lub inny), i tam przeprowadza odpowiednią weryfikację.

Warto w tym miejscu wyraźnie zaznaczyć, że weryfikacja użytkownika jest przeprowadzana przez Google. To w witrynie Google użytkownik podaje nazwę użytkownika, hasło i być może nawet odbywa się dwuetapowe uwierzytelnienie. Po tym usługa Google jest przygotowana do potwierdzenia tożsamości użytkownika. Przeglądarkę internetową jest przekazywany zaszyfrowany certyfikat potwierdzony przez Google oraz zawierający adres e-mail użytkownika.

Na tym etapie przeglądarka internetowa „wierzy”, że podany przez użytkownika adres e-mail należy do niego. Wspomniany wcześniej certyfikat może więc być ponownie stosowany w innych witrynach internetowych używających usługi Persona.

Teraz certyfikat jest łączony z nazwą domeny witryny, do której użytkownik chce się zalogować. Powstają dane nazywane „asercją” i są one przekazywane witrynie w celu weryfikacji.

To jest punkt między wywołaniem `onlogin` przez `navigator.id.request` i `navigator.id.watch` — asercja jest za pomocą żądania POST przekazywana do adresu URL w naszej witrynie; jest nim `accounts/login`.

W serwerze trzeba przeprowadzić operację weryfikacji asercji: czy to naprawdę jest dowód, że adres e-mail należy do użytkownika? Nasz serwer może to sprawdzić, ponieważ Google podpisało fragment asercji swoim kluczem publicznym. Można więc samodzielnie utworzyć kod przeznaczony do weryfikacji wspomnianej asercji lub też użyć do tego oferowanej przez Mozillę publicznej usługi.



Masz rację, myśląc, że zlecenie Mozilli zadania weryfikacji asercji całkowicie niweluje prywatność, ale taka jest *zasada*. Jeżeli chcesz, weryfikacją możesz się zająć samodzielnie. Implementację takiego rozwiązania pozostawiam Ci jako ćwiczenie. Więcej informacji na ten temat znajdziesz w *witrynie Mozilla*⁴. Znajdziesz tam sprytny klucz publiczny, dzięki któremu Google nie wie, do jakiej witryny internetowej chcesz się zalogować. Wspomniany klucz uniemożliwia także przeprowadzanie ataków typu replay itd. Sprytne.

Kod po stronie serwera — niestandardowe uwierzytelnienie

Kolejnym krokiem jest przygotowanie aplikacji do obsługi kont użytkowników:

```
$ python3 manage.py startapp accounts
```

Poniżej przedstawiono kod widoku odpowiedzialnego za obsługę żądań POST do `accounts/login`.

Plik `accounts/views.py`:

```
import sys
from django.contrib.auth import authenticate
from django.contrib.auth import login as auth_login
from django.shortcuts import redirect

def login(request):
    print('login view', file=sys.stderr)
    # user = PersonaAuthenticationBackend().authenticate(request.POST['assertion'])
    user = authenticate(assertion=request.POST['assertion'])
    if user is not None:
        auth_login(request, user)
    return redirect('/')
```

Wyraźnie widać, że to kod eksperymentalny i że zawiera umieszczone w komentarzach polecenia będące dowodem na wczesne eksperymenty zakończone niepowodzeniem.

Poniżej przedstawiono funkcję `authenticate()` zaimplementowaną jako niestandardowe rozwiązanie oparte na uwierzytelnianiu Django. (Ten kod można umieścić bezpośrednio w widoku, ale pokazane podejście jest zalecane przez Django. Dzięki temu system uwierzytelniania będzie mógł być ponownie wykorzystany, na przykład w witrynie administracyjnej).

⁴ https://developer.mozilla.org/en-US/Persona/Protocol_Overview

Plik *accounts/authentication.py*:

```
import requests
import sys
from accounts.models import ListUser

class PersonaAuthenticationBackend(object):

    def authenticate(self, assertion):
        # Wysłanie asercji do usługi weryfikacyjnej oferowanej przez Mozillę.
        data = {'assertion': assertion, 'audience': 'localhost'}
        print('sending to mozilla', data, file=sys.stderr)
        resp = requests.post('https://verifier.login.persona.org/verify', data=data)
        print('got', resp.content, file=sys.stderr)

        # Czy weryfikujący udzielił odpowiedzi?
        if resp.ok:
            # Przetworzenie odpowiedzi.
            verification_data = resp.json()

            # Sprawdzenie, czy asercja była prawidłowa.
            if verification_data['status'] == 'okay':
                email = verification_data['email']
                try:
                    return self.get_user(email)
                except ListUser.DoesNotExist:
                    return ListUser.objects.create(email=email)

    def get_user(self, email):
        return ListUser.objects.get(email=email)
```

Na podstawie komentarzy znajdujących się w kodzie nietrudno zgadnąć, że został on skopiowany bezpośrednio z witryny Mozilla.

W środowisku wirtualnym konieczne jest wydanie polecenia `pip install requests`. Jeżeli nigdy wcześniej nie używałeś biblioteki *Request*⁵, to powinieneś wiedzieć, że jest ona doskonałą alternatywą dla standardowej biblioteki narzędzi Pythona przeznaczonych do wykonywania żądań HTTP.

Aby zakończyć pracę nad dostosowaniem do własnych potrzeb uwierzytelniania w Django, należy jeszcze przygotować własny model użytkownika.

Plik *accounts/models.py*:

```
from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin
from django.db import models

class ListUser(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(primary_key=True)
    USERNAME_FIELD = 'email'
    #REQUIRED_FIELDS = ['email', 'height']

    objects = ListUserManager()

    @property
    def is_staff(self):
        return self.email == 'harry.percival@example.com'

    @property
    def is_active(self):
        return True
```

⁵ <http://docs.python-requests.org/en/latest/>

To właśnie nazywam minimalnym modelem użytkownika. Tylko jedna właściwość, żadnego nonsensu w stylu imię, nazwisko, nazwa użytkownika i przede wszystkim brak hasła! Obsługa haseł to problem pozostawiony innym! Warto przypomnieć ponownie, że przedstawiony powyżej kod nie jest gotowy do użycia w środowisku produkcyjnym — zawiera polecenia w komentarzach oraz na stałe zdefiniowany adres e-mail.



Na tym etapie zachęcam do przejrzenia dokumentacji Django poświęconej *uwierzytelnianiu*⁶.

Potrzebny jest nam model menedżera dla użytkownika.

Plik *accounts/models.py* (ch15l006):

```
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager, PermissionsMixin

class ListUserManager(BaseUserManager):

    def create_user(self, email):
        ListUser.objects.create(email=email)

    def create_superuser(self, email, password):
        self.create_user(email)
```

Poniżej przedstawiono widok logout.

Plik *accounts/views.py* (ch15l007):

```
from django.contrib.auth import login as auth_login, logout as auth_logout
[...]

def logout(request):
    auth_logout(request)
    return redirect('/')
```

Oto kilka adresów URL dla dwóch naszych widoków.

Plik *superlists/urls.py* (ch15l008):

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/', include('lists.urls')),
    url(r'^accounts/', include('accounts.urls')),
    # url(r'^admin/', include(admin.site.urls)),
)
```

I jeszcze kilka dodatkowych adresów URL.

Plik *accounts/urls.py*:

```
from django.conf.urls import patterns, url

urlpatterns = patterns('',
    url(r'^login$', 'accounts.views.login', name='login'),
    url(r'^logout$', 'accounts.views.logout', name='logout'),
)
```

Już prawie skończyliśmy. W pliku *settings.py* zmieniamy jeszcze system uwierzytelniania dla naszej nowej aplikacji accounts.

⁶ <https://docs.djangoproject.com/en/1.7/topics/auth/customizing/>

Plik `superlists/settings.py`:

```
INSTALLED_APPS = (
    #'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
    'accounts',
)

AUTH_USER_MODEL = 'accounts.ListUser'
AUTHENTICATION_BACKENDS = (
    'accounts.authentication.PersonaAuthenticationBackend',
)

MIDDLEWARE_CLASSES = (
    [...]
```

Wydajemy polecenie `makemigrations`, aby zastosować nowy model użytkownika:

```
$ python3 manage.py makemigrations
Migrations for 'accounts':
  0001_initial.py:
    - Create model ListUser
```

Następnie wydajemy polecenie `migrate` i tworzymy bazę danych:

```
$ python3 manage.py migrate
[...]
Running migrations:
  Applying accounts.0001_initial... OK
```

I to już wszystko. Teraz możesz uruchomić serwer programistyczny za pomocą `runserver` i zobaczyć, jak prezentuje się przygotowane rozwiązanie (patrz rysunek 15.1).

Oto gotowe rozwiązanie! Podczas pracy nad nim musiałem dość mocno się wysilić, między innymi przeprowadziłem debugowanie żądań Ajax w konsoli przeglądarki internetowej Firefox (zobacz rysunek 15.2), zmagałem się z wykonywaną w nieskończoność pętlą odświeżania strony oraz brakiem wielu atrybutów w przygotowanym niestandardowym modelu użytkownika (ponieważ niedokładnie zapoznałem się z dokumentacją), a nawet zastosowałem wersję beta Django, aby przezwyciężyć błąd, który na szczęście okazał się nieistotny.

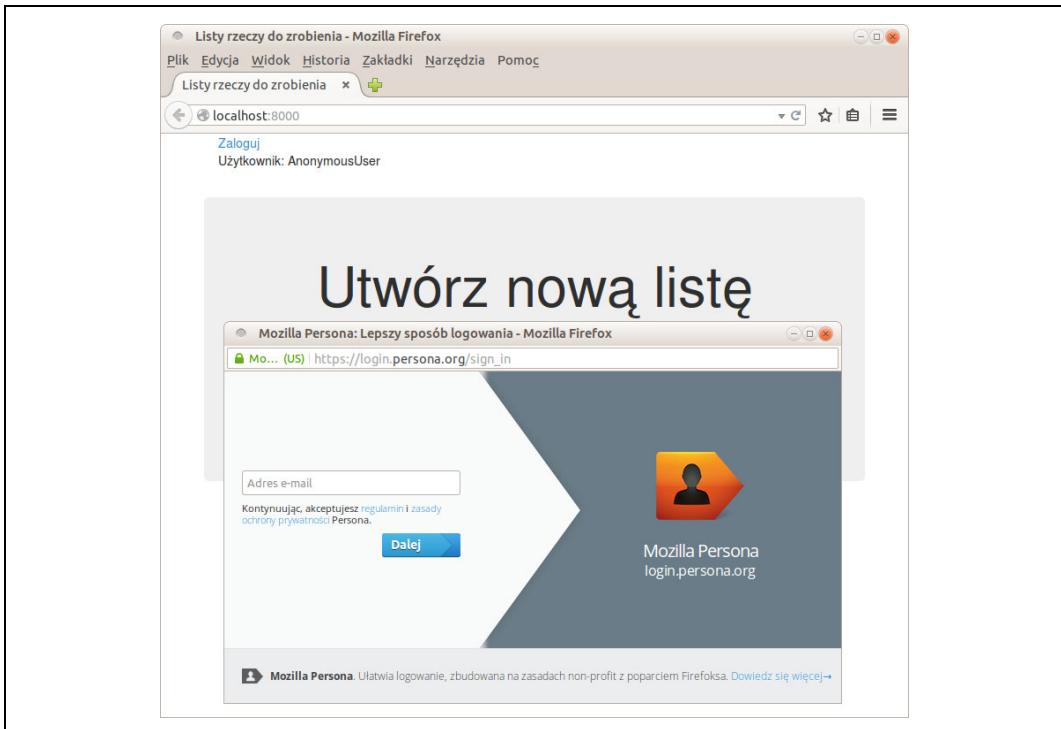


Jeżeli rozwiązanie nie działa po jego ręcznym wypróbowaniu i otrzymujesz w konsole błędy typu „audience mismatch”, wówczas upewnij się, że witrynę odwiedzasz przez adres `http://localhost:8000`, a nie `127.0.0.1`.

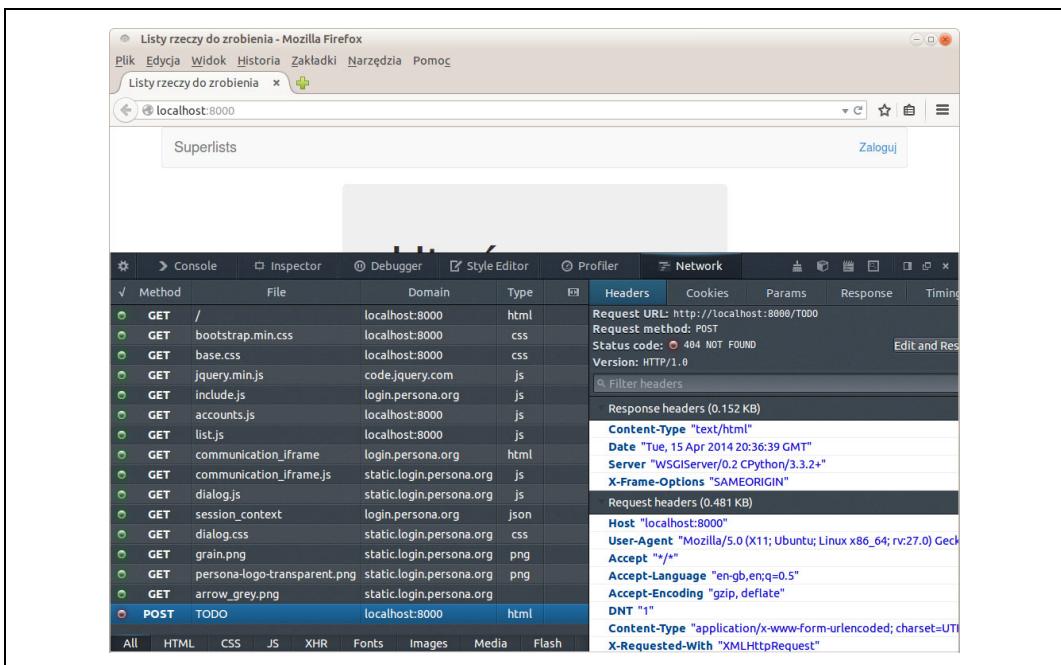
W tym momencie mamy działające rozwiązanie! Pliki przekazujemy więc do gałęzi eksperymentalnej w repozytorium:

```
$ git status
$ git add accounts
$ git commit -am "Własny system uwierzytelniania oparty na usłudze Persona."
```

Teraz przechodzimy do zamiany rozwiązania eksperymentalnego na zwykłe.



Rysunek 15.1. Działa, działa!



Rysunek 15.2. Debugowanie żądań Ajax w konsoli przeglądarki internetowej Firefox

Wyświetlanie błędów przez standardowe wyjście błędów (stderr)

Podczas przeprowadzania eksperymentów bardzo ważna jest możliwość przeglądania wyjątków zgłoszanych przez kod. Niestety domyślnie Django nie przekazuje do terminala wszystkich wyjątków, ale można to zmienić za pomocą zmiennej `LOGGING` w pliku `settings.py`.

Plik `superlists/settings.py` (ch15l011):

```
LOGGING = {  
    'version': 1,  
    'disable_existing_loggers': False,  
    'handlers': {  
        'console': {  
            'level': 'DEBUG',  
            'class': 'logging.StreamHandler',  
        },  
    },  
    'loggers': {  
        'django': {  
            'handlers': ['console'],  
        },  
    },  
    'root': {'level': 'INFO'},  
}
```

Do rejestracji danych Django używa modułu `logging` pochodzącego z biblioteki standardowej Pythona. Oferuje on wiele funkcji, ale jego poznanie wymaga całkiem sporego wysiłku. Nieco więcej informacji o wspomnianym module znajdziesz w rozdziale 17. oraz w dokumentacji⁷ Django.

Zamiana rozwiązania eksperymentalnego na zwykłe

Zamiana rozwiązania eksperymentalnego na zwykłe polega na przepisaniu kodu prototypu przy użyciu technik TDD. Na tym etapie powinniśmy mieć wystarczającą ilość informacji, aby „zastosować poprawną implementację”. Jaki jest więc krok pierwszy? To oczywiście testy funkcjonalne!

Pozostajemy w gałęzi eksperymentalnej i sprawdzamy, czy testy funkcjonalne zostaną zaliczone przez kod eksperymentalny. Następnie powrócimy do gałęzi master i testy funkcjonalne przekażemy do repozytorium.

Poniżej przedstawiono ogólny zarys testów funkcjonalnych.

Plik `functional_tests/test_login.py`:

```
from .base import FunctionalTest  
  
class LoginTest(FunctionalTest):  
  
    def test_login_with_persona(self):  
        # Edyta odwiedza wspaniałą witrynę superlists  
        # i po raz pierwszy zauważa łącze "Zaloguj".  
        self.browser.get(self.server_url)  
        self.browser.find_element_by_id('login').click()  
  
        # Na ekranie pojawia się okno dialogowe logowania za pomocą usługi Persona.
```

⁷ <https://docs.djangoproject.com/en/1.7/topics/logging/>

```

self.switch_to_new_window('Mozilla Persona') #❶
# Edyta loguje się, podając adres e-mail.
## Użycie mockmyid.com do przetestowania adresu e-mail.
self.browser.find_element_by_id(
    'authentication_email' #❷
).send_keys('edyta@mockmyid.com') #❸
self.browser.find_element_by_tag_name('button').click()

# Okno dialogowe usługi Persona zostaje zamknięte.
self.switch_to_new_window('To-Do')

# Edyta może zobaczyć, że została zalogowana.
self.wait_for_element_with_id('logout') #❹
navbar = self.browser.find_element_by_css_selector('.navbar')
self.assertIn('edyta@mockmyid.com', navbar.text)

```

- ❶❷ Test funkcjonalny wymaga kilku metod pomocniczych, które są dość często stosowane wraz z Selenium: oczekują na pewne zdarzenie. Kod wspomnianych metod znajdziesz nieco dalej w rozdziale.
- ❸ Identyfikator okna logowania usługi Persona odszukalem ręcznie, wyświetlając witrynę, a następnie poszukałem go za pomocą paska debugowania oferowanego przez przeglądarkę internetową Firefox (naciśnij klawisze *Ctrl+Shift+I*) — patrz rysunek 15.3.
- ❹ Zamiast użyć „prawdziwego” adresu e-mail i być zmuszonym do klikania na kolejnych ekranach procesu uwierzytelniania, zdecydowałem się na użycie „imitacji” dostawcy *MockMyID*⁸ to jeden z nich, możesz wypróbować także *Persona Test User*⁹.

Analiza infrastruktury testowej systemów oferowanych przez firmy trzecie

Testowanie powinno być częścią analizy systemów oferowanych przez firmy trzecie. Kiedy przeprowadzasz integrację z zewnętrzną usługą, powinieneś zastanowić się, jak będziesz ją sprawdzał za pomocą testów funkcjonalnych.

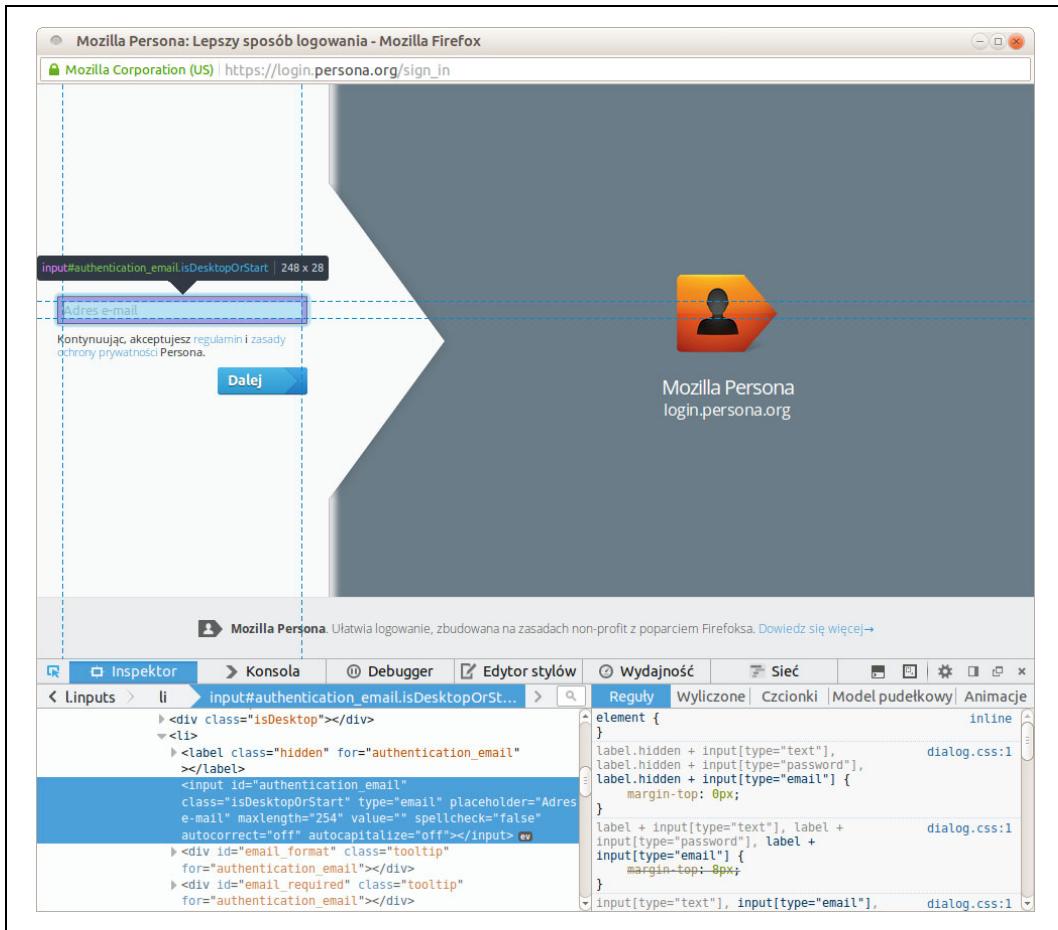
Bardzo często tej samej usługi można użyć zarówno w testach, jak i „rzeczywistym rozwiązaniu”. Jednak czasami trzeba będzie wykorzystać „testową” wersję usługi oferowanej przez firmę trzecią. W przypadku integracji z usługą Persona moglibyśmy użyć „prawdziwego” adresu e-mail. W początkowym szkicu rozdziału przygotowałem test funkcjonalny symulujący kolejne kliknięcia w witrynie Yahoo.com i logowanie się w utworzonym do tego celu koncie. Problem polega na tym, że test funkcjonalny całkowicie opiera się na pewnych szczegółach ekranów logowania do poczty Yahoo, które mogą ulec zmianie w każdej chwili.

Z kolei MockMyID i Persona Test User są powiązane z dokumentacją usługi Persona, działają niezawodnie i pozwalają na przetestowanie istotnych elementów integracji.

Prawdopodobnie nieco inaczej będziesz podchodził do sprawy w przypadku systemów obsługujących płatności. Jeżeli przystąpisz do integracji systemu obsługi płatności, to będzie jeden z najistotniejszych komponentów witryny i dlatego chcesz mieć pewność, że wszystko zostało dokładnie przetestowane... Jednak nie chcesz przecież przeprowadzać faktycznych transakcji na karcie kredytowej za każdym razem, gdy wykonasz testy funkcjonalne. Dlatego też większość dostawców oferuje „testowe” wersje API obsługującego płatności. Ich jakość bywa różna (nie chcę tutaj wymieniać żadnych nazw), więc upewnij się o ich dokładnym przeanalizowaniu.

⁸ <https://mockmyid.com/>

⁹ <http://personatestuser.org/>



Rysunek 15.3. Użycie narzędzi debugowania przeglądarki internetowej Firefox w celu odszukania identyfikatorów

Często stosowana technika Selenium — wyraźne oczekiwanie

Poniżej przedstawiono kod pierwszej z dwóch funkcji pomocniczych obsługujących „oczekiwanie”.

Plik `functional_tests/test_login.py` (ch15l014):

```
import time
[...]

def switch_to_new_window(self, text_in_title):
    retries = 60
    while retries > 0:
        for handle in self.browser.window_handles:
            self.browser.switch_to_window(handle)
            if text_in_title in self.browser.title:
                return
        retries -= 1
        time.sleep(0.5)
    self.fail('nie można znaleźć okna')
```

W powyższej funkcji „zdefiniowaliśmy” oczekiwanie — przeprowadzana jest iteracja przez wszystkie okna bieżącej przeglądarki internetowej w poszukiwaniu okna o konkretnym tytule. Jeżeli wspomniane okno nie zostanie znalezione, kod robi krótką przerwę i następnie próbuje ponownie. W trakcie każdej iteracji kod zmniejsza wartość licznika prób.

Tego rodzaju rozwiązanie jest często stosowane w testach Selenium, gdy dostępne jest API obsługujące oczekiwanie. Jednak to podejście nie zapewnia wystarczająco dobrej obsługi wszystkich przypadków i dlatego musielibyśmy zdefiniować własną funkcję. Kiedy wykonywane jest znacznie prostsze zadanie, takie jak oczekiwanie na pojawienie się na stronie elementu o wskazanym identyfikatorze, wówczas można użyć klasy `WebDriverWait`.

Plik `functional_tests/test_login.py` (ch15l015):

```
from selenium.webdriver.support.ui import WebDriverWait
[...]

def wait_for_element_with_id(self, element_id):
    WebDriverWait(self.browser, timeout=30).until(
        lambda b: b.find_element_by_id(element_id)
```

W ten sposób otrzymujemy to, co przez Selenium jest określane mianem „wyraźnego oczekiwania”. Jak zapewne pamiętasz, w metodzie `FunctionalTest.setUp()` zdefiniowaliśmy „niejawne oczekiwanie”. Wówczas zostało zdefiniowane trzysekundowe oczekiwanie, które jest wystarczające w większości przypadków. Jednak podczas oczekiwania na zewnętrzną usługę, taką jak Persona, czasami trzeba zwiększyć domyślnie ustalony czas oczekiwania.

Więcej przykładów znajdziesz w *dokumentacji Selenium*¹⁰, ale według mnie znacznie więcej pozytku przynosi analiza kodu źródłowego¹¹ — znajdziesz w nim doskonale komentarze.



Wywołanie `implicitly_wait()` jest zawodne, zwłaszcza w przypadku użycia JavaScript. W testach funkcjonalnych znacznie lepiej użyć wzorca „czekaj na”, jeśli zachodzi potrzeba sprawdzenia asynchronicznych interakcji na stronie. Jeszcze do tego powrócimy w rozdziale 20.

Po wykonaniu testów funkcjonalnych widzimy, że zostały zaliczone.

```
$ python3 manage.py test functional_tests.test_login
Creating test database for alias 'default'...
Not Found: /favicon.ico
login view
sending to mozilla {'assertion': [...]
[...]
got b'{"audience":"localhost","expires":...}
[...]
.
-----
Ran 1 test in 32.222s

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

¹⁰ http://docs.seleniumhq.org/docs/04_webdriver_advanced.jsp

¹¹ <https://code.google.com/p/selenium/source/browse/py/selenium/webdriver/support/wait.py>

Możesz nawet zobaczyć pewne dane wyjściowe procesu debugowania, które pozostały w eksperymentalnych implementacjach widoku. Nadeszła teraz pora na wycofanie wszystkich tymczasowych zmian i ich ponowne wprowadzenie pojedynczo, z wykorzystaniem pojęcia opartego na testach.

Wycofanie kodu eksperymentalnego

Wykonaj poniższe polecenia, aby wycofać kod eksperymentalny:

```
$ git checkout master # Powrót do gałęzi master.  
$ rm -rf accounts # Usunięcie całego kodu eksperymentalnego.  
$ git add functional_tests/test_login.py  
$ git commit -m "Testy funkcjonalne dla logowania z użyciem usługi Persona."
```

Teraz ponownie wykonujemy testy funkcjonalne i pozwalamy im kierować pracami programistycznymi:

```
$ python3 manage.py test functional_tests.test_login  
selenium.common.exceptions NoSuchElementException: Message: 'Unable to locate  
element: {"method":"id","selector":"login"}' ; Stacktrace:  
[...]
```

Pierwszym zadaniem jest dodanie łącza pozwalającego użytkownikowi na zalogowanie się. Nawiąsem mówiąc, preferuję poprzedzanie identyfikatorów HTML prefiksem `id_`. To tylko konwencja pozwalająca na łatwe odróżnianie klas i identyfikatorów w kodzie HTML oraz CSS. Zaczynamy więc od modyfikacji testu funkcjonalnego.

Plik `functional_tests/test_login.py` (ch15l017):

```
self.browser.find_element_by_id('id_login').click()  
[...]  
self.wait_for_element_with_id('id_logout')
```

Zajmujemy się łączem logowania, które obecnie nie wykonuje jeszcze żadnego zadania. Framework Bootstrap oferuje pewne wbudowane klasy przeznaczone dla pasków nawigacyjnych, ich więc użyjemy.

Plik `lists/templates/base.html`:

```
<div class="container">  
  
<nav class="navbar navbar-default" role="navigation">  
    <a class="navbar-brand" href="/">Superlists</a>  
    <a class="btn navbar-btn navbar-right" id="id_login" href="#">Zaloguj</a>  
</nav>  
  
<div class="row">  
[...]
```

Po upływie 30 sekund otrzymujemy następujący komunikat błędu:

```
AssertionError: nie można znaleźć okna
```

Możemy przejść dalej! Kolejne zadanie: jeszcze więcej kodu JavaScript.

Testy jednostkowe JavaScript obejmujące komponenty zewnętrzne — nasze pierwsze imitacje

Aby posunąć się nieco dalej w teście funkcjonalnym, na ekranie musimy wyświetlić okno usługi Persona. Konieczne jest więc pozbycie się z kodu JavaScript działającego po stronie klienta dodanego wcześniej kodu eksperymentalnego opartego na bibliotekach usługi Persona. Przeprowadzimy testy z użyciem testów jednostkowych JavaScript oraz imitacji.

Porządkowanie — katalog plików statycznych dla całej witryny

Na początek musimy zaprowadzić nieco porządku. Wewnątrz katalogu *superlists* tworzymy katalog przeznaczony dla plików statycznych o zasięgu całej witryny, a następnie przenosimy do niego wszystkie pliki Bootstrap CSS, kod QUnit, plik *base.css* itd. Struktura katalogów powinna przedstawiać się następująco:

```
$ tree superlists -L 3 -I __pycache__  
superlists  
    __init__.py  
    settings.py  
    static  
        base.css  
        bootstrap  
            css  
            fonts  
            js  
        tests  
            qunit.css  
            qunit.js  
    urls.py  
    wsgi.py  
  
6 directories, 7 files
```



Przed przeprowadzeniem operacji porządkowych, takich jak powyższa, zawsze przekazuj pliki do repozytorium.

Porządkowanie oznacza konieczność modyfikacji istniejących testów jednostkowych JavaScript.

Plik *lists/static/tests/tests.html* (ch15l020):

```
<link rel="stylesheet" href="../../superlists/static/tests/qunit.css">  
[...]  
<script src="http://code.jquery.com/jquery.min.js"></script>  
<script src="../../superlists/static/tests/qunit.js"></script>  
<script src="../../list.js"></script>
```

Upewniamy się, czy nadal działają, otwierając je w przeglądarce internetowej:

2 assertions of 2 passed, 0 failed.

Poniżej pokazano, jak wskazać w pliku *settings.py* pojawienie się nowego katalogu *static*.

Plik `superlists/settings.py`:

```
[...]
STATIC_ROOT = os.path.join(BASE_DIR, '../static')
STATICFILES_DIRS = (
    os.path.join(BASE_DIR, 'superlists', 'static'),
)
```



Zalecam ponowne ustawienie zmiennej `LOGGING`, podobnie jak to zrobiliśmy wcześniej w rozdziale. Nie ma potrzeby jej wyraźnego testowania, ponieważ bieżący zestaw testów poinformuje, jeśli dodanie wymienionej zmiennej cokolwiek uszkodzi w aplikacji. Jak się okaże w rozdziale 17., zmienna `LOGGING` jest niezwykle użyteczna w trakcie procesu usuwania błędów.

Wykonujemy testy jednostkowe dotyczące układu i stylów, aby sprawdzić, czy wszystkie style CSS nadal działają:

```
$ python3 manage.py test functional_tests.test_layout_and_styling
[...]
OK
```

Kolejnym krokiem jest utworzenie aplikacji o nazwie `accounts` przeznaczonej do przechowywania całego kodu związanego z logowaniem. W ten sposób docieramy do zadań wymagających użycia kodu JavaScript w usłudze Persona.

```
$ python3 manage.py startapp accounts
$ mkdir -p accounts/static/tests
```

Porządkowanie zakończone. Teraz jest dobry moment na przekazanie plików do repozytorium. Następnie spojrzymy na inny fragment eksperymentalnego kodu JavaScript:

```
var loginLink = document.getElementById('login');
if (loginLink) {
    loginLink.onclick = function() { navigator.id.request(); };
}
```

Imitacja: kto, co i dlaczego?

Chcemy, aby kliknięcie łącza logowania wywoływało funkcję `navigator.id.request()` dostarczaną przez usługę Persona.

W teście jednostkowym nie chcemy wywołania *rzeczywistej* funkcji oferowanej przez firmę trzecią, aby okna dialogowe usługi Persona nie były nieustannie wyświetlane na ekranie. Dlatego też wykorzystamy tak zwaną „imitację”: utworzymy „fikcyjną” implementację API firmy trzeciej, która będzie używana w testach.

Oznacza to konieczność zastąpienia prawdziwego obiektu `navigator` jego imitacją, którą samodzielnie opracujemy. Wspomniany obiekt imitacji dostarczy informacje o tym, co się dzieje.



Miałem nadzieję, że pierwszy przykład imitacji zostanie utworzony w Pythonie, ale wygląda na to, że to jednak będzie JavaScript. Cóż, nie ma innego wyjścia. Być może uznasz, że warto kilkukrotnie przeczytać dalszą część rozdziału, aby dokładnie zrozumieć wszystkie omawiane tutaj zagadnienia.

Przestrzenie nazw

W kontekście pliku *base.html* navigator to po prostu obiekt w zasięgu globalnym dostarczany przez skrypt *include.js*, który otrzymujemy z Mozilli. Testowanie zmiennej globalnej nastręcza wiele problemów, a więc możemy ją zamienić w zmienną lokalną przez przekazanie jej do funkcji inicjalizującej. Odpowiedni kod w pliku *base.html* będzie miał postać przedstawioną poniżej.

Plik *lists/templates/base.html*:

```
<script src="/static/accounts/accounts.js"></script>
<script>
$(document).ready(function() {
    Superlists.Accounts.initialize(navigator)
});</script>
```

W kodzie zdefiniowano, że przestrzeń nazw funkcji `initialize()` będzie znajdowała się w zagnieżdżonych obiektach `Superlists.Accounts`. JavaScript cierpi na skutek modelu programowania powiązanego z zasięgiem globalnym. Pewnego rodzaju konwencje nazw i przestrzenie nazw pomagają w zachowaniu wszystkiego pod kontrolą. Duża liczba bibliotek JavaScript może chcieć wywoływać funkcję `initialize()`, ale tylko niewiele będzie wywoływać `Superlists.Accounts.initialize()`¹².

Przedstawione wywołanie funkcji `initialize()` jest wystarczająco proste i jestem szczęśliwy, że nie wymaga żadnych testów jednostkowych.

Prosta imitacja dla testów jednostkowych dla naszej funkcji inicjującej

Samą funkcję `initialize()` na pewno *przetestujemy*. Skopuj szkielet kodu HTML dla testów, a następnie zmodyfikuj go w pokazany poniżej sposób.

Plik *accounts/static/tests/tests.html*:

```
<div id="qunit-fixture">
    <a id="id_login">Zaloguj</a>
</div>

<script src="http://code.jquery.com/jquery.min.js"></script>
<script src="../../superlists/static/tests/qunit.js"></script>
<script src="../accounts.js"></script>
<script>

/*global $, test, equal, sinon, Superlists */
test("Inicjalizacja połączenia przycisku logowania z navigator.id.request", function () {
    var requestWasCalled = false; #❶
    var mockRequestFunction = function () { requestWasCalled = true; }; #❷
    var mockNavigator = { #❸
        id: {
            request: mockRequestFunction
        }
    };
    var loginButton = $('#id_login');
    loginButton.click();
    equal(requestWasCalled, true);
});
```

¹² W świecie JavaScript nowym promykiem pozwalającym na uniknięcie problemów z przestrzeniami nazw jest skrypt *require.js*. To jeden z wielu tematów, których omówienie nie zmieściło się w książce, ale zdecydowanie powinieneś się o nim nieco więcej dowiedzieć.

```

    }
};

Superlists.Accounts.initialize(mockNavigator); #❸

$('#id_login').trigger('click'); #❹

equal(requestWasCalled, true); #❺
});

</script>

```

Jednym z najlepszych sposobów zrozumienia przedstawionego testu, a praktycznie każdego testu, jest przeanalizowanie go od końca. Pierwsze, co przyciąga naszą uwagę, to asercja.

- ❶ Przyjmujemy założenie, że wartością zmiennej `requestWasCalled` jest `true`. Sprawdzamy, czy w jakikolwiek sposób nastąpiło wywołanie funkcji `request()`.
- ❷ Kiedy była wywołana? Gdy zdarzenie `click` wystąpiło dla elementu o identyfikatorze `id_login`.
- ❸ Przed wyzwoleniem zdarzenia `click` wywołujemy funkcję `Superlists.Accounts.initialize()`, podobnie jak na prawdziwej stronie. Jedyna różnica polega na tym, że zamiast przekazać zmienną rzeczywistemu obiekowi globalnemu `navigator` biblioteki usługi `Persona`, przekazujemy ją imitacji o nazwie `mockNavigator`¹³.
- ❹ W tym wierszu następuje zdefiniowanie zwykłego obiektu JavaScript wraz z atrybutem o nazwie `id`, posiadającym z kolei atrybut o nazwie `request`, któremu przypisujemy wywołanie `mockRequestFunction()`.
- ❺ Definiujemy `mockRequestFunction()` jako bardzo prostą funkcję, której wywołanie spowoduje przypisanie wartości `true` zmiennej `requestWasCalled`.
- ❻ I na koniec (na początku?) upewniamy się, że pierwotnie wartością zmiennej `requestWasCalled` jest `false`.

Wynik będzie następujący: jedyny sposób na zaliczenie testu polega na tym, że funkcja `initialize()` dołącza zdarzenie `click` w elemencie `id_login` do metody `id_request()` przekazanego obiektu. Jeżeli uda się zaliczyć test za pomocą obiektu imitacji, wówczas zyskujemy pewność, że funkcja `initialize()` będzie działać prawidłowo po przekazaniu jej rzeczywistego obiektu na prawdziwej stronie.

Czy to ma sens? Poeksperymentujmy z testami i zobaczymy, co uda się nam osiągnąć.



Podczas testowania zdarzeń w elementach modelu DOM potrzebujemy rzeczywistego elementu wyzwalającego zdarzenie. We wspomnianym elemencie zarejestrujemy obserwatora zdarzeń. Jeśli o tym zapomnisz, powstanie szczególnie trudny do wykrycia błędów, ponieważ wywołanie `trigger()` zakończy się niczym, a Ty będziesz zastanawiał się, dlaczego rozwiązanie nie działa. Nie zapomnij więc o umieszczeniu znacznika `` gdzieś w elemencie `<div>` o identyfikatorze `qunit-fixture`.

¹³ Ten obiekt nazwalem „imitacją”, choć prawdopodobnie znacznie odpowiedniejszym określeniem będzie „spy”. W tej książce nie musimy się przejmować różnicami w nazewnictwie. Więcej informacji dotyczących ogólnych klas narzędzi określanych mianem „Test Doubles”, w tym między innymi omówienie różnic w nazewnictwie, znajdziesz w książce Emily Bache zatytułowanej *Mocks, Fakes and Stubs* (<https://leanpub.com/mocks-fakes-stubs>).

Pierwszy błąd przedstawia się następująco:

```
1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:35:
Superlists is not defined
```

To odpowiednik błędu `ImportError` występującego w Pythonie. Rozpoczynamy od wprowadzenia modyfikacji w pliku `accounts/static/accounts.js`.

Plik `accounts/static/accounts.js`:

```
window.Superlists = null;
```

Podobnie jak w Pythonie można użyć `Superlists = None`, ale tutaj decydujemy się na `window`.
→`Superlists = None`. Użycie `window` gwarantuje, że pobierany jest obiekt globalny:

```
1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:35:
Superlists is null
```

Przechodzimy do kolejnego kroku lub dwóch.

Plik `accounts/static/accounts.js`:

```
window.Superlists = {
  Accounts: {}  
};
```

Otrzymujemy następujące dane wyjściowe¹⁴:

```
Superlists.Accounts.initialize is not a function
```

Musimy więc zmienić wywołanie na funkcję.

Plik `accounts/static/accounts.js`:

```
window.Superlists = {
  Accounts: {
    initialize: function () {}
  }
};
```

Teraz otrzymujemy test zakończony niepowodzeniem zamiast po prostu błędu:

```
1. inicjalizacja dołączenia przycisku logowania do navigator.id.request (1, 0, 1)
```

```
1. failed
  Expected: true
  Result: false
```

Następnym krokiem jest odseparowanie definicji funkcji `initialize()` od miejsca, w którym jest eksportowana do przestrzeni nazw `Superlists`. Wygenerujemy też dane w konsoli (`console.log()`), co jest w języku JavaScript odpowiednikiem usuwania błędów za pomocą wywołań `print()` w kodzie. Dane wyświetlane w konsoli pokażą, jak wygląda wywołanie funkcji `initialize()`.

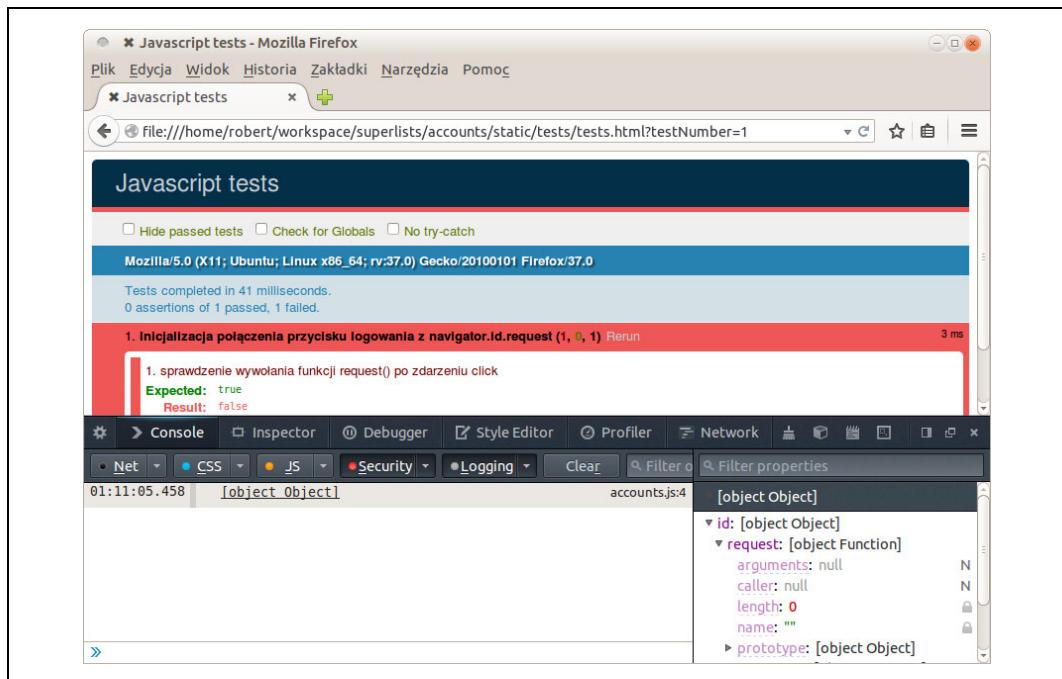
¹⁴ W rzeczywistości podczas konfigurowania przestrzeni nazw w pokazany powyżej sposób należy stosować wzorzec typu „dodaj lub utwórz”. Wówczas jeśli `window.Superlists` znajduje się w zasięgu, to rozszerzamy go, zamiast zastępować nowym. Wywołanie `window.Superlists = window.Superlists || {}` to jedno z możliwych sformułowań, inna możliwość to wywołanie `$.extend` w jQuery. W rozdziale znajduje się duża ilość materiału, a wspomniane zagadnienie to jeden z tematów, na którego dokładne omówienie po prostu zabrakło miejsca.

Plik accounts/static/accounts.js (ch15l028):

```
var initialize = function (navigator) {
    console.log(navigator);
};

window.Superlists = {
    Accounts: {
        initialize: initialize
    }
};
```

W przeglądarkach internetowych Firefox i Chrome naciśnięcie klawiszy *Ctrl+Shift+I* powoduje wyświetlenie konsoli JavaScript, w której możesz zobaczyć (patrz rysunek 15.4) wyświetcone informacje o obiekcie [object Object]. Jeżeli go klikniesz, zobaczysz właściwość zdefiniowaną w teście, czyli id zawierającą funkcję o nazwie request().



Rysunek 15.4. Usuwanie błędów za pomocą konsoli JavaScript

Teraz przekazujemy już funkcję i mamy zaliczony test.

Plik accounts/static/accounts.js (ch15l029):

```
var initialize = function (navigator) {
    navigator.id.request();
};
```

Wprawdzie test zostaje zaliczony, ale otrzymujemy nie do końca interesującą nas implementację. Wywołanie navigator.id.request() występuje zawsze zamiast jedynie po kliknięciu przycisku. Konieczne jest więc dopasowanie testów.

```
1 assertions of 1 passed, 0 failed.
1. inicjalizacja dołączenia przycisku logowania do navigator.id.request (0, 1, 1)
```

Zanim to jednak zrobimy, warto nieco poeksperymentować z kodem i przekonać się, czy naprawdę zrozumiałe jest to, co się tutaj dzieje. Jaki będzie skutek wprowadzenia poniższej modyfikacji?

Plik `accounts/static/accounts.js` (ch15l029-1):

```
var initialize = function (navigator) {
    navigator.id.request();
    navigator.id.doSomethingElse();
};
```

Otrzymamy następujące dane wyjściowe:

```
1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:35:
navigator.id.doSomethingElse is not a function
```

Jak możesz zobaczyć, przekazywany obiekt imitacji `navigator` jest całkowicie pod naszą kontrolą. Zawiera jedynie nadane mu atrybuty i metody. Jeśli chcesz, możesz teraz nieco poeksperymentować.

Plik `accounts/static/tests/tests.html`:

```
var mockNavigator = {
    id: {
        request: mockRequestFunction,
        doSomethingElse: function () { console.log("Wywołany!"); }
    }
};
```

Test zostaje zaliczony, a jeśli zairzysz do konsoli, to zobaczysz następujący wiersz:

```
[01:22:27.456] "Wywołany!"
```

Czy to pomogło Ci w zobaczeniu, co tak naprawdę się tutaj dzieje? Wycofujemy dwie ostatnie zmiany i zmieniamy test jednostkowy w taki sposób, aby sprawdzał, czy funkcja `request()` została wywołana tylko po wyzwoleniu zdarzenia `click`. Dodajemy także komunikaty błędów pomagające w ustaleniu, która z dwóch asercji `equal()` kończy się niepowodzeniem.

Plik `accounts/static/tests/tests.html` (ch15l032):

```
var mockNavigator = {
    id: {
        request: mockRequestFunction
    }
};
Superlists.Accounts.initialize(mockNavigator);
equal(requestWasCalled, false, 'sprawdzenie wywołania funkcji request() przed zdarzeniem click');
$('#id_login').trigger('click');
equal(requestWasCalled, true, 'sprawdzenie wywołania funkcji request() po zdarzeniu click');
```



Komunikaty asercji (trzeci argument wywołania `equal()`) są w QUnit tak naprawdę komunikatami „sukcesu”. Zamiast ich wyświetlania, jedynie w przypadku niepowodzenia testu, pojawiają się również po zaliczeniu testu. Dlatego też ich wydźwięk pozostaje pozytywny.

Teraz otrzymujemy znacznie czytelniejszy komunikat o niepowodzeniu testu:

```
1 assertions of 2 passed, 1 failed.
1.inicjalizacja dołczenia przycisku logowania do navigator.id.request (1, 1, 2)
  1. sprawdzenie wywołania funkcji request() przed zdarzeniem click
    Expected: false
    Result: true
```

Wprowadzamy teraz taką zmianę, aby wywołanie `navigator.id.request()` występowało jedynie po kliknięciu elementu o identyfikatorze `id_login`.

Plik `accounts/static/accounts.js` (ch15l033):

```
/*global $*/
var initialize = function (navigator) {
    $('#id_login').on('click', function () {
        navigator.id.request();
    });
};
```

Test zostaje zaliczony. Dobry początek! Spróbujemy teraz umieścić kod w szablonie.

Plik `lists/templates/base.html`:

```
<script src="http://code.jquery.com/jquery.min.js"></script>
<script src="https://login.persona.org/include.js"></script>
<script src="/static/accounts.js"></script>
<script src="/static/list.js"></script>
<script>
    /*global $, Superlists, navigator */
    $(document).ready(function () {
        Superlists.Accounts.initialize(navigator);
    });
</script>
</body>
```

Konieczne jest również dodanie aplikacji `accounts` do pliku `settings.py`, w przeciwnym razie pliki statyczne pozostaną niedostępne dla `accounts/static/accounts.js`.

Plik `superlists/settings.py`:

```
+++ b/superlists/settings.py
@@ -37,4 +37,5 @@ INSTALLED_APPS =
    'lists',
+
    'accounts',
)
```

Szybkie wykonanie testów funkcjonalnych... Niestety nie udało nam się przejść dalej. Aby dowiedzieć się dlaczego, ręcznie otwieramy witrynę, a następnie zagładamy do konsoli JavaScript:

```
[01:36:54.572] Error: navigator.id.watch must be called before
navigator.id.request @ https://login.persona.org/include.js:8
```

Bardziej zaawansowane imitacje

Potrzebujemy prawidłowego wywołania `navigator.id.watch()`. Jeżeli ponownie spojrzesz na nasz kod eksperymentalny, to zauważysz poniższy fragment:

```
var currentUser = '{{ user.email }}' || null;
var csrf_token = '{{ csrf_token }}';
console.log(currentUser);

navigator.id.watch({
    loggedInUser: currentUser, #❶
    onlogin: function(assertion) {
        $.post('/accounts/login', {assertion: assertion, csrfmiddlewaretoken: csrf_token}) #❷
            .done(function() { window.location.reload(); })
            .fail(function() { navigator.id.logout(); });
    }
});
```

```

    },
    onlogout: function() {
      $.post('/accounts/logout')
        .always(function() { window.location.reload(); });
    }
});

```

Funkcja `watch()` potrzebuje pewnych danych z zasięgu globalnego.

- ❶ Adres e-mail bieżącego użytkownika powinien być przekazany jako parametr `loggedInUser` wywołania `watch()`.
- ❷ Bieżący token CSRF do przekazania żądaniu Ajax POST kierowanego do widoku logowania¹⁵.

W kodzie mamy również dwa na stałe zdefiniowane adresy URL. Znacznie lepszym rozwiązaniem będzie ich pobieranie z Django, na przykład w następujący sposób:

```

var urls = {
  login: "{% url 'login' %}",
  logout: "{% url 'logout' %}",
};

```

To będzie trzeci parametr do przekazania z zasięgu globalnego. Ponieważ mamy funkcję `initialize()`, więc wyobraźmy sobie jej użycie w poniższy sposób:

```
Superlists.Accounts.initialize(navigator, user, token, urls);
```

Użycie imitacji sinon.js do sprawdzenia, czy API zostało prawidłowo wywołane

Jak się wcześniej przekonałeś, „opracowanie własnych” imitacji jest możliwe, a dzięki językowi JavaScript to zadanie należy do dość łatwych. Jednak wykorzystanie biblioteki imitacji pomoże w uniknięciu większego trudu. W świecie JavaScript najpopularniejsza tego typu biblioteka to *sinon.js*. Pobieramy ją (z witryny <http://sinonjs.org/>) i umieszczamy w katalogu *static* o zasięgu całej witryny:

```

$ tree superlists/static/tests/
superlists/static/tests/
  qunit.css
  qunit.js
  sinon.js

```

Następnie nową bibliotekę trzeba dołączyć w testach.

Plik `accounts/static/tests/tests.html`:

```

<script src="http://code.jquery.com/jquery.min.js"></script>
<script src="../../superlists/static/tests/qunit.js"></script>
<script src="../../superlists/static/tests/sinon.js"></script>
<script src="../accounts.js"></script>

```

Teraz już można zacząć tworzyć test używający obiektu oferowanego przez bibliotekę Sinon¹⁶.

¹⁵ Przy okazji zwróć uwagę na użycie wywołania `{{ csrf_token }}`, które daje nam niezmodyfikowany ciąg tekstowy tokenu, zamiast wywołania `{% csrf %}` zwracającego pełny znacznik HTML w postaci `<input type="hidden" name="itd..>`

¹⁶ Biblioteka Sinon oferuje również bardziej specjalizowane obiekty, takie jak „spy” i „stub”. Ponieważ imitacje potrafią to wszystko co spy i stub, więc uznałem, że mniejsza ilość terminologii ułatwia zrozumienie materiału.

Plik `accounts/static/tests/tests.html` (ch15l038):

```
test("inicjalizacja wywołań navigator.id.watch", function () {
  var user = 'bieżący użytkownik';
  var token = 'token csrf';
  var urls = {login: 'login url', logout: 'logout url'};
  var mockNavigator = {
    id: {
      watch: sinon.mock() #❶
    }
  };
  Superlists.Accounts.initialize(mockNavigator, user, token, urls);
  equal(
    mockNavigator.id.watch.calledOnce, #❷
    true,
    'sprawdzenie wywołania funkcji watch()'
  );
});
```

- ❶ Podobnie jak wcześniej tworzymy imitację obiektu `navigator`, ale teraz zamiast ręcznie przygotowywać interesującą nas funkcję, korzystamy z obiektu `sinon.mock()`.
- ❷ Następnie wymieniony obiekt rejestruje to wszystko, co zachodzi wewnątrz właściwości specjalnych, takich jak `calledOnce`, które możemy wykorzystać w asercjach.

Więcej informacji znajdziesz w dokumentacji biblioteki Sinon. Na stronie głównej¹⁷ znajduje się całkiem dobre ogólne omówienie tej biblioteki.

Poniżej przedstawiono oczekiwany wynik testu zakońzonego niepowodzeniem:

```
2 assertions of 3 passed, 1 failed.
```

```
1. inicjalizacja dołączenia przycisku logowania do navigator.id.request (0, 2, 2)
2. inicjalizacja wywołań navigator.id.watch (1, 0, 1)
  1. sprawdzenie wywołania funkcji watch()
    Expected: true
    Result: false
```

Dodajemy więc wywołanie funkcji `watch()`...

Plik `accounts/static/accounts.js`:

```
var initialize = function (navigator) {
  $('#id_login').on('click', function () {
    navigator.id.request();
  });
  navigator.id.watch();
};
```

To jednak powoduje uszkodzenie testu!

```
1 assertions of 2 passed, 1 failed.
```

```
1. inicjalizacja dołączenia przycisku logowania do navigator.id.request (1, 0, 1)
  1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:36:
missing argument 1 when calling function navigator.id.watch

2. inicjalizacja wywołań navigator.id.watch (0, 1, 1)
```

¹⁷ <http://sinonjs.org/>

Mamy wielką zagadkę — rozszyfrowanie znaczenia komunikatu informującego o braku pierwszego argumentu podczas wywoływania funkcji `navigator.id.watch()` zajęło mi nieco czasu. Okazało się¹⁸, że w przeglądarce internetowej Firefox funkcja `watch()` jest wywoływana w każdym obiekcie. Dlatego też trzeba przygotować jej imitację także dla poprzedniego testu.

Plik `accounts/static/tests/tests.html`:

```
test("inicjalizacja dołączenia przycisku logowania do navigator.id.request", function () {
    var requestWasCalled = false;
    var mockRequestFunction = function () { requestWasCalled = true; };
    var mockNavigator = [
        id: {
            request: mockRequestFunction,
            watch: function () {}
        }
    ];
    [...]
```

Po wprowadzeniu odpowiedniej zmiany testy znów są zaliczane:

```
3 assertions of 3 passed, 0 failed.
```

1. inicjalizacja dołączenia przycisku logowania do `navigator.id.request` (0, 2, 2)
2. inicjalizacja wywołań `navigator.id.watch` (0, 1, 1)

Sprawdzenie wywołania argumentów

Jeszcze nie wywołujemy prawidłowo funkcji `watch()`. Trzeba jej przekazać informacje o bieżącym użytkowniku, a także zdefiniować kilka wywołań zwrotnych dotyczących logowania i wylogowania. Pracę rozpoczynamy od użytkownika.

Plik `accounts/static/tests/tests.html` (ch15l042):

```
test("watch widzi bieżącego użytkownika", function () {
    var user = 'bieżący użytkownik';
    var token = 'token csrf';
    var urls = {login: 'login url', logout: 'logout url'};
    var mockNavigator = [
        id: {
            watch: sinon.mock()
        }
    ];
    Superlists.Accounts.initialize(mockNavigator, user, token, urls);
    var watchCallArgs = mockNavigator.id.watch.firstCall.args[0];
    equal(watchCallArgs.loggedInUser, user, 'sprawdzenie użytkownika');
});
```

Mamy bardzo podobną konfigurację (nawiasem mówiąc, to zapach kodu — w kolejnym teście będziemy chcieli pozbyć się pewnego powielonego kodu w teście). Następnie używamy właściwości `.firstCall.args[0]` w obiekcie imitacji, aby sprawdzić, czy funkcji `watch()` został przekazany parametr (`args` przedstawia listę argumentów pozycyjnych). Otrzymujemy następujące dane wyjściowe:

```
3. watch widzi bieżącego użytkownika (1, 0, 1)
   1. Died on test #1
@file:///workspace/superlists/accounts/static/tests/tests.html:72:
watchCallArgs is undefined
```

ponieważ obecnie nie przekazujemy żadnych argumentów funkcji `watch()`. Oto krok po kroku, co możemy zrobić.

¹⁸ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/watch

Plik *accounts/static/accounts.js* (ch15l043):

```
navigator.id.watch({});
```

Otrzymujemy znacznie czytelniejszy komunikat błędu:

```
3. watch widzi bieżącego użytkownika (1, 0, 1)
  1. sprawdzenie użytkownika
    Expected: "bieżący użytkownik"
    Result: undefined
```

Usuwamy problem.

Plik *accounts/static/accounts.js* (ch15l044):

```
var initialize = function (navigator, user, token, urls) {
  [...]
  navigator.id.watch({
    loggedInUser: user
  });
}
```

Doskonale!

```
4 assertions of 4 passed, 0 failed.
```

Konfiguracja QUnit i testowanie żądań Ajax

Kolejnym krokiem jest sprawdzenie wywołania zwrotnego `onlogin`, które zaczyna działać, gdy usługa Persona otrzyma pewne informacje uwierzytelniające użytkownika i chcemy je przekazać do naszego serwera w celu weryfikacji. W tym miejscu używane jest wywołanie Ajax (`$.post`). Wymienione żądania są trudne do przetestowania, ale biblioteka Sinon oferuje pomoc w postaci *Fake XMLHttpRequest*¹⁹.

W ten sposób radzimy sobie z rodzimą klasą `XMLHttpRequest` w języku JavaScript. Dobrą praktyką jest upewnienie się po zakończeniu testów, że używana jest rzeczywista klasa `XMLHttpRequest`, a nie jej imitacja. Mamy więc całkiem dobry powód do poznania metod `setup()` i `teardown()` oferowanych przez jQuery. Wymienione metody są używane w funkcji o nazwie `module()`, działającej trochę na zasadzie podobnej do klasy `unittest.TestCase`, i grupują wszystkie testy pojawiające się później.

Uwagi dotyczące technologii Ajax

Jeżeli wcześniej nie zetknąłeś się z technologią Ajax, tutaj znajdziesz jej niezwykle krótkie omówienie. Jednak zanim przejdziesz dalej, dobrym rozwiązaniem będzie poczytanie o technologii Ajax w innych źródłach.

Akronim Ajax oznacza „asynchroniczny JavaScript i XML”, choć obecnie fragment dotyczący XML jest już nieodpowiedni, ponieważ praktycznie zawsze wysyłany jest tekst lub dane w formacie JSON zamiast XML. Ajax pozwala działającemu po stronie klienta kodowi JavaScript na wysyłanie i otrzymywanie informacji za pomocą protokołu HTTP (żądania GET i POST), ale w sposób „asynchroniczny”, czyli bez blokowania i oczekiwania na odświeżenie strony.

W naszej aplikacji będziemy używać żądań Ajax w celu wykonywania żądań POST do widoku logowania, wysyłając mu otrzymane z usługi Persona informacje asercji. W kodzie wykorzystamy *wygodne funkcje jQuery Ajax*²⁰.

¹⁹ <http://sinonjs.org/docs/#server>

²⁰ <http://api.jquery.com/jQuery.post/>

Dodajemy omawiany „moduł” po pierwszym teście, ale jeszcze przed testem dla „inicjalizacji wywołań `navigator.id.watch()`”.

Plik `accounts/static/tests/tests.html` (ch15l045):

```
var user, token, urls, mockNavigator, requests, xhr; #❶
module("testy navigator.id.watch", {
    setup: function () {
        user = 'bieżący użytkownik'; #❷
        token = 'token csrf';
        urls = { login: 'login url', logout: 'logout url' };
        mockNavigator = {
            id: {
                watch: sinon.mock()
            }
        };
        xhr = sinon.useFakeXMLHttpRequest(); #❸
        requests = []; #❹
        xhr.onCreate = function (request) { requests.push(request); }; #❺
    },
    teardown: function () {
        mockNavigator.id.watch.reset(); #❻
        xhr.restore(); #❼
    }
});
test("inicjalizacja wywołań navigator.id.watch", function () {
    [...]
});
```

- ❶ Zmienne `user`, `token`, `urls` itd. są przekazywane do większego zasięgu, aby były dostępne dla wszystkich testów w pliku.
- ❷ Wymienione wcześniej zmienne inicjujemy wewnętrz funkcji `setup()`, która podobnie jak funkcja `setUp()` modułu `unittest` będzie wywołana przed każdym testem. To obejmuje również nasz `mockNavigator`.
- ❸ Wywołujemy również oferowaną przez bibliotekę Sinon funkcję `useFakeXMLHttpRequest()`, która poprawia możliwości przeglądarki internetowej w zakresie obsługi żądań Ajax.
- ❹ Jeszcze jedno zadanie związane z tworzeniem szkieletu kodu. Nakazujemy bibliotece Sinon pobieranie wszelkich żądań Ajax i ich umieszczenie w tablicy `requests`, aby mogły być analizowane w trakcie testów.
- ❺ Wreszcie przeprowadzamy operacje porządkowe — pomiędzy poszczególnymi testami następuje „wyzerowanie” imitacji dla funkcji `watch()`. W przeciwnym razie wywołania z jednego testu pojawiłyby się w innych.
- ❻ Na koniec przywracamy użycie obiektu `XMLHttpRequest` udostępnianego przez JavaScript.

Zyskaliśmy możliwość przepisania dwóch testów i ich znacznego skrócenia.

Plik `accounts/static/tests/tests.html` (ch15l046):

```
test("inicjalizacja wywołań navigator.id.watch", function () {
    Superlists.Accounts.initialize(mockNavigator, user, token, urls);
    equal(mockNavigator.id.watch.calledOnce, true, 'sprawdzenie wywołania funkcji watch');
});

test("watch widzi bieżącego użytkownika", function () {
    Superlists.Accounts.initialize(mockNavigator, user, token, urls);
    var watchCallArgs = mockNavigator.id.watch.firstCall.args[0];
    equal(watchCallArgs.loggedInUser, user, 'sprawdzenie użytkownika');
});
```

Testy nadal są zaliczane, ale ich nazwy zawierają prefiks wskazujący nazwę modułu:

4 assertions of 4 passed, 0 failed.

- 1.inicjalizacja dołączenia przycisku logowania do navigator.id.request (0, 2, 2)
2. testy navigator.id.watch:inicjalizacja wywołań navigator.id.watch (0, 1, 1)
3. testy navigator.id.watch:watch widzi bieżącego użytkownika (0, 1, 1)

Poniżej przedstawiono kod pozwalający na przetestowanie wywołań zwrotnych onlogin.

Plik accounts/static/tests/tests.html (ch15l047):

```
test("onlogin wykonuje żądanie ajax post do login url", function () {  
    Superlists.Accounts.initialize(mockNavigator, user, token, urls);  
    var onloginCallback = mockNavigator.id.watch.firstCall.args[0].onlogin; #❶  
    onloginCallback(); #❷  
    equal(requests.length, 1, 'sprawdzenie żądania ajax'); #❸  
    equal(requests[0].method, 'POST');  
    equal(requests[0].url, urls.login, 'sprawdzenie adresu url');  
});  
  
test("onlogin wysyła asercję wraz z tokenem csrf", function () {  
    Superlists.Accounts.initialize(mockNavigator, user, token, urls);  
    var onloginCallback = mockNavigator.id.watch.firstCall.args[0].onlogin;  
    var assertion = 'browser-id assertion';  
    onloginCallback(assertion);  
    equal(  
        requests[0].requestBody,  
        $.param({ assertion: assertion, csrfmiddlewaretoken: token }), #❹  
        'sprawdzenie danych żądania POST'  
    );  
});
```

- ❶ Zdefiniowana imitacja dla funkcji watch() pozwala na wyodrębnienie funkcji wywołania zwrotnego ustawionej jako onlogin.
- ❷ Rzeczywiste wywołanie funkcji w celu jej przetestowania.
- ❸ Oferowany przez bibliotekę Sinon serwer fakeXMLHttpRequest będzie przechwytywał wszystkie wykonywane żądania Ajax i umieści je w tablicy requests. To pozwoli później na sprawdzenie na przykład typu żądania oraz jego docelowego adresu URL.
- ❹ Rzeczywiste parametry żądania POST są przechowywane w .requestBody, ale zakodowane w postaci adresu URL (używają składni &klucz=wartość). Oferowana przez jQuery funkcja \$.param dekoduje adres URL i dlatego używamy jej w operacji porównania.

Oba testy kończą się spodziewanym niepowodzeniem:

4. navigator.id.watch tests: onlogin wykonuje żądanie ajax post do login url (1, 0, 1)
 1. Died on test #1

@file:///workspace/superlists/accounts/static/tests/tests.html:78:
onloginCallback is not a function
5. navigator.id.watch tests: onlogin wysyła asercję wraz z tokenem csrf (1, 0, 1)
 1. Died on test #1

@file:///workspace/superlists/accounts/static/tests/tests.html:90:
onloginCallback is not a function

Następny cykl to test jednostkowy i tworzenie kodu. Poniżej przedstawiłem komunikaty błędów, które były kolejno wyświetlane:

```

1. sprawdzenie żądania ajax
Expected: 1

...
3. sprawdzenie adresu url
Expected: "login url"

...
7 assertions of 8 passed, 1 failed.
1. sprawdzenie danych żądania POST
Expected:
"assertion=browser-id+assertion&csrfmiddlewaretoken=csrf+token"
Result: null

...
1. sprawdzenie danych żądania POST
Expected:
"assertion=browser-id+assertion&csrfmiddlewaretoken=csrf+token"
Result: "assertion=browser-id+assertion"

...
8 assertions of 8 passed, 0 failed.

```

Wreszcie udało mi się opracować przedstawiony poniżej kod.

Plik *accounts/static/accounts.js*:

```

navigator.id.watch({
  loggedInUser: user,
  onlogin: function (assertion) {
    $.post(
      urls.login,
      { assertion: assertion, csrfmiddlewaretoken: token }
    );
  }
});

```

Wylogowanie

W trakcie pisania tej książki stan kodu wylogowania (*onlogout*) w API funkcji *watch()* pozostawał niepewny. Wprawdzie kod działa, ale niekoniecznie nadaje się do naszych celów. Wykorzystaliśmy go jako funkcję, która nic nie robi, po prostu w charakterze miejsca zarezerwowanego. Poniżej przedstawiono minimalną wersję testu dla wylogowania.

Plik *accounts/static/tests/tests.html* (ch15l053):

```

test("onlogout to jedynie miejsce zarezerwowane", function () {
  Superlists.Accounts.initialize(mockNavigator, user, token, urls);
  var onlogoutCallback = mockNavigator.id.watch.firstCall.args[0].onlogout;
  equal(typeof onlogoutCallback, "function", "onlogout to powinna być funkcja");
});

```

Sama funkcja wylogowania przedstawia się niezwykle prosto, jak pokazano poniżej.

Plik *accounts/static/accounts.js* (ch15l054):

```

},
onlogout: function () {}
});

```

Więcej zagnieżdżonych wywołań zwrotnych! Testowanie kodu asynchronicznego

Do tego celu został przeznaczony JavaScript! Na szczęście biblioteka Sinon okazuje się tutaj pomocna. Nadal trzeba przetestować metody `post()` odpowiedzialne za obsługę logowania oraz zdefiniować pewne wywołania zwrotne przeznaczone do realizacji zadań po wykonaniu żądania POST:

```
.done(function() { window.location.reload(); })
.fail(function() { navigator.id.logout();});
```

Pominę tutaj testowanie `window.location.reload`, ponieważ to jest niepotrzebnie skomplikowane²¹ i uważam, że wymienione wywołanie może być przetestowane przez test Selenium. Przygotujemy test dla wywołania zwrotnego kończącego się niepowodzeniem, aby pokazać istnienie takiej możliwości.

Plik `accounts/static/tests/tests.html` (ch15l055):

```
test("niepowodzenie onlogin powinno wywoływać navigator.id.logout", function () {
  mockNavigator.id.logout = sinon.mock(); #❶
  Superlists.Accounts.initialize(mockNavigator, user, token, urls);
  var onloginCallback = mockNavigator.id.watch.firstCall.args[0].onlogin;
  var server = sinon.fakeServer.create(); #❷
  server.respondWith([403, {}, "permission denied"]); #❸

  onloginCallback();
  equal(mockNavigator.id.logout.called, false, 'nie należy jeszcze wylogowywać');

  server.respond(); #❹
  equal(mockNavigator.id.logout.called, true, 'należy wywołać logout');
});
```

- ❶ Tworzymy imitację dla interesującej funkcji `navigator.id.logout`.
- ❷ W celu symulacji odpowiedzi serwera Ajax używany jest oferowany przez bibliotekę Sinon `fakeServer`, który jest abstrakcją opartą na `fakeXMLHttpRequest`.
- ❸ Konfigurujemy serwer imitacji w celu udzielania odpowiedzi o kodzie stanu 403 (brak uprawnień) do symulacji sytuacji, gdy użytkownik jest nieuwierzytelny.
- ❹ Wyraźnie nakazujemy serwerowi imitacji wysłanie odpowiedzi. Tylko wówczas powinno pojawić się wywołanie operacji wylogowania.

To doprowadziło nas do poniższego kodu, nieco zmodyfikowanego względem kodu eksperymentalnego.

Plik `accounts/static/accounts.js` (ch15l056):

```
onlogin: function (assertion) {
  $.post(
    urls.login,
    { assertion: assertion, csrfmiddlewaretoken: token }
  ).fail(function () { navigator.id.logout(); });
},
onlogout: function () {}
```

²¹ Istnieje możliwość przygotowania imitacji dla `window.location.reload`. W takim przypadku trzeba zdefiniować (nieprzetestowaną) funkcję o nazwie `Superlists.Accounts.refreshPage()`, a następnie użyć jej jako imitacji do sprawdzenia, czy działa jak funkcja wywołania zwrotnego `.done()`.

Na końcu dodajemy wywołanie `window.location.reload` w celu sprawdzenia, czy nie został uszkodzony którykolwiek test jednostkowy.

Plik `accounts/static/accounts.js` (ch15l057):

```
navigator.id.watch({
  loggedInUser: user,
  onlogin: function (assertion) {
    $.post(
      urls.login,
      { assertion: assertion, csrfmiddlewaretoken: token }
    )
      .done(function () { window.location.reload(); })
      .fail(function () { navigator.id.logout(); });
  },
  onlogout: function () {}
});
```

Wszystkie testy nadal są zaliczane:

```
11 assertions of 11 passed, 0 failed.
```

Jeżeli połączone wywołania `.done()` i `.fail()` irytują Cię — mnie na pewno lekko irytują — wówczas można je przepisać na następującą postać:

```
var deferred = $.post(
  urls.login,
  { assertion: assertion, csrfmiddlewaretoken: token }
);
deferred.done(function () { window.location.reload(); })
deferred.fail(function () { navigator.id.logout(); });
```

Jednak kod wykonywany asynchronicznie zawsze jest nieco zagmatwany. Przedstawiony kod można odczytać następująco: „wykonaj żądanie POST do `urls.login` wraz z asercją i tokenem CSRF, po zakończeniu odśwież okno, natomiast w przypadku niepowodzenia wykonaj `navigator.id.logout`”. Więcej informacji na temat tak zwanych „obietnic” znajdziesz w tym²² artykule.

Dochodzimy do chwili prawdy: czy nasze testy funkcjonalne da się posunąć choć trochę do przodu? Najpierw trzeba zmodyfikować wywołanie funkcji `initialize()`.

Plik `lists/templates/base.html`:

```
<script>
/*global $, Superlists, navigator */
$(document).ready(function () {
  var user = "{{ user.email }}" || null;
  var token = "{{ csrf_token }}";
  var urls = {
    login: "TODO",
    logout: "TODO",
  };
  Superlists.Accounts.initialize(navigator, user, token, urls);
});
</script>
```

²² <http://otaqui.com/blog/1637/introducing-javascript-promises-aka-futures-in-google-chrome-canary/>

Następnie można już wykonać testy funkcjonalne...

```
$ python3 manage.py test functional_tests.test_login
Creating test database for alias 'default'...
Not Found: /favicon.ico
Not Found: /TODO
E
=====
ERROR: test_login_with_persona (functional_tests.test_login.LoginTest)

Traceback (most recent call last):
  File "/workspace/superlists/functional_tests/test_login.py", line 47, in
    test_login_with_persona
    self.wait_for_element_with_id('id_logout')
  File "/workspace/superlists/functional_tests/test_login.py", line 23, in
    wait_for_element_with_id
    lambda b: b.find_element_by_id(element_id)
[...]
selenium.common.exceptions.TimeoutException: Message: ''

Ran 1 test in 28.779s
FAILED (errors=1)
Destroying test database for alias 'default'...
```

Hura! Wiem, że zakończyły się niepowodzeniem, ale zobaczyliśmy okno dialogowe usługi Persona, a więc możemy je wykorzystać. W następnym rozdziale przechodzimy do kodu działającego po stronie serwera.

Kod eksperymentalny i imitacje w JavaScript

Kod eksperymentalny

Kod eksperymentalny służy do poznawania nowego API lub też poszukiwania nowego rozwiązania. W tego rodzaju kodzie można obyć się bez testów. Dobrym rozwiązaniem jest tworzenie kodu eksperymentalnego w oddzielnej gałęzi i powrót do gałęzi master podczas zmiany kodu eksperymentalnego na zwykły.

Imitacje

Imitacje w testach jednostkowych są używane, gdy mamy zewnętrzną zależność, której tak naprawdę nie chcemy używać w naszych testach. Imitacja służy do symulacji API opracowanego przez firmę trzecią. Wprawdzie istnieje możliwość „przygotowania” własnych imitacji w języku JavaScript, ale framework imitacji, taki jak Sinon, oferuje wiele użytecznych skrótów ułatwiających tworzenie (i co znacznie ważniejsze odczyt) Twoich testów.

Testy jednostkowe metod Ajax

Framework Sinon oferuje ogromną pomoc w tym zakresie. Ręczne przygotowywanie imitacji metod Ajax to prawdziwe utrapienie.

Uwierzytelnianie po stronie serwera i imitacje w Pythonie

Zajmiemy się teraz kodem po stronie serwera naszego systemu uwierzytelniania. W tym rozdziale nadal będziemy opierać się na imitacjach, ale tym razem w Pythonie. Ponadto do wiesz się, jak w Django dostosować do własnych potrzeb system uwierzytelniania.

Rzut oka na wersję eksperymentalną widoku logowania

Na końcu poprzedniego rozdziału mieliśmy opracowany kod działający po stronie klienta i odpowiedzialny za wysyłanie asercji uwierzytelniania do znajdującego się w serwerze widoku logowania. Pracę rozpoczęmy od utworzenia wspomnianego widoku, a następnie przejdziemy dalej do funkcji systemu uwierzytelniania.

Oto wersja eksperymentalna widoku logowania:

```
def persona_login(request):
    print('login view', file=sys.stderr)
    #user = PersonaAuthenticationBackend().authenticate(request.POST['assertion'])
    user = authenticate(assertion=request.POST['assertion']) #❶
    if user is not None:
        login(request, user) #❷
    return redirect('/')
```

- ❶ Funkcja `authenticate()` została dostosowana do potrzeb uwierzytelniania. Na razie to kod eksperymentalny, później zastąpimy go zwykłym. Zadaniem funkcji jest pobranie asercji od klienta, a następnie jej weryfikacja.
- ❷ Funkcja `login()` jest wbudowana w Django i odpowiada za obsługę logowania. Przechowuje obiekt sesji w serwerze powiązanym z plikami cookies użytkownika. Dlatego też w trakcie kolejnych żądań rozpoznaje użytkownika jako uwierzytelnionego.

Przedstawiona tutaj funkcja `authenticate()` będzie przez internet wykonywała wywołania do serwerów Mozilla. Ponieważ takie zachowanie w testach jednostkowych jest niepożądane, musimy przygotować imitację dla wymienionej funkcji.

Imitacje w Pythonie

Począwszy od wydania Python 3.3¹, cieszący się popularnością pakiet `mock` jest dodawany do biblioteki standardowej. Otrzymujesz niemal magiczny obiekt o nazwie `Mock` przypominający nieco pokazane w poprzednim rozdziale obiekty biblioteki `Sinon`, ale znacznie użyteczniejsze. Spójrz na poniższy fragment interaktywnej sesji Pythona:

```
>>> from unittest.mock import Mock
>>> m = Mock()
>>> m.any_attribute
<Mock name='mock.any_attribute' id='140716305179152'>
>>> m.foo
<Mock name='mock.foo' id='140716297764112'>
>>> m.any_method()
<Mock name='mock.any_method()' id='140716331211856'>
>>> m.foo()
<Mock name='mock.foo()' id='140716331251600'>
>>> m.called
False
>>> m.foo.called
True
>>> m.bar.return_value = 1
>>> m.bar()
1
```

Obiekt imitacji byłby idealnym rozwiązaniem do przygotowania imitacji dla wspomnianej wcześniej funkcji `authenticate()`. Poniżej zobaczysz, jak można to zrobić.

Testowanie widoku za pomocą imitacji funkcji uwierzytelnienia

(Ufam Ci, że przygotowałeś katalog `tests`. Nie zapomnij o usunięciu domyślnego pliku `tests.py`).

Plik `accounts/tests/test_views.py`:

```
from django.test import TestCase
from unittest.mock import patch

class LoginViewTest(TestCase):

    @patch('accounts.views.authenticate') #❶
    def test_calls_authenticate_with_assertion_from_post(
        self, mock_authenticate #❷
    ):
        mock_authenticate.return_value = None #❸
        self.client.post('/accounts/login', {'assertion': 'assert this'}) #❹
        mock_authenticate.assert_called_once_with(assertion='assert this')
```

- ❶ Dekorator o nazwie `patch` przypomina oferowaną przez bibliotekę `Sinon` funkcję `mock()`, którą wykorzystywaliśmy w poprzednim rozdziale. Pozwala on na określenie imitowanego obiektu. W omawianym przykładzie tworzymy imitację funkcji `authenticate()`, z której mamy zamiar korzystać w pliku `accounts/view.py`.
- ❷ Dekorator dodaje obiekt imitacji jako kolejny argument funkcji, w której został zastosowany.

¹ Jeżeli używasz wydania 3.2, uaktualnij Pythona do wersji 3.3. Natomiast jeśli musisz pozostać przy wcześniejszej wersji, wówczas zainstaluj pakiet `mock` za pomocą polecenia `pip3 install mock`, a następnie używaj `from mock` zamiast `from unittest.mock`.

- ❸ Następnie przystępujemy do konfiguracji obiektu imitacji, aby zachowywał się w pożądany sposób. Najprostsze rozwiązanie to zwrot wartości `None` przez funkcję `authenticate()`, a więc definiujemy atrybut specjalny `.return_value`. W przeciwnym razie wartością zwrotną byłaby inna imitacja, co niewątpliwie wprowadziłoby zamieszanie w widoku.
- ❹ Imitacje mogą definiować asercje. W omawianym przykładzie sprawdzamy, czy zostały wywołane oraz z jakimi wartościami.

Jakie uzyskaliśmy korzyści?

```
$ python3 manage.py test accounts
[...]
AttributeError: <module 'accounts.views' from
'./workspace/superlists/accounts/views.py'> does not have the attribute
'authenticate'
```

Podjęliśmy próbę poprawy czegoś, co jeszcze nie istnieje. Konieczne jest zimportowanie `authenticate` w pliku `views.py`².

Plik `accounts/views.py`:

```
from django.contrib.auth import authenticate
```

Otrzymujemy następujące dane wyjściowe:

```
AssertionError: Expected 'authenticate' to be called once. Called 0 times.
```

To jest oczekiwane niepowodzenie. Przygotowanie implementacji wymaga podania adresu URL wskazującego widok logowania.

Plik `superlists/urls.py`:

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/', include('lists.urls')),
    url(r'^accounts/', include('accounts.urls')),
    #url(r'^admin/', include(admin.site.urls)),
)
```

Plik `accounts/urls.py`:

```
from django.conf.urls import patterns, url

urlpatterns = patterns('',
    url(r'^login$', 'accounts.views.persona_login', name='persona_login'),
)
```

Czy minimalna wersja widoku wykonuje jakiekolwiek zadanie?

Plik `accounts/views.py`:

```
from django.contrib.auth import authenticate

def persona_login():
    pass
```

Tak:

```
TypeError: persona_login() takes 0 positional arguments but 1 was given
```

² Choć zamierzamy zdefiniować własną wersję funkcji `authenticate()`, nadal możemy importować kod z `django.contrib.auth`. Django automatycznie zastąpi go funkcją wskazaną w pliku `settings.py`. Korzyść z takiego rozwiązania jest następująca: jeśli później zdecydujesz się na użycie biblioteki firmy trzeciej dla funkcji `authenticate()`, plik `views.py` nie będzie wymagał żadnych zmian.

I jeszcze:

Plik *accounts/views.py* (ch16l008):

```
def persona_login(request):
    pass
```

Następnie:

ValueError: The view accounts.views.persona_login didn't return an HttpResponseRedirect object. It returned None instead.

Plik *accounts/views.py* (ch16l009):

```
from django.contrib.auth import authenticate
from django.http import HttpResponseRedirect

def persona_login(request):
    return HttpResponseRedirect()
```

I powracamy do poniższego komunikatu:

AssertionError: Expected 'authenticate' to be called once. Called 0 times.

Podejmujemy kolejną próbę.

Plik *accounts/views.py*:

```
def persona_login(request):
    authenticate()
    return HttpResponseRedirect()
```

I oczywiście otrzymujemy komunikat:

*AssertionError: Expected call: authenticate(assertion='assert this')
Actual call: authenticate()*

Także i powyższy problem można usunąć.

Plik *accounts/views.py*:

```
def persona_login(request):
    authenticate(assertion=request.POST['assertion'])
    return HttpResponseRedirect()
```

Dobrze. W ten sposób utworzyliśmy imitację jednej funkcji Pythona i przetestowaliśmy ją.

Sprawdzenie, czy widok faktycznie loguje użytkownika

Jednak funkcja `authenticate()` musi również faktycznie zalogować użytkownika za pomocą wywołania funkcji `auth.login()`, jeżeli `authenticate()` zwróci użytkownika. Odpowiedź nie może być więc pusta — ponieważ widok działa w technologii Ajax, odpowiedź nie musi zawierać kodu HTML, wystarczy ciąg tekstowy OK.

Plik *accounts/tests/test_views.py* (ch16l011):

```
from django.contrib.auth import get_user_model
from django.test import TestCase
from unittest.mock import patch
User = get_user_model() #❶

class LoginViewTest(TestCase):
    @patch('accounts.views.authenticate')
    def test_calls_authenticate_with_assertion_from_post(
        self, patched_authenticate):
        [...]
```

```

@patch('accounts.views.authenticate')
def test_returns_OK_when_user_found(
    self, mock_authenticate
):
    user = User.objects.create(email='a@b.com')
    user.backend = '' # Wymagane do działania auth_login.
    mock_authenticate.return_value = user
    response = self.client.post('/accounts/login', {'assertion': 'a'})
    self.assertEqual(response.content.decode(), 'OK')

```

- ❶ Powinienem wyjaśnić to użycie wywołania `get_user_model()` z `django.contrib.auth`. Jego zadaniem jest wyszukanie w projekcie modelu użytkownika, niezależnie od tego, czy używany jest standardowy model użytkownika, czy niestandardowy (jak to będzie w omawianym przykładzie).

Przedstawiony test zapewnia obsługę żądanej odpowiedzi. Teraz można faktycznie sprawdzić, czy użytkownik naprawdę został zalogowany. Odbywa się to przez analizę klienta testowego Django i sprawdzenie, czy nastąpiło poprawne ustawienie pliku cookie dla sesji.



Na tym etapie warto zapoznać się z dokumentacją Django poświęconą *uwierzytelnianiu*³.

Plik `accounts/tests/test_views.py` (ch16l012):

```

from django.contrib.auth import get_user_model, SESSION_KEY
[...]
@patch('accounts.views.authenticate')
def test_gets_logged_in_session_if_authenticate_returns_a_user(
    self, mock_authenticate
):
    user = User.objects.create(email='a@b.com')
    user.backend = '' # required for auth_login to work
    mock_authenticate.return_value = user
    self.client.post('/accounts/login', {'assertion': 'a'})
    self.assertEqual(self.client.session[SESSION_KEY], user.pk) #❶

@patch('accounts.views.authenticate')
def test_does_not_get_logged_in_if_authenticate_returns_None(
    self, mock_authenticate
):
    mock_authenticate.return_value = None
    self.client.post('/accounts/login', {'assertion': 'a'})
    self.assertNotIn(SESSION_KEY, self.client.session) #❷

```

- ❶ Klient testowy Django monitoruje sesję użytkownika. Jeżeli uwierzytelnienie użytkownika zakończyło się powodzeniem, wtedy należy sprawdzić, czy jego identyfikator (klucz podstawowy, w kodzie oznaczony jako `pk`) jest powiązany z sesją.
- ❷ Jeżeli użytkownik nie powinien być uwierzytelniony, wówczas sesja nie powinna zawierać `SESSION_KEY`.

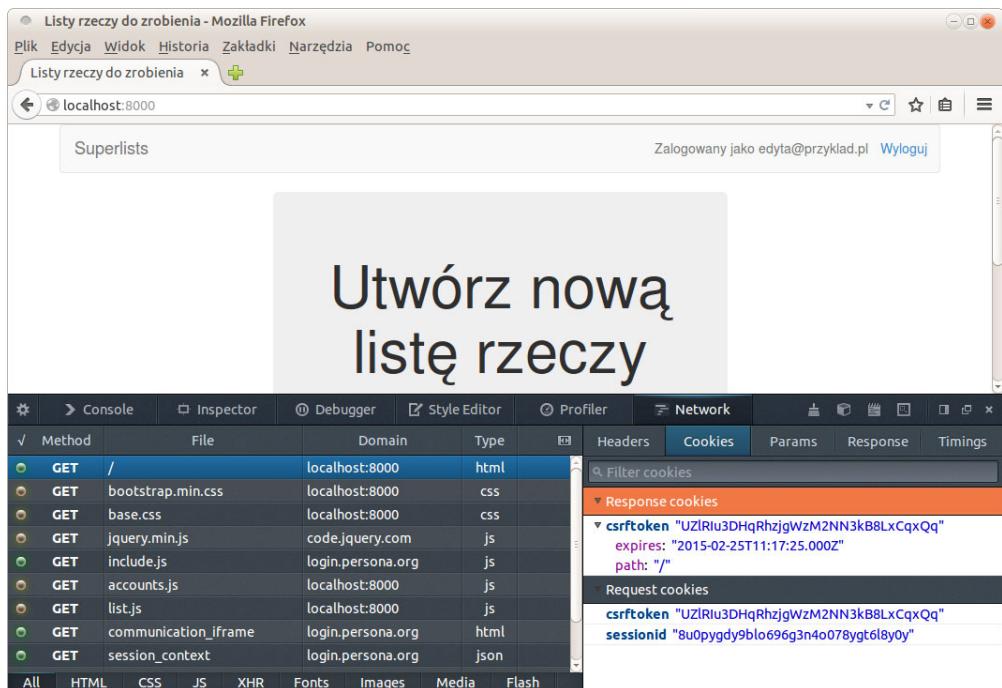
³ <https://docs.djangoproject.com/en/1.7/topics/auth/default/#how-to-log-a-user-in>

Sesje Django: jaki plik cookie użytkownika informuje serwer, że użytkownik jest uwierzytelniony?

Spróbuję teraz wyjaśnić temat sesji, cookies i uwierzytelniania w Django.

Ponieważ protokół HTTP jest bezstanowy, serwer potrzebuje rozwiązania pozwalającego mu na rozpoznawanie różnych klientów w trakcie każdego żądania. Adres IP może być współdzielony, dlatego też najczęściej stosowane podejście polega na nadaniu każdemu klientowi unikalowego identyfikatora, który jest przechowywany w pliku cookie i dołączany do każdego żądania. Serwer również przechowuje gdzieś identyfikator klienta (domyślnie w bazie danych) i dzięki temu może rozpoznać każde żądanie jako pochodzące od konkretnego klienta.

Jeżeli zalogujesz się do witryny, używając serwera programistycznego, zyskujesz możliwość przeanalizowania identyfikatora sesji, jak pokazano na rysunku 16.1. Domyślnie jest on przechowywany w kluczu sessionid.



Rysunek 16.1. Analiza pliku cookie sesji w narzędziach oferowanych przez przeglądarkę Firefox

Wspomniane pliki cookie sesji są tworzone dla wszystkich odwiedzających witrynę Django, niezależnie od tego, czy zostali zalogowani, czy nie.

Kiedy chcemy rozpoznać klienta jako zalogowanego i uwierzytelnionego użytkownika, wtedy zamiast w trakcie każdego żądania pytać go o nazwę użytkownika i hasło, serwer oznacza sesję klienta jako uwierzytelnioną i wiąże ją z przechowywanym w bazie danych identyfikatorem użytkownika.

Sesja to przypominająca słownik struktura danych, a identyfikator użytkownika jest przechowywany w kluczu nadanym przez `django.contrib.auth.SESSION_KEY`. Jeśli chcesz, możesz się o tym przekonać za pomocą konsoli polecenia `manage.py`:

```
$ python3 manage.py shell
[...]
In [1]: from django.contrib.sessions.models import Session

# Ponizej znajdziesz identyfikator sesji pochodzący z pliku cookie w Twojej przeglądarce internetowej.
In [2]: session = Session.objects.get(
    session_key="8u0pygdy9blo696g3n4o078ygt618y0y"
)

In [3]: print(session.get_decoded())
{'_auth_user_id': 'harry@mockmyid.com', '_auth_user_backend':
'accounts.authentication.PersonaAuthenticationBackend'}
```

W sesji użytkownika można przechowywać różne inne informacje, co pozwala na tymczasowe monitorowanie pewnego stanu. Takie podejście sprawdza się także w przypadku niezalogowanych użytkowników. Wewnątrz widoku wystarczy po prostu użyć obiektu `request.session`, którego działanie przypomina słownik. Więcej informacji na ten temat znajdziesz w dokumentacji Django poświęconej *sesjom*⁴.

Otrzymujemy dwa niepowodzenia:

```
$ python3 manage.py test accounts
[...]
    self.assertEqual(self.client.session[SESSION_KEY], user.pk)
KeyError: '_auth_user_id'

[...]
AssertionError: '' != 'OK'
+ OK
```

Funkcja Django, która zajmuje się logowaniem użytkownika (przez oznaczenie jego sesji), to `django.contrib.auth.login()`. Przechodzimy więc przez kilka kolejnych cykli TDD aż do osiągnięcia kodu w pokazanej poniżej postaci.

Plik `accounts/views.py`:

```
from django.contrib.auth import authenticate, login
from django.http import HttpResponseRedirect

def persona_login(request):
    user = authenticate(assertion=request.POST['assertion'])
    if user:
        login(request, user)
    return HttpResponseRedirect('OK')

...
OK
```

W tym momencie mamy działający widok logowania.

⁴ <https://docs.djangoproject.com/en/1.7/topics/http/sessions/>

Użycie imitacji do testowania logowania

Alternatywny sposób sprawdzenia poprawności działania funkcji logowania będzie obejmować użycie imitacji.

Plik `accounts/tests/test_views.py`:

```
from django.http import HttpRequest
from accounts.views import persona_login
[...]
@patch('accounts.views.login')
@patch('accounts.views.authenticate')
def test_calls_auth_login_if_authenticate_returns_a_user(
    self, mock_authenticate, mock_login
):
    request = HttpRequest()
    request.POST['assertion'] = 'asserted'
    mock_user = mock_authenticate.return_value
    login(request)
    mock_login.assert_called_once_with(request, mock_user)
```

Plusem tej wersji testu jest to, że nie musi polegać na magii klienta testowego Django oraz nie musi mieć żadnej wiedzy dotyczącej sposobu działania sesji Django. Musisz jedynie znać nazwę funkcji, którą chcesz wywołać.

Z kolei wadą powyższego rozwiązania jest przeprowadzanie testowania implementacji za pomocą sposobu działania funkcji. Wynika to ze ścisłego powiązania z określona nazwą funkcji logowania w Django oraz jej API.

Zmiana eksperimentalnej wersji uwierzytelniania na zwykłą — imitacja żądania internetowego

Kolejnym zadaniem jest praca nad kodem funkcji uwierzytelniania. Poniżej przedstawiono eksperimentalną wersję kodu:

```
class PersonaAuthenticationBackend(object):

    def authenticate(self, assertion):
        # Wysłanie asercji do usługi Mozilla odpowiedzialnej za weryfikację.
        data = {'assertion': assertion, 'audience': 'localhost'}
        print('sending to mozilla', data, file=sys.stderr)
        resp = requests.post('https://verifier.login.persona.org/verify', data=data)
        print('got', resp.content, file=sys.stderr)

        # Czy została udzielona odpowiedź?
        if resp.ok:
            # Przetworzenie odpowiedzi.
            verification_data = resp.json()

            # Sprawdzenie, czy asercja była prawidłowa.
            if verification_data['status'] == 'okay':
                email = verification_data['email']
                try:
                    return self.get_user(email)
                except ListUser.DoesNotExist:
                    return ListUser.objects.create(email=email)

    def get_user(self, email):
        return ListUser.objects.get(email=email)
```

Oto sposób działania powyższego kodu:

- Pobieramy asercję i wysyłamy ją do usługi Mozilli, używając do tego `requests.post`.
- Sprawdzamy kod odpowiedzi (`resp.ok`), a następnie szukamy `status=ok` w danych JSON odpowiedzi.
- Wyodrębniamy adres e-mail i dla danego adresu wyszukujemy istniejącego użytkownika lub też tworzymy nowego.

Polecenie if oznacza więcej testów

Dla tego rodzaju testów istnieje następująca reguła: każde polecenie `if` oznacza dodatkowy test i każda konstrukcja `try-except` oznacza dodatkowy test, więc powinniśmy mieć cztery testy. Rozpoczynamy od pierwszego.

Plik `accounts/tests/test_authentication.py`:

```
from unittest.mock import patch

from django.test import TestCase
from accounts.authentication import (
    PERSONA_VERIFY_URL, DOMAIN, PersonaAuthenticationBackend
)

class AuthenticateTest(TestCase):

    @patch('accounts.authentication.requests.post')
    def test_sends_assertion_to_mozilla_with_domain(self, mock_post):
        backend = PersonaAuthenticationBackend()
        backend.authenticate('an assertion')
        mock_post.assert_called_once_with(
            PERSONA_VERIFY_URL,
            data={'assertion': 'an assertion', 'audience': DOMAIN}
        )
```

W pliku `authentication.py` mamy jedynie kilka miejsc zarezerwowanych.

Plik `accounts/authentication.py`:

```
import requests

PERSONA_VERIFY_URL = 'https://verifier.login.persona.org/verify'
DOMAIN = 'localhost'

class PersonaAuthenticationBackend(object):

    def authenticate(self, assertion):
        pass
```

Na tym etapie należy wydać poniższe polecenie:

```
(virtualenv)$ pip install requests
```



Nie zapomnij o dodaniu `requests` dla pliku `requirements.txt`, w przeciwnym razie kolejne wdrożenie nie powiedzie się...

Zobaczmy, jak przebiega wykonanie testów!

```
$ python3 manage.py test accounts  
[...]  
AssertionError: Expected 'post' to be called once. Called 0 times.
```

Zaliczenie testów może nastąpić już po wykonaniu trzech kroków (upewnij się, że Testing Goat „widzi”, jak je kolejno wykonujesz!).

Plik *accounts/authentication.py*:

```
def authenticate(self, assertion):  
  
    requests.post(  
        PERSONA_VERIFY_URL,  
        data={'assertion': assertion, 'audience': DOMAIN}  
    )
```

Świetnie!

```
$ python3 manage.py test accounts  
[...]
```

Ran 5 tests in 0.023s

OK

Następnie sprawdzamy, czy funkcja `authenticate()` zwraca wartość `None`, jeżeli żądanie zawiera błąd.

Plik *accounts/tests/test_authentication.py* (ch16l020):

```
@patch('accounts.authentication.requests.post')  
def test_returns_none_if_response_errors(self, mock_post):  
    mock_post.return_value.ok = False  
    backend = PersonaAuthenticationBackend()  
  
    user = backend.authenticate('an assertion')  
    self.assertIsNone(user)
```

Otrzymujemy natychmiastowy efekt — aktualnie wartość `None` jest zwracana we wszystkich przypadkach!

Poprawki na poziomie klasy

W kolejnym kroku chcemy sprawdzić, czy odpowiedź w postaci danych JSON zawiera `status=okay`. Dodanie odpowiedniego testu oznacza powielenie kodu — zastosujmy więc regułę „do trzech razy sztuka”.

Plik *accounts/tests/test_authentication.py* (ch16l021):

```
@patch('accounts.authentication.requests.post') #❶  
class AuthenticateTest(TestCase):  
  
    def setUp(self):  
        self.backend = PersonaAuthenticationBackend() #❷  
  
    def test_sends_assertion_to_mozilla_with_domain(self, mock_post):  
        self.backend.authenticate('an assertion')  
        mock_post.assert_called_once_with(  
            PERSONA_VERIFY_URL,  
            data={'assertion': 'an assertion', 'audience': DOMAIN})
```

```

    )

def test_returns_none_if_response_errors(self, mock_post):
    mock_post.return_value.ok = False #❸
    user = self.backend.authenticate('an assertion')
    self.assertIsNone(user)

def test_returns_none_if_status_not_okay(self, mock_post):
    mock_post.return_value.json.return_value = {'status': 'not okay!'} #❹
    user = self.backend.authenticate('an assertion')
    self.assertIsNone(user)

```

- ❶ Dekorator patch można zastosować na poziomie klasy. Każda metoda testowa w tej klasie będzie miała zastosowaną poprawkę, a ponadto nastąpi wstrzyknięcie imitacji.
- ❷ Można już użyć funkcji `setUp()` w celu przygotowania wszelkich użytecznych zmiennych, które będą użyte we wszystkich testach.
- ❸❹ Teraz w poszczególnych testach nastąpi jedynie dostosowanie zmiennych konfiguracji — o ile będzie to konieczne — zamiast wczytywania dużej ilości powielonego kodu. Zastosowane tutaj rozwiążanie jest o wiele czytelniejsze.

Wszystko doskonale, a testy znów są zaliczone!

OK

Nadszedł moment na sprawdzenie pozytywnego przypadku, w którym funkcja `authenticate()` powinna zwrócić obiekt użytkownika. Oczekujemy, że test zakończy się niepowodzeniem.

Plik `accounts/tests/test_authentication.py` (ch16l022-1):

```

from django.contrib.auth import get_user_model
User = get_user_model()
[...]

def test_finds_existing_user_with_email(self, mock_post):
    mock_post.return_value.json.return_value = {'status': 'okay', 'email': 'a@b.com'}
    actual_user = User.objects.create(email='a@b.com')
    found_user = self.backend.authenticate('an assertion')
    self.assertEqual(found_user, actual_user)

```

Faktycznie, mamy test zakończony niepowodzeniem:

`AssertionError: None != <User: >`

Przystępujemy do tworzenia kodu. Pracę rozpoczynamy od implementacji „oszukującej”, w której po prostu pobieramy pierwszego użytkownika znalezioneego w bazie danych.

Plik `accounts/authentication.py` (ch16l023):

```

import requests
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

def authenticate(self, assertion):
    requests.post(
        PERSONA_VERIFY_URL,
        data={'assertion': assertion, 'audience': DOMAIN}
    )
    return User.objects.first()

```

Dzięki małemu oszustwu udaje nam się zaliczyć nowy test, żaden z pozostałych wciąż nie kończy się niepowodzeniem:

```
$ python3 manage.py test accounts  
[...]
```

Ran 8 tests in 0.030s

OK

Testy są zaliczane, ponieważ wynikiem wywołania `object.first()` jest `None`, jeśli baza danych nie zawiera żadnych użytkowników. Zmodyfikujemy testy, aby stały się nieco bardziej rzeczywiste. W tym celu upewniamy się, że baza danych zawiera przynajmniej jednego użytkownika dla wszystkich testów.

Plik `accounts/tests/test_authentication.py` (ch16l022-2):

```
def setUp(self):  
    self.backend = PersonaAuthenticationBackend()  
    user = User(email='other@user.com')  
    user.username = 'otheruser' #❶  
    user.save()
```

- ❶ Domyślnie użytkownicy Django mają atrybut `username`, który z założenia ma być unikalny. Dlatego też ta wartość jest po prostu miejscem zarezerwowanym pozwalającym na tworzenie wielu użytkowników. Na późniejszym etapie prac pozbędziemy się nazwy użytkownika i zastąpimy ją adresem e-mail, który będzie służył w charakterze klucza podstawowego.

Wynikiem są trzy niepowodzenia:

```
FAIL: test_finds_existing_user_with_email  
AssertionError: <User: otheruser> != <User: >  
[...]  
FAIL: test_returns_none_if_response_errors  
AssertionError: <User: otheruser> is not None  
[...]  
FAIL: test_returns_none_if_status_not_okay  
AssertionError: <User: otheruser> is not None
```

Rozpoczynamy od przypadków, gdy uwierzytelnienie powinno zakończyć się niepowodzeniem, czyli gdy odpowiedź zawiera błąd lub kod stanu jest inny niż okay. Przyjmujemy założenie, że na początku mamy poniższy kod.

Plik `accounts/authentication.py` (ch16l024-1):

```
def authenticate(self, assertion):  
    response = requests.post(  
        PERSONA_VERIFY_URL,  
        data={'assertion': assertion, 'audience': DOMAIN}  
    )  
    if response.json()['status'] == 'okay':  
        return User.objects.first()
```

Co nieco zaskakujące, pozwala to na zaliczenie dwóch testów, które wcześniej kończyły się niepowodzeniem:

```
AssertionError: <User: otheruser> != <User: >  
  
FAILED (failures=1)
```

Przechodzimy do ostatniego testu. Zaliczenie go wymaga pobrania odpowiedniego użytkownika. I tutaj mamy kolejne zaskakujące zaliczenie testu.

Plik `accounts/authentication.py` (ch16l024-2):

```
if response.json()['status'] == 'okay':  
    return User.objects.get(email=response.json()['email'])  
  
...  
  
OK
```

Strzeż się imitacji w porównaniach wartości boolowskich

Jak to się stało, że wykonanie `test_returns_none_if_response_errors()` nie kończy się niepowodzeniem?

Ponieważ przygotowaliśmy imitację `requests.post`, `response` jest obiektem `Mock`, który jak pamiętasz zwraca wszystkie atrybuty i właściwości podobnie jak inne obiekty `Mock`⁵. Dlatego też po wykonaniu przedstawionego poniżej polecenia:

Plik `accounts/authentication.py`:

```
if response.json()['status'] == 'okay':  
  
response tak naprawdę będzie imitacją, podobnie jak response.json() oraz response.json()  
→['status']. Odbywa się więc porównanie imitacji z ciągiem tekstowym okay, wynikiem  
porównania jest False i wartością zwrotną jest domyślnie None. Modyfikujemy nieco test, aby  
wyraźnie zdefiniować, że odpowiedź JSON będzie pustym słownikiem.
```

Plik `accounts/tests/test_authentication.py` (ch16l025):

```
def test_returns_none_if_response_errors(self, mock_post):  
    mock_post.return_value.ok = False  
    mock_post.return_value.json.return_value = {}  
    user = self.backend.authenticate('an assertion')  
    self.assertIsNone(user)
```

W ten sposób otrzymujemy:

```
if response.json()['status'] == 'okay':  
KeyError: 'status'
```

Wprowadzamy niewielką zmianę.

Plik `accounts/authentication.py` (ch16l026):

```
if response.ok and response.json()['status'] == 'okay':  
    return User.objects.get(email=response.json()['email'])
```

Testy zostają zaliczone:

```
OK
```

Doskonale! Teraz funkcja `authenticate()` działa w oczekiwany przez nas sposób.

⁵ W rzeczywistości tak się dzieje, ponieważ użyliśmy dekoratora `patch`, który zwraca `MagicMock`. To jest jeszcze inna wersja `mock`, która może zachowywać się jak słownik. Więcej informacji na ten temat znajdziesz na stronie <https://docs.python.org/3/library/unittest.mock.html#magicmock-and-magic-method-support>.

Utworzenie użytkownika, jeśli to konieczne

Kolejnym krokiem jest sprawdzenie, czy funkcja `authenticate()` otrzymuje poprawną asercję z Persony. Jednak w naszej bazie danych nie mamy rekordu użytkownika dla danej osoby, więc powinniśmy go utworzyć. Poniżej przedstawiono test dla wymienionego zadania.

Plik `accounts/tests/test_authentication.py` (ch16l027):

```
def testCreatesNewUserIfNecessaryForValidAssertion(self, mock_post):
    mock_post.return_value.json.return_value = {'status': 'okay', 'email': 'a@b.com'}
    found_user = self.backend.authenticate('an assertion')
    new_user = User.objects.get(email='a@b.com')
    self.assertEqual(found_user, new_user)
```

Operacja kończy się niepowodzeniem w kodzie aplikacji, gdy próbujemy znaleźć istniejącego użytkownika wraz z podanym adresem e-mail:

```
    return User.objects.get(email=response.json()['email'])
django.contrib.auth.models.DoesNotExist: User matching query does not exist.
```

Dodajemy więc konstrukcję `try-except`, która na początku zwraca „pustego” użytkownika.

Plik `accounts/authentication.py` (ch16l028):

```
if response.ok and response.json()['status'] == 'okay':
    try:
        return User.objects.get(email=response.json()['email'])
    except User.DoesNotExist:
        return User.objects.create()
```

Teraz działanie kodu również kończy się niepowodzeniem, ale kiedy `test` próbuje wyszukać nowego użytkownika poprzez jego adres e-mail:

```
new_user = User.objects.get(email='a@b.com')
django.contrib.auth.models.DoesNotExist: User matching query does not exist.
```

poprawka polega na przypisaniu prawidłowego adresu e-mail.

Plik `accounts/authentication.py` (ch16l029):

```
if response.ok and response.json()['status'] == 'okay':
    email = response.json()['email']
    try:
        return User.objects.get(email=email)
    except User.DoesNotExist:
        return User.objects.create(email=email)
```

Zmiana pozwala na zaliczenie testów:

```
$ python3 manage.py test accounts
[...]
Ran 9 tests in 0.019s
OK
```

Metoda `get_user()`

Kolejnym zadaniem jest opracowanie metody `get_user()` dla naszego systemu uwierzytelniania. Zadaniem metody jest pobieranie użytkownika na podstawie jego adresu e-mail lub zwrot wartości `None`, gdy nie uda się odnaleźć użytkownika. (W trakcie pisania książki to nie było zbyt dobrze udokumentowane, ale musimy zachować zgodność z interfejsem. Więcej informacji znajdziesz w *kodzie źródłowym*⁶).

⁶ <https://github.com/django/django/blob/1.6c1/django/contrib/auth/backends.py#L66>

Poniżej przedstawiono kilka testów dotyczących dwóch powyższych wymagań.

Plik *accounts/tests/test_authentication.py* (ch16l030):

```
class GetUserTest(TestCase):

    def test_gets_user_by_email(self):
        backend = PersonaAuthenticationBackend()
        other_user = User(email='other@user.com')
        other_user.username = 'otheruser'
        other_user.save()
        desired_user = User.objects.create(email='a@b.com')
        found_user = backend.get_user('a@b.com')
        self.assertEqual(found_user, desired_user)

    def test_returns_none_if_no_user_with_that_email(self):
        backend = PersonaAuthenticationBackend()
        self.assertIsNone(
            backend.get_user('a@b.com')
        )
```

Oto pierwsze niepowodzenie:

```
AttributeError: 'PersonaAuthenticationBackend' object has no attribute
'get_user'
```

Tworzymy więc miejsce zarezewowane.

Plik *accounts/authentication.py* (ch16l031):

```
class PersonaAuthenticationBackend(object):

    def authenticate(self, assertion):
        [...]

    def get_user(self):
        pass
```

Teraz otrzymujemy następujący komunikat błędu:

```
TypeError: get_user() takes 1 positional argument but 2 were given
```

Dlatego też modyfikujemy metodę `get_user()`.

Plik *accounts/authentication.py* (ch16l032):

```
def get_user(self, email):
    pass
```

Wynik modyfikacji:

```
    self.assertEqual(found_user, desired_user)
AssertionError: None != <User: >
```

Kolejna zmiana (wprowadzamy je powoli, aby sprawdzić, czy test kończy się niepowodzeniem w oczekiwany sposób).

Plik *accounts/authentication.py* (ch16l033):

```
def get_user(self, email):
    return User.objects.first()
```

Udaje się zaliczyć pierwszą asercję:

```
    self.assertEqual(found_user, desired_user)
AssertionError: <User: otheruser> != <User: >
```

Wywołujemy funkcję `get()` wraz z argumentem w postaci adresu e-mail.

Plik `accounts/authentication.py` (ch16l034):

```
def get_user(self, email):
    return User.objects.get(email=email)
```

W ten sposób zaliczamy testy.

Teraz niepowodzeniem kończy się test dla przypadku, gdy wartością zwrotną ma być `None`:

```
ERROR: test_returns_none_if_no_user_with_that_email
[...]
django.contrib.auth.models.DoesNotExist: User matching query does not exist.
```

To zmusza nas do uzupełnienia metody w przedstawiony poniżej sposób.

Plik `accounts/authentication.py` (ch16l035):

```
def get_user(self, email):
    try:
        return User.objects.get(email=email)
    except User.DoesNotExist:
        return None # ❶
```

- ❶ W tym miejscu można użyć `pass` i wówczas funkcja domyślnie zwróci wartość `None`. Ponieważ potrzebujemy funkcji zwracającej wartość `None`, lepszym rozwiązaniem jest wyraźny zwrot wymienionej wartości.

Testy zostają zaliczone:

```
OK
```

Mamy działający system uwierzytelniania!

```
$ python3 manage.py test accounts
[...]
Ran 11 tests in 0.020s
OK
```

Możemy przystąpić do zdefiniowania własnego modelu użytkownika.

Minimalny niestandardowy model użytkownika

We wbudowanym w Django modelu użytkownika przyjęto wiele założeń dotyczących tego, jakie informacje o użytkowniku mają być monitorowane, począwszy od wyraźnej rejestracji imienia i nazwiska aż po wymóg użycia nazwy użytkownika. Należą do przeciwników przechowywania informacji o użytkownikach, o ile nie ma takiej konieczności. Dlatego też model przechowujący jedynie adres e-mail to dla mnie rozwiązanie doskonałe.

Plik `accounts/tests/test_models.py`:

```
from django.test import TestCase
from django.contrib.auth import get_user_model

User = get_user_model()

class UserModelTest(TestCase):

    def test_user_is_valid_with_email_only(self):
        user = User(email='a@b.com')
        user.full_clean() # Nie powinien być zgłoszony.
```

Wynikiem jest oczekiwane niepowodzenie:

```
django.core.exceptions.ValidationError: {'username': ['This field cannot be blank.'], 'password': ['This field cannot be blank.']}
```

Hasło? Nazwa użytkownika? Co to takiego?

Plik *accounts/models.py*:

```
from django.db import models

class User(models.Model):
    email = models.EmailField()
```

Model dodajemy w pliku *settings.py* za pomocą zmiennej o nazwie `AUTH_USER_MODEL`. Przy okazji dodajemy również nowo przygotowany system uwierzytelniania.

Plik *superlists/settings.py* (ch16l039):

```
AUTH_USER_MODEL = 'accounts.User'
AUTHENTICATION_BACKENDS = (
    'accounts.authentication.PersonaAuthenticationBackend',
)
```

Teraz Django zgłasza błąd, ponieważ potrzebuje pewnych metadanych dotyczących niestandardowego modelu użytkownika:

```
AttributeError: type object 'User' has no attribute 'REQUIRED_FIELDS'
```

Och. Daj spokój, Django, to jest tylko jedna właściwość i powinieneś samodzielnie znaleźć odpowiedź na zadane pytanie. Ale dobrze, oto żądaną informację.

Plik *accounts/models.py*:

```
class User(models.Model):
    email = models.EmailField()
    REQUIRED_FIELDS = ()
```

Czy to kolejne głupie pytanie?⁷

```
AttributeError: type object 'User' has no attribute 'USERNAME_FIELD'
```

Dodajemy więc odpowiedni kod.

Plik *accounts/models.py*:

```
class User(models.Model):
    email = models.EmailField()
    REQUIRED_FIELDS = ()
    USERNAME_FIELD = 'email'
```

Kolejny błąd dotyczy bazy danych:

```
django.db.utils.OperationalError: no such table: accounts_user
```

Jak zwykle tego rodzaju błąd wymusza przeprowadzenie migracji:

```
$ python3 manage.py makemigrations
System check identified some issues:
```

```
WARNINGS:
accounts.User: (auth.W004) 'User.email' is named as the 'USERNAME_FIELD', but
```

⁷ Móglbyś w tym miejscu zapytać: skoro uważasz Django za niedopracowany framework, dlaczego nie zgłosisz błędów i nie poprosisz o ich poprawienie? To powinna być całkiem prosta poprawka. Obiecuję, że to zrobię, gdy tylko zakończę pisanie książki. Na razie muszą wystarczyć jedynie moje komentarze.

```
it is not unique.  
HINT: Ensure that your authentication backend(s) can handle non-unique  
usernames.  
Migrations for 'accounts':  
  0001_initial.py:  
    - Create model User
```

Trzymamy się tej myśli i sprawdzamy, czy testy zostaną zaliczone.

Małe rozczarowanie

W międzyczasie otrzymujemy kilka dziwnych i nieoczekiwanych niepowodzeń:

```
$ python3 manage.py test accounts  
[...]  
ERROR: test_gets_logged_in_session_if_authenticate_returns_a_user  
[...]  
ERROR: test_returns_OK_when_user_found  
[...]  
    user.save(update_fields=['last_login'])  
[...]  
ValueError: The following fields do not exist in this model or are m2m fields:  
last_login
```

Wygląda na to, że Django naciska, aby w modelu użytkownika umieścić właściwość `last_login`. No cóż, marzenie o składającym się tylko z jednej właściwości modelu użytkownika prysnęło. Mimo wszystko nadal lubię przygotowany model.

Plik `accounts/models.py`:

```
from django.db import models  
from django.utils import timezone  
  
class User(models.Model):  
    email = models.EmailField()  
    last_login = models.DateTimeField(default=timezone.now)  
    REQUIRED_FIELDS = ()  
    USERNAME_FIELD = 'email'
```

Otrzymujemy kolejny błąd bazy danych. Usuwamy migrację i przeprowadzamy ją ponownie:

```
$ rm accounts/migrations/0001_initial.py  
$ python3 manage.py makemigrations  
System check identified some issues:  
[...]  
Migrations for 'accounts':  
  0001_initial.py:  
    - Create model User
```

Wprawdzie testy zostają zaliczone, ale otrzymujemy pewne ostrzeżenia:

```
$ python3 manage.py test accounts  
[...]  
System check identified some issues:  
  
WARNINGS:  
accounts.User: (auth.W004) 'User.email' is named as the 'USERNAME_FIELD', but  
it is not unique.  
[...]
```

Ran 12 tests in 0.041s

OK

Testy jako dokumentacja

Pójdzmy na całość i zamieńmy właściwość adresu e-mail na klucz podstawowy, co pozwoli na pozbycie się automatycznie generowanej kolumny `id`.

Wprawdzie ostrzeżenie będzie prawdopodobnie wystarczającym uzasadnieniem dla wprowadzenia wspomnianej zmiany, ale lepiej jednak przygotować odpowiedni test.

Plik `accounts/tests/test_models.py` (ch16l043):

```
def test_email_is_primary_key(self):
    user = User()
    self.assertFalse(hasattr(user, 'id'))
```

Test będzie pomagał przypomnieć o użyciu adresu e-mail jako klucza podstawowego, gdy w przyszłości powrócisz do pracy nad kodem.

```
    self.assertFalse(hasattr(user, 'id'))
AssertionError: True is not false
```



Testy mogą stanowić formę dokumentacji kodu źródłowego — wyrażają wymagania stawiane poszczególnym klasom i funkcjom. Jeśli zapomnisz, dlaczego coś zostało zrobione w określony sposób, szybkie spojrzenie na testy zapewni Ci odpowiedź. Dlatego też ważne jest, aby testom nadawać jasne i opisowe nazwy metod.

Poniżej przedstawiono implementację (zobacz, co się stanie, jeśli najpierw użyjesz `unique=True`).

Plik `accounts/models.py` (ch16l044):

```
email = models.EmailField(primary_key=True)
```

Rozwiążanie działa:

```
$ python3 manage.py test accounts
[...]
Ran 13 tests in 0.021s
OK
```

Ostatnim krokiem jest usunięcie migracji w celu upewnienia się, że wszystko zostało uwzględnione:

```
$ rm accounts/migrations/0001_initial.py
$ python3 manage.py makemigrations
Migrations for 'accounts':
  0001_initial.py:
    - Create model User
```

Komunikaty ostrzeżeń nie są dłużej wyświetlane!

Użytkownicy są uwierzytelnieni

Zanim nasz model będzie można uznać za ukończony, potrzebuje jeszcze jednej właściwości. Standardowy użytkownik Django ma API oferujące wiele metod⁸. Większość z nich nie potrzebuje, ale jedna jest szczególnie użyteczna: `.is_authenticated()`.

Plik `accounts/tests/test_models.py` (ch16l045):

```
def test_is_authenticated(self):
    user = User()
    self.assertTrue(user.is_authenticated())
```

⁸ <https://docs.djangoproject.com/en/1.7/ref/contrib/auth/#methods>

Po wprowadzeniu powyższej zmiany otrzymujemy komunikat:

```
AttributeError: 'User' object has no attribute 'is_authenticated'
```

Oto kod naszego niezwykle prostego modelu użytkownika.

Plik `accounts/models.py`:

```
class User(models.Model):
    email = models.EmailField(primary_key=True)
    last_login = models.DateTimeField(default=timezone.now)
    REQUIRED_FIELDS = ()
    USERNAME_FIELD = 'email'

    def is_authenticated(self):
        return True
```

Model działa bez problemów:

```
$ python3 manage.py test accounts
[...]
Ran 14 tests in 0.021s
OK
```

Chwila prawdy — czy testy funkcjonalne zostaną zaliczone?

Sądzę, że jesteśmy gotowi do przeprowadzenia testu funkcjonalnego, w którego trakcie wykorzystamy nasz szablon. Przede wszystkim szablon powinien wyświetlać odmienne komunikaty dla użytkownika zalogowanego i niezalogowanego.

Plik `lists/templates/base.html`:

```
<nav class="navbar navbar-default" role="navigation">
    <a class="navbar-brand" href="/">Superlists</a>
    {% if user.email %}
        <a class="btn navbar-btn navbar-right" id="id_logout" href="#">Wyloguj</a>
        <span class="navbar-text navbar-right">Zalogowany jako {{ user.email }}</span>
    {% else %}
        <a class="btn navbar-btn navbar-right" id="id_login" href="#">Zaloguj</a>
    {% endif %}
</nav>
```

Uroczo. Następnie wykorzystamy różne zmienne kontekstu w celu wywołania funkcji `initialize()`.

Plik `lists/templates/base.html`:

```
<script>
/*global $, Superlists, navigator */
$(document).ready(function () {
    var user = "{{ user.email }}" || null;
    var token = "{{ csrf_token }}";
    var urls = {
        login: "{% url 'persona_login' %}",
        logout: "TODO",
    };
    Superlists.Accounts.initialize(navigator, user, token, urls);
});
</script>
```

Jaki będzie skutek wykonania testu funkcjonalnego?

```
$ python3 manage.py test functional_tests.test_login
Creating test database for alias 'default'...
[...]
Ran 1 test in 26.382s
```

OK

Wspaniale!

Do tego momentu wstrzymywałem się z przekazaniem plików do repozytorium, aby mieć pewność, że wszystko działa. Teraz można przeprowadzić serię oddzielnych operacji przekazania plików — dla widoku logowania, systemu uwierzytelniania, modelu użytkownika oraz po wykorzystaniu szablonu. Ewentualnie możesz uznać, że skoro wszystkie wprowadzone powyżej zmiany są ze sobą powiązane i nie funkcjonują oddzielnie, rozsądne będzie przeprowadzenie pojedynczej dużej operacji przekazania plików do repozytorium:

```
$ git status
$ git add .
$ git diff --staged
$ git commit -am "System uwierzytelniania oparty na Persona + własny model użytkownika."
```

Zakończenie testu funkcjonalnego, przetestowanie wylogowania

Rozbudujemy test funkcjonalny o sprawdzenie, czy zachowywany jest stan „zalogowany”. Tego nie definiujemy jedynie w kodzie JavaScript działającym po stronie klienta — serwer również musi „wiedzieć”, że użytkownik jest zalogowany, i utrzymać ten stan, nawet jeśli użytkownik odświeży stronę. Ponadto przetestujemy możliwość wylogowania się użytkownika.

Rozpoczynamy od utworzenia przedstawionego poniżej kodu.

Plik *functional_tests/test_login.py*:

```
# Odświeżenie strony. Użytkownik widzi, że to prawdziwa sesja logowania,
# a nie tylko jednorazowa operacja na danej stronie.
self.browser.refresh()
self.wait_for_element_with_id('id_logout')
navbar = self.browser.find_element_by_css_selector('.navbar')
self.assertIn('edyta@mockmyid.com', navbar.text)
```

Następnie po czterech powtórzeniach bardzo podobnego kodu i utworzeniu kilku funkcji pomocniczych otrzymujemy poniższy kod.

Plik *functional_tests/test_login.py* (ch16l050):

```
def wait_to_be_logged_in(self):
    self.wait_for_element_with_id('id_logout')
    navbar = self.browser.find_element_by_css_selector('.navbar')
    self.assertIn('edyta@mockmyid.com', navbar.text)

def wait_to_be_logged_out(self):
    self.wait_for_element_with_id('id_login')
    navbar = self.browser.find_element_by_css_selector('.navbar')
    self.assertNotIn('edyta@mockmyid.com', navbar.text)
```

Rozbudowujemy test funkcjonalny, jak pokazano w poniższym kodzie.

Plik *functional_tests/test_login.py* (ch16l049):

```
[...]
# Zamknięcie okna systemu Persona.
self.switch_to_new_window('To-Do')

# Użytkownik widzi, że jest zalogowany.
self.wait_to_be_logged_in()

# Odświeżenie strony. Użytkownik widzi, że to prawdziwa sesja logowania,
# a nie tylko jednorazowa operacja na danej stronie
self.browser.refresh()
self.wait_to_be_logged_in()

# Przerażony tą funkcją użytkownik kliką przycisk wylogowania.
self.browser.find_element_by_id('id_logout').click()
self.wait_to_be_logged_out()

# Stan "wylogowany" został zachowany także po odświeżeniu strony.
self.browser.refresh()
self.wait_to_be_logged_out()
```

Uznałem, że poprawienie komunikatu niepowodzenia w funkcji *wait_for_element_with_id()* pomaga w zrozumieniu tego, co tak naprawdę się dzieje.

Plik *functional_tests/test_login.py*:

```
def wait_for_element_with_id(self, element_id):
    WebDriverWait(self.browser, timeout=30).until(
        lambda b: b.find_element_by_id(element_id),
        'Nie można znaleźć elementu o id {}'.format(
            element_id, self.browser.find_element_by_tag_name('body').text
        )
    )
```

Po wprowadzeniu usprawnień widzimy, że przyczyną niepowodzenia testu jest niedziałający przycisk wylogowania:

```
$ python3 manage.py test functional_tests.test_login
  File "/workspace/superlists/functional_tests/test_login.py", line 36, in
    wait_to_be_logged_out
    [...]
  selenium.common.exceptions.TimeoutException: Message: 'Nie można znaleźć elementu
  o id id_login. Zawartość strony to Superlists\nWyloguj\nZalogowany jako
  edyta@mockmyid.com\nUtwórz nową listę rzeczy do zrobienia'
```

Faktyczna implementacja przycisku wylogowania jest bardzo prosta. Możemy wykorzystać wbudowany w Django *widok wylogowania*⁹, który powoduje usunięcie sesji użytkownika i przekierowanie go na wskazaną stronę.

Plik *accounts/urls.py*:

```
urlpatterns = patterns '',
    url(r'^login$', 'accounts.views.persona_login', name='persona_login'),
    url(r'^logout$', 'django.contrib.auth.views.logout', {'next_page': '/'}, name='logout'),
)
```

⁹ <https://docs.djangoproject.com/en/1.7/topics/auth/default/#module-django.contrib.auth.views>

W szablonie *base.html* zmieniamy przycisk wylogowania na zwykłe łącze URL.

Plik *lists/templates/base.html*:

```
<a class="btn navbar-btn navbar-right" id="id_logout" href="{% url 'logout' %}">Wyloguj </a>
```

W ten sposób testy funkcjonalne zostają zaliczone. Tak naprawdę zaliczony jest pełny zestaw testów:

```
$ python3 manage.py test functional_tests.test_login
[...]
OK
$ python3 manage.py test
[...]
Ran 54 tests in 78.124s
```

OK

W kolejnym rozdziale zaczniemy wykorzystywać przygotowany system logowania. Wcześniej przekaż pliki do repozytorium i przypomnij sobie pewne pojęcia.

Imitacje w Pythonie

Biblioteka imitacji

Michael Foord (pracował dla firmy, która utworzyła PythonAnywhere, zanim do niej dołączyłem) opracował doskonałą bibliotekę „imitacji”. Obecnie ta biblioteka została zintegrowana ze standardową biblioteką Pythona 3. We wspomnianej bibliotece imitacji znajdziesz wszystko, co może być potrzebne podczas stosowania imitacji w Pythonie.

Dekorator patch

Moduł `unittest.mock` oferuje funkcję o nazwie `patch()` przeznaczoną do „imitowania” dowolnego obiektu z testowanego modułu. Ta funkcja jest najczęściej stosowana jako dekorator dla metody testowej lub nawet na poziomie klasy — wówczas ma zastosowanie we wszystkich metodach danej klasy.

Imitacje mają wartość true i mogą maskować błędy

Pamiętaj, że imitowane komponenty mogą zachowywać się odmiennie w poleceniach `if`. Imitacje mają wartość `true`, a ponadto mogą maskować błędy, ponieważ zawierają wszystkie atrybuty i metody.

Zbyt duża liczba imitacji to zapach kodu

Zbyt duża liczba imitacji w testach oznacza bardzo ścisłe powiązanie z ich implementacjami. Czasami nie da się tego uniknąć. Ogólnie rzecz biorąc, staraj się zorganizować kod w taki sposób, aby nie istniała konieczność stosowania zbyt wielu imitacji.

Konfiguracja testu, rejestracja i debugowanie po stronie serwera

Mamy już funkcjonujący system uwierzytelniania i chcemy go wykorzystać do identyfikacji użytkowników, aby każdy z nich mógł wyświetlać utworzone przez siebie listy rzeczy do zrobienia.

Konieczne jest opracowanie testów funkcjonalnych potwierdzających zalogowanie użytkownika. Zamiast definiować przejście testu przez wszystkie elementy okna dialogowego Persona (co może być niezwykle czasochłonne), decydujemy się na pominięcie tej części.

W tym momencie na myśl powinna przyjść separacja zadań. Testy funkcjonalne nie są jak testy jednostkowe, czyli najczęściej nie wymagają użycia pojedynczej asercji. Jednak pod względem koncepcyjnym powinny być przeznaczone do testowania pojedynczej funkcjonalności. Nie trzeba tworzyć oddzielnych testów funkcjonalnych do przetestowania mechanizmów logowania i wylogowania użytkownika. Jeżeli uda się znaleźć sposób „oszukania” i pominięcia tej części, wówczas mniej czasu zajmie oczekiwanie na wykonanie powielonych ścieżek testów.



Nie przesadzaj z unikaniem powielonego kodu w testach funkcjonalnych. Jedną z zalet testu funkcjonalnego jest możliwość przechwycenia dziwnych i nieprzewidywalnych interakcji zachodzących między różnymi komponentami aplikacji.

Pominięcie procesu logowania przez wstępne utworzenie sesji

Bardzo często zdarza się, że użytkownik powraca do witryny internetowej i nadal ma ustawione cookie, czyli jest „wstępnie uwierzytelniony”. Tak więc wspomniane „oszustwo” nie przedstawia wcale tak nierzeczywistej sytuacji. Poniżej przedstawiono sposób przygotowania testów.

Plik `functional_tests/test_my_lists.py`:

```
from django.conf import settings
from django.contrib.auth import BACKEND_SESSION_KEY, SESSION_KEY, get_user_model
User = get_user_model()
from django.contrib.sessions.backends.db import SessionStore

from .base import FunctionalTest

class MyListsTest(FunctionalTest):

    def create_pre_authenticated_session(self, email):
        user = User.objects.create(email=email)
        session = SessionStore()
        session[SESSION_KEY] = user.pk #❶
        session[BACKEND_SESSION_KEY] = settings.AUTHENTICATION_BACKENDS[0]
        session.save()
        ## W celu ustawienia cookie najpierw trzeba odwiedzić witrynę.
        ## Strony o kodzie błędu 404 są wczytywane najszybciej!
        self.browser.get(self.server_url + "/404_no_such_url/")
        self.browser.add_cookie(dict(
            name=settings.SESSION_COOKIE_NAME,
            value=session.session_key, #❷
            path='/',
        ))
```

- ❶ W bazie danych tworzymy obiekt sesji. Klucz sesji jest kluczem podstawowym obiektu użytkownika (którym w rzeczywistości jest jego adres e-mail).
- ❷ Następnie do przeglądarki internetowej dodajemy cookie dopasowane do sesji w serwerze. W trakcie kolejnych odwiedzin witryny serwer powinien rozpoznać odwiedzającego jako zalogowanego użytkownika.

Zwrót uwagę, że przedstawione rozwiązanie może działać tylko dlatego, że używamy `LiveServerTestCase`, stąd obiekty `User` i `Session` są tworzone w tej samej bazie danych, która jest używana przez serwer testowy. Później zmodyfikujemy nieco rozwiązanie, aby działało także z bazą danych w serwerze prowizorycznym.

Oparta na JSON konfiguracja testu jest uznawana za nieszkodliwą

Kiedy w bazie danych wstępnie umieszczamy dane testowe, jak w omawianym przykładzie za pomocą obiektu `User` i powiązanego z nim obiektu `Session`, jest to czynność określana mianem „konfiguracja testu”.

Django oferuje wbudowaną możliwość zapisu obiektów bazy danych jako JSON (za pomocą `manage.py dumpdata`) oraz ich automatyczne wczytywanie w testach za pomocą atrybutu `fixtures` klas `TestCase`.

Coraz więcej osób uważa, że nie należy używać *konfiguracji testu opartej na JSON*¹. To staje się koszmarem w przypadku zmiany modelu. Zamiast tego dane można wczytywać bezpośrednio za pomocą Django ORM lub wykorzystać narzędzie takie jak `factory_boy`².

¹ <http://blog.muhuk.com/2012/04/09/carl-meyers-testing-talk-at-pycon-2012.html#.U5XCBpRdXcQ>

² <https://factoryboy.readthedocs.org/en/latest/>

Sprawdzamy rozwiązanie

Aby sprawdzić działanie rozwiązania, można użyć funkcji `wait_to_be_logged_in()` zdefiniowanej w poprzednim kroku. W celu uzyskania dostępu do wymienionej funkcji z poziomu innego testu oraz metod należy ją umieścić w `FunctionalTest`. Przy okazji nieco ją zmodyfikujemy, umożliwiając obsługę parametru przekazującego funkcji dowolny adres e-mail.

Plik `functional_tests/base.py` (ch17l002-2):

```
from selenium.webdriver.support.ui import WebDriverWait
[...]

class FunctionalTest(StaticLiveServerCase):
    [...]

    def wait_for_element_with_id(self, element_id):
        [...]

    def wait_to_be_logged_in(self, email):
        self.wait_for_element_with_id('id_logout')
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertIn(email, navbar.text)

    def wait_to_be_logged_out(self, email):
        self.wait_for_element_with_id('id_login')
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertNotIn(email, navbar.text)
```

To oznacza konieczność wprowadzenia zmian także w pliku `test_login.py`.

Plik `functional_tests/test_login.py` (ch17l003):

```
TEST_EMAIL = 'edyta@mockmyid.com'
[...]

def test_login_with_persona(self):
    [...]

    self.browser.find_element_by_id(
        'authentication_email'
    ).send_keys(TEST_EMAIL)
    self.browser.find_element_by_tag_name('button').click()

    [...]

    # Użytkownik widzi, że został zalogowany.
    self.wait_to_be_logged_in(email=TEST_EMAIL)
    [...]
    self.wait_to_be_logged_in(email=TEST_EMAIL)
    [...]
    self.wait_to_be_logged_out(email=TEST_EMAIL)
    [...]
    self.wait_to_be_logged_out(email=TEST_EMAIL)
```

Sprawdzimy teraz, czy przypadkiem nie uszkodziliśmy jakiejkolwiek funkcjonalności. Ponownie wykonujemy więc test logowania:

```
$ python3 manage.py test functional_tests.test_login
[...]
OK
```

W tym momencie można utworzyć miejsce zarezerwowane dla testu `test_my_lists()` i tym samym przekonać się, czy wstępne uwierzytelnienie naprawdę działa.

Plik `functional_tests/test_my_lists.py` (ch17l004):

```
def test_logged_in_users_lists_are_saved_as_my_lists(self):
    email = 'edyta@przyklad.pl'

    self.browser.get(self.server_url)
    self.wait_to_be_logged_out(email)

    # Edyta jest zalogowanym użytkownikiem.
    self.create_pre_authenticated_session(email)

    self.browser.get(self.server_url)
    self.wait_to_be_logged_in(email)
```

Otrzymujemy następujące dane wyjściowe:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
OK
```

To doskonała chwila na przekazanie plików do repozytorium:

```
$ git add functional_tests
$ git commit -m "Miejsce zarezerwowane test_my_lists i przeniesienie operacji sprawdzenia logowania."
```

Dowód znajdziesz w praktyce — użycie wersji prowizorycznej do wychwycenia błędów

Wszystko sprawdza się doskonale podczas lokalnego wykonywania testów funkcjonalnych, ale jak to będzie wyglądać w przypadku serwera prowizorycznego? Wychwycimy nieoczekiwany błąd (przecież do tego celu jest przeznaczony serwer prowizoryczny!), a następnie będziemy musieli ustalić, w jaki sposób zarządzać bazą danych w serwerze testowym.

```
$ cd deploy_tools
$ fab deploy --host=superlists-staging.ottg.eu
[...]
```

Ponownie uruchamiamy Gunicorn:

```
edyta@serwer: sudo restart gunicorn-superlists-staging.ottg.eu
```

Oto co się stanie po uruchomieniu testów funkcjonalnych:

```
$ python3 manage.py test functional_tests \
--liveserver=superlists-staging.ottg.eu
=====
ERROR: test_login_with_persona (functional_tests.test_login.LoginTest)
-----
Traceback (most recent call last):
  File "/workspace/functional_tests/test_login.py", line 50, in
    test_login_with_persona
  [...]
      self.wait_for_element_with_id('id_logout')
  [...]
selenium.common.exceptions.TimeoutException: Komunikat: 'Nie można znaleźć elementu o id id_logout. Zawartość strony to Superlists\nZaloguj\nUtwórz nową listę rzeczy do zrobienia'
```

```
=====
ERROR: test_logged_in_users_lists_are_saved_as_my_lists
(functional_tests.test_my_lists.MyListsTest)
-----
Traceback (most recent call last):
  File "/workspace/functional_tests/test_my_lists.py", line 34, in
    test_logged_in_users_lists_are_saved_as_my_lists
      self.wait_to_be_logged_in(email)
  [...]
  selenium.common.exceptions.TimeoutException: Komunikat: 'Nie można znaleźć elementu
o id id_logout. Zawartość strony to Superlists\nZaloguj\nUtwórz nową listę rzeczy do
zrobienia'
```

Nie można się zalogować — ani za pomocą rzeczywistej sesji Persona, ani za pomocą wstępnie uwierzytelnionej. Mamy do czynienia z pewnego rodzaju błędem.

Rozważyłem już możliwość cofnięcia się do poprzedniego rozdziału, poprawienia go i udowadniania, że nic się nie stało. Uznałem jednak, że pozostawienie tego błędu doskonale zilustruje potrzebę wykonywania testów względem środowiska prowizorycznego. Wprowadzenie tego rodzaju błędu w witrynie udostępnianej użytkownikom byłoby niezwykle kłopotliwe.

Pomijając powyższe, nabędziesz praktykę w usuwaniu błędów w kodzie działającym po stronie serwera.

Konfiguracja rejestracji danych

W celu wyśledzenia pojawiającego się błędu musimy skonfigurować Gunicorn do rejestracji danych. Za pomocą vi lub nano zmodyfikuj w serwerze konfigurację Gunicorn.

Plik `/etc/init/gunicorn-superlists-staging.ottg.eu.conf` w serwerze:

```
[...]
exec ../virtualenv/bin/gunicorn \
  --bind unix:/tmp/superlists-staging.ottg.eu.socket \
  --access-logfile ../access.log \
  --error-logfile ../error.log \
  superlists.wsgi:application
```

Powyższa konfiguracja powoduje, że dane o dostępie i błędach są umieszczone w katalogu `~/sites/$NAZWAWITRYNY`. Następnie w funkcji `authenticate()` dodamy pewne wywołania pomagające w usuwaniu błędów (pamiętaj, że robimy to w serwerze).

Plik `accounts/authentication.py`:

```
def authenticate(self, assertion):
    logging.warning('wejście do funkcji authenticate')
    response = requests.post(
        PERSONA_VERIFY_URL,
        data={'assertion': assertion, 'audience': settings.DOMAIN})
    logging.warning('otrzymano odpowiedź od persona')
    logging.warning(response.content.decode())
[...]
```



Użycie rejestracji danych na „najwyższym poziomie” — `logging.warning()` — jest ogólnie rzeczą biorąc, nie najlepszym pomysłem. Na końcu rozdziału opracujemy znacznie solidniejszą konfigurację rejestracji danych.

Ponadto trzeba się upewnić, że w pliku *settings.py* nadal znajduje się ustawienie LOGGING odpowiedzialne za faktyczne wysyłanie danych do konsoli.

Plik *superlists/settings.py*:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
        },
    },
    'root': {'level': 'INFO'},
}
```

Ponownie uruchamiamy serwer Gunicorn i wykonujemy test funkcjonalny lub próbujemy ręcznie się zalogować. Następnie za pomocą poniższego polecenia możemy obserwować dzienniki zdarzeń serwera:

```
edyta@serwer: $ tail -f error.log # O ile znajdujesz się w katalogu ~/sites/$NAZWA_WITRYNY.
[...]
WARNING:root:{"status":"failure","reason":"audience mismatch: domain mismatch"}
```

Możesz nawet odkryć, że strona została uwięziona w „pętli przekierowania”, ponieważ Persona nieustannie próbuje ponownie wysłać asercję.

Okazało się, że przeoczyłem ważną część systemu Persona — uwierzytelnianie pozostaje ważne jedynie dla określonych domen. W pliku *accounts/authentication.py* pozostawiliśmy na stałe zdefiniowaną domenę *localhost*.

Plik *accounts/authentication.py*:

```
PERSONA_VERIFY_URL = 'https://verifier.login.persona.org/verify'
DOMAIN = 'localhost'
User = get_user_model()
```

To można zmienić w serwerze.

Plik *accounts/authentication.py*:

```
DOMAIN = 'superlists-staging.ottg.eu'
```

I sprawdzić, czy działa, przeprowadzając w tym celu ręczne logowanie. Działa.

Usunięcie błędu systemu Persona

Poniżej przedstawiono procedurę usunięcia błędu przez poprawę kodu w komputerze lokalnym. Rozpoczynamy od przeniesienia definicji zmiennej DOMAIN do pliku *settings.py*, w którym będziemy mogli później użyć skryptu wdrożenia i nadpisać ustawienia.

Plik *superlists/settings.py* (ch17l011):

```
# Poniższe ustawienie będzie później nadpisane przez skrypt wdrożenia.
DOMAIN = "localhost"

ALLOWED_HOSTS = [DOMAIN]
```

Po wprowadzeniu zmiany wykonujemy testy.

Plik *accounts/tests/test_authentication.py*:

```
@@ -1,12 +1,14 @@
 from unittest.mock import patch
+from django.conf import settings
 from django.contrib.auth import get_user_model
 from django.test import TestCase
 User = get_user_model()

 from accounts.authentication import (
-    PERSONA_VERIFY_URL, DOMAIN, PersonaAuthenticationBackend
+    PERSONA_VERIFY_URL, PersonaAuthenticationBackend
)

+
 @patch('accounts.authentication.requests.post')
 class AuthenticateTest(TestCase):
@@ -21,7 +23,7 @@ class AuthenticateTest(TestCase):
     self.backend.authenticate('an assertion')
     mock_post.assert_called_once_with(
         PERSONA_VERIFY_URL,
-        data={'assertion': 'an assertion', 'audience': DOMAIN}
+        data={'assertion': 'an assertion', 'audience': settings.DOMAIN}
     )

```

Następnie zmieniamy implementację.

Plik *accounts/authentication.py*:

```
@@ -1,8 +1,8 @@
 import requests
+from django.conf import settings
 from django.contrib.auth import get_user_model
 User = get_user_model()

 PERSONA_VERIFY_URL = 'https://verifier.login.persona.org/verify'
-DOMAIN = 'localhost'

@@ -11,7 +11,7 @@ class PersonaAuthenticationBackend(object):
     def authenticate(self, assertion):
         response = requests.post(
             PERSONA_VERIFY_URL,
-            data={'assertion': assertion, 'audience': DOMAIN}
+            data={'assertion': assertion, 'audience': settings.DOMAIN}
         )
         if response.ok and response.json()['status'] == 'okay':
             email = response.json()['email']
```

Ponownie wykonujemy testy, tak dla pewności:

```
$ python3 manage.py test accounts
[...]
Ran 14 tests in 0.053s
OK
```

Kolejnym krokiem jest uaktualnienie pliku *fabfile.py* i modyfikacja ustawień domeny w *settings.py* oraz usunięcie dwóch wierszy sed w *ALLOWED_HOSTS*.

Plik *deploy_tools/fabfile.py*:

```
def _update_settings(source_folder, site_name):
    settings_path = source_folder + '/superlists/settings.py'
    sed(settings_path, "DEBUG = True", "DEBUG = False")
```

```
sed(settings_path, 'DOMAIN = "localhost"', 'DOMAIN = "%s"' % (site_name,))
secret_key_file = source_folder + '/superlists/secret_key.py'
if not exists(secret_key_file):
    [...]
```

Ponownie przeprowadzamy wdrożenie i zauważamy sed w danych wyjściowych.

```
$ fab deploy --host=superlists-staging.ottg.eu
[...]
[superlists-staging.ottg.eu] run: sed -i.bak -r -e s/DOMAIN =
"localhost"/DOMAIN = "superlists-staging.ottg.eu"/g "$(echo
/home/harry/sites/superlists-staging.ottg.eu/source/superlists/settings.py)"
[...]
```

Zarządzanie testową bazą danych w serwerze prowizorycznym

Teraz ponownie wykonujemy testy funkcjonalne i otrzymujemy kolejne niepowodzenie: próba utworzenia wstępnie uwierzytelnionej sesji nie działa, a więc test `test_my_list()` kończy się niepowodzeniem:

```
$ python3 manage.py test functional_tests \
--liveserver=superlists-staging.ottg.eu

ERROR: test_logged_in_users_lists_are_saved_as_my_lists
(functional_tests.test_my_lists.MyListsTest)
[...]
selenium.common.exceptions.TimeoutException: Komunikat: 'Nie można znaleźć elementu
o id id_logout. Zawartość strony to Superlists\nZaloguj\nUtwórz nową listę rzeczy do
zrobienia'

Ran 7 tests in 72.742s

FAILED (errors=1)
```

Funkcja narzędziowa `create_pre_authenticated_session()` działa jedynie w lokalnej bazie danych, stąd niepowodzenie testu. Zobaczmy, jak testy mogą zarządzać bazą danych w serwerze.

Polecenie Django służące do tworzenia sesji

W celu wykonywania różnych zadań w serwerze konieczne jest opracowanie samodzielnego skryptu, który można uruchamiać z poziomu powłoki w serwerze, prawdopodobnie za pomocą Fabric.

Podczas tworzenia samodzielnego skryptu działającego w środowisku Django i potrafiącego współpracować z bazą danych konieczne będzie rozwiązywanie pewnych kwestii, takich jak poprawne ustawienie zmiennej środowiskowej `DJANGO_SETTINGS_MODULE` i poprawne pobranie `sys.path`. Zamiast mieszać we wspomnianych ustawieniach, Django pozwala na utworzenie własnych „narzędzi zarządzania” (poleceń wydawanych za pomocą `python manage.py`), które zajmą się wprowadzaniem odpowiednich zmian. Wspomniane narzędzia znajdują się w katalogu `management/commands` w aplikacji:

```
$ mkdir -p functional_tests/management/commands
$ touch functional_tests/management/__init__.py
$ touch functional_tests/management/commands/__init__.py
```

Szkieletem dla narzędzia zarządzania jest klasa dziedzicząca po `django.core.management.BaseCommand` i definiująca metodę o nazwie `handle()`.

Plik `functional_tests/management/commands/create_session.py`:

```
from django.conf import settings
from django.contrib.auth import BACKEND_SESSION_KEY, SESSION_KEY, get_user_model
User = get_user_model()
from django.contrib.sessions.backends.db import SessionStore
from django.core.management.base import BaseCommand

class Command(BaseCommand):

    def handle(self, email, *_, **__):
        session_key = create_pre_authenticated_session(email)
        self.stdout.write(session_key)

    def create_pre_authenticated_session(email):
        user = User.objects.create(email=email)
        session = SessionStore()
        session[SESSION_KEY] = user.pk
        session[BACKEND_SESSION_KEY] = settings.AUTHENTICATION_BACKENDS[0]
        session.save()
        return session.session_key
```

Kod funkcji `create_pre_authenticated_session()` został wzięty z pliku `test_my_lists.py`. Funkcja `handle()` pobiera adres e-mail podany jako pierwszy argument, a zwraca klucz sesji, który ma być dodany do cookies w przeglądarce internetowej. Narzędzie zarządzania wyświetla polecenie w powłoce; wypróbuj:

```
$ python3 manage.py create_session a@b.com
Unknown command: 'create_session'
```

Jeszcze jeden krok. Konieczne jest dodanie `functional_tests` do pliku `settings.py`, aby wyraźnie wskazać, że to rzeczywista aplikacja, która może zawierać narzędzia zarządzające oraz testy.

Plik `superlists/settings.py`:

```
+++ b/superlists/settings.py
@@ -42,6 +42,7 @@ INSTALLED_APPS = (
    'lists',
    'south',
    'accounts',
+   'functional_tests',
)
```

Teraz wszystko działa:

```
$ python3 manage.py create_session a@b.com
qns1ckvp2aga7tm6xuivyb0ob1akzzwl
```

Test funkcjonalny uruchamiający w serwerze narzędzie zarządzania

Następnym krokiem jest modyfikacja testu `test_my_lists()`, aby w serwerze lokalnym wywoływał funkcję lokalną, natomiast w serwerze prowizorycznym uruchamiał narzędzie zarządzania.

Plik *functional_tests/test_my_lists.py* (ch17l016):

```
from django.conf import settings
from .base import FunctionalTest
from .server_tools import create_session_on_server
from .management.commands.create_session import create_pre_authenticated_session

class MyListsTest(FunctionalTest):
    def create_pre_authenticated_session(self, email):
        if self.against_staging:
            session_key = create_session_on_server(self.server_host, email)
        else:
            session_key = create_pre_authenticated_session(email)
        ## W celu ustawienia cookie najpierw trzeba odwiedzić witrynę.
        ## Strony o kodzie błędu 404 są wczytywane najszybciej!
        self.browser.get(self.server_url + "/404_no_such_url/")
        self.browser.add_cookie(dict(
            name=settings.SESSION_COOKIE_NAME,
            value=session_key,
            path='/',
        ))
    [...]
```

Zobaczmy, jak sprawdzić, kiedy kod jest wykonywany w serwerze prowizorycznym. Atrybut `self.against_staging` jest ustawiany w pliku *base.py*.

Plik *functional_tests/base.py* (ch17l017):

```
from .server_tools import reset_database #❶
class FunctionalTest(StaticLiveServerCase):

    @classmethod
    def setUpClass(cls):
        for arg in sys.argv:
            if 'liveserver' in arg:
                cls.server_host = arg.split('=')[1] #❷
                cls.server_url = 'http://' + cls.server_host
                cls.against_staging = True #❸
                return
        super().setUpClass()
        cls.against_staging = False
        cls.server_url = cls.live_server_url

    @classmethod
    def tearDownClass(cls):
        if not cls.against_staging:
            super().tearDownClass()

    def setUp(self):
        if self.against_staging:
            reset_database(self.server_host) #❹
            self.browser = webdriver.Firefox()
            self.browser.implicitly_wait(3)
```

❸❹ Zamiast przechowywać po prostu `cls.server_url`, przechowujemy również atrybuty `server_host` i `against_staging` po wykryciu argumentu `liveserver` w powłoce.

❻❺ Konieczny jest sposób pozwalający na wyzerowanie serwerowej bazy danych między poszczególnymi testami. Wkrótce wyjaśnię logikę kodu tworzącego sesję, co jednocześnie powinno wyjaśnić przedstawione rozwiązanie.

Dodatkowy krok za pomocą modułu subprocess

Ponieważ nasze testy są przeznaczone dla Pythona 3, to nie możemy bezpośrednio wywoływać funkcji narzędzia Fabric, które jest przeznaczone dla Pythona 2. Zamiast tego trzeba wykonać dodatkowy krok i wywołać polecenie fab jako nowy proces, podobnie jak ma to miejsce w po-włoce podczas przeprowadzania wdrożenia serwera. Poniżej przedstawiono odpowiedni kod umieszczony w module o nazwie `server_tools`.

Plik `functional_tests/server_tools.py`:

```
from os import path
import subprocess
THIS_FOLDER = path.abspath(path.dirname(__file__))

def create_session_on_server(host, email):
    return subprocess.check_output(
        [
            'fab',
            'create_session_on_server:email={}'.format(email), #❶❷
            '--host={}'.format(host),
            '--hide=everything,status', #❸
        ],
        cwd=THIS_FOLDER
    ).decode().strip() #❹

def reset_database(host):
    subprocess.check_call(
        ['fab', 'reset_database', '--host={}'.format(host)],
        cwd=THIS_FOLDER
    )
```

Oto omówienie użycia modułu `subprocess` do wywołania pewnych funkcji Fabric za pomocą polecenia `fab`.

- ❶ Jak możesz zobaczyć, składnia powłoki dla argumentów funkcji `fab` jest całkiem prosta — dwukropek, a następnie zmienna i argument.
- ❷ Nawiąsem mówiąc, to jest pierwsze wystąpienie „nowego stylu” składni formatowania ciągu tekstuowego. Zamiast `%s` użyto nawiasu klamrowego. Osobiście wolę stary styl, ale powinieneś poznać oba, jeśli już od jakiegoś czasu zajmujesz się programowaniem w Pythonie. Jeżeli chcesz się dowiedzieć więcej na temat nowego stylu składni, spójrz na przykłady zaprezentowane w *dokumentacji Pythona*³.
- ❸❹ Z powodu komunikacji między Fabric i subprocessami jesteśmy zmuszeni do zachowania szczególnej ostrożności podczas wyodrębniania ciągu tekstuowego klucza sesji z danych wyjściowych polecenia, ponieważ jest ono wykonywane w serwerze.

Jeżeli używasz własnej nazwy użytkownika i hasła, może wystąpić konieczność modyfikacji wywołania do modułu `subprocess`. Kiedy korzystasz ze zautomatyzowanego skryptu wdrożenia, upewnij się o dopasowaniu wymienionych danych do argumentów polecenia `fab`.



Gdy czytasz tę książkę, narzędzie Fabric może być już w pełni dostosowane do Pythona 3. W takim przypadku przeczytaj o użyciu menedżerów kontekstu Fabric w celu bezpośredniego wywoływania funkcji Fabric w kodzie aplikacji.

³ <https://docs.python.org/3/library/string.html#format-examples>

Na koniec spójrz na plik *fabfile.py* definiujący dwa polecenia, które chcemy wykonywać po stronie serwera. Pierwsze służy do zerowania bazy danych, natomiast drugie do konfiguracji sesji.

Plik *functional_tests/fabfile.py*:

```
from fabric.api import env, run

def _get_base_folder(host):
    return '~/sites/' + host

def _get_manage_dot_py(host):
    return '{path}/virtualenv/bin/python {path}/source/manage.py'.format(
        path=_get_base_folder(host)
    )

def reset_database():
    run('{manage_py} flush --noinput'.format(
        manage_py=_get_manage_dot_py(env.host)
    ))

def create_session_on_server(email):
    session_key = run('{manage_py} create_session {email}'.format(
        manage_py=_get_manage_dot_py(env.host),
        email=email,
    ))
    print(session_key)
```

Czy przedstawione rozwiązanie ma sens? Mamy funkcję przeznaczoną do tworzenia sesji w bazie danych. Jeżeli kod jest wykonywany lokalnie, wówczas wspomniana funkcja jest wywoływana bezpośrednio. Natomiast jeśli kod jest wykonywany w serwerze, wtedy mamy kilka przeskóków: używamy modułu subprocess, aby uzyskać dostęp do narzędzia Fabric za pomocą polecenia fab, co pozwala na wydanie polecień zarządzających, które z kolei wywołują tę samą funkcję w serwerze.

Jak to można zilustrować za pomocą stylu ASCII-Art?

W przypadku lokalnego wykonania kodu mamy:

```
MyListsTest
.create_pre_authenticated_session --> .management.commands.create_session
                                         .create_pre_authenticated_session
```

W przypadku wykonania kodu w serwerze prowizorycznym:

```
MyListsTest
.create_pre_authenticated_session      .management.commands.create_session
                                         .create_pre_authenticated_session
                                         |
                                         \|/
                                         server_tools
                                         .create_session_on_server      run manage.py create_session
                                         |
                                         \||/
                                         subprocess.check_output -->  fab -->  fabfile.create_session_on_server
```

Nieważne. Zobaczmy, czy przedstawione rozwiązanie działa. Najpierw sprawdzamy, czy na pewno nie uszkodziliśmy żadnej funkcjonalności:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
OK
```

Kolejnym krokiem jest sprawdzenie po stronie serwera. W pierwszej kolejności trzeba przekazać kod:

```
$ git push # Najpierw trzeba przekazać zmiany do repozytorium.  
$ cd deploy_tools  
$ fab deploy --host=superlists-staging.ottg.eu
```

Teraz wykonujemy test — zwróć uwagę na możliwość użycia edyta@ w specyfikacji argumentu liveserver:

```
$ python3 manage.py test functional_tests.test_my_lists \  
--liveserver=edyta@superlists-staging.ottg.eu  
Creating test database for alias 'default'...  
[superlists-staging.ottg.eu] Executing task 'reset_database'  
~/sites/superlists-staging.ottg.eu/source/manage.py flush --noinput  
[superlists-staging.ottg.eu] out: Syncing...  
[superlists-staging.ottg.eu] out: Creating tables ...  
[...]  
.  
-----  
Ran 1 test in 25.701s
```

OK

Wszystko wygląda dobrze! Możemy ponownie wykonać wszystkie testy, aby mieć większą pewność...

```
$ python3 manage.py test functional_tests \  
--liveserver=edyta@superlists-staging.ottg.eu  
Creating test database for alias 'default'...  
[superlists-staging.ottg.eu] Executing task 'reset_database'  
[...]  
Ran 7 tests in 89.494s  
OK  
Destroying test database for alias 'default'...
```

Hura!



Pokazałem jeden ze sposobów zarządzania testową bazą danych. Możesz eksperymentować z innymi, na przykład jeśli używasz MySQL lub PostgreSQL, wówczas nawiąż połączenie z serwerem przez SSH i prowadź bezpośrednią komunikację z bazą danych, wykorzystując przekierowanie portu. Następnie możesz poprawić settings.DATABASES podczas testu funkcjonalnego i pracować z tunelowanym portem.

Ostrzeżenie — zachowaj ostrożność i nie wykonuj testu względem serwera produkcyjnego

Weszliśmy na niebezpieczny obszar i teraz mamy kod, który może mieć bezpośredni wpływ na bazę danych znajdującej się w serwerze. Musisz być wyjątkowo ostrożny, aby przypadkiem nie usunąć produkcyjnej bazy danych przez wykonanie testów funkcjonalnych względem niewłaściwego hosta.

Na tym etapie zdecyduj się na wprowadzenie pewnych środków ostrożności. Na przykład wersje prowizoryczną i produkcyjną umieść w oddzielnych serwerach i upewnij się, że do uwierzytelnienia używają innej pary kluczowej z innymi hasłami.

Niebezpieczeństwo jest podobne jak podczas wykonywania testów względem klonowanych danych produkcyjnych. Przypomnij sobie moją krótką historię o przypadkowym wysłaniu tysiące powielonych faktur do klientów. Ucz się na moich błędach.

Zachowanie kodu odpowiedzialnego za rejestrację danych

Zanim zakończymy pracę, warto zachować kod odpowiedzialny za rejestrację danych. Przygotowany tutaj kod rejestracji danych dobrze byłoby zachować w systemie kontroli wersji, aby móc go wykorzystać w przyszłości podczas ewentualnego usuwania błędów w funkcji logowania. Problemy z logowaniem mogą wskazywać, że dzieje się coś niedobrego.

Rozpoczynamy od zapisania konfiguracji Gunicorn w pliku szablonu w module `deploy_tools`.

Plik `deploy_tools/gunicorn-upstart.template.conf`:

```
[...]
chdir /home/elspeth/sites/SITENAME/source

exec ../virtualenv/bin/gunicorn \
    --bind unix:/tmp/SITENAME.socket \
    --access-logfile ../access.log \
    --error-logfile ../error.log \
    superlists.wsgi:application
```

Użycie konfiguracji hierarchicznej rejestracji danych

Kiedy wcześniej majstrowaliśmy przy wywołaniu `logging.warning()`, wykorzystaliśmy rejestrowanie danych na najwyższym poziomie. To nie jest najlepszy pomysł, ponieważ przygotowane przez firmy trzecie moduły mogą tutaj namieszać. Zwyczajowym rozwiązaniem jest użycie komponentu rejestracji danych o nazwie takiej samej, jaką ma aktualny plik. W tym celu stosuje się następujące polecenie:

```
logger = logging.getLogger(__name__)
```

Konfiguracja rejestracji danych jest hierarchiczna, a więc można zdefiniować „nadzędne” komponenty rejestracji danych dla modułów najwyższego poziomu, a wszystkie znajdujące się w nich moduły Pythona będą dziedziczyć po wspomnianej konfiguracji.

Poniżej pokazano dodanie do pliku `settings.py` konfiguracji komponentów rejestracji danych.

Plik `superlists/settings.py`:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
        },
        'accounts': {
            'handlers': ['console'],
        },
    },
}
```

```

    'lists': {
        'handlers': ['console'],
    },
},
'root': {'level': 'INFO'},
}

```

Teraz `accounts.models`, `accounts.views`, `accounts.authentication` i wszystkie pozostałe moduły będą dziedziczyć `logging.StreamHandler()` po nadzorującym komponentem rejestracji danych o nazwie `accounts`.

Niestety struktura projektu Django wyklucza możliwość zdefiniowania najwyższego poziomu komponentu rejestracji danych przeznaczonego dla całego projektu (pominając root), a więc trzeba zdefiniować po jednym tego rodzaju komponentem dla poszczególnych aplikacji.

Poniżej pokazano przykład testu przeznaczonego do sprawdzenia zachowania komponentu rejestrującego dane.

Plik `accounts/tests/test_authentication.py` (ch17l023):

```

import logging
[...]
@patch('accounts.authentication.requests.post')
class AuthenticateTest(TestCase):
    [...]

    def test_logs_non_okay_responses_from_persona(self, mock_post):
        response_json = {
            'status': 'not okay', 'reason': 'eg, audience mismatch'
        }
        mock_post.return_value.ok = True
        mock_post.return_value.json.return_value = response_json #❶

        logger = logging.getLogger('accounts.authentication') #❷
        with patch.object(logger, 'warning') as mock_log_warning: #❸
            self.backend.authenticate('an assertion')

        mock_log_warning.assert_called_once_with(
            'Persona mówi nie. Dane JSON: {}'.format(response_json) #❹
        )

```

- ❶ Konfigurujemy test wraz z pewnymi danymi, które powinny wywołać operację rejestracji danych.
- ❷ Pobieramy komponent rejestracji danych przeznaczony dla testowanego modułu.
- ❸ Używamy wywołania `patch.object()` w celu tymczasowej imitacji funkcji ostrzeżenia. Dzięki użyciu `with` otrzymujemy menedżer kontekstu dla testowanej funkcji.
- ❹ Następnie możemy wykonywać asercje.

Otrzymujemy następujące dane wyjściowe:

```
AssertionError: Expected 'warning' to be called once. Called 0 times.
```

Wypróbowujemy teraz rozwiązanie, aby upewnić się, że naprawdę testujemy to, co chcemy przetestować.

Plik *accounts/authentication.py* (ch17l024):

```
import logging
logger = logging.getLogger(__name__)
[...]
if response.ok and response.json()['status'] == 'okay':
    [...]
else:
    logger.warning('foo')
```

Wynikiem jest oczekiwane niepowodzenie:

```
AssertionError: Expected call: warning("Persona mówi nie. Dane JSON: {'status': 'not okay', 'reason': 'eg, audience mismatch'}")
Actual call: warning('foo')
```

Przechodzimy teraz do rzeczywistej implementacji.

Plik *accounts/authentication.py* (ch17l025):

```
else:
    logger.warning(
        ' Persona mówi nie. Dane JSON: {}'.format(response.json())
    )
$ python3 manage.py test accounts
[...]
Ran 15 tests in 0.033s

OK
```

Możesz sobie łatwo wyobrazić, jak na obecnym etapie przetestować więcej kombinacji, gdy chcesz uzyskać różne komunikaty błędów dla `response.ok != True` itd.

Podsumowanie

Skonfigurowaliśmy testy do działania w komputerze lokalnym i serwerze, a ponadto przygotowaliśmy znacznie solidniejszą konfigurację rejestracji danych.

Zanim jednak wdrożymy kod w witrynę produkcyjną, lepiej dajmy użytkownikom to, czego naprawdę chcą. W kolejnym rozdziale dowiesz się, jak umożliwić użytkownikom zapisywanie ich list na stronie „Moje listy”.

Konfiguracja testu i rejestracja danych

Zachowaj ostrożność podczas pozbywania się powielonego kodu z testów funkcjonalnych

Test funkcjonalny nie musi testować każdego fragmentu aplikacji. W omawianym przykładzie chcieliśmy uniknąć pełnego procesu logowania dla każdego testu funkcjonalnego wymagającego uwierzytelnionego użytkownika. Dlatego też konfigurację testu wykorzystaliśmy do małego „oszustwa” i pominiecia tej części testu. W tworzonych testach funkcjonalnych być może będziesz chciał pomijać inne fragmenty aplikacji. Pamiętaj jednak o jednym: testy funkcjonalne mają przechwytywać nieprzewidywalne interakcje między poszczególnymi fragmentami aplikacji, a więc zachowaj dużą ostrożność podczas radykalnego unikania powielonego kodu.

Konfiguracje testów

Konfiguracja testu odnosi się tutaj do danych, które muszą być przygotowane przed rozpoczęciem testu. Bardzo często oznacza to umieszczenie pewnych informacji w bazie danych. Jak zobaczyłeś (cookie w przeglądarce internetowej), to mogą być także inne rodzaje przygotowań.

Unikaj konfiguracji w formacie JSON

Django niezwykle ułatwia zapis i przywracanie w formacie JSON (oraz innych) danych z bazy danych. W tym celu używane są polecenia zarządzania o nazwach dumpdata i loaddata. Większość programistów jest przeciwna tego rodzaju konfiguracji testów, ponieważ są niezwykle uciążliwe w obsłudze, gdy zmianie ulegnie schemat bazy danych. Lepszym rozwiązaniem jest użycie ORM lub narzędzia takiego jak `factory_boy`⁴.

Konfiguracja testu powinna również działać zdalnie

Klasa `LiveServerTestCase` bardzo ułatwia pracę z testową bazą danych za pomocą Django ORM, gdy testy są wykonywane lokalnie. Praca z bazą danych znajdująca się w serwerze prowizorycznym nie jest już taka łatwa. Jednym z rozwiązań jest użycie polecień zarządzania, jak pokazano w rozdziale. Sprawdź jednak inne rozwiązanie i zachowaj ostrożność.

Używaj komponentów rejestracji danych o nazwach takich jak testowane moduły

Główny komponent rejestracji danych to pojedynczy obiekt globalny dostępny dla każdej biblioteki wczytanej przez proces Pythona, a więc nigdy nie masz nad nim pełnej kontroli. Zamiast tego stosuj wzorzec `logging.getLogger(__name__)` w celu pobrania unikatowego komponentu dla danego modułu. Wspomniany komponent będzie dziedziczył po konfiguracji najwyższego poziomu.

Testuj ważne komunikaty mechanizmu rejestracji danych

Jak mogłeś się przekonać, komunikaty w procesie rejestracji danych mogą mieć znaczenie krytyczne podczas usuwania błędów w środowisku produkcyjnym. Jeżeli komunikat jest wystarczająco ważny, aby zachować go w bazie, to prawdopodobnie jest również wystarczająco ważny, aby go przetestować. Dobrą regułą jest to, że wszystko powyżej poziomu `logging.INFO` zdecydowanie wymaga testu. Użycie wywołania `patch.object()` w komponencie rejestracji danych dla testowanego modułu jest wygodnym sposobem na przeprowadzanie testów jednostkowych.

⁴ <https://factoryboy.readthedocs.org/en/latest/>

Kończymy „Moje listy” — podejście Outside-In

W tym rozdziale chciałbym przedstawić podejście Outside-In w TDD. W zasadzie to podejście, które stosujemy w książce. Używana przez nas „podwójna pętla” procesu TDD, gdy najpierw tworzymy test funkcjonalny, a następnie testy jednostkowe, stanowi manifestację podejścia Outside-In — projektujemy system od zewnętrz i tworzymy kod w warstwach. Teraz wyraźnie zastosujemy podejście Outside-In, wyjaśnione też zostaną kwestie, które mogą się pojawiać podczas jego użycia.

Alternatywa, czyli podejście Inside-Out

Alternatywą dla Outside-In jest podejście Inside-Out, czyli sposób, w jaki większość osób intuicyjnie pracuje, zanim napotka TDD. Po przygotowaniu projektu naturalną inklinacją jest jego implementacja, począwszy od znajdujących się w samym środku komponentów działających na najniższym możliwym poziomie.

Na przykład gdy zmagamy się z naszym bieżącym problemem dostarczenia użytkownikom strony z zapisanymi listami rzeczy do zrobienia, może wystąpić pokusa, aby pracę rozpoczęć od dodania atrybutu „właściciel” do obiektu modelu List, tłumacząc to tym, że atrybut wspomnianego rodzaju „oczywiście” będzie niezbędny. Mając ten atrybut, można przystąpić do modyfikacji innych warstw kodu, takich jak widoki i szablony, wykorzystując przy tym nowy atrybut. Na końcu można dodać adres URL prowadzący do nowego widoku.

Powyższe podejście wydaje się komfortowe, ponieważ nigdy nie pracujesz z żadnym fragmentem kodu zależnym od czegoś, co nie zostało jeszcze zaimplementowane. Każde zadanie wykonane na niższym poziomie jednocześnie stanowi solidny fundament, na którym następnie powstaje kolejna warstwa.

Jednak podejście Inside-Out ma także pewne słabe punkty.

Dlaczego preferowane jest podejście Outside-In?

Najbardziej oczywisty problem związanego z podejściem Inside-Out to brak możliwości stosowania technik TDD. Pierwsze niepowodzenie testu funkcjonalnego może wynikać z braku adresu URL, ale decydujemy się na zignorowanie tego i dodajemy atrybuty do obiektów modelu bazy danych.

W gowie możemy mieć pomysły na żądane zachowanie wewnętrznych warstw, takich jak modele bazy danych. Bardzo często wspomniane pomysły będą całkiem dobre, ale to jedynie spekulacje dotyczące tego, co będzie naprawdę potrzebne, ponieważ jeszcze nie zostały zbudowane zewnętrzne warstwy używające wymyślonych funkcji.

Problem będący skutkiem powyższego podejścia to powstanie wewnętrznych komponentów o możliwościach znacznie ogólniejszych lub dużo większych niż faktycznie potrzebne. Ich opracowanie oznacza marnotrawstwo czasu i stanowi źródło niepotrzebnego zwiększenia poziomu skomplikowania projektu. Inny często pojawiający się problem polega na tworzeniu wewnętrznych komponentów wraz z API wygodnym z punktu widzenia wewnętrznego projektu. Niestety później okazuje się, że API jest nieodpowiednie dla wywołań pochodzących z zewnętrznych warstw, które chcemy utworzyć. Co gorsza, może się okazać, że masz wewnętrzne komponenty, które w rzeczywistości nie rozwiązują koniecznych do rozwiązania problemów w zewnętrznych warstwach.

Z drugiej strony, podejście Outside-In pozwala na użycie każdej warstwy do przygotowania najwygodniejszego API, które można uzyskać z warstwy znajdującej się niżej. Zobaczmy takie podejście w praktyce.

Test funkcjonalny dla strony Moje listy

Podczas przygotowywania poniższego testu funkcjonalnego pracę rozpoczynamy od najbardziej zewnętrznej warstwy (prezentacja), a następnie przechodzimy do funkcji widoku (tak zwane kontrolery). Na końcu zajmujemy się najbardziej wewnętrznymi warstwami, w tym przypadku to będzie kod modelu.

Doskonale wiemy, że kod funkcji `create_pre_authenticated_session()` działa, a więc możemy przystąpić do utworzenia testu funkcjonalnego wyszukującego stronę „Moje listy”.

Plik `functional_tests/test_my_lists.py`:

```
def test_logged_in_users_lists_are_saved_as_my_lists(self):
    # Edyta jest zalogowanym użytkownikiem.
    self.create_pre_authenticated_session('edyta@przyklad.pl')

    # Przeszła na stronę główną i zaczęła tworzyć nową listę.
    self.browser.get(self.server_url)
    self.get_item_input_box().send_keys('Przygotować siatkę\n')
    self.get_item_input_box().send_keys('Kupić przynętę\n')
    first_list_url = self.browser.current_url

    # Po raz pierwszy zauważała łącze "Moje listy".
    self.browser.find_element_by_link_text('Moje listy').click()

    # Zobaczyła, że jej lista tam się znajduje.
    # Nazwa listy pochodzi od pierwszego elementu na niej.
```

```

self.browser.find_element_by_link_text('Przygotować siatkę').click()
self.assertEqual(self.browser.current_url, first_list_url)

# Postanowiła utworzyć kolejną listę.
self.browser.get(self.server_url)
self.get_item_input_box().send_keys('Przygotowania do wypyawy\n')
second_list_url = self.browser.current_url

# Strona "Moje listy" zawierała teraz tę nową listę.
self.browser.find_element_by_link_text('Moje listy').click()
self.browser.find_element_by_link_text('Przygotowania do wypyawy').click()
self.assertEqual(self.browser.current_url, second_list_url)

# Wylogowała się. Opcja "Moje listy" zniknęła.
self.browser.find_element_by_id('id_logout').click()
self.assertEqual(
    self.browser.find_elements_by_link_text('Moje listy'),
    []
)

```

Po uruchomieniu testu pierwszy błąd powinien przedstawić się następująco:

```
selenium.common.exceptions NoSuchElementException: Message: 'Unable to locate
element: {"method":"link text","selector":"Moje listy"}' ; Stacktrace:
```

Warstwa zewnętrzna — prezentacja i szablony

Test kończy się niepowodzeniem, a komunikat informuje o braku możliwości odszukania łącza o nazwie *Moje listy*. Rozwiążanie problemu można przeprowadzić w warstwie prezentacyjnej, a dokładniej w pliku *base.html*, w pasku nawigacyjnym. Oto minimalna zmiana kodu do wprowadzenia.

Plik *lists/templates/base.html* (ch18l002-1):

```
{% if user.email %}
    <ul class="nav navbar-nav">
        <li><a href="#">Moje listy</a></li>
    </ul>
    <a class="btn navbar-btn navbar-right" id="id_logout" [...]
```

Oczywiście dodane łącze prowadzi donikąd, ale przynajmniej pozwala na przejście do kolejnego błędu:

```
self.browser.find_element_by_link_text('Przygotować siatkę').click()
[...]
selenium.common.exceptions NoSuchElementException: Message: 'Unable to locate
element: {"method":"link text","selector":"Przygotować siatkę"}' ; Stacktrace:
```

Z komunikatu wynika, że powinniśmy przystąpić do utworzenia strony wyświetlającej wszystkie listy użytkownika. Pracę rozpoczynamy od podstaw — adresu URL oraz szablonu miejsca zarezerwowanego dla niego.

Warto przypomnieć ponownie o możliwości zastosowania podejścia Outside-In, czyli w warstwie prezentacyjnej wraz z adresem URL i niczym więcej.

Plik *lists/templates/base.html* (ch18l002-2):

```
<ul class="nav navbar-nav">
    <li><a href="{% url 'my_lists' user.email %}">Moje listy</a></li>
</ul>
```

Przejście o jedną warstwę w dół do funkcji widoku (kontroler)

Poprzedni krok powoduje wygenerowanie błędu szablonu, a więc z warstwy prezentacyjnej i adresu URL przechodzimy w dół do warstwy kontrolera, czyli funkcji widoku w Django.

Jak zwykle zaczynamy od przygotowania testu.

Plik *lists/tests/test_views.py* (ch18l003):

```
class MyListsTest(TestCase):  
  
    def test_my_lists_url_renders_my_lists_template(self):  
        response = self.client.get('/lists/users/a@b.com/')  
        self.assertTemplateUsed(response, 'my_lists.html')
```

Otrzymujemy następujący komunikat błędu:

```
AssertionError: No templates used to render the response
```

Usunięciem problemu zajmujemy się na poziomie prezentacji, w pliku *urls.py*.

Plik *lists/urls.py*:

```
urlpatterns = patterns('',  
    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),  
    url(r'^new$', 'lists.views.new_list', name='new_list'),  
    url(r'^users/(.+)/$', 'lists.views.my_lists', name='my_lists'),  
)
```

Test kończy się kolejnym niepowodzeniem, co wskazuje na konieczność przejścia w dół na następny poziom:

```
django.core.exceptions.ViewDoesNotExist: Could not import lists.views.my_lists.  
View does not exist in module lists.views.
```

Z warstwy prezentacji przechodzimy do warstwy widoków, a następnie tworzymy minimalne miejsce zarezerwowane.

Plik *lists/views.py* (ch18l005):

```
def my_lists(request, email):  
    return render(request, 'my_lists.html')
```

Przygotowujemy także minimalny szablon.

Plik *lists/templates/my_lists.html*:

```
{% extends 'base.html' %}  
  
{% block header_text %}Moje listy{% endblock %}
```

Testy jednostkowe zostają zaliczone, ale test funkcjonalny nadal pozostaje w tym samym punkcie i informuje, że na stronie „Moje listy” nie wyświetlono żadnych list. Lista ma być możliwym do kliknięcia łączem o nazwie takiej samej jak pierwszy element listy:

```
selenium.common.exceptions NoSuchElementException: Message: 'Unable to locate  
element: {"method":"link text","selector":"Przygotować siatkę"}' ; Stacktrace:
```

Kolejne zaliczenie — podejście Outside-In

Na każdym etapie pozwalamy, aby test funkcjonalny wskazywał zadania, które powinny być wykonane jako kolejne.

Ponownie rozpoczynamy od warstwy zewnętrznej (w szablonie), przygotowujemy kod szablonu, za pomocą którego strona „Moje listy” będzie działała w oczekiwany sposób. W trakcie pracy zaczynamy określać API, którego będziemy chcieli użyć z kodu znajdującego się w warstwie zdefiniowanej niżej.

Szybka restrukturyzacja hierarchii dziedziczenia szablonu

Obecnie w naszym szablonie bazowym nie ma żadnego miejsca na umieszczenie nowej treści. Ponadto strona „Moje listy” nie wymaga formularza tworzenia nowego elementu. Dlatego umieszczamy go w bloku i powodujemy, że staje się opcjonalny.

Plik *lists/templates/base.html* (ch18l007-1):

```
<div class="text-center">
    <h1>{% block header_text %}{% endblock %}</h1>

    {% block list_form %}
        <form method="POST" action="{% block form_action %}{% endblock %}">
            {{ form.text }}
            {% csrf_token %}
            {% if form.errors %}
                <div class="form-group has-error">
                    <div class="help-block">{{ form.text.errors }}</div>
                </div>
            {% endif %}
        </form>
    {% endblock %}

</div>
```

Plik *lists/templates/base.html* (ch18l007-2):

```
<div class="row">
    <div class="col-md-6 col-md-offset-3">
        {% block table %}
        {% endblock %}
    </div>
</div>

<div class="row">
    <div class="col-md-6 col-md-offset-3">
        {% block extra_content %}
        {% endblock %}
    </div>
</div>

</div>
<script src="http://code.jquery.com/jquery.min.js"></script>
[...]
```

Projektowanie API za pomocą szablonu

Tymczasem w pliku *my_lists.html* nadpisujemy `list_form` i wskazujemy, że powinien być pusty...

Plik *lists/templates/my_lists.html*:

```
{% extends 'base.html' %}

{% block header_text %}Moje listy{% endblock %}

{% block list_form %}{% endblock %}
```

Następnie możemy przystąpić do pracy wewnątrz bloku `extra_content`.

Plik *lists/templates/my_lists.html*:

```
[...]

{% block list_form %}{% endblock %}

{% block extra_content %}
    <h2>Listy utworzone przez {{ owner.email }}</h2> #❶
    <ul>
        {% for list in owner.list_set.all %} #❷
            <li><a href="{{ list.get_absolute_url }}">{{ list.name }}</a></li> #❸
        {% endfor %}
    </ul>
{% endblock %}
```

W szablonie podjęliśmy kilka decyzji projektowych, które będą filtrowały sobie drogę w dół kodu:

- ❶ Zmienna o nazwie `owner` ma w szablonie przedstawiać użytkownika.
- ❷ Chcemy mieć możliwość iteracji list utworzonych przez użytkownika. Do tego celu wykorzystamy `owner.list_set.all` (wiem, że wymienione wywołanie jest oferowane przez Django ORM).
- ❸ Chcemy, aby wywołanie `list.name` wyświetliło „nazwę” listy, czyli obecnie tekst jej pierwszego elementu.



Podejście Outside-In jest czasami określane mianem „myślienie życzeniowe” i teraz już wiesz, skąd wzięło się to określenie. Rozpoczynamy od tworzenia kodu na wyższych poziomach, opierając się na tym, co chcemy osiągnąć na niższych poziomach, nawet jeśli one jeszcze nie istnieją!

Możemy ponownie uruchomić testy funkcjonalne, sprawdzić, czy na pewno niczego nie uszkodziliśmy, i przekonać się, że nie posunęliśmy się do przodu:

```
$ python3 manage.py test functional_tests
[...]
selenium.common.exceptions NoSuchElementException: Message: 'Unable to locate
element: {"method":"link text","selector":"Przygotować siatkę"}' ; Stacktrace:
-----
Ran 7 tests in 77.613s

FAILED (errors=1)
```

Cóż, nie posunęliśmy się do przodu, ale też przynajmniej niczego nie zepsuliśmy. Czas na przekazanie zmian do repozytorium:

```
$ git add lists  
$ git diff --staged  
$ git commit -m "Adres URL, miejsce zarezerwowane dla widoku i pierwsze kroki w szablonie  
dla my_lists."
```

Przejście w dół do kolejnej warstwy — co widok przekazuje szablonowi?

Poniżej przedstawiono test widoków.

Plik *lists/tests/test_views.py* (ch18l011):

```
from django.contrib.auth import get_user_model  
User = get_user_model()  
[...]  
class MyListsTest(TestCase):  
  
    def test_my_lists_url_renders_my_lists_template(self):  
        [...]  
  
        def test_passes_correct_owner_to_template(self):  
            User.objects.create(email='wrong@owner.com')  
            correct_user = User.objects.create(email='a@b.com')  
            response = self.client.get('/lists/users/a@b.com/')  
            self.assertEqual(response.context['owner'], correct_user)
```

Otrzymujemy następujący komunikat błędu:

```
KeyError: 'owner'
```

Oto zmiany konieczne do wprowadzenia.

Plik *lists/views.py* (ch18l012):

```
from django.contrib.auth import get_user_model  
User = get_user_model()  
[...]  
  
def my_lists(request, email):  
    owner = User.objects.get(email=email)  
    return render(request, 'my_lists.html', {'owner': owner})
```

Nasz nowy test zostaje zaliczony, ale jednocześnie poprzedni test powoduje wygenerowanie błędu. Musimy po prostu dodać użytkownika dla testu, jak pokazano poniżej.

Plik *lists/tests/test_views.py* (ch18l013):

```
def test_my_lists_url_renders_my_lists_template(self):  
    User.objects.create(email='a@b.com')  
    [...]
```

Teraz już wszystko w porządku:

```
OK
```

Kolejne „wymaganie” z warstwy widoku — nowe listy powinny „zapamiętywać” swego właściciela

Zanim przejdziemy w dół do warstwy modelu, w warstwie widoku musimy zająć się jeszcze jednym fragmentem kodu, który będzie używał modelu. Mianowicie potrzebny jest pewien sposób, aby nowe listy były przypisywane ich właścielowi, jeśli tworzący je użytkownik jest zalogowany w witrynie.

Oto pierwsze podejście do utworzenia testu.

Plik *lists/tests/test_views.py* (ch18l014):

```
from django.http import HttpRequest
[...]
from lists.views import new_list
[...]

class NewListTest(TestCase):
    [...]

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        request = HttpRequest()
        request.user = User.objects.create(email='a@b.com')
        request.POST['text'] = 'new list item'
        new_list(request)
        list_ = List.objects.first()
        self.assertEqual(list_.owner, request.user)
```

Powyższy test używa niezmodyfikowanej funkcji widoku i ręcznie skonstruowanego obiektu `HttpRequest`, ponieważ przygotowanie testu w taki właśnie sposób jest nieco łatwiejsze. Wprawdzie klient testowy Django zawiera funkcję pomocniczą o nazwie `login()`, ale ona nie działa z zewnętrznymi usługami uwierzytelniania. Alternatywne podejście polega na ręcznym utworzeniu obiektu sesji (podobnie jak ma to miejsce w testach funkcjonalnych) lub użyciu imitacji. Jednak sądzę, że obie wymienione możliwości prowadzą do brzydszego rozwiązania niż tutaj prezentowane. Jeżeli jesteś ciekaw, test możesz utworzyć jeszcze inaczej.

Wykonanie testu kończy się niepowodzeniem:

```
AttributeError: 'List' object has no attribute 'owner'
```

Aby usnąć błąd, można zapisać kod w przedstawiony poniżej sposób.

Plik *lists/views.py* (ch18l015):

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List()
        list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

To jednak nie działa, ponieważ jeszcze nie wiemy, jak zapisać właściciela listy:

```
self.assertEqual(list_.owner, request.user)
AttributeError: 'List' object has no attribute 'owner'
```

Czy przejść do kolejnej warstwy, gdy test kończy się niepowodzeniem?

W celu zaliczenia powyższego testu zapisanego w obecnej postaci konieczne jest przejście w dół do warstwy modelu. To jednak oznacza wykonywanie kolejnych zadań, gdy ma się test kończący się niepowodzeniem, co na pewno nie jest idealnym rozwiązaniem.

Alternatywa polega na modyfikacji testu w taki sposób, aby stał się bardziej *odizolowany* od warstwy znajdującej się poniżej, co pozwoli na użycie imitacji.

Z jednej strony użycie imitacji wiąże się z większym wysiłkiem i może prowadzić do powstania testów trudniejszych w odczycie. Z drugiej strony wyobraź sobie, że nasza aplikacja mogłaby być znacznie bardziej skomplikowana i zawierać więcej warstw pośrednich. Wyobraź sobie pozostawienie trzech, czterech lub pięciu warstw testów kończących się niepowodzeniem, a Ty czekasz na przejście do najniższej warstwy, aby zaimplementować funkcję o znaczeniu krytycznym. Kiedy testy kończą się niepowodzeniem, nie ma pewności, że dana warstwa faktycznie działa. Musimy zaczekać aż do przejścia do kolejnej warstwy znajdującej się poniżej.

To jest decyzja, którą prawdopodobnie będziesz musiał podjąć we własnych projektach. Przeanalizujmy oba podejścia. Rozpoczynamy od drogi na skróty i pozostawiamy test kończący się niepowodzeniem. W następnym rozdziale powrócimy do dokładnie tego samego miejsca i zobaczymy, jak wszystko może się potoczyć, gdy test będzie znacznie bardziej odizolowany.

Przekazujemy pliki do repozytorium, a operację przekazania oznaczamy *tagiem*, aby w kolejnym rozdziale móc łatwo powrócić do tego samego punktu:

```
$ git commit -am"Widok new_list próbuje przypisać właściciela, ale nie może."  
$ git tag revisit_this_point_with_isolated_tests
```

Przejście w dół do warstwy modelu

Nasz projekt z zastosowaniem podejścia Outside-In definiuje dwa wymagania dla warstwy modelu. Pierwsze to możliwość przypisania właściciela listy za pomocą atrybutu `.owner`. Drugie to możliwość uzyskania dostępu do właściciela listy za pomocą API `owner.list_set.all()`.

Zaczynamy od przygotowania testu.

Plik `lists/tests/test_models.py` (ch18l018):

```
from django.contrib.auth import get_user_model  
User = get_user_model()  
[...]  
  
class ListModelTest(TestCase):  
  
    def test_get_absolute_url(self):  
        [...]  
  
    def test_lists_can_have_owners(self):  
        user = User.objects.create(email='a@b.com')  
        list_ = List.objects.create(owner=user)  
        self.assertIn(list_, user.list_set.all())
```

Wynikiem jest niepowodzenie nowego testu jednostkowego:

```
list_ = List.objects.create(owner=user)
[...]
TypeError: 'owner' is an invalid keyword argument for this function
```

Naiwna implementacja może przedstawiać się następująco:

```
from django.conf import settings
[...]
class List(models.Model):
    owner = models.ForeignKey(settings.AUTH_USER_MODEL)
```

Chcemy mieć pewność, że właściciel listy jest opcjonalny. Wyraźne wskazanie tego faktu jest lepsze od niejawnego, a testy stanowią formę dokumentacji, więc przygotowujemy odpowiedni test.

Plik *lists/tests/test_models.py* (ch18l020):

```
def test_list_owner_is_optional(self):
    List.objects.create() # Nie powinno nastąpić zgłoszenie.
```

Prawidłowa implementacja została przedstawiona poniżej.

Plik *lists/models.py*:

```
from django.conf import settings
[...]
class List(models.Model):
    owner = models.ForeignKey(settings.AUTH_USER_MODEL, blank=True, null=True)

    def get_absolute_url(self):
        return reverse('view_list', args=[self.id])
```

Wykonanie testów w tym momencie kończy się zwykłym błędem bazy danych:

```
return Database.Cursor.execute(self, query, params)
django.db.utils.OperationalError: table lists_list has no column named owner_id
```

Konieczne jest przeprowadzenie migracji:

```
$ python3 manage.py makemigrations
Migrations for 'lists':
  0006_list_owner.py:
    - Add field owner to list
```

Już prawie skończyliśmy, jeszcze tylko kilka błędów:

```
ERROR: test_redirects_after_POST (lists.tests.test_views.NewListTest)
[...]
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f364795ef90>>":
"List.owner" must be a "User" instance.
ERROR: test_saving_a_POST_request (lists.tests.test_views.NewListTest)
[...]
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f364795ef90>>":
"List.owner" must be a "User" instance.
```

Teraz powracamy do warstwy widoku w celu przeprowadzenia drobnych prac porządkowych. Zwróć uwagę na obecność starego testu dla widoku *new_list*, gdy nie mamy zalogowanego użytkownika. Właściciel listy powinien być zapisany tylko wtedy, gdy użytkownik jest faktycznie zalogowany. W tym miejscu użyteczna okazuje się funkcja *.is_authenticated()*, którą zdefiniowaliśmy w rozdziale 16. (Kiedy użytkownik nie jest zalogowany, Django przedstawia go za pomocą klasy o nazwie *AnonymousUser*, dla której wartością zwrótną funkcji *.is_authenticated()* zawsze będzie *False*).

Plik *lists/views.py* (ch18l023):

```
if form.is_valid():
    list_ = List()
    if request.user.is_authenticated():
        list_.owner = request.user
    list_.save()
    form.save(for_list=list_)
    [...]
```

Testy zostały zaliczone!

```
$ python3 manage.py test lists
Creating test database for alias 'default'...
-----
Ran 39 tests in 0.237s

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

To jest odpowiedni moment na przekazanie plików do repozytorium:

```
$ git add lists
$ git commit -m"Lista może mieć właściciela, który zostanie zapisany podczas jej tworzenia."
```

Ostatni krok — uzyskanie z poziomu szablonu dostępu do właściciela za pomocą API .name

Ostatnim krokiem w projekcie opartym na podejściu Outside-In jest możliwość uzyskania z poziomu szablonu dostępu do „nazwy” listy, którą w tym momencie jest tekst pierwszego jej elementu.

Plik *lists/tests/test_models.py* (ch18l024):

```
def test_list_name_is_first_item_text(self):
    list_ = List.objects.create()
    Item.objects.create(list=list_, text='first item')
    Item.objects.create(list=list_, text='second item')
    self.assertEqual(list_.name, 'first item')
```

Plik *lists/models.py* (ch18l025):

```
@property
def name(self):
    return self.item_set.first().text
```

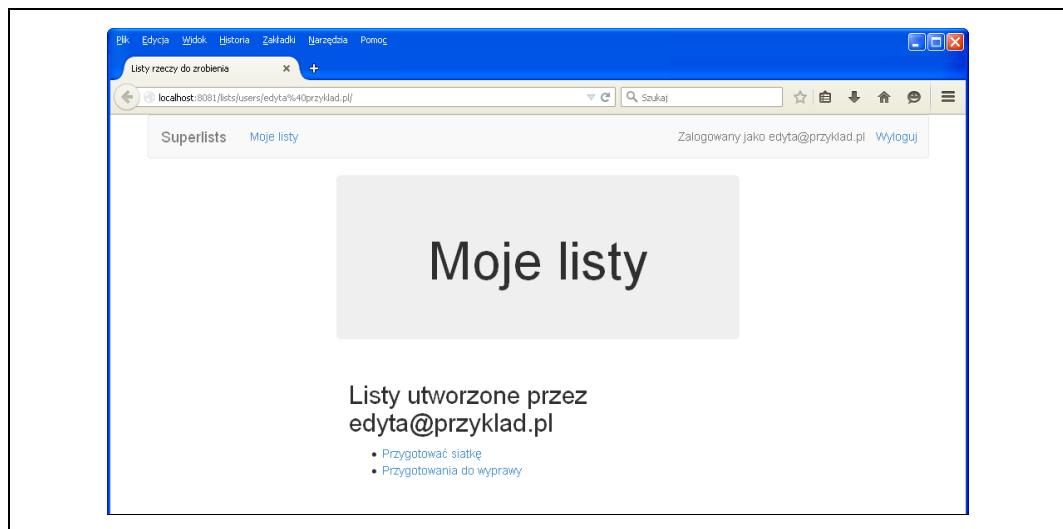
Dekorator @property w Pythonie

Jeżeli wcześniej nie spotkałeś się z dekoratorem `@property`, to powinieneś wiedzieć, że transformuje on metodę klasy w taki sposób, aby dla świata zewnętrznego jawiła się jako atrybut.

Ta funkcja języka oferuje potężne możliwości, ponieważ niezwykle ułatwia implementację „kacznego typowania” (ang. *duck typing*) i zmianę implementacji właściwości bez zmiany interfejsu klasy. Innymi słowy, jeśli zdecydujemy się zmienić `.name` na „rzeczywisty” atrybut modelu przechowywany jako tekst w bazie danych, wówczas będzie można to zrobić bez problemu. Pozostała część kodu nadal będzie mogła po prostu uzyskać dostęp do `.name` i pobrać nazwę listy, nie mając nawet żadnych informacji dotyczących sposobu implementacji atrybutu.

Oczywiście w języku szablonów Django `.name` nadal będzie wywoływać metodę, nawet jeśli nie zawiera dekoratora `@property`. To jednak jest cecha Django i nie ma zastosowania ogólnie w Pythonie...

Możesz wierzyć lub nie, wszystkie testy zostały zaliczone i mamy działającą stronę „Moje listy”, jak pokazano na rysunku 18.1.



Rysunek 18.1. Strona „Moje listy” (i jednocześnie dowód, że test przeprowadziłem w Windows)

```
$ python3 manage.py test functional_tests  
[...]  
Ran 7 tests in 93.819s
```

OK

Wiemy, że tutaj posłużyliśmy się drobnym oszustwem. Testing Goat uważnie nas obserwuje. Na jednej warstwie pozostawiliśmy test kończący się niepowodzeniem i zabraliśmy się za implementację jego zależności w warstwie znajdującej się poniżej. Zobaczmy, jak taka sytuacja będzie się przedstawiać, gdy zastosujemy lepszą izolację testu...

Podejście Outside-In w TDD

Podejście Outside-In w TDD

Oparta na testach metodologia tworzenia kodu począwszy od warstw „zewnętrznych” (prezentacja, GUI) w kierunku „wewnętrznych” krok po kroku, poprzez warstwy widoku i kontrolera w kierunku do warstwy modelu. Pomyśl polega na tym, aby kod tworzyć na podstawie zadań, do których jest przeznaczony, a nie próbować określić wymagania od podstaw.

Myślenie życzeniowe

Podejście Outside-In jest czasami określane mianem „myślęcie życzeniowe”. Tak naprawdę każda technika TDD wykorzystuje myślenie życzeniowe. Pracę zawsze zaczynamy od utworzenia testów dla komponentów, które jeszcze nie istnieją.

Wady podejścia Outside-In

Podejście Outside-In nie jest złotym środkiem. Zachęca do skoncentrowania się na elementach natychmiast widocznych dla użytkownika, ale nie przypomina automatycznie o konieczności utworzenia innych testów o znaczeniu krytycznym, które są jednak mniej widoczne dla użytkownika, na przykład związanych z zapewnieniem bezpieczeństwa. O nich musisz pamiętać sam.

Izolacja i „słuchanie” testów

W poprzednim rozdziale podjęliśmy decyzję o pozostawieniu kończącego się niepowodzeniem testu jednostkowego w warstwie widoku i przejściu do tworzenia kolejnych testów oraz kodu w warstwie modelu, aby móc zaliczyć wspomniany test.

Zdecydowaliśmy się na takie rozwiązanie z powodu prostoty budowanej aplikacji. Warto jednak tutaj dodać, że tego rodzaju decyzja w skomplikowanej aplikacji może być niebezpieczna. Przejście do pracy nad kodem działającym na niższym poziomie bez wcześniejszego upewnienia się o *prawidłowym* funkcjonowaniu kodu na wyższym poziomie to ryzykowna strategia.



Jestem wdzięczny Gary'emu Bernhardtowi, który zapoznał się z wczesną wersją poprzedniego rozdziału i zachęcił mnie do dokładniejszego przedstawienia tematu izolacji testu.

Zapewnienie izolacji między warstwami aplikacji wymaga nieco większego wysiłku (i w większym stopniu użycia budzących lęk imitacji!), choć jednocześnie może pomóc w poprawieniu projektu, o czym się przekonasz w rozdziale.

Powrót do miejsca, w którym podjęliśmy decyzję — warstwa widoku zależy od nieutworzonego jeszcze kodu modelu

Powróćmy teraz do miejsca, w którym w poprzednim rozdziale podjęliśmy ważną decyzję, gdy nie można było zmusić do prawidłowego działania widoku `new_list` z powodu braku atrybutu `.owner`.

Mamy możliwość faktycznego cofnięcia się w czasie i przywrócenia wcześniejszego kodu. Dzięki temu zobaczysz, co się stanie, gdy wykorzystamy bardziej odizolowane testy.

```
$ git checkout -b more-isolation # Galążka na potrzeby tego eksperymentu.  
$ git reset --hard revisit_this_point_with_isolated_tests
```

Oto jak przedstawia się test, którego wykonanie kończy się niepowodzeniem.

Plik *lists/tests/test_views.py*:

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

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        request = HttpRequest()
        request.user = User.objects.create(email='a@b.com')
        request.POST['text'] = 'nowy element listy'
        new_list(request)
        list_ = List.objects.first()
        self.assertEqual(list_.owner, request.user)
```

Poniżej pokazano naszą próbę znalezienia rozwiązania.

Plik *lists/views.py*:

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List()
        list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

Na tym etapie test widoku kończy się niepowodzeniem, ponieważ nie mamy jeszcze warstwy modelu.

```
self.assertEqual(list_.owner, request.user)
AttributeError: 'List' object has no attribute 'owner'
```



Nie zobaczyłeś powyższego komunikatu błędu, dopóki faktycznie nie przywróciłeś starego kodu oraz pliku *lists/models.py*. Zdecydowanie powinieneś to zrobić, ponieważ celem niniejszego rozdziału jest między innymi sprawdzenie, czy można tworzyć testy dla nieistniejącej jeszcze warstwy modelu.

Pierwsza próba użycia imitacji w celu zapewnienia izolacji

Listy nie mają jeszcze właścicieli, ale dzięki użyciu odrabiny imitacji możemy pozwolić testom warstwy widoku „udawać”, że mają właścicieli.

Plik *lists/tests/test_views.py* (ch19l003):

```
from unittest.mock import Mock, patch

from django.http import HttpRequest
from django.test import TestCase
[...]

@patch('lists.views.List') #❶
def test_list_owner_is_saved_if_user_is_authenticated(self, mockList):
    mock_list = List.objects.create() #❷
    mock_list.save = Mock()
    mockList.return_value = mock_list
    request = HttpRequest()
```

```
request.user = User.objects.create() #❸
request.POST['text'] = 'nowy element listy'

new_list(request)
self.assertEqual(mock_list.owner, request.user) #❹
```

- ❶ Imitujemy funkcję `List()`, aby mogła uzyskać dostęp do wszystkich list tworzonych przez widok.
- ❷ Następnie tworzymy rzeczywisty obiekt `List` do użycia w widoku. To musi być rzeczywisty obiekt `List`, w przeciwnym razie próba zapisu obiektu `Item` przez widok zakończy się niepowodzeniem na skutek błędu klucza zewnętrznego (to wskazuje na jedynie częściowe odizolowanie testu).
- ❸ W obiekcie żądania definiujemy rzeczywistego użytkownika.
- ❹ Tworzymy asercję sprawdzającą, czy lista ma ustawiony atrybut `.owner`.

Jeżeli teraz wykonamy test, powinien zostać zaliczony:

```
$ python3 manage.py test lists
[...]
Ran 37 tests in 0.145s
OK
```

Jeśli test nie został zaliczony, upewnij się, że kod widoków w pliku `views.py` jest dokładnie taki sam jak pokazany. Użyte powinno być wywołanie `List()`, a nie `Lists.object.create`.



Użycie imitacji oznacza powiązanie z określonymi sposobami zastosowania API. To jest jeden z wielu kompromisów, na które trzeba iść podczas użycia obiektów imitacji.

Użycie `side_effect` do sprawdzenia sekwencji zdarzeń

Problem z omawianym testem polega na tym, że może być on zaliczony przez nieprawidłowy kod utworzony pomyłkowo. Wyobraź sobie, że przypadkowo wywołujemy metodę `save()` przed przypisaniem właściciela listy.

Plik `lists/views.py`:

```
if form.is_valid():
    list_ = List()
    list_.save()
    list_.owner = request.user
    form.save(for_list=list_)
    return redirect(list_)
```

Test w obecnej postaci nadal będzie zaliczony:

OK

Dlatego też trzeba sprawdzić, nie tylko to, czy właściciel został przypisany liście, ale również czy przypisanie nastąpiło *przed* wywołaniem metody `save()` w obiekcie listy.

Poniżej pokazano, jak przetestować sekwencję zdarzeń za pomocą imitacji. Można przygotować imitację funkcji, a następnie wykorzystać tę imitację do sprawdzenia stanu w chwili wywoływania funkcji.

Plik *lists/tests/test_views.py* (ch19l005):

```
@patch('lists.views.List')
def test_list_owner_is_saved_if_user_is_authenticated(self, mockList):
    mock_list = List.objects.create()
    mock_list.save = Mock()
    mockList.return_value = mock_list
    request = HttpRequest()
    request.user = Mock()
    request.user.is_authenticated.return_value = True
    request.POST['text'] = 'nowy element listy'

    def check_owner_assigned(): #❶
        self.assertEqual(mock_list.owner, request.user) #❷
        mock_list.save.side_effect = check_owner_assigned #❸

    new_list(request)

    mock_list.save.assert_called_once_with() #❹
```

- ❶❷ Definiujemy funkcję wykonującą asercję określającą czynność, która ma zostać wykonana jako pierwsza: sprawdzenie, czy został przypisany właściciel listy.
- ❸ Wspomnianą funkcję sprawdzającą przypisujemy jako `side_effect` do czynności, która ma zostać wykonana jako druga. Kiedy widok wywoła imitację metody `save()`, wtedy wykonana zdefiniowaną asercję. Upewniamy się o przygotowaniu tej konfiguracji przed rzeczywistym wywołaniem testowanej funkcji.
- ❹ Na koniec upewniamy się, że funkcja z przypisanyem `side_effect` faktycznie została wywołana, na przykład jak w `.save()`. W przeciwnym razie wywołanie asercji może nigdy nie nastąpić.



Dwa najczęściej popełniane błędy podczas użycia `side_effect` to zbyt późne przypisanie, na przykład *po* wywołaniu testowanej funkcji, oraz brak sprawdzenia, czy funkcja z `side_effect` została faktycznie wywołana. Muszę w tym miejscu dodać, że oba wymienione błędy popełniłem kilkakrotnie w trakcie pisania rozdziału.

Jeżeli na tym etapie kod omawiany powyżej wciąż masz „uszkodzony”, wówczas wywołanie metody `save()` w niewłaściwej kolejności podczas operacji przypisania właściciela spowoduje wygenerowanie następującego komunikatu o niepowodzeniu testu:

```
ERROR: test_list_owner_is_saved_if_user_is_authenticated
(lists.tests.test_views.NewListTest)
[...]
  File "/workspace/superlists/lists/views.py", line 17, in new_list
    list_.save()
[...]
  File "/workspace/superlists/lists/tests/test_views.py", line 84, in
check_owner_assigned
    self.assertEqual(mock_list.owner, request.user)
AttributeError: 'List' object has no attribute 'owner'
```

Zwrót uwagi, że niepowodzenie następuje w trakcie próby zapisu, a następnie wejścia do funkcji `side_effect`.

Zaliczenie testu może nastąpić po zmianie kodu na przedstawiony poniżej.

Plik *lists/views.py*:

```
if form.is_valid():
    list_ = List()
    list_.owner = request.user
    list_.save()
    form.save(for_list=list_)
    return redirect(list_)

...
OK
```

Ojej, to będzie naprawdę brzydkie test!

Posłuchaj testu — brzydkie test oznacza konieczność refaktoryzacji

Kiedy będziesz zmuszony tworzyć tego rodzaju test i zobacysz, że działa z oporami, to prawdopodobnie testy próbują przekazać coś ważnego. Dziesięć wierszy kodu odpowiedzialnych za konfigurację (trzy dla imitacji użytkownika, cztery dla obiektu żądania i trzy dla funkcji `side_effect`) to zdecydowanie zbyt wiele.

Test próbuje wskazać, że widok wykonuje zbyt dużą ilość pracy, zajmując się utworzeniem formularza, obiektu nowej listy oraz decydując o ewentualnym przypisaniu właściciela liście.

Wcześniej już widziałeś, że widoki mogą stać się prostsze i łatwiejsze do zrozumienia dzięki przeniesieniu pewnej ilości kodu do klasy. Dlaczego widok ma się zajmować tworzeniem obiektu listy? Prawdopodobnie to zadanie może być wykonane przez wywołanie `ItemForm.save()`. Ponadto dlaczego widok ma decydować, czy zapisywać `request.user`? Przecież to zadanie również może być wykonane przez formularz.

Kiedy formularzowi zlecamy więcej zadań do wykonania, prawdopodobnie należy mu nadać nową nazwę, na przykład `NewListForm` — ta nazwa lepiej oddaje zadania realizowane przez formularz.

Plik *lists/views.py*:

```
# Nie wprowadzaj jeszcze poniższego kodu, na razie rozważamy potencjalne rozwiązania.
```

```
def new_list(request):
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user) # Utworzenie obiektów List i Item.
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

To wygląda na interesujące rozwiązanie! Zobaczmy, jak można dotrzeć do tego stanu przez użycie w pełni izolowanych testów.

Ponowne utworzenie testów dla widoku, tym razem w pełni odizolowanych

Pierwsza próba utworzenia zestawu testów dla widoku zakończyła się powstaniem wysoce zintegrowanych testów. Ich zaliczenie wymagało użycia w pełni funkcjonalnych warstw bazy danych i formularzy. Podjęliśmy próbę odizolowania testów i teraz będziemy kontynuować ten wysiłek.

Pozostawienie starych zintegrowanych testów jako punktu odniesienia

Nazwę starej klasy `NewListTest` zmieniamy na `NewListViewIntegratedTest`, pozbywamy się kodu próbującego zapisać właściciela, przywracamy wersję zintegrowaną i na razie pomijamy ten test.

Plik `lists/tests/test_views.py` (ch19l008):

```
import unittest
[...]

class NewListViewIntegratedTest(TestCase):

    def test_saving_a_POST_request(self):
        [...]

    @unittest.skip
    def test_list_owner_is_saved_if_user_is_authenticated(self):
        request = HttpRequest()
        request.user = User.objects.create(email='a@b.com')
        request.POST['text'] = 'nowy element listy'
        new_list(request)
        list_ = List.objects.first()
        self.assertEqual(list_.owner, request.user)
```



Czy spotkałeś się z pojęciem „test integracji” i zastanawiasz się, czym się różni od „testu zintegrowanego”? Zajrzyj do ramki z definicjami w rozdziale 22.

```
$ python3 manage.py test lists
[...]
Ran 37 tests in 0.139s
OK
```

Nowy zestaw w pełni odizolowanych testów

Rozpoczynamy od pustych testów i sprawdzamy, czy odizolowane testy mogą być wykorzystane podczas tworzenia zmiennika widoku `new_list`. Nowemu widokowi nadajemy nazwę `new_list2`, tworzymy go obok poprzedniego, a gdy będziemy gotowi, wówczas zamienimy widoki i zobaczymy, czy zintegrowane testy nadal są zaliczane.

Plik *lists/views.py* (ch19l009):

```
def new_list(request):
    [...]
def new_list2(request):
    pass
```

Myślimy w kategoriach współpracy

Aby ponownie utworzone testy pozostały w pełni odizolowane, konieczna jest zmiana myślenia o testach. Przestań się zastanawiać, jak w rzeczywistości widok wpłynie na komponenty takie jak baza danych. Zastanów się lepiej nad obiekttami, z którymi widok ma współpracować, oraz nad sposobami współpracy widoku ze wspomnianymi obiekttami.

W nowym świecie widok będzie współdziałał przede wszystkim z obiektem formularza. Tworzymy więc jego imitację, aby mieć pełną kontrolę nad nim i możliwość zdefiniowania (myślenie życzeniowe) sposobu działania formularza.

Plik *lists/tests/test_views.py* (ch19l010):

```
from lists.views import new_list, new_list2
[...]
@patch('lists.views.NewListForm') #❶
class NewListViewUnitTestCase(unittest.TestCase): #❷

    def setUp(self):
        self.request = HttpRequest()
        self.request.POST['text'] = 'nowy element listy' #❸

    def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
        new_list2(self.request)
        mockNewListForm.assert_called_once_with(data=self.request.POST) #❹
```

- ❶ Klasa `TestCase` w Django znacznie ułatwia tworzenie zintegrowanych testów. Ponieważ chcemy mieć pewność, że tworzymy „czyste”, odizolowane testy jednostkowe, więc użyjemy jedynie `unittest.TestCase`.
- ❷ Przygotowujemy imitację klasy `NewListForm` (która nawet jeszcze nie istnieje). Wymieniona imitacja będzie używana we wszystkich testach i dlatego tworzymy ją na poziomie klasy.
- ❸ W metodzie `setUp()` przygotowujemy podstawowe żądanie POST, ręcznie tworzymy żądanie, zamiast używać (przesadnie zintegrowanego) klienta testowego Django.
- ❹ Przeprowadzamy pierwsze sprawdzenie nowego widoku, który inicjalizuje formularz `NewListForm` za pomocą odpowiedniego konstruktora. Dane używane w trakcie inicjalizacji pochodzą z żądania.

Test kończy się niepowodzeniem z powodu braku formularza `NewListForm` w widoku.

```
AttributeError: <module 'lists.views' from
'/workspace/superlists/lists/views.py'> does not have the attribute
'NewListForm'
```

Tworzymy miejsce zarezerwowane dla formularza.

Plik *lists/views.py* (ch19l011):

```
from lists.forms import ExistingListItemForm, ItemForm, NewListForm
[...]
```

I dalej...

Plik *lists/forms.py* (ch19l012):

```
class ItemForm(forms.ModelForm):
    [...]
    class NewListForm(object):
        pass
    class ExistingListItemForm(ItemForm):
        [...]
```

Teraz otrzymujemy oczekiwane niepowodzenie testu:

```
AssertionError: Expected 'NewListForm' to be called once. Called 0 times.
```

Przeprowadzamy więc implementację w przedstawiony poniżej sposób.

Plik *lists/views.py* (ch19l012-2):

```
def new_list2(request):
    NewListForm(data=request.POST)
$ python3 manage.py test lists
[...]
Ran 38 tests in 0.143s
OK
```

Kontynuujemy pracę. Jeżeli formularz jest prawidłowy, ma być przeprowadzona operacja zapisu.

Plik *lists/tests/test_views.py* (ch19l013):

```
@patch('lists.views.NewListForm')
class NewListViewUnitTestCase(unittest.TestCase):

    def setUp(self):
        self.request = HttpRequest()
        self.request.POST['text'] = 'nowy element listy'
        self.request.user = Mock()

    def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
        new_list2(self.request)
        mockNewListForm.assert_called_once_with(data=self.request.POST)

    def test_saves_form_with_owner_if_form_valid(self, mockNewListForm):
        mock_form = mockNewListForm.return_value
        mock_form.is_valid.return_value = True
        new_list2(self.request)
        mock_form.save.assert_called_once_with(owner=self.request.user)
```

Dochodzimy do przedstawionego poniżej fragmentu kodu.

Plik *lists/views.py* (ch19l014):

```
def new_list2(request):
    form = NewListForm(data=request.POST)
    form.save(owner=request.user)
```

W przypadku poprawności formularza widok ma zwrócić przekierowanie do obiektu utworzonego przez ten formularz. Tworzymy więc kolejną imitację dla widoku, czyli funkcję `redirect()`.

Plik `lists/tests/test_views.py` (ch19l015):

```
@patch('lists.views.redirect') #❶
def test_redirects_to_form_returned_object_if_form_valid(
    self, mock_redirect, mockNewListForm #❷
):
    mock_form = mockNewListForm.return_value
    mock_form.is_valid.return_value = True #❸

    response = new_list2(self.request)

    self.assertEqual(response, mock_redirect.return_value) #❹
    mock_redirect.assert_called_once_with(mock_form.save.return_value) #❺
```

- ❶ Tworzymy imitację funkcji `redirect()`. Tym razem to imitacja na poziomie metody.
- ❷ Dekorator `patch` będzie zastosowany jako pierwszy, a więc nowa imitacja zostanie wstrzyknięta do metody przed `mockNewListForm`.
- ❸ Określamy, że testowany jest przypadek, gdy formularz jest prawidłowy.
- ❹ Sprawdzamy, czy odpowiedzią udzieloną przez widok jest wynik wywołania funkcji `redirect()`.
- ❺ Sprawdzamy też, czy funkcja przekierowania została wywołana wraz z obiektem zwroconym przez formularz w trakcie operacji zapisu.

Docieramy do przedstawionego poniżej kodu.

Plik `lists/views.py` (ch19l016):

```
def new_list2(request):
    form = NewListForm(data=request.POST)
    list_ = form.save(owner=request.user)
    return redirect(list_)

$ python3 manage.py test lists
[...]
Ran 40 tests in 0.163s
OK
```

Przechodzimy do przypadku niepowodzenia — jeżeli formularz jest nieprawidłowy, chcemy wygenerowania szablonu strony głównej.

Plik `lists/tests/test_views.py` (ch19l017):

```
@patch('lists.views.render')
def test_renders_home_template_with_form_if_form_invalid(
    self, mock_render, mockNewListForm
):
    mock_form = mockNewListForm.return_value
    mock_form.is_valid.return_value = False

    response = new_list2(self.request)

    self.assertEqual(response, mock_render.return_value)
    mock_render.assert_called_once_with(
        self.request, 'home.html', {'form': mock_form}
    )
```

Wynikiem jest następujący komunikat błędu:

```
AssertionError: <django.http.response.HttpResponseRedirect object at  
0x7f8d3f338a50> != <MagicMock name='render()' id='140244627467408'>
```



Podczas użycia metod asercji w imitacji, na przykład `assert_called_once_with()`, szczególnie ważne jest upewnienie się o wykonaniu testu i sprawdzenie, jak wygląda jego niepowodzenie. Zbyt łatwo można popełnić błąd w nazwie funkcji asercji i wywołać metodę imitacji, która nic nie robi. W moim przypadku to było wywołanie `assert_called_once_with()` z trzema literami s w słowie assert. Spróbuj i przekonaj się sam.

W kodzie testu wprowadziliśmy celowy błąd, aby mieć pewność, że testy są dokładne.

Plik `lists/views.py` (ch19l018):

```
def new_list2(request):  
    form = NewListForm(data=request.POST)  
    list_ = form.save(owner=request.user)  
    if form.is_valid():  
        return redirect(list_)  
    return render(request, 'home.html', {'form': form})
```

Testy są zaliczone, choć nie powinny! Wykonujemy jeszcze jeden test.

Plik `lists/tests/test_views.py` (ch19l019):

```
def test_does_not_save_if_form_invalid(self, mockNewListForm):  
    mock_form = mockNewListForm.return_value  
    mock_form.is_valid.return_value = False  
    new_list2(self.request)  
    self.assertFalse(mock_form.save.called)
```

Ten test kończy się niepowodzeniem:

```
    self.assertFalse(mock_form.save.called)  
AssertionError: True is not false
```

Dochodzimy do eleganckiego, ukończonego widoku.

Plik `lists/views.py`:

```
def new_list2(request):  
    form = NewListForm(data=request.POST)  
    if form.is_valid():  
        list_ = form.save(owner=request.user)  
        return redirect(list_)  
    return render(request, 'home.html', {'form': form})  
  
...  
  
$ python3 manage.py test lists  
[...]  
Ran 42 tests in 0.163s  
OK
```

Przejście w dół do warstwy formularzy

Jak dotąd opracowaliśmy funkcję widoku opartą na nawet jeszcze nieistniejącej wersji formularza `NewItemForm`.

Metoda `save()` formularza jest potrzebna w celu utworzenia nowej listy i nowego elementu na podstawie testu zweryfikowanych danych POST formularza. Jeżeli mielibyśmy po prostu użyć ORM, wówczas kod mógłby być podobny do przedstawionego poniżej:

```
class NewListForm(models.Form):

    def save(self, owner):
        list_ = List()
        if owner:
            list_.owner = owner
        list_.save()
        item = Item()
        item.list = list_
        item.text = self.cleaned_data['text']
        item.save()
```

Powyższa implementacja zależy od dwóch klas warstwy modelu: `Item` i `List`. Jak będzie więc wyglądał odizolowany test?

```
class NewListFormTest(unittest.TestCase):

    @patch('lists.forms.List') #❶
    @patch('lists.forms.Item') #❷
    def test_saveCreatesNewListAndItemFromPostData(
        self, mockItem, mockList #❸
    ):
        mock_item = mockItem.return_value
        mock_list = mockList.return_value
        user = Mock()
        form = NewListForm(data={'text': 'tekst nowego elementu'})
        form.is_valid() #❹

        def check_item_text_and_list():
            self.assertEqual(mock_item.text, 'tekst nowego elementu')
            self.assertEqual(mock_item.list, mock_list)
            self.assertTrue(mock_list.save.called)
        mock_item.save.side_effect = check_item_text_and_list #❺

        form.save(owner=user)

        self.assertTrue(mock_item.save.called) #❻
```

- ❶❷❸ Tworzymy imitacje dwóch komponentów pochodzących z warstwy modelu i używanych przez formularz.
 - ❹ Konieczne jest wywołanie `is_valid()`, aby wypełnić formularz słownikiem `.cleaned_data`, w którym są przechowywane zweryfikowane dane.
 - ❺ Używamy metody `side_effect` w celu zagwarantowania, że w trakcie operacji zapisu obiektu nowego elementu tak naprawdę nastąpi zapis obiektu `List` wraz z odpowiednim tekstem elementu.
 - ❻ Jak zwykle dokładnie sprawdzamy, czy funkcja `side_effect` faktycznie została wywołana.
- Fuj! Jaki brzydki test!

Nadal słuchaj testów — usunięcie kodu ORM z aplikacji

Warto przypomnieć ponownie, że testy próbują nam przekazać pewną wiadomość: mechanizm ORM w Django trudno imitować, a klasa formularza ma zbyt wiele informacji o sposobie działania wspomnianego ORM. Czy programowanie oparte na myśleniu życzeniowym będzie skutkowało prostszym API, które będzie mogło być użyte przez formularz? Co sądzisz o poniższym fragmencie kodu?

```
def save(self):
    List.create_new(first_item_text=self.cleaned_data['text'])
```

Myślenie życzeniowe podpowiada: przygotuj metodę pomocniczą w klasie `List`¹, a będzie ona hermetyzowała całą logikę niezbędną do zapisu obiektu nowej listy i powiązanego z nią pierwszego elementu.

Przygotujemy test dla metody pomocniczej.

Plik `lists/tests/test_forms.py` (ch19l021):

```
import unittest
from unittest.mock import patch, Mock
from django.test import TestCase

from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_LIST_ERROR,
    ExistingListItemForm, ItemForm, NewListForm
)
from lists.models import Item, List
[...]

class NewListFormTest(unittest.TestCase):

    @patch('lists.forms.List.create_new')
    def test_saveCreatesNewListFromPostDataIfUserNotAuthenticated(
        self, mock_List_create_new
    ):
        user = Mock(is_authenticated=lambda: False)
        form = NewListForm(data={'text': 'tekst nowego elementu'})
        form.is_valid()
        form.save(owner=user)
        mock_List_create_new.assert_called_once_with(
            first_item_text='tekst nowego elementu'
        )
```

Przy okazji możemy też przetestować sytuację, gdy użytkownik jest uwierzytelniony.

Plik `lists/tests/test_forms.py` (ch19l022):

```
@patch('lists.forms.List.create_new')
def test_saveCreatesNewListWithOwnerIfUserAuthenticated(
    self, mock_List_create_new
):
    user = Mock(is_authenticated=lambda: True)
    form = NewListForm(data={'text': 'tekst nowego elementu'})
    form.is_valid()
    form.save(owner=user)
    mock_List_create_new.assert_called_once_with(
        first_item_text='tekst nowego elementu', owner=user
    )
```

¹ Równie dobrze mogłyby to być samodzielna funkcja. Jednak umieszczenie jej w klasie modelu oznacza, że znajduje się ona w odpowiednim miejscu, a ponadto daje wskazówki dotyczące przeznaczenia danej funkcji.

Jak możesz zobaczyć, powstał znacznie czytelniejszy test. Przystępujemy do implementacji naszego nowego formularza. Na początku mamy polecenie `import`.

Plik `lists/forms.py` (ch19l023):

```
from lists.models import Item, List
```

Teraz imitacja podpowiada, aby utworzyć miejsce zarezerwowane dla metody `create_new()`.

```
AttributeError: <class 'lists.models.List'> does not have the attribute  
'create_new'
```

Plik `lists/models.py`:

```
class List(models.Model):  
  
    def get_absolute_url(self):  
        return reverse('view_list', args=[self.id])  
  
    def create_new():  
        pass
```

Ukrycie kodu ORM w metodach pomocniczych

Jedną z technik powstających na podstawie użycia odizolowanych testów była „metoda pomocnicza ORM”.

Oferowany przez Django mechanizm ORM pozwala na szybkie wykonywanie zadań za pomocą względnie czytelnej składni (zdecydowanie bardziej eleganckiej niż zwykły kod SQL!). Jednak programiści dążą do minimalizacji ilości kodu ORM w aplikacji, w szczególności do jego usunięcia z warstw widoków i formularzy.

Jednym z powodów wspomnianego dążenia jest łatwiejsze testowanie tak uproszczonych warstw. Kolejny powód to wymuszenie tworzenia funkcji pomocniczych, które znacznie czytelniej wyrażają logikę domeny. Porównaj poniższy kod:

```
list_ = List()  
list_.save()  
item = Item()  
item.list = list_  
item.text = self.cleaned_data['text']  
item.save()
```

z następującym:

```
List.create_new(first_item_text=self.cleaned_data['text'])
```

Większa czytelność dotyczy zapytań zarówno odczytujących, jak i zapisujących dane. Spójrz na poniższy kod:

```
Book.objects.filter(in_print=True, pub_date__lte=datetime.today())
```

i porównaj go z metodą pomocniczą, na przykład:

```
Book.all_available_books()
```

Podczas tworzenia funkcji pomocniczych można im nadać nazwy wyrażające przeznaczenie w kategoriach domeny biznesowej. Dzięki temu kod w rzeczywistości staje się znacznie czytelniejszy, a ponadto pozwala nam na umieszczenie wszystkich wywołań ORM w warstwie modelu. W ten sposób poszczególne komponenty aplikacji są jeszcze luźniej powiązane.

Po kilku krokach powinniśmy mieć gotową metodę `save()` formularza, na przykład jak przedstawiono poniżej.

Plik `lists/forms.py` (ch19l025):

```
class NewListForm(ItemForm):

    def save(self, owner):
        if owner.is_authenticated():
            List.create_new(first_item_text=self.cleaned_data['text'], owner=owner)
        else:
            List.create_new(first_item_text=self.cleaned_data['text'])
```

Testy zostają zaliczone:

```
$ python3 manage.py test lists
Ran 44 tests in 0.192s
OK
```

Wreszcie przechodzimy w dół do warstwy modelu

W warstwie modelu nie musimy już tworzyć odizolowanych testów — celem warstwy modelu jest integracja z bazą danych, a więc odpowiednie wydaje się przygotowanie zintegrowanych testów.

Plik `lists/tests/test_models.py` (ch19l026):

```
class ListModelTest(TestCase):

    def test_get_absolute_url(self):
        list_ = List.objects.create()
        self.assertEqual(list_.get_absolute_url(), '/lists/%d/' % (list_.id,))

    def test_create_newCreatesListAndFirstItem(self):
        List.create_new(first_item_text='tekst nowego elementu')
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'tekst nowego elementu')
        new_list = List.objects.first()
        self.assertEqual(new_item.list, new_list)
```

Otrzymujemy następujący komunikat błędu:

```
TypeError: create_new() got an unexpected keyword argument 'first_item_text'
```

W ten sposób dochodzimy do pierwszej implementacji, która może przedstawiać się, jak pokazano poniżej.

Plik `lists/models.py` (ch19l027):

```
class List(models.Model):

    def get_absolute_url(self):
        return reverse('view_list', args=[self.id])

    @staticmethod
    def create_new(first_item_text):
        list_ = List.objects.create()
        Item.objects.create(text=first_item_text, list=list_)
```

Zwróć uwagę na możliwość poruszania się w dół aż do warstwy modelu, co stanowi eleganckie rozwiązanie dla warstw widoków i formularzy, choć model `List` nadal nie obsługuje przypisania właściciela.

Przetestujemy teraz przypadek, gdy lista powinna mieć właściciela. Dodajemy pokazany poniżej kod.

Plik *lists/tests/test_models.py* (ch19l028):

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

def test_create_new_optionally_saves_owner(self):
    user = User.objects.create()
    List.create_new(first_item_text='tekst nowego elementu', owner=user)
    new_list = List.objects.first()
    self.assertEqual(new_list.owner, user)
```

Przy okazji możemy przygotować testy dla nowego atrybutu właściciela.

Plik *lists/tests/test_models.py* (ch19l029):

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

    def test_lists_can_have_owners(self):
        List(owner=User()) # Nie powinien być zgłoszony.

    def test_list_owner_is_optional(self):
        List().full_clean() # Nie powinien być zgłoszony.
```

Dwa powyższe testy są niemal identyczne z testami użytymi w poprzednim rozdziale. Zmodyfikowałem je nieco, aby w rzeczywistości nie zapisywały obiektów — na potrzeby testu w zupełności wystarczające będzie umieszczenie obiektów w pamięci.



Użycie podczas testów przechowywanych w pamięci (niezapisanych) obiektów modelu pozwala na ich znacznie szybsze wykonywanie.

Otrzymujemy następujące dane wyjściowe:

```
$ python3 manage.py test lists
[...]
ERROR: test_create_new_optionally_saves_owner
TypeError: create_new() got an unexpected keyword argument 'owner'
[...]
ERROR: test_lists_can_have_owners (lists.tests.test_models.ListModelTest)
TypeError: 'owner' is an invalid keyword argument for this function
[...]
Ran 48 tests in 0.204s
FAILED (errors=2)
```

Implementujemy testy podobnie jak w poprzednim rozdziale.

Plik *lists/models.py* (ch19l030-1):

```
from django.conf import settings
[...]

class List(models.Model):
    owner = models.ForeignKey(settings.AUTH_USER_MODEL, blank=True, null=True)
    [...]
```

Aż do chwili przeprowadzenia migracji będziemy otrzymywać komunikaty o niepowodzeniu wynikające z różnego rodzaju niespójności danych:

```
django.db.utils.OperationalError: no such column: lists_list.owner_id  
FAILED (errors=28)
```

Przeprowadzenie migracji zmniejsza liczbę niepowodzeń do trzech:

```
ERROR: test_create_new_optionally_saves_owner  
TypeError: create_new() got an unexpected keyword argument 'owner'  
[...]  
ValueError: Cannot assign "<SimpleLazyObject:  
<django.contrib.auth.models.AnonymousUser object at 0x7f5b2380b4e0>>":  
"List.owner" must be a "User" instance.  
ValueError: Cannot assign "<SimpleLazyObject:  
<django.contrib.auth.models.AnonymousUser object at 0x7f5b237a12e8>>":  
"List.owner" must be a "User" instance.
```

Zajmiemy się teraz pierwszym, które jest związane z metodą `create_new()`.

Plik `lists/models.py` (ch19l030-3):

```
@staticmethod  
def create_new(first_item_text, owner=None):  
    list_ = List.objects.create(owner=owner)  
    Item.objects.create(text=first_item_text, list=list_)
```

Powrót do widoków

Dwa ze starych, zintegrowanych testów dla warstwy widoków kończą się niepowodzeniem. Co się dzieje?

```
ValueError: Cannot assign "<SimpleLazyObject:  
<django.contrib.auth.models.AnonymousUser object at 0x7fbad1cb6c10>>":  
"List.owner" must be a "User" instance.
```

Aha, stary widok nie ma wystarczających wskazówek, co powinien zrobić z właścicielem listy.

Plik `lists/views.py`:

```
if form.is_valid():  
    list_ = List()  
    list_.owner = request.user  
    list_.save()
```

Na tym etapie zdajemy sobie sprawę, że stary kod nie pasował zbytnio do naszego celu. Poprawimy go, aby wszystkie testy zostały zaliczone.

Plik `lists/views.py` (ch19l031):

```
def new_list(request):  
    form = ItemForm(data=request.POST)  
    if form.is_valid():  
        list_ = List()  
        if request.user.is_authenticated():  
            list_.owner = request.user  
        list_.save()  
        form.save(for_list=list_)  
        return redirect(list_)  
    else:  
        return render(request, 'home.html', {"form": form})  
  
def new_list2(request):  
    [...]
```



Jedną z zalet testów zintegrowanych jest to, że mogą pomóc w wychwyceniu mniej przewidywalnych interakcji, na przykład jak przedstawiona. Zapomnieliśmy przygotować test dla sytuacji, gdy użytkownik nie jest uwierzytelniony. Ponieważ jednak testy zintegrowane używają całego stosu aż do początku, wypłyną błędy znalezione w warstwie modelu. W ten sposób zostaniemy poinformowani, że o czymś zapomnieliśmy:

```
$ python3 manage.py test lists  
[...]  
Ran 48 tests in 0.175s  
OK
```

Moment prawdy (i ryzyko związane z imitacjami)

Spróbujemy teraz wyłączyć stary widok i aktywować nowy. Zamianę można przeprowadzić w pliku *urls.py*.

Plik *lists/urls.py*:

```
[...]  
url(r'^new$', 'lists.views.new_list2', name='new_list'),
```

Należy również usunąć `unittest.skip` z klasy zintegrowanych testów i wskazać jej nasz nowy widok (`new_list2`), aby przekonać się, czy kod odpowiedzialny za przypisywanie właścicieli faktycznie działa.

Plik *lists/tests/test_views.py* (ch19l033):

```
class NewListViewIntegratedTest(TestCase):  
  
    def test_saving_a_POST_request(self):  
        [...]  
  
    def test_list_owner_is_saved_if_user_is_authenticated(self):  
        request = HttpRequest()  
        request.user = User.objects.create(email='a@b.com')  
        request.POST['text'] = 'nowy element listy'  
        new_list2(request)  
        list_ = List.objects.first()  
        self.assertEqual(list_.owner, request.user)
```

Jaki będzie wynik wykonania testów? O nie!

```
ERROR: test_list_owner_is_saved_if_user_is_authenticated  
[...]  
ERROR: test_saving_a_POST_request  
[...]  
ERROR: test_redirects_after_POST  
(lists.tests.test_views.NewListViewIntegratedTest)  
  File "/workspace/superlists/lists/views.py", line 30, in new_list2  
    return redirect(list_)  
[...]  
TypeError: argument of type 'NoneType' is not iterable  
  
FAILED (errors=3)
```

Oto niezwykle ważna lekcja dotycząca izolacji testu: podejście może okazać się pomocne w przygotowaniu dobrego projektu dla poszczególnych warstw, ale na pewno nie przeprowadzi automatycznej weryfikacji integracji między poszczególnymi warstwami.

W omawianym przypadku widok oczekuje, że wartością zwrotną formularza będzie element listy.

Plik *lists/views.py*:

```
list_ = form.save(owner=request.user)
return redirect(list_)
```

My jednak zapomnieliśmy o zwrocie czegokolwiek.

Plik *lists/forms.py*:

```
def save(self, owner):
    if owner.is_authenticated():
        List.create_new(first_item_text=self.cleaned_data['text'], owner=owner)
    else:
        List.create_new(first_item_text=self.cleaned_data['text'])
```

Potraktowanie interakcji między warstwami jak kontraktów

Ostatecznie nawet jeśli będziemy tworzyć tylko odizolowane testy jednostkowe, to testy funkcjonalne będą w stanie wychwycić powyższą wpadkę. Idealnym rozwiązaniem byłoby szybko dowiadywać się o tego rodzaju problemach — wykonanie testów funkcjonalnych może zająć kilka minut lub nawet godzin, gdy aplikacja zostanie znacznie rozbudowana. Czy istnieje jakikolwiek sposób na uniknięcie wspomnianych wcześniej problemów, zanim jeszcze się pojawią?

Pod względem metodologicznym rozwiązanie polega na potraktowaniu interakcji między warstwami jak kontraktów. Kiedy imitujemy pewne zachowanie w jednej warstwie, trzeba pamiętać o powstaniu niejawnego połączenia między warstwami. Wspomniana imitacja w jednej warstwie prawdopodobnie powinna przełożyć się na test w warstwie znajdującej się poniżej.

Oto fragment kontraktu, o którym zapomnieliśmy.

Plik *lists/tests/test_views.py*:

```
@patch('lists.views.redirect')
def test_redirects_to_form_returned_object_if_form_valid(
    self, mock_redirect, mockNewListForm
):
    mock_form = mockNewListForm.return_value
    mock_form.is_valid.return_value = True

    response = new_list2(self.request)

    self.assertEqual(response, mock_redirect.return_value)
    mock_redirect.assert_called_once_with(mock_form.save.return_value) #❶
```

- ❶ Imitacja funkcji `form.save()` zwraca obiekt, który jest przeznaczony do użycia w widoku.

Identyfikacja niejawnych kontraktów

Warto przejrzeć poszczególne testy w `NewListViewUnitTest` i sprawdzić, która imitacja wskazuje na istnienie niejawnego kontraktu.

Plik `lists/tests/test_views.py`:

```
def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
    [...]
    mockNewListForm.assert_called_once_with(data=self.request.POST) #❶

def test_saves_form_with_owner_if_form_valid(self, mockNewListForm):
    mock_form = mockNewListForm.return_value
    mock_form.is_valid.return_value = True #❷
    new_list2(self.request)
    mock_form.save.assert_called_once_with(owner=self.request.user) #❸

def test_does_not_save_if_form_invalid(self, mockNewListForm):
    [...]
    mock_form.is_valid.return_value = False #❹
    [...]

@patch('lists.views.redirect')
def test_redirects_to_form_returned_object_if_form_valid(
    self, mock_redirect, mockNewListForm
):
    [...]
    mock_redirect.assert_called_once_with(mock_form.save.return_value) #❺

def test_renders_home_template_with_form_if_form_invalid(
    [...]
```

- ❶ Inicjacja formularza powinna nastąpić przez przekazanie mu danych w postaci żądania POST.
- ❷❸ Formularz powinien mieć funkcję `is_valid()` zwracającą wartość `True` lub `False`, w zależności od danych wejściowych.
- ❹ Formularz powinien mieć także metodę `.save()` akceptującą `request.user`. Użytkownik może, ale nie musi być zalogowany, a zadaniem formularza jest jego odpowiednie obsłużenie.
- ❺ Wartością zwrotną metody `.save()` powinien być obiekt nowej listy, do której zostanie przekierowany użytkownik.

Jeżeli dokładnie przeanalizujesz przygotowane przez nas testy formularza, to zobacysz, że tylko punkt 3. jest wyraźnie testowany. W przypadku punktów 1. i 2. mamy szczęście, ponieważ są to funkcje domyślne `ModelForm` w Django i będą sprawdzone przez testy przeznaczone dla klasy nadzędnej `ItemForm`.

Jednak punkt 4. prześlizguje się przez сито, to znaczy pozostaje nieprzetestowany.



Kiedy w TDD stosujesz podejście Outside-In wraz z odizolowanymi testami, wówczas dla poszczególnych testów powinieneś monitorować niejawne założenia dotyczące kontraktu, które powinny być implementowane przez kolejną warstwę. Pamiętaj też o ich kolejnym przetestowaniu później. Dobrym rozwiązaniem będzie sporządzenie odpowiednich notatek w zeszycie lub też utworzenie miejsca zarezerwowanego dla testu wraz z wywołaniem `self.fail()`.

Usunięcie przeoczonego problemu

Dodajemy nowy test sprawdzający, czy formularz zwraca nowo zapisaną listę.

Plik *lists/tests/test_forms.py* (ch19l038-1):

```
@patch('lists.forms.List.create_new')
def test_save_returns_new_list_object(self, mock_List_create_new):
    user = Mock(is_authenticated=lambda: True)
    form = NewListForm(data={'text': 'tekst nowego elementu'})
    form.is_valid()
    response = form.save(owner=user)
    self.assertEqual(response, mock_List_create_new.return_value)
```

Powyżej przedstawiono całkiem dobry przykład — mamy niejawny kontrakt z `List.create_new` i chcemy, aby wymienione wywołanie zwracało obiekt nowej listy. Dodajemy miejsce zarezerwowane dla testu.

Plik *lists/tests/test_models.py* (ch19l038-2):

```
class ListModelTest(TestCase):
    [...]
    def test_create_returns_new_list_object(self):
        self.fail()
```

Otrzymujemy niepowodzenie testu wraz z komunikatem informującym o potrzebie poprawy funkcji zapisu formularza:

```
AssertionError: None != <MagicMock name='create_new()' id='139802647565536'>
FAILED (failures=2, errors=3)
```

Możemy wprowadzić przedstawiony poniżej kod.

Plik *lists/forms.py* (ch19l039-1):

```
class NewListForm(ItemForm):

    def save(self, owner):
        if owner.is_authenticated():
            return List.create_new(first_item_text=self.cleaned_data['text'],
                                  owner=owner)
        else:
            return List.create_new(first_item_text=self.cleaned_data['text'])
```

To dopiero początek. Teraz przechodzimy do miejsca zarezerwowanego dla testu:

```
[...]
FAIL: test_create_returns_new_list_object
      self.fail()
AssertionError: None
FAILED (failures=1, errors=3)
```

Podpieramy się nim.

Plik *lists/tests/test_models.py* (ch19l039-2):

```
def test_create_returns_new_list_object(self):
    returned = List.create_new(first_item_text='tekst nowego elementu')
    new_list = List.objects.first()
    self.assertEqual(returned, new_list)

...
AssertionError: None != <List: List object>
```

I możemy dodać wartość zwrotną.

Plik *lists/models.py* (ch19l039-3):

```
@staticmethod
def create_new(first_item_text, owner=None):
    list_ = List.objects.create(owner=owner)
    Item.objects.create(text=first_item_text, list=list_)
    return list_
```

W ten sposób testy zostają zaliczone:

```
$ python3 manage.py test lists
[...]
Ran 50 tests in 0.169s

OK
```

Jeszcze jeden test

Opierając się na testach, opracowaliśmy kod przeznaczony do zapisu właściciela listy. Jednak test funkcjonalny nie jest jeszcze zaliczony:

```
$ python3 manage.py test functional_tests.test_my_lists
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"link text","selector":"Przygotować siatkę"}' ; Stacktrace:
```

Powodem niepowodzenia jest konieczność zaimplementowania ostatniej funkcji, czyli atrybutu `.name` w obiektach list. Pracę ponownie zaczynamy od pobrania testu i kodu z poprzedniego rozdziału.

Plik *lists/tests/test_models.py* (ch19l040):

```
def test_list_name_is_first_item_text(self):
    list_ = List.objects.create()
    Item.objects.create(list=list_, text='pierwszy element')
    Item.objects.create(list=list_, text='drugi element')
    self.assertEqual(list_.name, 'pierwszy element')
```

(Warto przypomnieć raz jeszcze — to jest test w warstwie modelu, a więc można użyć ORM. Wprawdzie można sobie wyobrazić przygotowanie testu opartego na imitacji, ale nie ma takiej potrzeby).

Plik *lists/models.py* (ch19l041):

```
@property
def name(self):
    return self.item_set.first().text
```

Test funkcjonalny wreszcie zostaje zaliczony!

```
$ python3 manage.py test functional_tests.test_my_lists
Ran 1 test in 21.428s

OK
```

Porządkowanie, czyli co zachować z pakietu testów zintegrowanych?

Skoro wszystko działa, to możemy usunąć pewne powielone testy i zdecydować, co powinno pozostać ze starych testów zintegrowanych.

Usunięcie powielonego kodu w warstwie formularzy

Możemy się pozbyć testu dla starej metody zapisu w ItemForm.

Plik *lists/tests/test_forms.py*:

```
--- a/lists/tests/test_forms.py
+++ b/lists/tests/test_forms.py
@@ -23,14 +23,6 @@ class ItemFormTest(TestCase):

    self.assertEqual(form.errors['text'], [EMPTY_LIST_ERROR])

-   def test_form_save_handles_saving_to_a_list(self):
-       list_ = List.objects.create()
-       form = ItemForm(data={'text': 'dowolne zadanie'})
-       new_item = form.save(for_list=list_)
-       self.assertEqual(new_item, Item.objects.first())
-       self.assertEqual(new_item.text, 'dowolne zadanie')
-       self.assertEqual(new_item.list, list_)
```

W rzeczywistym kodzie można się także pozbyć dwóch zbędnych metod zapisu zdefiniowanych w pliku *forms.py*.

Plik *lists/forms.py*:

```
--- a/lists/forms.py
+++ b/lists/forms.py
@@ -22,11 +22,6 @@ class ItemForm(forms.ModelForm):

    self.fields['text'].error_messages['required'] = EMPTY_LIST_ERROR

-   def save(self, for_list):
-       self.instance.list = for_list
-       return super().save()
-
-
 class NewListForm(ItemForm):

@@ -52,8 +47,3 @@ class ExistingListItemForm(ItemForm):

    e.error_dict = {'text': [DUPLICATE_ITEM_ERROR]}
    self._update_errors(e)
-
-
-   def save(self):
-       return forms.ModelForm.save(self)
-
```

Usunięcie starej implementacji widoku

W tym momencie możemy zupełnie usunąć stary widok `new_list`, a następnie nazwę widoku `new_list2` zmienić na `new_list`.

Plik `lists/tests/test_views.py`:

```
-from lists.views import new_list, new_list2
+from lists.views import new_list

class HomePageTest(TestCase):
@@ -75,7 +75,7 @@ class NewListViewIntegratedTest(TestCase):
    request = HttpRequest()
    request.user = User.objects.create(email='a@b.com')
    request.POST['text'] = 'nowy element listy'
-    new_list2(request)
+    new_list(request)
    list_ = List.objects.first()
    self.assertEqual(list_.owner, request.user)

@@ -91,21 +91,21 @@ class NewListViewUnitTest(unittest.TestCase):

    def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
-        new_list2(self.request)
+        new_list(self.request)

[.. kilka dalszych]
```

Plik `lists/urls.py`:

```
--- a/lists/urls.py
+++ b/lists/urls.py
@@ -2,6 +2,6 @@ from django.conf.urls import patterns, url

 urlpatterns = patterns('',
     url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
-    url(r'^new$', 'lists.views.new_list2', name='new_list'),
+    url(r'^new$', 'lists.views.new_list', name='new_list'),
     url(r'^users/(.+)$', 'lists.views.my_lists', name='my_lists'),
 )
```

Plik `lists/views.py` (ch19l047):

```
def new_list(request):
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user)
    [...]
```

Wykonaj testy, aby sprawdzić, czy wszystkie nadal są zaliczane:

OK

Usunięcie zbędnego kodu w warstwie formularzy

Na końcu musimy zdecydować, co (o ile cokolwiek) zachować z pakietu testów zintegrowanych.

Jedną z możliwości jest wyrzucenie ich wszystkich i poleganie na tym, że testy funkcjonalne będą w stanie wychwycić wszelkie problemy związane z integracją. Takie podejście jest jak najbardziej poprawne.

Z drugiej strony widziałeś, że testy zintegrowane mogą ostrzec, gdy popełnisz nawet niewielką pomyłkę w zintegrowanych warstwach. Można więc zachować kilka testów „na wszelki wypadek”, co skróci czas oczekiwania na wynik wykonania testów.

Proponuję pozostawić trzy wymienione poniżej testy.

Plik *lists/tests/test_views.py* (ch19l048):

```
class NewListViewIntegratedTest(TestCase):

    def test_saving_a_POST_request(self):
        self.client.post(
            '/lists/new',
            data={'text': 'nowy element listy'}
        )
        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'nowy element listy')

    def test_for_invalid_input_doesnt_save_but_shows_errors(self):
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertEqual(List.objects.count(), 0)
        self.assertContains(response, escape(EMPTY_LIST_ERROR))

    def test_saves_list_owner_if_user_logged_in(self):
        request = HttpRequest()
        request.user = User.objects.create(email='a@b.com')
        request.POST['text'] = 'nowy element listy'
        new_list(request)
        list_ = List.objects.first()
        self.assertEqual(list_.owner, request.user)
```

Jeżeli w ogóle zdecydujesz się na zachowanie jakichkolwiek testów warstw pośrednich, wówczas wybór trzech powyższych testów jest dobry, ponieważ sprawdzają one większość zadań „integracji” — testują pełny stos, począwszy od żądania, aż po rzeczywistą bazę danych, a ponadto sprawdzają trzy najważniejsze przypadki użycia naszego widoku.

Podsumowanie

— testy odizolowane kontra zintegrowane

Narzędzie testowe Django niezwykle łatwo pozwala przygotować testy zintegrowane. Silnik testów utworzy w pamięci szybko działającą wersję bazy danych i będzie ją zerował pomiędzy poszczególnymi testami. Klasa `TestCase` i klient testów ułatwiają testowanie widoków, sprawdzenie, czy zmodyfikowano obiekty bazy danych, potwierdzenie działania mapowania adresów URL, a także analizę generowania szablonów. Domyślnie oferowane możliwości Django pomagają w łatwym rozpoczęciu stosowania testów, które będą sprawdzać cały stos wywołań w aplikacji.

Z drugiej strony, wspomniane testy zintegrowane niekoniecznie oferują wszystkie korzyści, jakie można osiągnąć za pomocą rygorystycznych testów jednostkowych i podejścia `Outside-In` w kategoriach projektu.

Spójrz na przykład omawiany w rozdziale, a następnie porównaj jego wersje przed i po zmianie.

Przed zmianą.

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List()
        if not isinstance(request.user, AnonymousUser):
            list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

Po zmianie.

```
def new_list(request):
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user)
        return redirect(list_)
    return render(request, 'home.html', {'form': form})
```

Jeżeli nie zdecydowalibyśmy się na odizolowanie testu, czy zajelibyśmy się refaktoryzacją funkcji widoku? W pierwszej wersji rozdziału nie zdecydowałem się na to. Chciałbym powiedzieć, że zdecydowałbym się „w rzeczywistym projekcie”, ale tak naprawdę wcale nie mam pewności. Tworzenie testów odizolowanych pozwala zobaczyć, które fragmenty kodu są skomplikowane.

Niech poziom skomplikowania będzie Twoim przewodnikiem

Uważam, że zwiększenie się stopnia skomplikowania aplikacji to punkt, w którym tworzenie testów odizolowanych warte jest wysiłku. Przykład przedstawiony w książce jest wyjątkowo prosty, w takich sytuacjach często nie warto ponosić wspomnianego wysiłku. Nawet w przypadku przykładu przedstawionego w tym rozdziale mogę sobie powiedzieć, że tak naprawdę nie potrzebuję testów odizolowanych.

Jednak gdy aplikacja stanie się nieco bardziej skomplikowana, zaczyna pojawiać się dodatkowe warstwy między widokami i modelami, a Ty złapiesz się na tworzeniu metod pomocniczych i własnych klas, wtedy prawdopodobnie przygotowanie testów odizolowanych będzie miało sens i będzie z korzyścią dla projektu.

Czy powiniensem tworzyć oba rodzaje testów?

Mamy opracowany zestaw testów funkcyjnych, których celem jest poinformowanie o pełnieniu wszelkich pomyłek w trakcie integracji poszczególnych fragmentów kodu. Tworzenie testów odizolowanych może pomóc w przygotowaniu lepszej jakości kodu, a także w trakcie sprawdzania poprawności na większym poziomie dokładności. Czy dodatkowa warstwa testów integracji będzie służyła jakimkolwiek dodatkowym celom?

Sądzę, że odpowiedź brzmi: potencjalnie tak, o ile zapewnią krótszy cykl dostarczania informacji oraz pomogą we wskazywaniu pojawiających się problemów integracji. Dane generowane przez wspomniane testy mogą dostarczać czytelniejszych informacji o problemach niż te, które uzyskujesz na przykład z testu funkcyjnego.

Może się nawet znaleźć powód do umieszczenia testów odizolowanych w oddzielnym zestawie. Tego rodzaju zestaw szybko wykonywanych, odizolowanych testów jednostkowych nie wymaga nawet użycia `manage.py`, ponieważ nie przeprowadzają one żadnych operacji porządkowych

w bazie danych. Zawsze masz metodę `tearDown()` udostępnianą przez silnik testów Django oraz warstwę testów funkcjonalnych, które komunikują się z serwerem prowizorycznym. Zastosowanie testów może być warte wysiłku, gdy osiągniesz korzyści w poszczególnych warstwach.

To jest tylko moje zdanie. Mam nadzieję, że materiał przedstawiony w rozdziale wyraźnie zaprezentował wady i zalety testów odizolowanych.

Do przodu!

Jesteśmy zadowoleni z nowej wersji, więc warto ją umieścić w gałęzi master:

```
$ git add .
$ git commit -m"Dodanie za pomocą formularza właściciela listy. Więcej odizolowanych testów."
$ git checkout master
$ git checkout -b master-noforms-noisolation-bak # Opcjonalna kopia zapasowa.
$ git checkout master
$ git reset --hard more-isolation # Gałąź master zawiera teraz kod opracowany w naszej gałęzi.
```

Wykonanie tych testów funkcjonalnych zabiera irytująco dużo czasu. Zastanawiam się, czy istnieje jakiekolwiek rozwiązanie tego problemu?

Wady i zalety różnego typu testów oraz usunięcia kodu ORM z aplikacji

Testy funkcjonalne

- Z punktu widzenia użytkownika są najlepszą gwarancją, że aplikacja naprawdę działa prawidłowo.
- Niestety oznaczają wolniejszy cykl zgłaszania informacji o problemach.
- Ponadto niekoniecznie pomagają w tworzeniu przejrzystego kodu.

Testy zintegrowane (oparte na ORM lub kliencie testów Django)

- Można je szybko przygotować.
- Są łatwe do zrozumienia.
- Ostrzegają o wszelkich problemach związanych z integracją.
- Nie zawsze jednak prowadzą do powstania dobrego projektu (to zależy od Ciebie!).
- Zwykle działają wolniej niż testy odizolowane.

Testy odizolowane (stosujące imitacje)

- Wymagają największej ilości pracy.
- Mogą być trudniejsze w odczycie i zrozumieniu.
- Są jednak najlepsze, jeśli chcesz przygotować lepszy projekt.
- Są wykonywane najszybciej.

Usunięcie kodu ORM z aplikacji

Kiedy usiłujesz przygotować testy odizolowane, jednym z wymogów jest konieczność usunięcia kodu ORM z miejsc takich jak widoki i formularze. Wspomniany kod musi być umieszczony w funkcjach pomocniczych lub metodach. Zaletą na pewno jest brak powiązania aplikacji z kodem ORM, a ponadto kod aplikacji staje się znacznie czytelniejszy. Podobnie jak w innych sytuacjach zawsze należy się zastanowić, czy dodatkowy wysiłek będzie wart uzyskanego efektu końcowego.

Ciągła integracja

Wraz z rozbudowywaniem witryny nieubłaganie wydłuża się także czas potrzebny na wykonanie wszystkich testów funkcjonalnych. Jeżeli taka sytuacja się utrzyma, wówczas istnieje niebezpieczeństwo, że przestaniemy się nimi przejmować.

Zamiast pozwolić na ziszczenie się takiego scenariusza, lepiej zautomatyzować wykonywanie testów funkcjonalnych przez konfigurację „ciągłej integracji” (ang. *continuous integration*) lub też serwera ciągłej integracji. W ten sposób podczas codziennej pracy nad aplikacją można wykonać tylko testy funkcjonalne dla aktualnie opracowywanej funkcjonalności, natomiast serwerowi ciągłej integracji pozostawić automatyczne wykonywanie wszystkich testów i informowanie nas o potencjalnych problemach. Testy jednostkowe powinny być na tyle szybkie, by można było sobie pozwolić na ich wykonywanie co kilka sekund.

Obecnie dość często wybieranym serwerem ciągłej integracji jest Jenkins. Choć serwer Jenkins został utworzony w Javie, czasem ulega awarii i jest nieatrakcyjny wizualnie, to i tak ma ogromną rzeszę użytkowników oraz doskonały ekosystem wtyczek. Dlatego też zdecydowałem się na jego użycie.

Instalacja serwera Jenkins

Dostępnych jest wiele usług hostingowych oferujących ciągłą integrację, które w takim przypadku najczęściej dostarczają gotowy do użycia serwer Jenkins. Wypróbowałem Sauce Labs, Travis, Circle-CI, ShiningPanda, ale znajdziesz jeszcze wiele innych. Tutaj przyjąłem założenie, że instalujemy wszystko w serwerze pozostającym pod naszą kontrolą.



Nie jest dobrym pomysłem instalacja Jenkins w tym samym serwerze, w którym mamy serwer prowizoryczny lub produkcyjny. Pomijając inne wady takiego rozwiązania, trzeba również pamiętać, że Jenkins powinien mieć możliwość ponownego uruchomienia serwera prowizorycznego.

Zainstalujemy najnowszą wersję Jenkins z oficjalnego repozytorium, ponieważ wersja dostępna dla Ubuntu ma kilka irytujących błędów związanych z obsługą Unicode, a także domyślnie nie jest skonfigurowana do nasłuchiwanego publicznego internetu.

```
# Polecenia pobrane z witryny Jenkins.  
użytkownik@serwer:$ wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key |\\  
    sudo apt-key add -  
użytkownik@serwer:$ echo deb http://pkg.jenkins-ci.org/debian binary/ | sudo tee \\  
    /etc/apt/sources.list.d/jenkins.list  
użytkownik@serwer:$ sudo apt-get update  
użytkownik@serwer:$ sudo apt-get install jenkins
```

Przy okazji instalujemy jeszcze kilka innych zależności:

```
użytkownik@serwer:$ sudo apt-get install git firefox python3 python-virtualenv xvfb
```



W czasie pisania książki wtyczka shiningpanda pozostawała *niezgodna*¹ z Pythonem 3.4². Działa jednak doskonale z Pythonem 3.3 i dlatego zalecam użycie nieco starszej dystrybucji Ubuntu, w której domyślana wersja Pythona 3 jest nieco starsza. Ubuntu Saucy (13.10) sprawdza się doskonale, natomiast wydanie Trusty (14.04) już nie.

Powinieneś mieć możliwość przejścia do adresu URL serwera na porcie 8080, jak pokazano na rysunku 20.1.

A screenshot of a Mozilla Firefox browser window showing the Jenkins dashboard. The title bar says "Dashboard [Jenkins] - Mozilla Firefox". The address bar shows "jenkins.ottg.eu:8080". The main content area displays the Jenkins logo and the message "Witamy w Jenkins!". It includes links for "New Item", "Ludzie", "Historia zadań", "Manage Jenkins", and "Credentials". Below these are sections for "Kolejka Budowania" (empty) and "Stan Wykonawcy Zadań" (two Bezzczynny tasks). A note at the bottom encourages users to "utworzyć nowe zadanie". The footer contains links for "Pomóż nam przetłumaczyć stronę", "Strona wygenerowana: 2015-04-26 17:34:21", "REST API", and "Jenkins ver. 1.610".

Rysunek 20.1. Kamerdyner? To takie staroświeckie...

¹ <https://issues.jenkins-ci.org/browse/JENKINS-22902>

² Autor wtyczki wydał jej poprawioną wersję zgodną z Pythonem 3.4 — przyp. tłum.

Konfiguracja zabezpieczeń w Jenkins

Pierwszym krokiem jest konfiguracja uwierzytelnienia, ponieważ nasz serwer jest dostępny w publicznym internecie:

- Wybierz *Manage Jenkins/Configure Global Security/Enable security*.
- Wybierz opcje *Jenkins' own user database* i *Matrix-based security*.
- Wyłącz wszystkie uprawnienia dla użytkownika anonimowego (*Anonymous*).
- Dodaj użytkownika dla siebie i nadaj mu wszystkie uprawnienia (patrz rysunek 20.2).

Rysunek 20.2. Zmiana uprawnień użytkowników...

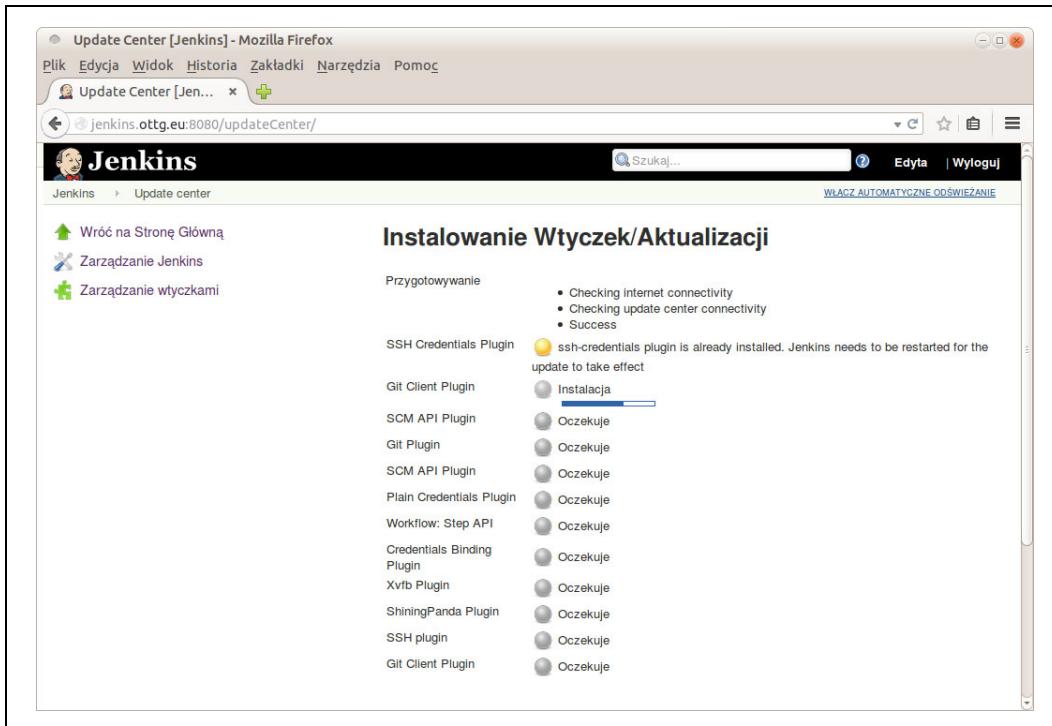
- Na kolejnym ekranie będziesz miał możliwość utworzenia konta o nazwie dopasowanej do użytkownika i przypisania mu hasła³.

Dodanie wymaganych wtyczek

Kolejnym krokiem jest instalacja kilku wtyczek ułatwiających pracę z Git, Pythonem i wirtualnymi ekranami (patrz rysunek 20.3):

- Wybierz opcję *Manage Jenkins/Zarządzaj dodatkami/Dostępne*.

³ Jeżeli pominiesz ten ekran, nadal możesz się „zarejestrować”, o ile użyjesz podanej wcześniej nazwy użytkownika. Odpowiednie konto będzie już skonfigurowane.



Rysunek 20.3. Instalacja wtyczek...

Potrzebne są nam wtyczki:

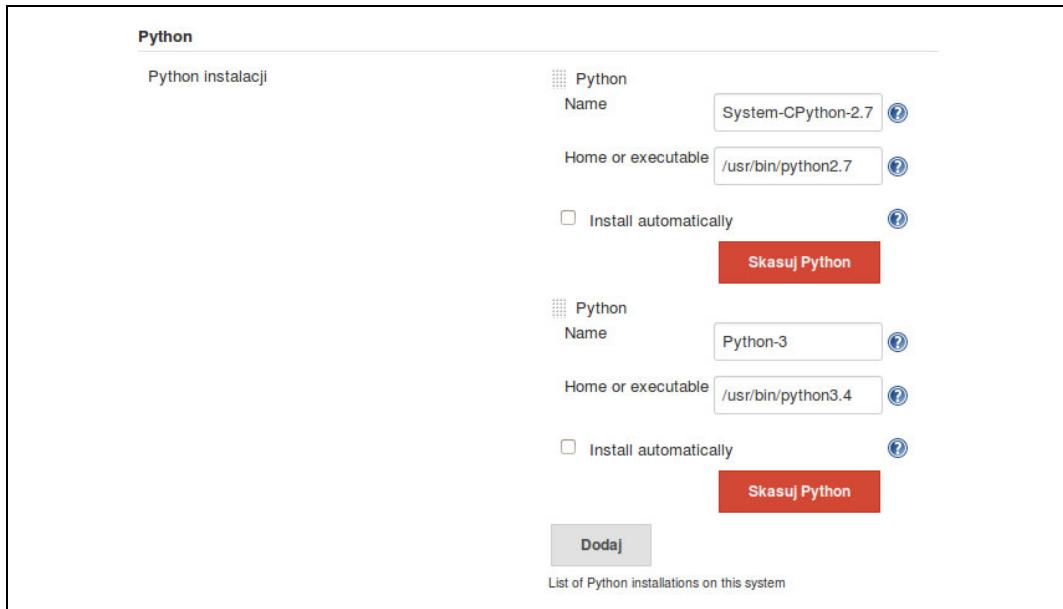
- Git,
- ShiningPanda,
- Xvfb.

Po przeprowadzeniu instalacji wtyczek ponownie uruchom serwer Jenkins za pomocą przycisku wyświetlanego na ostatnim ekranie lub wydając następujące polecenie poziomu powłoki: `sudo service jenkins restart`.

Poinformuj Jenkins, gdzie znajdzie Pythona 3 i Xvfb

Wtyczce ShiningPanda trzeba wskazać położenie Pythona 3 (to najczęściej katalog `/usr/bin/python3`, ale możesz to sprawdzić, wydając polecenie `which python3`) — wybierz *Manage Jenkins/Skonfiguruj system*.

- *Python/Python instalacji/Dodaj* (patrz rysunek 20.4).
- *Xvfb instalacji/Dodaj* — jako katalog instalacyjny podaj `/usr/bin`.

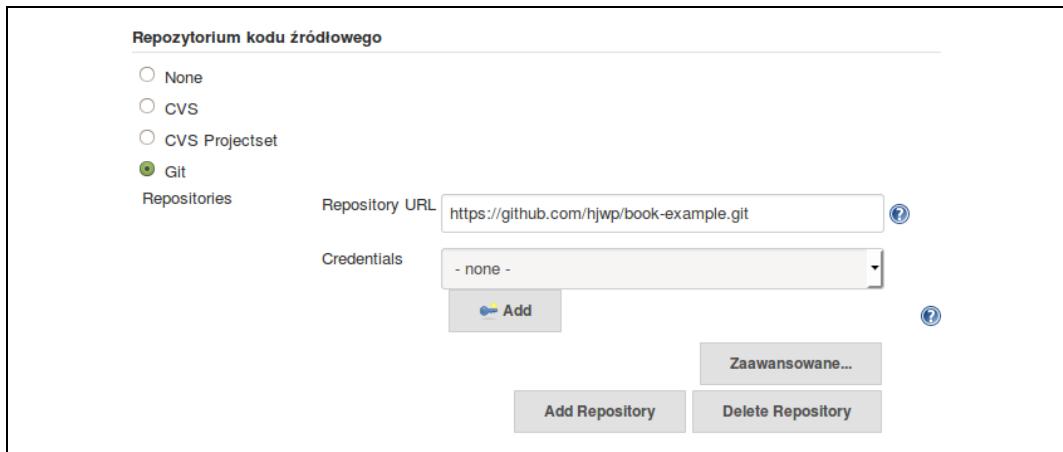


Rysunek 20.4. Gdzie się podzielił Python?

Konfiguracja projektu

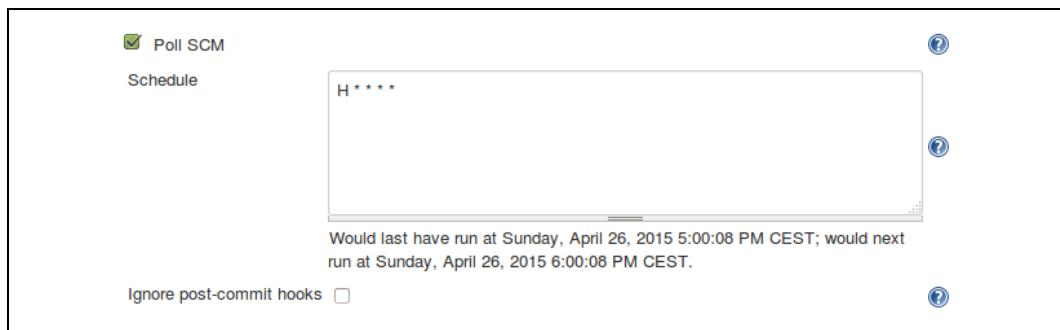
Po przeprowadzeniu podstawowej konfiguracji Jenkins możemy przejść do przygotowania projektu:

- Wybierz *New Item/Freestyle*.
- Dodaj repozytorium Git, jak pokazano na rysunku 20.5.



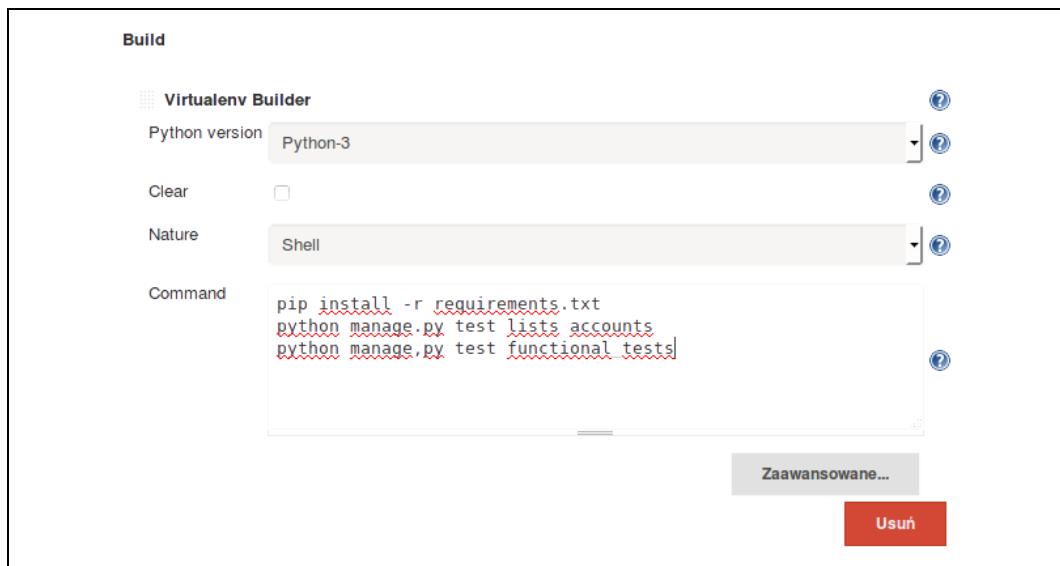
Rysunek 20.5. Pobranie kodu z repozytorium Git

- Zdefiniuj pobieranie kodu co godzinę (patrz rysunek 20.6). Sprawdź tekst pomocy, ponieważ istnieje wiele innych możliwości inicjowania komplikacji kodu.



Rysunek 20.6. Sprawdzenie w GitHub, czy pojawiły się zmiany w kodzie

- Uruchom testy wewnętrz środowiska wirtualnego Pythona 3.
- Oddzielnie wykonaj testy jednostkowe i funkcjonalne (patrz rysunek 20.7).



Rysunek 20.7. Kolejne kroki komplikacji w środowisku wirtualnym

Pierwsza komplikacja

Kliknij przycisk *Build Now*, a następnie spójrz na okno *Console Output*. Powinieneś zobaczyć dane wyjściowe podobne do przedstawionych poniżej:

```
Started by user harry
Building in workspace /var/lib/jenkins/jobs/Superlists/workspace
Fetching changes from the remote Git repository
Fetching upstream changes from https://github.com/hjwp/book-example.git
Checking out Revision d515acebf7e173f165ce713b30295a4a6ee17c07 (origin/master)
```

```
[workspace] $ /bin/sh -xe /tmp/shiningpanda7260707941304155464.sh
+ pip install -r requirements.txt
Requirement already satisfied (use --upgrade to upgrade): Django==1.7 in
/var/lib/jenkins/shiningpanda/jobs/ddclaed1/virtualenvs/d41d8cd9/lib/python3.3/site-packages
(from -r requirements.txt (line 1))
Downloading/unpacking South==0.8.2 (from -r requirements.txt (line 2))
    Running setup.py egg_info for package South

Requirement already satisfied (use --upgrade to upgrade): gunicorn==17.5 in
/var/lib/jenkins/shiningpanda/jobs/ddclaed1/virtualenvs/d41d8cd9/lib/python3.3/site-packages
(from -r requirements.txt (line 3))
Downloading/unpacking requests==2.0.0 (from -r requirements.txt (line 4))
    Running setup.py egg_info for package requests

Installing collected packages: South, requests
    Running setup.py install for South

    Running setup.py install for requests

Successfully installed South requests
Cleaning up...
+ python manage.py test lists accounts
.....
-----
Ran 51 tests in 0.323s

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...
+ python manage.py test functional_tests
ImportError: No module named 'selenium'
Build step 'Virtualenv Builder' marked build as failure
```

Ach. W środowisku wirtualnym potrzebujemy jeszcze Selenium.

Dodajemy teraz ręcznie Selenium⁴:

```
pip install -r requirements.txt
pip install selenium==2.39
python manage.py test accounts lists
python manage.py test functional_tests
```



Niektórzy lubią użyć pliku o nazwie *test-requirements.txt* przeznaczonego do wskazania pakietów niezbędnych dla testów, a nie dla głównej aplikacji.

Co teraz?

```
File
"/var/lib/jenkins/shiningpanda/jobs/ddclaed1/virtualenvs/d41d8cd9/lib/python3.
line 100, in _wait_until_connectable
    self._get_firefox_output())
selenium.common.exceptions.WebDriverException: Message: 'The browser appears to
have exited before we could connect. The output was: b"\n(process:19757):
GLib-CRITICAL **: g_slice_set_config: assertion `sys_page_size == 0\'
failed\nError: no display specified\n"'
```

⁴ W chwili pisania książki najnowsza wersja Selenium (2.41) sprawiała pewne trudności i dlatego zdecydowałem się na użycie wersji 2.39. Zachęcam jednak do eksperymentów z nowszymi wersjami.

Konfiguracja ekranu wirtualnego, aby testy funkcjonalne można było wykonywać bez monitora

Jak możesz zobaczyć w powyższych danych wyjściowych, przeglądarka internetowa Firefox nie może zostać uruchomiona z powodu braku ekranu.

Istnieją dwa sposoby rozwiązyania tego problemu. Pierwszy polega na przejęciu do użycia przeglądarki niewymagającej ekranu, takiej jak PhantomJS lub SlimerJS. Tego rodzaju narzędzia zdecydowanie mają swoje zalety — między innymi działają szybciej — choć mają też wady. Przede wszystkim to nie są „prawdziwe” przeglądarki internetowe, stąd brak gwarancji, że wychwycą wszystkie dziwne zachowania przeglądarek internetowych, z których korzystają użytkownicy. Ponadto wymienione przeglądarki działają nieco inaczej wewnątrz Selenium, co oznacza konieczność ponownego utworzenia znacznej ilości kodu testu funkcjonalnego.



Przeglądarki niewymagające ekranu potraktowałbym jako narzędzia „jedynie dla programistów”, ponieważ pozwalają na przyśpieszenie wykonywania testów funkcjonalnych w komputerze programisty. Z kolei do testów w serwerze ciągłej integracji wykorzystałbym rzeczywiste przeglądarki internetowe.

Drugi sposób polega na konfiguracji ekranu wirtualnego. Po prostu udajemy, że do serwera jest dołączony ekran, co pozwoli na uruchomienie przeglądarki internetowej Firefox. Istnieje kilka narzędzi przeznaczonych do tego celu, tutaj wykorzystamy Xvfb (X Virtual Framebuffer)⁵, ponieważ jest łatwe w instalacji i użyciu, a ponadto oferuje wygodną wtyczkę dla Jenkins.

Powracamy do projektu, ponownie klikamy *Configure* i szukamy sekcji o nazwie *Build Environment*. Użycie ekranu wirtualnego jest niezwykle łatwe i sprowadza się do zaznaczenia pola wyboru *Start Xvfb before the build, and shut it down after*, jak pokazano na rysunku 20.8.

Build Environment

Start Xvfb before the build, and shut it down after.

Execute shell script on remote host using ssh

Use secret text(s) or file(s)

Zaawansowane...

Rysunek 20.8. Czasami konfiguracja jest łatwa

Teraz komplikacja przebiegła znacznie lepiej:

```
[...]
Xvfb starting$ /usr/bin/Xvfb :2 -screen 0 1024x768x24 -fbdir
/var/lib/jenkins/2013-11-04_03-27-221510012427739470928xvfb
[...]
+ python manage.py test lists accounts
.....
Ran 51 tests in 0.410s
```

⁵ Sprawdź także projekt pyvirtualdisplay (<https://pypi.python.org/pypi/PyVirtualDisplay>) — to inne rozwiązanie przeznaczone do kontrolowania ekranów wirtualnych z poziomu Pythona.

```

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...
+ pip install selenium
Requirement already satisfied (use --upgrade to upgrade): selenium in
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/python3.3/site-packages
Cleaning up...

+ python manage.py test functional_tests
.....F.
=====
FAIL: test_logged_in_users_lists_are_saved_as_my_lists
(functional_tests.test_my_lists.MyListsTest)

-----
Traceback (most recent call last):
  File
"/var/lib/jenkins/jobs/Superlists/workspace/functional_tests/test_my_lists.py",
line 44, in test_logged_in_users_lists_are_saved_as_my_lists
    self.assertEqual(self.browser.current_url, first_list_url)
AssertionError: 'http://localhost:8081/accounts/edyta@przyklad.pl/' !=
'http://localhost:8081/lists/1/'
- http://localhost:8081/accounts/edyta@przyklad.pl/
+ http://localhost:8081/lists/1/

-----
Ran 7 tests in 89.275s

FAILED (errors=1)
Creating test database for alias 'default'...
[{'secure': False, 'domain': 'localhost', 'name': 'sessionid', 'expiry': 1920011311, 'path': '/', 'value': 'a8d8bbde33nreq6gihw8a7r1cc8bf02k'}]
Destroying test database for alias 'default'...
Build step 'Virtualenv Builder' marked build as failure
Xvfb stopping
Finished: FAILURE

```

Było już całkiem blisko! Jednak analiza powyższego niepowodzenia wymaga zrzutów ekranu.



Jak możesz zobaczyć w powyższych danych wyjściowych, błąd został spowodowany przez stan wyścigu, co oznacza, że nie zawsze będzie możliwy do reprodukcji. Był może zobaczyś inny błąd lub w ogóle nie wystąpi żaden. W każdym bądź razie przedstawione poniżej narzędzia przeznaczone do wykonywania zrzutów ekranu i rozwiązywania problemów związanych ze stanem wyścigu okażą się użyteczne. Czytaj dalej.

Wykonanie zrzutów ekranu

W celu analizy nieoczekiwanych niepowodzeń występujących w zdalnym komputerze dobrze jest mieć zrzut ekranu z chwili wystąpienia błędu. Przyda się także kod HTML strony. Wspomniane dane możemy otrzymać za pomocą własnej logiki w metodzie `tearDown()` klasy testu funkcjonalnego. Konieczna jest odrobina introspekcji we wnętrzu modułu `unittest` i atrybutu prywatnego o nazwie `_outcomeForDoCleanups`, ale rozwiązanie działa.

Plik *functional_tests/base.py* (ch20l006):

```
import os
from datetime import datetime
SCREEN_DUMP_LOCATION = os.path.abspath(
    os.path.join(os.path.dirname(__file__), 'screendumps'))
)
[...]
def tearDown(self):
    if self._test_has_failed():
        if not os.path.exists(SCREEN_DUMP_LOCATION):
            os.makedirs(SCREEN_DUMP_LOCATION)
        for ix, handle in enumerate(self.browser.window_handles):
            self.windowid = ix
            self.browser.switch_to_window(handle)
            self.take_screenshot()
            self.dump_html()
        self.browser.quit()
    super().tearDown()

def _test_has_failed(self):
    # Dla Pythona 3.4. W wersji 3.3 można po prostu użyć self._outcomeForDoCleanups.success:
    for method, error in self._outcome.errors:
        if error:
            return True
    return False
```

Najpierw tworzymy katalog na zrzuty ekranu, o ile to konieczne. Następnie przeprowadzana jest iteracja przez wszystkie otwarte karty i strony w przeglądarce internetowej. Dzięki użyciu pewnych metod Selenium — `get_screenshot_as_file()` i `browser.page_source()` — otrzymujemy zrzut ekranu i kod HTML strony.

Plik *functional_tests/base.py* (ch20l007):

```
def take_screenshot(self):
    filename = self._get_filename() + '.png'
    print('wykonywanie zrzutu ekranu do pliku', filename)
    self.browser.get_screenshot_as_file(filename)

def dump_html(self):
    filename = self._get_filename() + '.html'
    print('zapis strony HTML do pliku', filename)
    with open(filename, 'w') as f:
        f.write(self.browser.page_source)
```

Poniższy fragment kodu jest przeznaczony do generowania unikatowych identyfikatorów nazwy pliku zawierającej nazwę testu, klasę oraz znacznik czasu.

Plik *functional_tests/base.py* (ch20l008):

```
def _get_filename(self):
    timestamp = datetime.now().isoformat().replace(':', '.')[:19]
    return '{folder}/{classname}_{method}-{window{windowid}}-{timestamp}'.format(
        folder=SCREEN_DUMP_LOCATION,
        classname=self.__class__.__name__,
        method=self._testMethodName,
        windowid=self._windowid,
        timestamp=timestamp
    )
```

Kod możesz najpierw przetestować lokalnie, celowo uszkadzając jeden z testów, na przykład za pomocą wywołania `self.fail()`. Otrzymasz wówczas dane wyjściowe podobne do przedstawionych poniżej:

[...]
wykonywanie zrzutu ekranu do pliku /workspace/superlists/functional_tests/screendumps/
↳ MyListsTest.test_logged_in_users_lists_are_saved_as_my_lists-window0-2014-03-09T11.19.12.png
zapis strony HTML do pliku /workspace/superlists/functional_tests/screendumps/
↳ MyListsTest.test_logged_in_users_lists_are_saved_as_my_lists-window0-2014-03-09T11.19.12.html

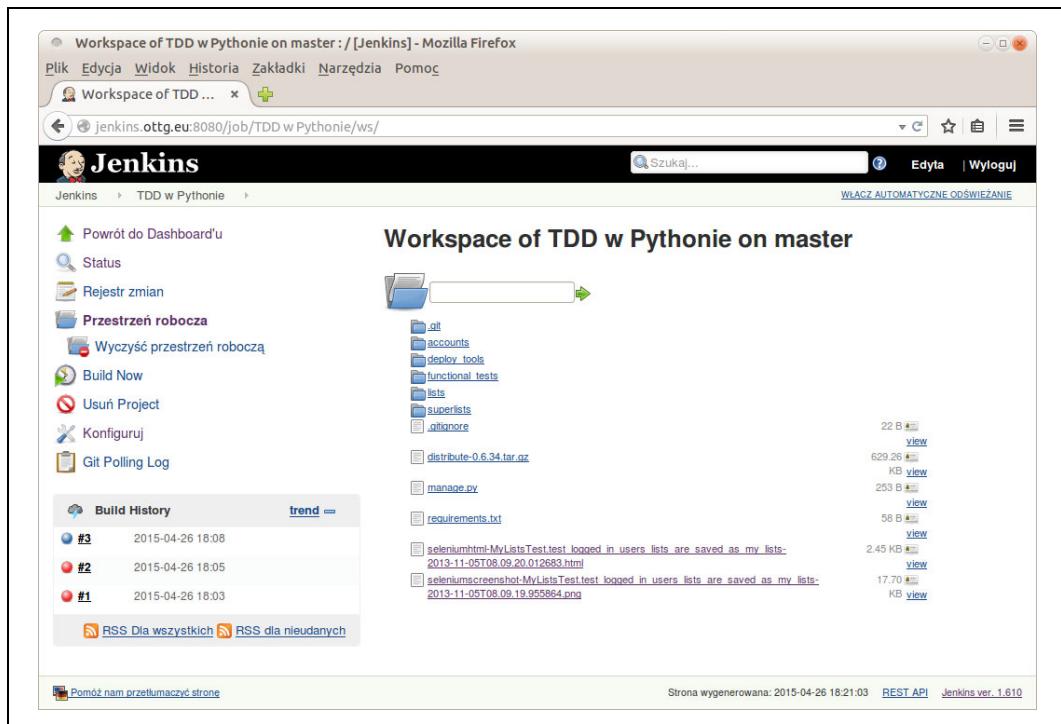
Powracamy do wywołania `self.fail()`, a następnie przekazujemy kod do repozytorium, skąd zostanie pobrany do serwera ciągłej integracji:

```
$ git diff # Polecenie pokazuje zmiany w pliku base.py.
$ echo "functional_tests/screendumps" >> .gitignore
$ git commit -am "Dodanie do testu funkcjonalnego operacji wykonywania zrzutu ekranu w przypadku
↳niepowodzenia testu."
$ git push
```

Po ponownym przeprowadzeniu komplikacji w Jenkins otrzymamy dane wyjściowe podobne do poniższych:

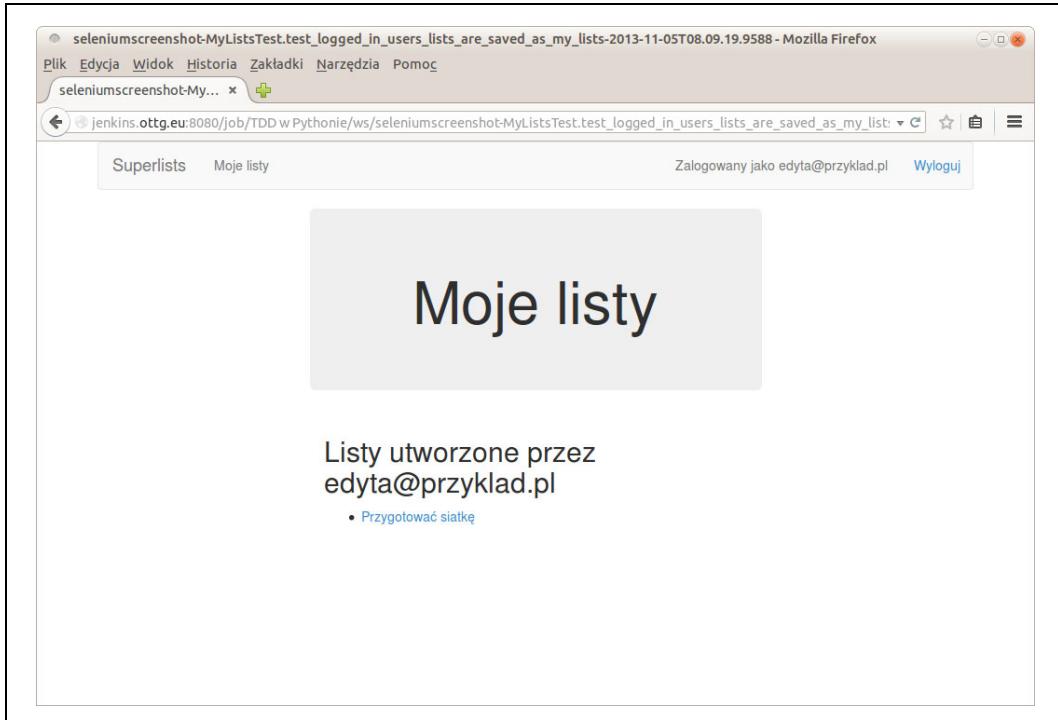
```
wykonywanie zrzutu ekranu do /var/lib/jenkins/jobs/Superlists/workspace/functional_tests/
↳screendumps/LoginTest.test_login_with_persona-window0-2014-01-22T17.45.12.png
zapis strony HTML do pliku /var/lib/jenkins/jobs/Superlists/workspace/functional_tests/
↳screendumps/LoginTest.test_login_with_persona-window0-2014-01-22T17.45.12.html
```

Wygenerowane pliki możemy przejrzeć w tak zwanej przestrzeni roboczej, czyli w katalogu (patrz rysunek 20.9) utworzonym przez Jenkins do przechowywania kodu źródłowego i uruchamiania w nim testów.



Rysunek 20.9. Przestrzeń robocza projektu

Następnie możemy już wyświetlić zrobiony zrzut ekranu (patrz rysunek 20.10).



Rysunek 20.10. Zrzut ekranu wygląda normalnie

No cóż, zrzut ekranu nie bardzo nam pomógł w rozwiązaniu problemu.

Najczęstszy problem w Selenium — stan wyścigu

Kiedy otrzymasz nieoczekiwane niepowodzenie w trakcie testu Selenium, jedną z jego przyczyn może być ukryty stan wyścigu. Spójrzmy na fragment kodu odpowiedzialny za niepowodzenie.

Plik `functional_tests/test_my_lists.py`:

```
# Zobaczyła, że jej lista tam się znajduje.  
# Nazwa listy pochodzi od pierwszego elementu na niej.  
self.browser.find_element_by_link_text('Przygotować siatkę').click()  
self.assertEqual(self.browser.current_url, first_list_url)
```

Natychmiast po kliknięciu łącza *Przygotować siatkę* nakazujemy Selenium sprawdzenie, czy bieżący adres URL odpowiada adresowi URL pierwszej listy. Jednak adresy nie są takie same:

```
AssertionError: 'http://localhost:8081/accounts/edyta@przyklad.pl/' !=  
'http://localhost:8081/lists/1/'
```

Wygląda na to, że bieżącym adresem URL wciąż jest adres strony „Moje listy”. Co się dzieje?

Czy pamiętasz, jak w rozdziale 2. ustawiliśmy dla przeglądarki internetowej `implicitly_wait`? Czy pamiętasz, jak wspomniałem, że to jest zawodne rozwiązanie?

Ustawienie `implicitly_wait` sprawdza się dobrze w przypadku wszelkich wywołań `find_element` w Selenium, ale nie dla `browser.current_url`. Selenium nie „czeka” po nakazaniu kliknięcia elementu. Dlatego też przeglądarka internetowa nie zakończyła wczytywania nowej strony i wartością `current_url` nadal jest adres poprzedniej strony. Potrzebny jest nam bardziej niezawodny kod oczekiwania, podobny do zastosowanego na różnych stronach systemu Persona.

Na tym etapie nadeszła pora na przygotowanie funkcji pomocniczej obsługującej „oczekiwanie”. Aby przekonać się, jak będzie działała, pomocne będzie określenie naszych oczekiwania względem wspomnianej funkcji (podejście Outside-In).

Plik `functional_tests/test_my_lists.py` (ch20l012):

```
# Zobaczyła, że jej lista tam się znajduje.  
# Nazwa listy pochodzi od pierwszego elementu na niej.  
self.browser.find_element_by_link_text('Przygotować siatkę').click()  
self.wait_for(  
    lambda: self.assertEqual(self.browser.current_url, first_list_url)  
)
```

Wywołanie `assertEqual` zamienimy na funkcję `lambda`, a następnie przekażemy funkcji pomocniczej `wait_for()`.

Plik `functional_tests/base.py` (ch20l013):

```
import time  
from selenium.common.exceptions import WebDriverException  
[...]  
  
def wait_for(self, function_with_assertion, timeout=DEFAULT_WAIT):  
    start_time = time.time()  
    while time.time() - start_time < timeout:  
        try:  
            return function_with_assertion()  
        except (AssertionError, WebDriverException):  
            time.sleep(0.1)  
    # Jeszcze jedna próba, która spowoduje zgłoszenie błędów, jeśli nie zostaną wcześniej usunięte.  
    return function_with_assertion()
```

Funkcja pomocnicza `wait_for()` próbuje wykonać asercje, ale zamiast pozwolić na niepowodzenie funkcji w przypadku niepowodzenia asercji przechwytuje `AssertionError` zgłoszony przez `assertEqual`, czeka przez chwilę, a następnie ponawia próbę. Pętla `while` jest wykonywana przez podaną ilość czasu. Przechwytywane są wszelkie wyjątki `WebDriverException`, które mogą zostać zgłoszone, jeśli element jeszcze nie pojawił się na stronie. Po upłynięciu podanego czasu funkcja podejmuje jeszcze jedną próbę wykonania asercji, tym razem bez konstrukcji `try-except`. Jeżeli więc nadal występuje błąd `AssertionError`, wówczas test kończy się niepowodzeniem.



Widziałeś, że Selenium dostarcza `WebDriverWait` jako narzędzie przeznaczone do obsługi oczekiwania, choć jego działanie jest nieco restrykcyjne. Przedstawiona powyżej opracowana przez nas wersja pozwala na przekazanie funkcji wykonującej asercję modułu `unittest` i zachowuje wszystkie zalety czytelnych komunikatów błędów, jakie generuje wymieniony moduł.

Czas oczekiwania został dodany jako argument opcjonalny, opieramy się na stałej, która będzie zdefiniowana w pliku `base.py`. Wykorzystamy ją także w pierwotnym wywołaniu `implicitly_wait`.

Plik *functional_tests/base.py* (ch20l014):

```
[...]
DEFAULT_WAIT = 5
SCREEN_DUMP_LOCATION = os.path.abspath(
    os.path.join(os.path.dirname(__file__), 'screendumps')
)
class FunctionalTest(StaticLiveServerCase):
    [...]
    def setUp(self):
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(DEFAULT_WAIT)
```

Teraz możemy ponownie wykonać test i sprawdzić, czy nadal jest zaliczany w komputerze lokalnym:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
Ran 1 test in 9.594s
OK
```

Tak dla pewności celowo uszkadzamy test, aby zobaczyć jego niepowodzenie.

Plik *functional_tests/test_my_lists.py* (ch20l015):

```
self.wait_for(
    lambda: self.assertEqual(self.browser.current_url, 'barf')
)
```

Zgodnie z oczekiwaniami otrzymujemy komunikat błędu:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
AssertionError: 'http://localhost:8081/lists/1/' != 'barf'
```

Zauważamy kilkusekundową pauzę na stronie. Cofamy ostatnią zmianę, a następnie przekazujemy pliki do repozytorium:

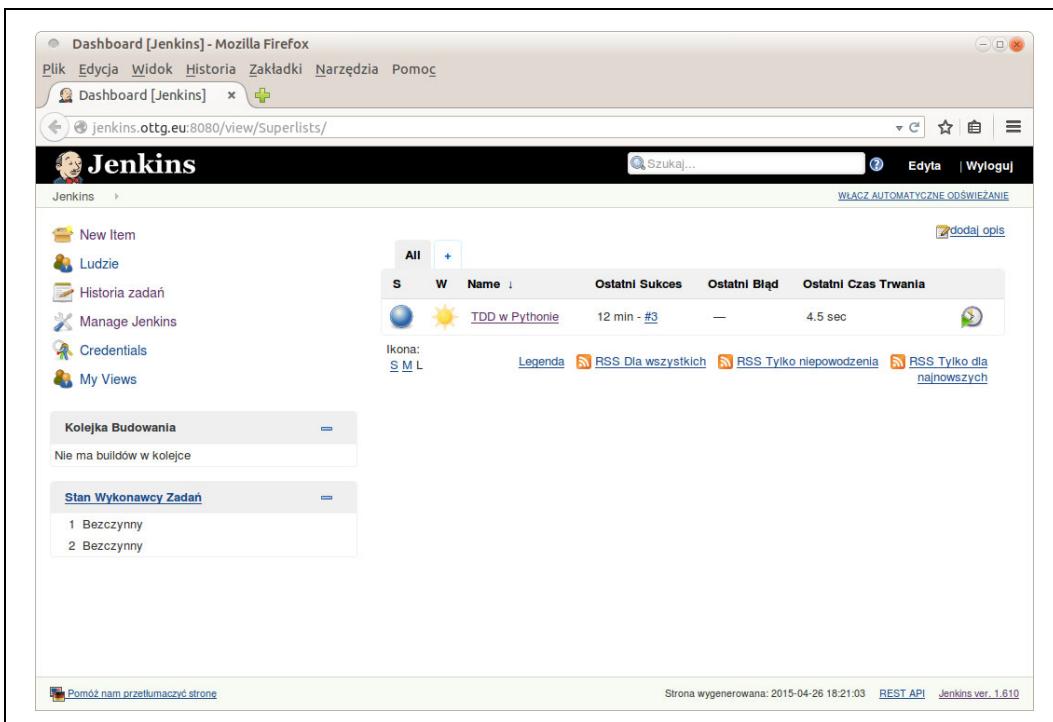
```
$ git diff # Polecenie pokazuje zmiany w plikach base.py i test_my_lists.py
$ git commit -am"Użycie funkcji wait_for() do sprawdzenia adresu URL w my_lists."
$ git push
```

Teraz możemy ponownie powtórzyć operację w Jenkins, używając przycisku *Build Now!*, i potwierdzić działanie rozwiązania, jak pokazano na rysunku 20.11.

Do wskazania udanej komplikacji Jenkins używa koloru niebieskiego zamiast zielonego, co może być pewnym rozczarowaniem. Jednak spójrz na słońce przebijające się przez chmury: to niewątpliwie radosny widok. W serwerze Jenkins mamy wyświetlony współczynnik komplikacji zakończonych powodzeniem do zakończonych niepowodzeniem. W naszym przykładzie to wygląda dobrze.

Wykonanie testów QUnit w Jenkins za pomocą PhantomJS

Mamy jeszcze zestaw testów, o którym niemal zapomnieliśmy — testy JavaScript. Obecnie naszym „silnikiem testów” jest rzeczywista przeglądarka internetowa. Aby zlecić ich wykonywanie serwerowi Jenkins, potrzebujemy silnika testów działającego z poziomu powłoki. W ten sposób pojawią się szansa na użycie PhantomJS.



Rysunek 20.11. Widok przedstawia się znacznie lepiej

Instalacja node

Czas przestać się oszukiwać, że nie korzystamy z kodu JavaScript. Zajmujemy się programowaniem sieciowym, co oznacza także użycie języka JavaScript. To doprowadza nas do instalacji node.js w komputerze. Tak po prostu musi być.

Zastosuj się do poleceń przedstawionych w witrynie *node.js*⁶. Dostępne są pakiety instalacyjne dla systemów Windows i Mac oraz repozytoria dla popularnych dystrybucji systemu Linux⁷.

Po instalacji node.js można zainstalować PhantomJS:

```
$ npm install -g phantomjs # Opcja -g oznacza "dla całego systemu". Może wystąpić potrzeba użycia sudo.
```

Kolejnym krokiem jest konfiguracja silnika testów QUnit i PhantomJS. Dostępnych jest kilka (samodzielnie utworzyłem nawet prosty silnik, aby przetestować listingi QUnit zaprezentowane w książce), ale prawdopodobnie najlepszym będzie jeden ze wskazanych na stronie *wtyczek QUnit*⁸. W chwili pisania książki jego repozytorium znajdowało się pod adresem <https://github.com/jonkemp/qunit-phantomjs-runner>. Jedynym wymaganym plikiem jest *runner.js*.

⁶ <https://nodejs.org/download/>

⁷ Upewnij się o pobraniu najnowszej wersji node.js. W dystrybucji Ubuntu skorzystaj z PPA zamiast z domyślnego pakietu instalacyjnego.

⁸ <http://qunitjs.com/plugins/>

Ostatecznie powinieneś mieć następującą strukturę plików:

```
$ tree superlists/static/tests/
superlists/static/tests/
  qunit.css
  qunit.js
  runner.js
  sinon.js

0 directories, 4 files
```

Wypróbujmy rozwiązanie:

```
$ phantomjs superlists/static/tests/runner.js lists/static/tests/tests.html
Took 24ms to run 2 tests. 2 passed, 0 failed.
$ phantomjs superlists/static/tests/runner.js accounts/static/tests/tests.html
Took 29ms to run 11 tests. 11 passed, 0 failed.
```

Aby uzyskać pewność, celowo uszkadzamy cokolwiek.

Plik *lists/static/list.js* (ch20l019):

```
$(‘input’).on(‘keypress’, function () {
  //$(‘.has-error’).hide();
});
```

Zgodnie z oczekiwaniami otrzymujemy niepowodzenie testu:

```
$ phantomjs superlists/static/tests/runner.js lists/static/tests/tests.html
Test failed: undefined: errors should be hidden on keypress
  Failed assertion: expected: false, but was: true
    at file:///workspace/superlists/superlists/static/tests/qunit.js:556
    at file:///workspace/superlists/lists/static/tests/tests.html:26
  at file:///workspace/superlists/superlists/static/tests/qunit.js:203
    at file:///workspace/superlists/superlists/static/tests/qunit.js:361
    at process
  (file:///workspace/superlists/superlists/static/tests/qunit.js:1453)
    at file:///workspace/superlists/superlists/static/tests/qunit.js:479
Took 27ms to run 2 tests. 1 passed, 1 failed.
```

Wszystko w porządku. Cofamy ostatnią zmianę, przekazujemy pliki do repozytorium, stamtąd do serwera Jenkins, któremu pozwalamy na komplikację:

```
$ git checkout lists/static/list.js
$ git add superlists/static/tests/runner.js
$ git commit -m" Dodanie phantomjs dla testów JavaScript."
$ git push
```

Dodanie kolejnych kroków kompilacji w Jenkins

Przeprowadź ponownie konfigurację projektu i dodaj kroki dla poszczególnych testów JavaScript, jak pokazano na rysunku 20.12.



Rysunek 20.12. Dodanie w Jenkins kroków dla testów jednostkowych JavaScript

Konieczne jest również zainstalowanie PhantomJS w serwerze:

```
edyta@serwer:$ sudo add-apt-repository -y ppa:chris-lea/node.js
edyta@serwer:$ sudo apt-get update
edyta@serwer:$ sudo apt-get install nodejs
edyta@serwer:$ sudo npm install -g phantomjs
```

I to już tyle. Pełna komplikacja w procesie ciągłej integracji obsługuje nasze wszystkie testy!

```
Started by user harry
Building in workspace /var/lib/jenkins/jobs/Superlists/workspace
Fetching changes from the remote Git repository
Fetching upstream changes from https://github.com/hjwp/book-example.git
Checking out Revision 936a484038194b289312ff62f10d24e6a054fb29 (origin/chapter_1
Xvfb starting$ /usr/bin/Xvfb :1 -screen 0 1024x768x24 -fbdir /var/lib/jenkins/20
[workspace] $ /bin/sh -xe /tmp/shiningpanda7092102504259037999.sh

+ pip install -r requirements.txt
[...]

+ python manage.py test lists
.....
-----
Ran 33 tests in 0.229s

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

+ python manage.py test accounts
.....
-----
Ran 18 tests in 0.078s

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

[workspace] $ /bin/sh -xe /tmp/hudson2967478575201471277.sh
+ phantomjs superlists/static/tests/runner.js lists/static/tests/tests.html
Took 32ms to run 2 tests. 2 passed, 0 failed.
+ phantomjs superlists/static/tests/runner.js accounts/static/tests/tests.html
Took 47ms to run 11 tests. 11 passed, 0 failed.

[workspace] $ /bin/sh -xe /tmp/shiningpanda7526089957247195819.sh
+ pip install selenium
Requirement already satisfied (use --upgrade to upgrade): selenium in /var/lib/
Cleaning up...
[workspace] $ /bin/sh -xe /tmp/shiningpanda2420240268202055029.sh
+ python manage.py test functional_tests
.....
-----
Ran 7 tests in 76.804s

OK
```

Dobrze wiedzieć, że niezależnie od tego, jak będę leniwy w zakresie wykonania pełnego zestawu testów w komputerze, serwer ciągłej integracji złapie mnie na tym. To kolejny agent Testing Goat w cyberprzestrzeni, który nas obserwuje...

Więcej zadań do wykonania za pomocą serwera ciągłej integracji

W tym rozdziale jedynie dotknąłem tematu możliwości Jenkins i serwerów ciągłej integracji. Na przykład mogą działać jeszcze sprytniej i monitorować repozytorium pod kątem nowych plików przekazanych do repozytorium.

Prawdopodobnie znacznie bardziej interesująca jest możliwość użycia serwera ciągłej integracji do automatyzacji testów prowizorycznych, a także zwykłych testów funkcjonalnych. Jeżeli wszystkie testy funkcjonalne zostaną zaliczone, wówczas można dodać kolejny krok komplikacji wdrażający kod do serwera prowizorycznego. Następnie ponownie wykonuje się testy funkcjonalne względem wspomnianego serwera — automatyzując tym samym kolejny krok procesu i zapewniając automatyczną aktualizację serwera prowizorycznego najnowszą wersją kodu.

Niektórzy nawet używają serwera ciągłej integracji jako sposobu wdrażania wydań produkcyjnych.

Podpowiedzi dotyczące ciągłej integracji oraz najlepszych praktyk podczas użycia Selenium

Jak najwcześniej skonfiguruj serwer ciągłej integracji dla projektu

Gdy tylko wykonanie testów funkcjonalnych zajmie więcej niż kilka sekund, znajdź sposób na uniknięcie wykonywania ich wszystkich. To zadanie zleć serwerowi ciągłej integracji, aby mieć pewność, że wszystkie testy jednak będą wykonywane.

Skonfiguruj wykonywanie zrzutów ekranu i kopii kodu HTML stron w przypadku niepowodzeń

Debugowanie niepowodzeń testu będzie łatwiejsze, jeśli możesz zobaczyć wygląd strony w momencie niepowodzenia. To jest szczególnie użyteczne podczas analizy niepowodzeń w serwerze ciągłej integracji, choć również pomocne w przypadku testów wykonywanych lokalnie.

Zdefiniuj oczekiwanie w testach Selenium

Oferowane przez Selenium wywołanie `implicitly_wait` ma zastosowanie jedynie dla funkcji `find_element()`, ale nawet wówczas może być zawodne, bo na przykład znajduje element pochodzący ze starej strony. Opracuj funkcję pomocniczą `wait_for()`, wykonaj pewne działania w witrynie, a następnie zastosuj pewien rodzaj mechanizmu oczekiwania, aby sprawdzić, czy wspomniane działania mają jakikolwiek efekt.

Znajdź sposób na powiązanie serwerów ciągłej integracji i prowizorycznego

Testy używające klasy `LiveServerTestCase` doskonale sprawdzają się w środowiskach programistycznych, ale prawdziwą gwarancję daje dopiero wykonanie testów względem rzeczywistego serwera. Znajdź sposób na wykorzystanie serwera ciągłej integracji w celu wdrożenia kodu do serwera prowizorycznego i wykonywania względem niego testów funkcjonalnych. Efektem ubocznym będzie przetestowanie zautomatyzowanych skryptów wdrożenia.

Token serwisów społecznościowych, wzorzec strony i ćwiczenie dla czytelnika

Czy żarty o tym, że wszystko musi być powiązane z serwisami społecznościowymi, nie wydają się już niemodne? Wszystko powinno być przetestowane na piątkę, big data, listy zbierz-więcej-kliknięć, „10 rzeczy wymienionych przez tego inspirującego nauczyciela, które zmienią twój stosunek do...”, bla, bla, bla. Listy rzeczy do zrobienia, nieważne, czy inspirujące, czy jednak nie, bardzo często są współdzielone. Pozwólmy więc użytkownikom na dzielenie się swoimi listami z innymi użytkownikami.

Przy okazji poprawimy testy funkcjonalne, rozpoczynając implementację w Selenium wzorca interakcja-oczekiwanie, który przedstawiono w poprzednim rozdziale. Ponadto poeksperymentujemy z tak zwanym wzorcem strony.

Następnie zamiast dokładnie podać to, co powinieneś zrobić, pozwolę Ci utworzyć testy jednostkowe i samodzielnie przetestować kod aplikacji. Nie martw się, nie pozostawię Cię zupełnie samego! Ogólnie przedstawię konieczne kroki i podam kilka wskazówek oraz podpowiedzi.

Test funkcjonalny z wieloma użytkownikami i funkcja addCleanup()

Rozpoczynamy pracę i będziemy potrzebować dwóch użytkowników dla tego testu funkcjonalnego.

Plik `functional_tests/test_sharing.py`:

```
from selenium import webdriver
from .base import FunctionalTest

def quit_if_possible(browser):
    try:
        browser.quit()
    except:
        pass

class SharingTest(FunctionalTest):

    def test_logged_in_users_lists_are_saved_as_my_lists(self):
        # Edyta jest zalogowanym użytkownikiem.
        self.create_pre_authenticated_session('edyta@przyklad.pl')
        edith_browser = self.browser
        self.addCleanup(lambda: quit_if_possible(edith_browser))
```

```

# Jej przyjaciel Orfeusz również korzysta z witryny.
oni_browser = webdriver.Firefox()
self.addCleanup(lambda: quit_if_possible(oni_browser))
self.browser = oni_browser
self.create_pre_authenticated_session('orfeusz@przyklad.pl')

# Edyta przechodzi na stronę główną i zaczyna tworzyć listę.
self.browser = edith_browser
self.browser.get(self.server_url)
self.get_item_input_box().send_keys('Pomoc\n')

# Zauważa opcję "Udostępnij tę listę".
share_box = self.browser.find_element_by_css_selector('input[name=email]')
self.assertEqual(
    share_box.get_attribute('placeholder'),
    'twoj-przyjaciel@przyklad.pl'
)

```

Warto zwrócić uwagę na interesującą funkcję `addCleanup()`, której dokumentację znajdziesz tutaj¹. Ta funkcja może być używana w charakterze alternatywy dla funkcji `tearDown()`, służący do zwolnienia zasobów wykorzystywanych podczas testu. Najbardziej użyteczna okazuje się, gdy w trakcie testu zasób jest zaalokowany jedynie w połowie. Nie trzeba więc marnować czasu w funkcji `tearDown()` na ustalenie, co wymaga, a co nie wymaga operacji porządkowych.

Funkcja `addCleanup()` jest wywoływaną po `tearDown()` i dlatego konieczne jest użycie konstrukcji `try-except` dla `quit_if_possible()`. Jeżeli w chwili zakończenia testu wywołanie `self.browser` będzie miało przypisaną wartość `edith_browser` lub `oni_browser`, wówczas test zostanie zakończony przez funkcję `tearDown()`.

Konieczne jest również przeniesienie funkcji `create_pre_authenticated_session()` z pliku `test_my_lists.py` do `base.py`.

Dobrze, zobaczymy, czy przygotowane rozwiązanie działa:

```

$ python3 manage.py test functional_tests.test_sharing
[...]
Traceback (most recent call last):
  File "/workspace/superlists/functional_tests/test_sharing.py", line 29, in
    test_logged_in_users_lists_are_saved_as_my_lists
    share_box = self.browser.find_element_by_css_selector('input[name=email]')
[...]
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate
element: {"method":"css selector","selector":"input[name=email]"}';

```

Doskonale! Wydaje się, że przechodzimy przez operację utworzenia dwóch sesji użytkowników i następuje oczekiwane niepowodzenie — brakuje danych wejściowych w postaci adresu e-mail osoby, której ma zostać udostępniona lista.

Na tym etapie przekazujemy kod do repozytorium, ponieważ mamy przynajmniej miejsce zarezerwowane na test funkcjonalny, przeprowadziliśmy użyteczną modyfikację funkcji `create_pre_authenticated_session()`, a ponadto rozpoczęliśmy proces refaktoryzacji testu funkcjonalnego:

```

$ git add functional_tests
$ git commit -m "Nowy test funkcjonalny dla udostępniania listy, przeniesienie do pliku base.py
→operacji tworzenia sesji."

```

¹ <https://docs.python.org/3/library/unittest.html#unittest.TestCase.addCleanup>

Implementacja w Selenium wzorca interakcja-oczekiwanie

Zanim będziemy kontynuować pracę, spójrzmy na interakcje w witrynie, dla których przygotowaliśmy testy funkcjonalne.

Plik `functional_tests/test_sharing.py`:

```
# Edyta przechodzi na stronę główną i zaczyna tworzyć listę.  
self.browser = edith_browser  
self.browser.get(self.server_url)  
self.get_item_input_box().send_keys('Pomoc\n') #❶  
  
# Zauważ opcję "Udostępnij tę listę".  
share_box = self.browser.find_element_by_css_selector('input[name=email]') #❷  
self.assertEqual(  
    share_box.get_attribute('placeholder'),  
    'twoj-przyjaciel@przyklad.pl'  
)
```

- ❶ Interakcja z witryną.
- ❷ Założenie dotyczące aktualnego stanu strony.

W poprzednim rozdziale dowiedziałeś się, jak niebezpieczne będzie przyjęcie zbyt daleko idących założeń dotyczących stanu przeglądarki internetowej po przeprowadzeniu interakcji (na przykład `send_keys`). Teoretycznie `implicitly_wait` daje pewność, że jeśli wywołanie `find_element_by_css_selector()` nie znajdzie najpierw elementu `input[name=email]`, wtedy próba zostanie kilkakrotnie powtórzona. Jednak i takie podejście może się nie udać. Wyobraź sobie, że na poprzedniej stronie znajdował się element `<input>` o atrybutie `name=email`, ale z innym tekstem podpowiedzi. W takim przypadku otrzymamy dziwne niepowodzenie, ponieważ Selenium może teoretycznie wziąć element z poprzedniej strony podczas wczytywania nowej. To doprowadzi do zgłoszenia wyjątku `StaleElementException`.



Nieoczekiwane błędy `StaleElementException` w Selenium często oznaczają powstanie pewnego rodzaju stanu wyścigu. W takim przypadku prawdopodobnie powinieneś wyraźnie zdefiniować wzorzec interakcja-oczekiwanie.

Zawsze rozważne będzie stosowanie wzorca „oczekiwanie na”, gdy występuje potrzeba sprawdzenia wyniku właśnie przeprowadzonej interakcji. Oto przykładowe rozwiązanie.

Plik `functional_tests/test_sharing.py`:

```
self.get_item_input_box().send_keys('Pomoc\n')  
# Zauważ opcję "Udostępnij tę listę".  
self.wait_for(  
    lambda: self.assertEqual(  
        self.browser.find_element_by_css_selector(  
            'input[name=email]'  
        ).get_attribute('placeholder'),  
        'twoj-przyjaciel@przyklad.pl'  
    )
```

Wzorzec strony

Czy wiesz, co byłoby jeszcze lepszym rozwiązaniem? To jest okazja do zastosowania podejścia „do trzech razy sztuka, a później refaktoryzacja”. Podobnie jak wiele innych, także ten test zaczyna się od utworzenia przez użytkownika nowej listy. Co się stanie, jeśli przygotujemy funkcję pomocniczą przeznaczoną do utworzenia nowej listy i ta funkcja będzie wywoływała funkcję `wait_for()` oraz pobierała dane wejściowe elementów listy?

Wcześniej spotkałeś się już z przykładem użycia metod pomocniczych w klasie bazowej `FunctionalTest`, ale jeśli będziemy kontynuować wykorzystywanie zbyt wielu z nich, klasa bardzo szybko się rozrośnie. Pracowałem już z bazową klasą testów funkcjonalnych, która miała wielkość przekraczającą 1500 wierszy kodu. To było niezwykle nieporęczne rozwiązanie.

Jeden z akceptowanych wzorców podziału kodu pomocniczego testów funkcjonalnych nosi nazwę *wzorzec strony*². Oznacza przygotowanie obiektów przedstawiających różne strony witryny i wyznaczenie jednego miejsca przeznaczonego do przechowywania informacji o sposobach interakcji między wspomnianymi obiektyami.

Zobaczmy, jak możemy utworzyć obiekty `Page` dla stron głównej i list. Poniżej przedstawiono obiekt `Page` dla strony głównej.

Plik `functional_tests/home_and_list_pages.py`:

```
class HomePage(object):

    def __init__(self, test):
        self.test = test #❶

    def go_to_home_page(self): #❷
        self.test.browser.get(self.test.server_url)
        self.test.wait_for(self.get_item_input)
        return self #❸

    def get_item_input(self):
        return self.test.browser.find_element_by_id('id_text')

    def start_new_list(self, item_text): #❹
        self.go_to_home_page()
        inputbox = self.get_item_input()
        inputbox.send_keys(item_text + '\n')
        list_page = ListPage(self.test) #❺
        list_page.wait_for_new_item_in_list(item_text, 1) #❻
        return list_page #❼
```

- ❶ Inicjalizacja wraz z obiektem przedstawiającym bieżący test. W ten sposób zyskujemy możliwość przeprowadzania asercji, uzyskania dostępu do egzemplarza przeglądarki za pomocą `self.test.browser`, a także użycia funkcji `wait_for()`.
- ❷ Większość obiektów `Page` ma funkcję w stylu „przejdz do tej strony”. Zwróć uwagę, że to implementuje wzorzec interakcja-oczekiwanie. Najpierw pobieramy adres URL strony, a następnie oczekujemy na dany element na stronie głównej.
- ❸ Wartość zwrotna `self` jest użyta po prostu dla wygody i pozwala na łączenie metod³.

² http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern

³ https://en.wikipedia.org/wiki/Method_chaining

- ④ Tutaj znajduje się funkcja odpowiedzialna za utworzenie nowej listy. Przechodzi na stronę główną, odszukuje element `<input>` i przekazuje mu tekst nowego elementu oraz znak nowego wiersza. Następnie czeka, aby sprawdzić, czy interakcja została zakończona. Jak możesz zobaczyć, w rzeczywistości do implementacji mechanizmu oczekiwania wykorzystaliśmy inny obiekt `Page`.
- ⑤ Obiekt `ListPage`, którego kod zostanie wkrótce przedstawiony. Jego inicjalizacja przebiega podobnie jak `HomePage`.
- ⑥ Obiekt `ListPage` jest używany do wykonania `wait_for_new_item_in_list()`. Podajemy oczekiwany tekst elementu oraz jego oczekiwane położenie na liście.
- ⑦ Na koniec komponentowi wywołującemu zwracamy obiekt `list_page`, ponieważ prawdopodobnie uzna go za użyteczny (do tego wkrótce powróćmy).

Poniżej przedstawiono klasę `ListPage`.

Plik `functional_tests/home_and_list_pages.py` (ch21l006):

```
[...]
class ListPage(object):

    def __init__(self, test):
        self.test = test

    def get_list_table_rows(self):
        return self.test.browser.find_elements_by_css_selector(
            '#id_list_table tr'
        )

    def wait_for_new_item_in_list(self, item_text, position):
        expected_row = '{}: {}'.format(position, item_text)
        self.test.wait_for(lambda: self.test.assertIn(
            expected_row,
            [row.text for row in self.get_list_table_rows()]
        ))
```



Zwykle najlepsze rozwiązanie to przygotowanie oddzielnych plików dla poszczególnych obiektów `Page`. W omawianym przykładzie obiekty `HomePage` i `ListPage` są ze sobą tak blisko powiązane, że łatwiej jest przechowywać je razem.

Zobaczmy, jak można to wykorzystać w naszym teście.

Plik `functional_tests/test_sharing.py` (ch21l007):

```
from .home_and_list_pages import HomePage
[...]
# Edyta przechodzi na stronę główną i zaczyna tworzyć listę.
self.browser = edith_browser
list_page = HomePage(self).start_new_list('Pomoc')
```

Kontynuujemy pracę nad modyfikacją testu i używamy obiektu `Page`, gdy tylko wystąpi potrzeba uzyskania dostępu do elementów ze strony list.

Plik `functional_tests/test_sharing.py` (ch21l008):

```
# Zauważ opcję "Udostępnij tę listę".
share_box = list_page.get_share_box()
self.assertEqual(
```

```

        share_box.get_attribute('placeholder'),
        'twoj-przyjaciel@przyklad.pl'
    )

# Edyta udostępnia swoją listę.
# Strona zostaje uaktualniona i informuje o udostępnieniu listy Orfeuszowi.
list_page.share_list_with('orfeusz@przyklad.pl')

```

Do klasy `ListPage` dodajemy trzy kolejne funkcje.

Plik `functional_tests/home_and_list_pages.py` (ch21l009):

```

def get_share_box(self):
    return self.test.browser.find_element_by_css_selector(
        'input[name=email]')
)

def get_shared_with_list(self):
    return self.test.browser.find_elements_by_css_selector(
        '.list-sharee'
)

def share_list_with(self, email):
    self.get_share_box().send_keys(email + '\n')
    self.test.wait_for(lambda: self.test.assertIn(
        email,
        [item.text for item in self.get_shared_with_list()])
)

```

Idea przyświecająca powstaniu wzorca strony była następująca: przechwycenie wszystkich informacji o danej stronie witryny. Jeżeli później zajdzie potrzeba wprowadzenia zmian na tej stronie — nawet w postaci prostych modyfikacji kodu HTML tworzącego układ strony — wtedy zmiany przeprowadzasz w jednym miejscu, a następnie odpowiednio dostosowujesz testy funkcjonalne; nie musisz zagłębiać się w dziesiątki testów funkcjonalnych.

Kolejnym krokiem będzie przeprowadzenie refaktoryzacji testów funkcjonalnych. Nie pokażę tego tutaj, to zadanie pozostawiam Ci do wykonania jako rodzaj ćwiczenia. Dzięki temu poznasz różnice między DRY i czytelnością testów...

Rozszerzenie testu funkcjonalnego na drugiego użytkownika i stronę „Moje listy”

Podamy teraz nieco więcej szczegółów dotyczących testów funkcji udostępniania list. Na stronie listy rzeczy do zrobienia Edyta widzi, że dana lista jest „współzielona” z Orfeuszem. Z kolei Orfeusz może się zalogować do witryny i na stronie „moje listy” widzi listę Edyty, prawdopodobnie w sekcji typu „Listy, które zostały mi udostępnione”.

Plik `functional_tests/test_sharing.py` (ch21l010):

```

[...]
list_page.share_list_with('orfeusz@przyklad.pl')

# W przeglądarce internetowej Orfeusz przechodzi na stronę list.
self.browser = oni_browser
HomePage(self).go_to_home_page().go_to_my_lists_page()

# Widzi tam listę udostępnioną przez Edytę!
self.browser.find_element_by_link_text('Pomoc').click()

```

Oznacza to konieczność przygotowania kolejnej funkcji w klasie HomePage.

Plik *functional_tests/home_and_list_pages.py* (ch21l011):

```
class HomePage(object):  
    [...]  
  
    def go_to_my_lists_page(self):  
        self.test.browser.find_element_by_link_text('Moje listy').click()  
        self.test.wait_for(lambda: self.test.assertEqual(  
            self.test.browser.find_element_by_tag_name('h1').text,  
            'Moje listy'  
        ))
```

Warto w tym miejscu podkreślić, że powyższa funkcja powinna być dostępna w *test_my_lists.py* wraz z prawdopodobnie obiektem MyListsPage. Przygotowanie odpowiedniej implementacji to zadanie dla Ciebie!

Orfeusz również może dodawać elementy na liście udostępnionej przez Edytkę.

Plik *functional_tests/test_sharing.py* (ch21l012):

```
# Na stronie listy Orfeusz widzi, że to jest lista Edyty.  
self.wait_for(lambda: self.assertEqual(  
    list_page.get_list_owner(),  
    'edyta@przyklad.pl'  
)  
  
# Dodaje element do listy.  
list_page.add_new_item('Witaj, Edyto!')  
  
# Kiedy Edyta odświeży stronę, widzi element dodany przez Orfeusza.  
self.browser = edith_browser  
self.browser.refresh()  
list_page.wait_for_new_item_in_list('Witaj, Edyto!', 2)
```

Poniżej przedstawiono kilka dodatkowych zmian w obiekcie Page.

Plik *functional_tests/home_and_list_pages.py* (ch21l013):

```
ITEM_INPUT_ID = 'id_text'  
[...]  
  
class HomePage(object):  
    [...]  
  
    def get_item_input(self):  
        return self.test.browser.find_element_by_id(ITEM_INPUT_ID)  
  
class ListPage(object):  
    [...]  
  
    def get_item_input(self):  
        return self.test.browser.find_element_by_id(ITEM_INPUT_ID)  
  
    def add_new_item(self, item_text):  
        current_pos = len(self.get_list_table_rows())  
        self.get_item_input().send_keys(item_text + '\n')  
        self.wait_for_new_item_in_list(item_text, current_pos + 1)  
  
    def get_list_owner(self):  
        return self.test.browser.find_element_by_id('id_list_owner').text
```

Nadeszła pora, aby wykonać test funkcjonalny i sprawdzić, czy wszystko działa.

```
$ python3 manage.py test functional_tests.test_sharing
```

```
share_box = list_page.get_share_box()  
[...]  
selenium.common.exceptions.NoSuchElementException: Message: 'Unable to locate  
element: {"method":"css selector","selector":"input[name=email]"}' ;
```

Wynikiem jest oczekiwane niepowodzenie. Nie mamy jeszcze elementu <input> przeznaczonego na podanie adresu e-mail osoby, której ma zostać udostępniona lista. Przekazujemy pliki do repozytorium.

```
$ git add functional_tests  
$ git commit -m "Utworzenie obiektów Page dla strony głównej i list, użycie ich we współdzielonym  
teście funkcjonalnym."
```

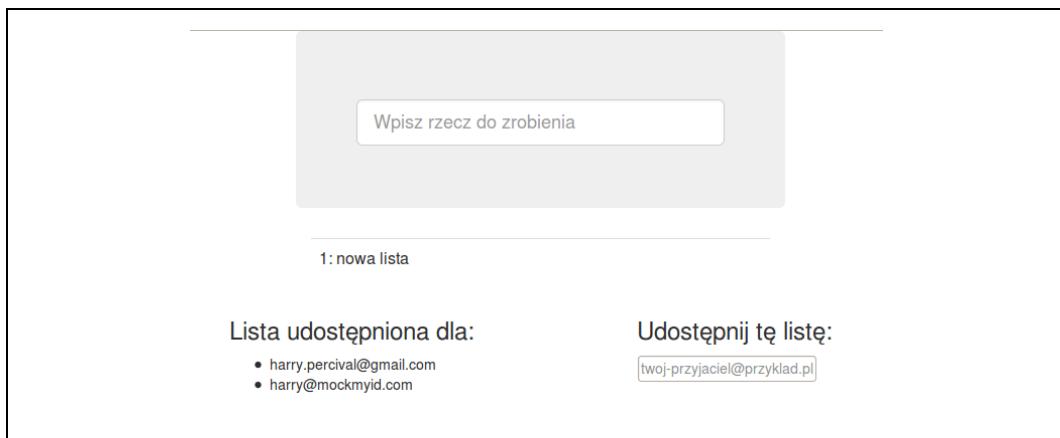
Ćwiczenie dla czytelnika

Oto ogólne omówienie kroków niezbędnych do implementacji nowej funkcji.

1. W pliku *list.html* potrzebna jest nowa sekcja. Na początku to będzie formularz wraz z elementem <input> dla adresu e-mail. To powinno posunąć test funkcjonalny o jeden krok do przodu.
2. Potrzebny jest widok, do którego zostanie wysłany formularz. Rozpocznij od zdefiniowania adresu URL w szablonie, na przykład w postaci *lists/<identyfikator_listy>/share*.
3. Pierwszy test jednostkowy. Wystarczające może być umieszczenie miejsca zarezerwowanego dla widoku. Wspomniany widok ma odpowiadać na żądania POST. Odpowiedzią powinno być przekierowanie z powrotem na stronę listy. Test może więc być nazwany na przykład *ShareList Test.test_post_redirects_to_lists_page()*.
4. Rozbudowujemy miejsce zarezerowane dla widoku. Wystarczą dwa wiersze kodu wyszukującego listę i przekierowującego użytkownika do niej.
5. Następnie możemy przygotować nowy test jednostkowy odpowiedzialny za utworzenie użytkownika i listy, wykonanie żądania POST wraz z adresem e-mail użytkownika oraz sprawdzenie, czy użytkownik został dodany do *list.shared_with.all()* — to jest podobny sposób użycia ORM jak na stronie „Moje listy”. Atrybut *shared_with* jeszcze nie istnieje, stosujemy tutaj podejście Outside-In.
6. Zanim test zostanie zaliczony, konieczne jest przejście w dół do warstwy modelu. Kolejny test (w pliku *test_models.py*) może sprawdzać, czy dana lista ma metodę *shared_with.add()*, która może być wywołana wraz z adresem e-mail użytkownika. Następnie sprawdzamy zbiór wynikowy *shared_with.all()*, który powinien zawierać danego użytkownika.
7. Potrzebujemy *ManyToManyField*. Prawdopodobnie otrzymasz komunikat błędu dotyczący *related_name*. Rozwiążanie znajdziesz, jeśli zajrzesz się do dokumentacji Django.
8. Konieczne będzie przeprowadzenie migracji bazy danych.
9. Na tym etapie testy modelu powinny zostać zaliczone. Przechodzimy do poprawy testu widoku.
10. Zobaczysz niepowodzenie testu przekierowania widoku, ponieważ nie jest wysyłane poprawne żądanie POST. Musisz zignorować nieprawidłowe dane wejściowe lub też zmodyfikować test w taki sposób, aby wysyłał poprawne żądanie POST.

- Następnie przechodzimy na poziom szablonu. Na stronie „Moje listy” powinien znaleźć się element `` wraz z pętlą `for` obsługującą listy udostępnione przez użytkownika. Na stronie list trzeba również podać użytkownika, któremu została udostępniona lista, oraz właściciela danej listy. Zajrzyj do testu funkcjonalnego, znajdziesz tam odpowiednie klasy i identyfikatory do użycia. Jeżeli chcesz, możesz przygotować krótkie testy jednostkowe do sprawdzenia wymienionych danych.
- Uruchomienie witryny za pomocą runserver pomoże w wychwyceniu ewentualnych błędów, a także w dopracowaniu układu wizualnego strony. Jeżeli uruchamiasz przeglądarkę internetową w trybie prywatnym, wówczas będziesz mógł mieć jednocześnie zalogowanych wielu użytkowników.

Na końcu powinieneś otrzymać wynik podobny do pokazanego na rysunku 21.1.



Rysunek 21.1. Możliwość udostępniania list

Wzorzec strony i prawdziwe ćwiczenie dla czytelnika

Zastosuj DRY we własnych testach funkcjonalnych

Kiedy zestaw testów funkcjonalnych rozrośnie się, zobaczysz, że nieunikniona jest sytuacja, w której różne testy używają podobnych fragmentów interfejsu użytkownika. Spróbuj uniknąć umieszczania stałych, na przykład klas lub identyfikatorów znaczników HTML, w poszczególnych elementach interfejsu użytkownika powielanych w testach funkcjonalnych.

Wzorzec strony

Przeniesienie metod pomocniczych do klasy bazowej `FunctionalTest` może być nieporęczne. Rozważ użycie poszczególnych obiektów `Page` do przechowywania całej logiki odpowiedzialnej za obsługę konkretnych fragmentów witryny.

Ćwiczenie dla czytelnika

Mam nadzieję, że naprawdę je wykonasz! Spróbuj zastosować podejście „Outside-In” i sięgaj po inne rozwiązania, gdy zapędzisz się w kozi róg. Prawdziwym ćwiczeniem dla czytelnika będzie oczywiście zastosowanie technik TDD w kolejnym projekcie. Mam nadzieję, że to sprawi Ci radość!

W następnym rozdziale znajdziesz podsumowanie oraz „najlepsze praktyki” w zakresie wykonywania testów.

Szybkie testy, wolne testy i gorąca lawa

Baza danych to gorąca lawa!

— Casey Kinsey¹

Aż do rozdziału 19. praktycznie wszystkie testy „jednostkowe” przedstawione w książce prawdopodobnie powinny być nazywane testami *integracyjnymi*, ponieważ opierają się na bazie danych lub używają klienta testów Django. Wspomniany klient wykonuje jednak zbyt wiele zadań w warstwach pośrednich mieszczących się między żądaniami, odpowiedziami i funkcjami widoku.

Można się spotkać ze stwierdzeniem, że prawdziwy test jednostkowy zawsze powinien być odizolowany, ponieważ jest przeznaczony do przetestowania pojedynczej jednostki oprogramowania.

Niektórzy weterani podejścia TDD twierdzą, że należy dążyć do tworzenia „czystych”, odizolowanych testów jednostkowych zamiast testów integracyjnych, o ile istnieje taka możliwość. To jest przedmiotem nieustannej (czasami gorącej) debaty w środowisku osób stosujących programowanie sterowane testami.

Będąc zaledwie żółtodziobem, nie zdołałem poznać wszystkich subtelności powyższego stwierdzenia. Jednak w tym rozdziale spróbuje wyjaśnić, dlaczego niektórzy są przekonani o jego słuszności. Ponadto postaram się przedstawić pewne podpowiedzi, kiedy można pozwolić sobie na kombinowanie z testami integracyjnymi (muszę przyznać, że często sam tak robię!), a także podpowiem, kiedy warto dążyć do przygotowania bardziej „czystych” testów jednostkowych.

Jeżeli wybaczysz mi terminologię filozoficzną, zastosuję tutaj strukturę dialektyki heglowskiej:

- Teza. W przypadku „czystych” testów jednostkowych oznacza to, że są szybkie.
- Antyteza. Pewne ryzyko powiązane z (naiwnym) podejściem testowania za pomocą „czystych” testów jednostkowych.
- Synteza. Analiza w zakresie najlepszych praktyk, na przykład „porty i adaptery” lub „architektura functional core, imperative shell”, a także czego po prostu oczekujemy od testów.

¹ <https://www.youtube.com/watch?v=bsmFVb8guMU>

Terminologia — różne typy testów

Testy odizolowane („czyste” testy jednostkowe) kontra testy zintegrowane

Podstawowym przeznaczeniem testu jednostkowego powinna być weryfikacja poprawności logiki aplikacji. Test *odizolowany* jest przeznaczony do sprawdzenia dokładnie jednego fragmentu kodu, a jego sukces lub niepowodzenie nie ma wpływu na żaden zewnętrzny kod. Stąd nazwa „czysty” test jednostkowy — sprawdzenie na przykład pojedynczej funkcji przez test utworzony w taki sposób, że tylko wspomniana funkcja może doprowadzić do jego niepowodzenia. Jeżeli działanie funkcji zależy od innego systemu, którego awaria oznacza także awarię testu, wówczas mamy test *zintegrowany*. Wspomnianym systemem może być system zewnętrzny, na przykład baza danych, lub inną funkcję, nad którą nie mamy kontroli. Niezależnie od tego, jeżeli awaria systemu prowadzi do awarii testu, oznacza to brak właściwej izolacji testu, który tym samym nie jest „czystym” testem jednostkowym. To niekoniecznie zła wiadomość, ale może oznaczać, że test wykonuje więcej niż jedno zadanie.

Testy integracji

Test integracji sprawdza, czy kontrolowany przez Ciebie kod jest prawidłowo zintegrowany z pewnym zewnętrznym systemem, nad którym nie masz kontroli. Testy *integracji* zwykle są również testami *zintegrowanymi*.

Testy systemu

Podczas gdy test integracji sprawdza integrację z zewnętrznym systemem, test systemu sprawdza integrację wielu systemów w aplikacji. Na przykład sprawdzane jest połączenie z bazą danych, powiązanie z plikami statycznymi, a także konfiguracja serwera w sposób zapewniający sprawne współdziałanie całości.

Testy funkcjonalne i testy akceptacji

Test akceptacji ma sprawdzić działanie systemu z punktu widzenia użytkownika („czy użytkownik zaakceptuje to zachowanie?”). Trudno jest przygotować test akceptacji, który nie uwzględnia pełnego stosu i nie będzie testem typu E2E (ang. *end-to-end*). Nasze testy funkcjonalne były używane w charakterze testów zarówno akceptacji, jak i systemu.

Teza — testy jednostkowe są niezwykle szybkie, mają także inne zalety

Często słyszałeś o szybkości wykonywania testów jednostkowych. Wprawdzie nie sądzę, aby to była podstawowa zaleta testów jednostkowych, ale warto lepiej poznać wspomnianą kwestię szybkości.

Szybsze testy oznaczają szybsze tworzenie kodu

Można powiedzieć, że im szybciej są wykonywane testy jednostkowe, tym lepiej. I do pewnego stopnia im szybsze są *wszystkie* testy, tym lepiej.

W książce przedstawiłem cykl TDD, czyli testowanie-tworzenie kodu. Na początek przedstawiłem sposób pracy stosowany w podejściu TDD i przechodzenie między tworzeniem niewielkich ilości kodu oraz wykonywaniem testów. W efekcie testy jednostkowe mogą być wykonywane wielokrotnie w ciągu minuty, natomiast testy funkcjonalne wielokrotnie w ciągu dnia.

Upraszczając: im dłużej trwa wykonanie testów jednostkowych, tym więcej czasu musisz poświęcić na czekanie na ich zakończenie, co przekłada się na wolniejsze prace nad kodem aplikacji. To jednak tylko jedna strona medalu.

Uczucie błogostanu

Przez chwilę pomyślmy w sposób socjologiczny. Jesteśmy programistami, mamy własną kulturę i religię. Znalazło się w niej wiele kongregacji, w tym także kult podejścia TDD, do stosowania którego zostałeś właśnie wprowadzony. Mamy wiernych użytkowników vi oraz heretyków korzystających z emacs. W jednej kwestii możemy się zgodzić — w konkretnej praktyce duchowej, swoistej transcendentalnej medytacji, istnieje uczucie błogostanu. To jest uczucie charakteryzujące się absolutnym skupieniem i koncentracją, w trakcie której nie czujesz upływu czasu, kod w naturalny sposób spływa z Twoich palców, a problemy są na tyle skomplikowane, aby mobilizować, i jednocześnie nie są w stanie nas pokonać...

Nie ma żadnych szans na osiągnięcie wspomnianego błogostanu, jeżeli nieustannie trzeba będzie długo oczekiwać na wykonanie testów. Każde oczekiwanie przekraczające kilka sekund powoduje rozproszenie uwagi, oderwanie od aktualnie wykonywanego zadania i błogostan pryska. Błogostan to bardzo delikatny stan; gdy zostanie przerwany, powrót do niego zajmie przynajmniej kwadrans.

Wolne testy nie są wykonywane zbyt często, co przekłada się na gorszej jakości kod

Jeżeli testy są wykonywane wolno i prowadzą do dekoncentracji, wówczas pojawia się niebezpieczeństwo unikania testów, co może prowadzić do powstawania błędów. Ewentualnie skutkiem będzie niechęć do przeprowadzenia refaktoryzacji kodu, ponieważ jak już wiesz, refaktoryzacja oznacza długie oczekiwanie na wykonanie wszystkich testów. W każdym z powyższych przypadków skutkiem będzie powstanie kodu o zdecydowanie gorszej jakości.

Teraz jest dobrze, ale wraz z upływem czasu testy zintegrowane są wykonywane coraz wolniej

Być może stwierdzisz: dobrze, ale nasz zestaw zawiera wiele zintegrowanych testów (ponad 50), których wykonanie zabiera jedynie 0,2 sekundy.

Jednak pamiętaj, że opracowaliśmy bardzo prostą aplikację. Gdy zaczniesz pracować nad bardziej skomplikowanymi projektami, a baza danych będzie zawierała więcej tabel i kolumn, wówczas testy zintegrowane zaczną być wykonywane coraz wolniej. Ponadto operacja zerowania przez Django bazy danych między poszczególnymi testami będzie zabierać coraz więcej czasu.

Nie zabieraj mi tego

Gary Bernhardt, czyli programista posiadający znacznie większe niż ja doświadczenie w zakresie testów, doskonale podsumował powyższe w wystąpieniu zatytułowanym *Szybki test, wolny test*². Zachęcam Cię do jego obejrzenia.

² <https://www.youtube.com/watch?v=RAxiiRPHS9k>

Testy jednostkowe pozwalają przygotować dobry projekt

Prawdopodobnie najważniejsze jest, aby zapamiętać lekcję z rozdziału 19. Przejście przez proces tworzenia dobrych, odizolowanych testów jednostkowych może pomóc w przygotowaniu lepszego kodu poprzez zmuszenie nas do wykrycia zależności i zachętą do stosowania luźniejszej architektury w sposób niemożliwy do użycia w przypadku testów zintegrowanych.

Problemy związane z czystymi testami jednostkowymi

Wszystko sprowadza się do ogromnego „ale”. Tworzenie odizolowanych testów jednostkowych wiąże się z pewnym ryzykiem, zwłaszcza jeśli (podobnie jak ja) nie masz biegłości w stosowaniu podejścia TDD.

Testy odizolowane mogą być trudniejsze w odczycie i zapisie

Przypomnij sobie pierwszy utworzony przez nas test odizolowany. Czy jego kod nie wydawał Ci się zagmatwany? Oczywiście to uległo zmianie po przeprowadzeniu refaktoryzacji na postać formularzy, ale wyobraź sobie, co by było, gdybyśmy nie zastosowali takiego rozwiązania. W kodzie pozostałby trudny w odczycie test. Nawet w ostatecznej wersji testów mogą znajdować się pewne mniej zrozumiałe fragmenty.

Testy odizolowane nie testują automatycznie integracji

Jak zobaczyłeś nieco wcześniej, testy odizolowane z natury testują jedynie sprawdzaną jednostkę kodu, zupełnie w izolacji. Nie zajmują się testowaniem integracji między poszczególnymi jednostkami kodu.

To jest doskonale znany problem i istnieją różne sposoby złagodzenia jego skutków. Jak mogłeś zobaczyć, odbywa się to jednak kosztem większego wysiłku ponoszonego przez programistę. Konieczne jest monitorowanie relacji między jednostkami kodu w celu identyfikacji występujących między poszczególnymi komponentami niejawnych kontraktów, które trzeba będzie honorować. Ponadto powstaje konieczność opracowania testów dla tych kontraktów jako wewnętrznej funkcjonalności danej jednostki kodu.

Testy jednostkowe rzadko przechytują nieoczekiwane błędy

Testy jednostkowe pomogą w wyłapywaniu błędów typu pomyłka o jeden, a także błędów logicznych. To są typy błędów popełnianych całym czas, a więc spodziewanych. Natomiast testy jednostkowe nie ostrzegają o pewnych mniej oczekiwanych błędach. Nie przypomną o konieczności przeprowadzenia migracji bazy danych. Nie poinformują, że warstwa pośrednia przeprowadza pewne sprytne oczyszczanie encji HTML w sposób zakłócający generowanie danych... Na myśl przychodzi tutaj powiedzenie Donalda Rumsfelda o znanych niewiadomych.

Testy oparte na imitacji stają się ścisłe powiązane z implementacją

Wreszcie testy oparte na imitacji stają się ścisłe powiązane z implementacją. Jeżeli do utworzenia obiektów zdecydujesz się na użycie `List.objects.create()`, natomiast imitacje oczekują użycia `List()` i `.save()`, wówczas testy zakończą się niepowodzeniem, nawet jeśli rzeczywisty

efekt działania kodu pozostanie taki sam. Jeżeli nie zachowasz ostrożności, wówczas staniesz się przeciwnikiem jednej z największych zalet testów, czyli zachęty do przeprowadzenia refaktoryzacji. Gdy będziesz chciał zmienić wewnętrzne API, wtedy może okazać się, że to wymaga również zmiany dziesiątek testów opartych na imitacjach oraz przetestowania kontraktów.

Zwróć uwagę, że to może być większy problem podczas pracy z API pozostającym poza Twoją kontrolą. Być może pamiętasz, jak musieliszy się nagimnastykować, aby przetestować budowany formularz — użyliśmy dwóch klas modelu Django oraz funkcji `side_effect` do sprawdzania stanu środowiska zewnętrznego aplikacji. Jeżeli tworzysz kod pozostający pod Twoją całkowitą kontrolą, wówczas prawdopodobnie opracujesz wewnętrzne API w taki sposób, aby było jasne i wymagało mniejszego wysiłku podczas jego testowania.

Jednak wszystkie wymienione problemy można pokonać

Jednak piewcy izolacji mogą upierać się, że wszelkie problemy da się pokonać i trzeba jedynie nabrać większej wprawy w tworzeniu odizolowanych testów. Czy pamiętasz o uczuciu błogostanu? Uczucie błogostanu!

W którym miejscu się znajdujemy?

Synteza — jakie mamy oczekiwania wobec testów?

Zatrzymajmy się na chwilę i zastanówmy — jakich korzyści wynikających z użycia testów oczekujemy? Dlaczego tworzymy je najpierw?

Poprawność

Naszym celem jest opracowanie aplikacji pozbawionej błędów — w logice działającej na niskim poziomie, błędów typu pomyłka o jeden oraz błędów na wysokim poziomie — aby oprogramowanie dostarczyło użytkownikom żądanej przez nich funkcjonalności. Ponadto chcemy się dowiedzieć o ewentualnym wprowadzeniu regresji, która może uszkodzić działającą funkcjonalność, oraz ogólnie dowiadywać się o problemach, zanim użytkownicy je zauważą. Oczekujemy od testów informacji, że aplikacja działa poprawnie.

Czytelny, łatwy w obsłudze kod

Chcemy, aby tworzony kod spełniał reguły takie jak YAGNI i DRY. Kod powinien jasno wyrażać intencje programisty, być podzielony na sensowne komponenty doskonale definiujące obowiązki oraz być łatwy do zrozumienia. Oczekujemy od testów nieustannej zachęty do refaktoryzacji aplikacji, aby nigdy nie mieć obaw przed jej usprawnieniem. Byłyby również dobrze, gdyby testy aktywnie pomagały w znalezieniu właściwego projektu aplikacji.

Produktywna praca

Wreszcie chcemy, aby testy pomogły w szybkiej i efektywnej pracy. Oczekujemy od testów pomocy w postaci zmniejszenia stresu podczas programowania oraz zapewnienia ochrony przed głupimi pomyłkami. Testy powinny pomóc programistę pozostać w stanie „błogostanu” nie dlatego, że lubi ten stan, ale ponieważ zapewnia on wysoką produktywność.

Test powinien jak najszybciej udzielać informacji zwrotnych, aby możliwie szybko można było wypróbowywać nowe pomysły. Nie chcemy odbierać testów bardziej jako przeszkody niż pomocy podczas opracowywania kodu.

Oceń testy pod kątem korzyści, jakich oczekujesz dzięki ich użyciu

Nie sądzę, aby istniały jakiekolwiek uniwersalne reguły dotyczące ilości testów koniecznych do utworzenia, a także sposobu zachowania odpowiedniej równowagi między testami funkcjonalnymi, zintegrowanymi i odizolowanymi. Okoliczności zmieniają się w zależności od projektów. Jednak myśląc o testach i zadając sobie pytania odnośnie do oczekiwanych korzyści, można podjąć pewne decyzje.

Cel	Rozważania
Poprawność	<ul style="list-style-type: none">• Czy mam wystarczającą ilość testów funkcjonalnych, aby zachować pewność, że aplikacja <i>naprawdę</i> działa z punktu widzenia użytkownika?• Czy wystarczająco dokładnie przetestowałem wszystkie sytuacje skrajne? To wygląda na zadanie dla działających na niskim poziomie testów odizolowanych.• Czy mam testy sprawdzające, czy wszystkie komponenty właściwie ze sobą współdziałają? Czy mogę do tego celu użyć pewnych testów zintegrowanych, czy wystarczające będzie zastosowanie jedynie testów funkcjonalnych?
Czytelny, łatwy w obsłudze kod	<ul style="list-style-type: none">• Czy moje testy dają zięchetę do refaktoryzacji kodu często i bez obaw?• Czy moje testy pchają mnie w kierunku dobrego projektu aplikacji? Jeżeli mam dużą ilość testów zintegrowanych i kilka odizolowanych, to czy jakiekolwiek fragmenty aplikacji, dla których utworzę dodatkowe testy odizolowane, dostarczą użyteczniejsze informacje o projekcie aplikacji?
Produktywna praca	<ul style="list-style-type: none">• Czy cykl otrzymywania informacji zwrotnych jest tak szybki, jak tego oczekuję? Kiedy jestem ostrzegany o błędach i czy istnieje jakikolwiek praktyczny sposób, aby ostrzegać mnie wcześniej?• Jeżeli mam dużo działających na wysokim poziomie testów funkcjonalnych, których wykonanie zabiera dużo czasu, i muszę oczekwać całą noc na otrzymanie informacji o przypadkowych regresjach, to czy istnieje jakikolwiek możliwość opracowania szybciej wykonywanych testów — być może testów zintegrowanych — pozwalających na szybsze uzyskanie informacji zwrotnych?• Czy jeśli zachodzi potrzeba, mogę wykonać jedynie część testów z pakietu?• Czy poświęcam zbyt dużo czasu, oczekując na wykonania testów, a tym samym przeznaczam mniejszą ilość czasu na produktywną pracę?

Rozwiązania architektoniczne

Istnieją pewne rozwiązania architektoniczne, które mogą pomóc w osiągnięciu jak największych korzyści z pakietu testów, a w szczególności pomagają unikać pewnych wad testów odizolowanych.

Głównym celem wspomnianych rozwiązań jest próba określenia granic systemu, czyli miejsc, w których Twój kod rozpoczyna pracę z zewnętrznymi systemami — takimi jak baza danych, system plików, internet, interfejs użytkownika — i próbuje oddzielić je od podstawowej logiki biznesowej tworzonej aplikacji.

Porty i adaptery, czysta architektura i architektura heksagonalna

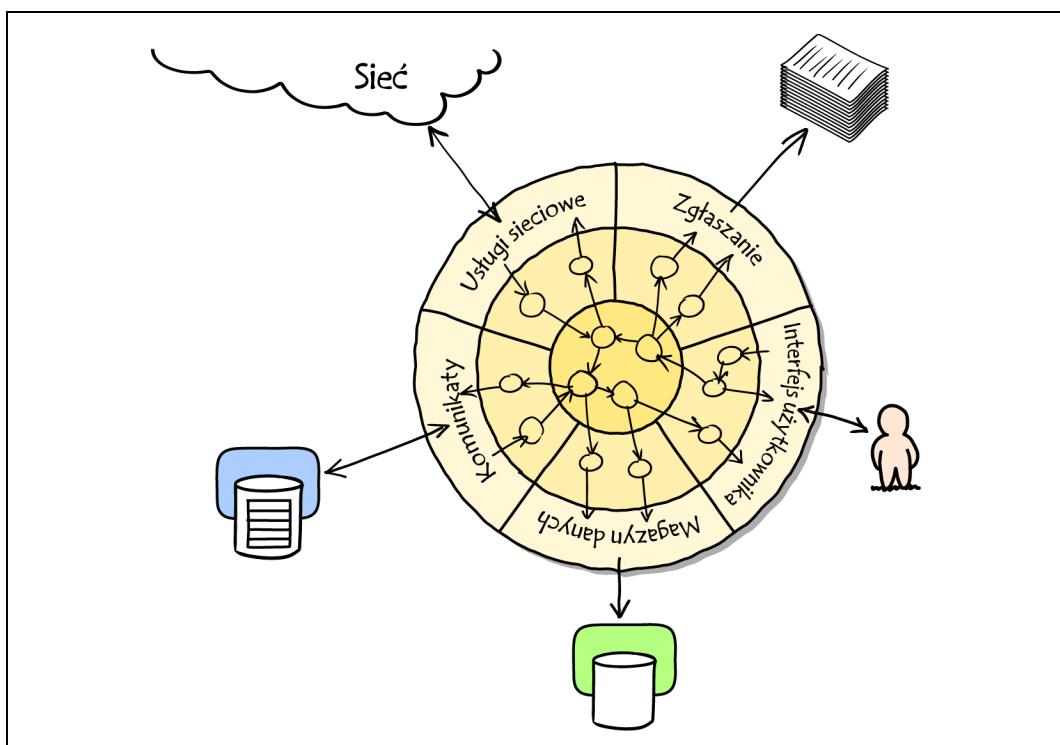
Testy zintegrowane są najbardziej użyteczne na *granicach* systemu, czyli w miejscach, w których Twój kod rozpoczyna pracę z zewnętrznymi systemami, takimi jak baza danych, system plików lub komponenty interfejsu użytkownika.

Istnieją granice, które mają negatywny wpływ na izolację testu, a imitacje są tutaj najgorsze, ponieważ ich ograniczenia prawdopodobnie będą najbardziej irytujące, jeśli testy są ściśle powiązane z implementacją lub gdy zachodzi konieczność uzyskania większej gwarancji o poprawnej integracji komponentów.

Z kolei kod w *jadrze* aplikacji — czyli skoncentrowany wyłącznie na logice biznesowej i regułach biznesowych, a przy tym pozostający pod Twoją całkowitą kontrolą — nie wymaga aż tak dokładnego sprawdzania za pomocą testów zintegrowanych, ponieważ całkowicie kontrolujesz ten kod.

Jednym ze sposobów osiągnięcia zamierzonych celów jest minimalizacja ilości kodu wymaganego do pracy z granicami. Następnie testujemy podstawową logikę biznesową za pomocą testów odizolowanych oraz sprawdzamy punkty integracji za pomocą testów zintegrowanych.

Steve Freeman i Nat Pryce w książce *Growing Object-Oriented Software Guided by Tests* nazwali takie podejście „portami i adapterami” (patrz rysunek 22.1).



Rysunek 22.1. Porty i adaptery (diagram opracowany przez Nata Pryce'a)

Tak naprawdę w stronę architektury portów i adapterów zaczeliśmy dążyć w rozdziale 19., gdy uznaliśmy, że tworzenie odizolowanych testów jednostkowych zachęciło nas do wyciągnięcia kodu ORM z głównej aplikacji i jego ukrycia w funkcjach pomocniczych w warstwie modelu.

Tego rodzaju podejście jest czasami określane mianem „czysta architektura” lub też „architektura heksagonalna”. Więcej informacji na ten temat znajdziesz w źródłach wymienionych na końcu rozdziału.

Architektura Functional Core, Imperative Shell

Gary Bernhardt poszedł jeszcze dalej i zaproponował architekturę, której nadał nazwę Functional Core, Imperative Shell. W tym przypadku „powłoka” aplikacji — miejsce, w którym zachodzą interakcje z granicami — stosuje paradygmat programowania imperatywnego i może być sprawdzana za pomocą testów zintegrowanych, testów akceptacji, a nawet (uwaga) wcale nie być sprawdzana, jeśli będzie zachowana na minimalnym poziomie. Jednak podstawowa część aplikacji jest w rzeczywistości tworzona za pomocą paradygmatu programowania funkcyjonalnego (pełnego, bez „efektów ubocznych”), co pozwala na wykorzystanie w pełni odizolowanych, „czystych” testów jednostkowych, całkowicie pozbawionych imitacji.

Więcej informacji na temat tego podejścia znajdziesz w przygotowanej przez Gary'ego prezentacji zatytułowanej *Boundaries*³ (czyli granice).

Podsumowanie

Spróbowałem przedstawić ogólne omówienie bardziej zaawansowanych kwestii, które mogą pojawiać się podczas stosowania procesu TDD. Opanowanie tych tematów to coś, co wymaga wielu lat praktyki. Dlatego też nie czuję się w pełni upoważniony do wypowiadania się w tym zakresie. Gorąco zachęcam Cię do zachowania pewnej dozy ostrożności i próby ustalenia, co tak naprawdę sprawdza się w Twoim przypadku. I jeszcze najważniejsze, zapoznaj się z opiniami prawdziwych ekspertów!

Poniżej wymieniłem źródła, w których znajdziesz więcej informacji dotyczących testów.

³ <https://www.youtube.com/watch?v=eOYal8elnZk>

Dalsza lektura

Szybkie testy, wolne testy i granice

Wystąpienia Gary'ego Bernhardta na konferencjach Pycon w latach 2012⁴ i 2013⁵. Warto również obejrzeć przygotowane przez niego screencasty (<https://www.destroyallsoftware.com/screencasts>).

Porty i adaptery

Steve Freeman i Nat Pryce omówili to podejście w swej książce (*Growing Object-Oriented Software Guided by Tests*). Poświęcone temu wystąpienie znajdziesz tutaj⁶. Zapoznaj się także z przedstawionym przez wujka Boba omówieniem czystej architektury⁷ oraz zaprezentowanym przez Alistair Cockburnem omówieniem architektury heksagonalnej⁸.

Gorąca lawa

Casey Kinsey przedstawił tutaj⁹ ostrzeżenie, aby unikać bazy danych, gdy tylko istnieje taka możliwość.

Odwrócona piramida

Pomysł, aby projekt kończył się ze zbyt dużym współczynnikiem wolnych, wykonywanych na wysokim poziomie testów do testów jednostkowych, a także wizualna metafora dla wysiłku odwrócenia tego współczynnika (<http://watirmelon.com/tag/testing-pyramid/>).

Testy zintegrowane to przekrót

Na stronie <http://blog.thecodewhisperer.com/2010/10/16/integrated-tests-are-a-scam/> znajduje się sława rozprawa J.B. Rainsbergera o tym, że testy zintegrowane mogą zrujnować Ci życie. Prezentację wideo możesz obejrzeć tutaj¹⁰ i tutaj¹¹ (są dostępne dwie prezentacje, żadna z nich nie jest szczytem kinematografii). Następnie na stronie <http://www.jbrains.ca/permalink/using-integration-tests-mindfully-a-case-study> zapoznaj się z kilkoma postami dotyczącymi omawianego tematu, zwłaszcza z postem broniącym testów akceptacji (ja nazywam je testami funkcjonalnymi). Natomiast na stronie <http://www.jbrains.ca/permalink/part-2-some-hidden-costs-of-integration-tests> znajdziesz analizę pokazującą, jak wolno wykonywane testy zabijają produktywność.

Podejście pragmatyczne

Martin Fowler (autor książki poświęconej refaktoryzacji) prezentuje rozsądnie zrównoważone, pragmatyczne podejście¹².

⁴ <https://www.youtube.com/watch?v=RAxiiRPHS9k>

⁵ <https://www.youtube.com/watch?v=eOYal8elnZk>

⁶ <https://vimeo.com/83960706>

⁷ <http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html>

⁸ <http://alistair.cockburn.us/Hexagonal+architecture>

⁹ <https://www.youtube.com/watch?v=bsmFVb8guMU>

¹⁰ <http://www.infoq.com/presentations/integration-tests-scam>

¹¹ <https://vimeo.com/80533536>

¹² <http://martinfowler.com/bliki/UnitTest.html>

Kieruj się Testing Goat!

Powracamy do Testing Goat.

Już słyszę, jak jęczysz: *Harry, Testing Goat przestał być zabawny jakieś 17 rozdziałów temu*. Bądź cierpliwy, ponieważ chcę napisać tutaj coś ważnego.

Testowanie jest trudne

Sądzę, że zwrot „kieruj się Testing Goat” „chwycił” mnie, gdy po raz pierwszy się z nim spotkałem, ponieważ testowanie jest trudne. Nie chodzi o testy same w sobie, ale ich *nieustanne* stosowanie.

Zawsze łatwiejsza będzie droga na skróty i pominięcie kilku testów. Pod względem psychologicznym nieustanne stosowanie testów jest dwukrotnie trudniejsze, ponieważ korzyści płynące z testów pojawiają się w znacznie innym miejscu i czasie, a nie w chwili poniesienia wysiłku ich utworzenia. Test, nad którym teraz pracujesz, nie przyniesie natychmiastowej nagrody, a jedynie będzie pomocny na późniejszym etapie prac. Być może po kilku miesiącach pomoże uniknąć wprowadzenia błędów w trakcie refaktoryzacji lub też wychwyci regresję po uaktualnieniu zależności. Mamy jeszcze inną możliwość — że test odpłaci w sposób trudny do zmierzenia, zachęcając do tworzenia lepiej zaprojektowanego kodu. Ty jednak będziesz się starał wmówić sobie, że tak elegancki kod mógłbyś utworzyć również bez użycia testów.

Sam zacząłem popełniać ten błąd podczas tworzenia frameworka testów na potrzeby książki. To jest całkiem skomplikowana bestia i ma swoje testy. Jednak w kilku miejscach zdecydowałem się pójść na skróty i otrzymałem rozwiązanie, które nie jest doskonałe. Teraz żałuję przyjętego podejścia, ponieważ framework okazał się zupełnie niewygodny i brzydki (najedzie dzień, w którym kod frameworka udostępnięę jako open source, i wtedy będziesz mógł sam się o tym przekonać).

Zachowuj kolor zielony w serwerze ciągłej integracji

Innym obszarem wymagającym naprawdę ciężkiej pracy jest ciągła integracja. W rozdziale 20. zobaczyłeś, że w ciągłej integracji czasami pojawiają się dziwne i nieprzewidywane błędy. Kiedy na nie spojrzyś i pomyślisz: „To przecież działa doskonale w moim komputerze”, wówczas pojawia się silna pokusa po prostu ich zignorowania. Jeżeli nie zachowasz wyjątkowej ostrożności, zaczniesz tolerować niepowodzenia zestawu testów w ciągłej integracji,

wtedy bardzo szybko kompilacje przeprowadzane przez serwer ciągłej integracji staną się bezużyteczne. Ponadto będzie można odnieść wrażenie, że przywrócenie prawidłowego działania wymaga zbyt dużego nakładu pracy. Nie wpadaj w tę pułapkę. Nie ustawaj w wysiłkach, a znajdziesz powód niepowodzenia testu i sposób na jego usunięcie — wtedy w serwerze ciągłej integracji znów pojawi się kolor zielony.

Bądź dumny z testów, podobnie jak z kodu

Jedną z rzeczy pomagających przestać myśleć o testach jako mało ważnych dodatkach do „rzeczywistego” kodu jest potraktowanie ich jako części końcowego produktu, który będziesz. Ten fragment kodu również powinienny być dopracowany, przyjemny pod względem estetycznym, abyś mógł być z niego dumny...

Zrób tak, ponieważ do tego nawołuje Testing Goat. Zrób tak, ponieważ wiesz, że przyniesie to korzyści, choć prawdopodobnie nie od razu. Zrób to z poczucia obowiązku i profesjonalizmu lub po prostu z czystej przekory. Zrób tak, ponieważ to jest dobra praktyka. Zrób tak, ponieważ w ten sposób programowanie przyniesie Ci więcej radości.

Nie zapomnij o napiwku

Powstanie tej książki byłoby niemożliwe bez pomocy wydawcy, czyli wspaniałego O'Reilly Media. Jeżeli czytasz bezpłatne wydanie w internecie, rozważ zakup drukowanego egzemplarza¹. Jeżeli nie potrzebujesz dla siebie, to może sprezentujesz go przyjacielowi?

Przedstaw się

Mam nadzieję, że lektura książki sprawiła Ci przyjemność. Skontaktuj się ze mną i napisz, co o niej myślisz.

Harry.

- @hjwp
- obeythetestinggoat@gmail.com



¹ <http://helion.pl/ksiazki/tddwpr.htm>

DODATKI



PythonAnywhere

Czy planujesz użycie PythonAnywhere podczas lektury książki? W tym dodatku znajdziesz kilka podpowiedzi i wskazówek, jak zapewnić działanie całości. Skoncentrujemy się przede wszystkim na testach Selenium/Firefox, działaniu serwera testowego i zrzutach ekranu.

Jeżeli jeszcze tego nie zrobiłeś, załącz konto w PythonAnywhere. Na potrzeby książki w zupełności wystarczy to bezpłatne.

Sesje Firefox Selenium przy użyciu Xvfb

Warto pamiętać, że PythonAnywhere to środowisko działające wyłącznie w konsoli, a więc nie oferuje ekranu, na którym można wyświetlić okno przeglądarki internetowej Firefox. Jednak bardzo łatwo można utworzyć ekran wirtualny.

W rozdziale 1., gdy tworzyliśmy nasz pierwszy test, okazało się, że nie wszystko działa zgodnie z oczekiwaniemi. Poniżej przedstawiono kod pierwszego testu, możesz go wprowadzić za pomocą edytora w PythonAnywhere:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://localhost:8000')
assert 'Django' in browser.title
```

Jednak gdy spróbujesz go wykonać (w powłoce bash), wówczas otrzymasz błąd:

```
$ python3 functional_tests.py
Traceback (most recent call last):
File "tests.py", line 3, in <module>
    browser = webdriver.Firefox()
File "/usr/local/lib/python3.3/site-packages/selenium/webdriver/firefox/webdrive
self.binary, timeout),
File "/usr/local/lib/python3.3/site-packages/selenium/webdriver/firefox/extensio
self.binary.launch_browser(self.profile)
File "/usr/local/lib/python3.3/site-packages/selenium/webdriver/firefox/firefox_
self._wait_until_connectable()
File "/usr/local/lib/python3.3/site-packages/selenium/webdriver/firefox/firefox_
self._get_firefox_output())
selenium.common.exceptions.WebDriverException: Message: 'The browser appears to
have exited before we could connect. The output was: Error: no display
specified\n'
```

Rozwiązaniem jest użycie Xvfb, czyli X Virtual Framebuffer. To spowoduje uruchomienie ekranu „wirtualnego”, który może być wykorzystany przez przeglądarkę internetową Firefox, nawet jeśli serwer nie oferuje rzeczywistego ekranu.



Jeżeli zamiast powyższego otrzymasz błąd „ImportError, no module named selenium”, wówczas wydaj polecenie `pip3 install --user selenium`.

Polecenie `xvfb-run` spowoduje uruchomienie następnego polecenia w Xvfb. Wydanie poniższego zakończy się oczekiwanyem niepowodzeniem:

```
$ xvfb-run python3 functional_tests.py
Traceback (most recent call last):
File "tests.py", line 11, in <module>
assert 'Django' in browser.title
AssertionError
```

Konfiguracja Django jako aplikacji sieciowej PythonAnywhere

Następnym krokiem jest konfiguracja Django. Zamiast użycia polecenia `django-admin.py start project` zalecam skorzystanie z opcji umieszczonej na karcie *Web* w PythonAnywhere. Dodaj nową aplikację sieciową, wybierz Django i Python 3, a następnie podaj *superlists* jako nazwę projektu.

Teraz zamiast uruchamiać serwer testowy z konsoli w komputerze lokalnym (`localhost:8000`), możesz wykorzystać rzeczywisty adres URL aplikacji sieciowej PythonAnywhere:

```
browser.get('http://nazwa_użytkownika.pythonanywhere.com')
```



W celu uaktualnienia witryny musisz pamiętać o kliknięciu przycisku *Reload Web App* po wprowadzeniu zmiany w kodzie.

Teraz wszystko powinno działać lepiej¹.

Porządek w katalogu /tmp

Selenium i Xvfb mają tendencję do pozostawiania ogromnej ilości śmieci w katalogu `/tmp`, zwłaszcza w przypadku nieeleganckiego zakończenia pracy (teraz już wiesz, dlaczego wcześniej używałem konstrukcji `try-finally`).

¹ Zamiast przedstawionego rozwiązania mógłbyś uruchomić serwer programistyczny Django z poziomu konsoli. Jednak problem polega na tym, że konsole PythonAnywhere nie zawsze są uruchomione w tym samym serwerze, a więc nie ma żadnej gwarancji, że konsola, w której wykonujesz testy, będzie znajdowała się w tym samym serwerze, w którym został uruchomiony serwer programistyczny Django. Ponadto po uruchomieniu serwera w konsoli nie ma łatwego sposobu na wizualne sprawdzenie, jak wygląda witryna.

W rzeczywistości ilość pozostawianych przez nie śmieci jest na tyle duża, że może szybko doprowadzić do wykorzystania całej przydzielonej Ci pamięci masowej. Warto więc często usuwać zawartość katalogu `/tmp` za pomocą poniższego polecenia:

```
$ rm -rf /tmp/*
```

Zrzuty ekranu

W rozdziale 5. zasugerowałem użycie `time.sleep()` w celu wstrzymania testu funkcjonalnego, aby można było zobaczyć, co przeglądarka Selenium pokazuje na ekranie. Takiego podejścia nie można zastosować w przypadku PythonAnywhere, ponieważ okno przeglądarki jest wyświetlane na ekranie wirtualnym. Zamiast tego możesz przeanalizować wdrożoną witrynę lub „uwierzyć mi na słowo” w kwestii tego, co powinieneś zobaczyć na ekranie.

W przypadku ekranu wirtualnego najlepszym sposobem na oglądanie wizualnych efektów testów jest użycie zrzutów ekranu. Zajrzyj do rozdziału 20., a znajdziesz tam kilka przykładów przedstawiających takie podejście.

Rozdział poświęcony wdrożeniom

Kiedy dotrzesz do rozdziału 8., będziesz mógł kontynuować użycie PythonAnywhere lub nauczyć się, jak zbudować „prawdziwy” serwer. Zalecam to drugie rozwiązanie, ponieważ oferuje ono większe możliwości.

Jeżeli naprawdę chcesz pozostać przy PythonAnywhere, jednym z rozwiązań będzie wdrożenie drugiej kopii aplikacji w innej domenie. Do tego będzie potrzebna własna domena oraz płatne konto w PythonAnywhere. Nawet jeśli nie spełnisz tych warunków, to nadal możesz upewnić się o wykonywaniu testów funkcjonalnych w trybie „prowizorycznym” względem rzeczywistej witryny, zamiast używać wątkowanego serwera z `LiveServerTestCase`.



Jeżeli używasz PythonAnywhere podczas lektury książki, chciałbym się dowiedzieć, na jakie rozwiązanie się zdecydowałeś. Wyślij mi wiadomość e-mail na adres obeythetestinggoat@gmail.com.

Widoki oparte na klasach Django

Ten dodatek stanowi uzupełnienie dla rozdziału 12., w którym zaimplementowaliśmy formularze Django przeznaczone do weryfikacji oraz przeprowadziliśmy refaktoryzację widoków. Na końcu wspomnianego rozdziału nasze widoki nadal używały funkcji.

Nowością w świecie Django są jednak widoki oparte na klasach. W tym dodatku przeprowadzimy refaktoryzację aplikacji, aby móc użyć widoków opartych na klasach zamiast funkcji widoków. Precyzując: użyjemy *ogólnych* widoków opartych na klasach.

Ogólne widoki oparte na klasach

Istnieje pewna różnica między widokami opartymi na klasach i *ogólnymi* widokami opartymi na klasach. Te pierwsze to po prostu jeszcze inny sposób zdefiniowania funkcji widoku. W tego rodzaju definiowaniu przyjęto kilka założeń dotyczących sposobu działania widoków. Oferują one jedną ogromną przewagę nad funkcjami widoku: możliwość tworzenia podklas. Oczywiście kosztem wymienionej możliwości jest mniejsza czytelność widoku niż w przypadku widoków opartych na funkcjach. Podstawowe przeznaczenie *zwykłych* widoków opartych na klasach to użycie w sytuacji, gdy kilka widoków wykorzystuje tę samą logikę. W ten sposób stosujemy zasadę DRY. W przypadku widoków opartych na funkcjach trzeba będzie użyć funkcji pomocniczych lub dekoratorów. W teorii użycie struktury klasy może pomóc w przygotowaniu bardziej eleganckiego rozwiązania.

Ogólne widoki oparte na klasach próbują dostarczyć łatwe w odczycie rozwiązanie przeznaczone do typowych zastosowań, takich jak pobieranie obiektu z bazy danych i przekazanie go szablonowi, pobieranie listy obiektów, zapis danych wejściowych użytkownika przekazanych przez żądanie POST przy użyciu ModelForm itd. Wydaje się, że wymienione przypadki dotyczą aplikacji budowanej w książce, ale jak się wkrótce przekonasz, diabeł tkwi w szczegółach.

W tym miejscu powinienem stwierdzić, że nie stosowałem zbyt często żadnego rodzaju widoków opartych na klasach. Na pewno dostrzegam ich sens i potencjalnie wiele zastosowań w aplikacjach Django, w których doskonale się sprawdzają ogólne widoki oparte na klasach. Jednak gdy tylko nieco wykroczyzsz poza prosty sposób użycia — na przykład zechcesz wykorzystać więcej niż tylko jeden model — wówczas okaże się, że widoki oparte na klasach mogą prowadzić do powstania kodu znacznie bardziej skomplikowanego w odczycie niż w przypadku klasycznych funkcji widoku (oczywiście to kwestia dyskusyjna).

Ponieważ jesteśmy zmuszeni do stosowania wielu opcji dopasowujących do naszych potrzeb widoki oparte na klasach, implementacja tego rodzaju widoków w omawianej tutaj aplikacji dostarczy sporo informacji o sposobie ich działania i testowania za pomocą testów jednostkowych.

Mam nadzieję, że same testy jednostkowe używane do sprawdzenia widoków opartych na funkcjach będą działały równie dobrze w przypadku widoków opartych na klasach. Zobaczmy, jak to wygląda w praktyce.

Strona główna jako widok FormView

Nasza strona główna po prostu wyświetla formularz w szablonie:

```
def home_page(request):
    return render(request, 'home.html', {'form': ItemForm()})
```

Przeglądając dostępne *opcje*¹, zobaczymy, że Django oferuje ogólny widok o nazwie `FormView`, który teraz wykorzystamy.

Plik `lists/views.py` (ch31l001):

```
from django.views.generic import FormView
[...]

class HomePageView(FormView):
    template_name = 'home.html'
    form_class = ItemForm
```

Wskazujemy szablon i widok przeznaczone do użycia. Kolejnym krokiem jest uaktualnienie pliku `urls.py` i zastąpienie poniższym wiersza `lists.views.home_page`.

Plik `superlists/urls.py` (ch31l002):

```
url(r'^$', HomePageView.as_view(), name='home'),
```

Teraz możemy wykonać wszystkie testy. To było łatwe...

```
$ python3 manage.py test lists
[...]
```

```
Ran 34 tests in 0.119s
OK
```

```
$ python3 manage.py test functional_tests
[...]
```

```
Ran 4 tests in 15.160s
OK
```

Jak dotąd wszystko przebiegło dobrze. Jednowierszową funkcję widoku zastąpiliśmy dwuwierszową klasą, ale kod nadal pozostaje czytelny. To dobra chwila na przekazanie kodu do repozytorium...

¹ <https://docs.djangoproject.com/en/1.6/ref/class-based-views/>

Użycie form_valid w celu dostosowania widoku CreateView do naszych potrzeb

Kolejnym krokiem jest modyfikacja widoku używanego do utworzenia zupełnie nowej listy. Obecnie odpowiada za to funkcja `new_list()`. Poniżej przedstawiono kod w jego dotychczasowej postaci.

Plik `lists/views.py`:

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list = List.objects.create()
        form.save(for_list=list)
        return redirect(list)
    else:
        return render(request, 'home.html', {"form": form})
```

Przeglądamy dostępne do użycia widoki oparte na klasach, prawdopodobnie chcemy użyć `CreateView`. Wiemy, że używamy klasy `ItemForm`, a więc zobaczymy, jak wszystko połączyć w całość oraz czy testy nam w tym pomogą.

Plik `lists/views.py` (ch31l003):

```
from django.views.generic import FormView, CreateView
[...]

class NewListView(CreateView):
    form_class = ItemForm

    def new_list(request):
        [...]
```

W pliku `views.py` pozostawiamy starą funkcję widoku, co pozwoli na skopiowanie z niej kodu. Gdy wszystko będzie już działało, wspomnianą funkcję można usunąć. Sama funkcja pozostaje nieszkodliwa po zmianie mapowania adresów URL, które przeprowadzamy w pliku `urls.py`.

Plik `lists/urls.py` (ch31l004):

```
from django.conf.urls import patterns, url
from lists.views import NewListView

urlpatterns = patterns('',
    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^new$', NewListView.as_view(), name='new_list'),
)
```

Po ponownym wykonaniu testów otrzymujemy poniższe błędy:

```
$ python3 manage.py test lists

ERROR: test_for_invalid_input_passes_form_to_template
(lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'

ERROR: test_for_invalid_input_renders_home_template (lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
```

```
either a definition of 'template_name' or an implementation of
'get_template_names()'

ERROR: test_invalid_list_items_arent_saved (lists.tests.ListViewTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'

ERROR: test_redirects_after_POST (lists.tests.ListViewTest)
TypeError: save() missing 1 required positional argument: 'for_list'

ERROR: test_saving_a_POST_request (lists.tests.ListViewTest)
TypeError: save() missing 1 required positional argument: 'for_list'

ERROR: test_validation_errors_are_shown_on_home_page (lists.tests.ListViewTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'

Ran 34 tests in 0.125s

FAILED (errors=6)
```

Rozpoczynamy od błędu informującego o braku szablonu — może wystarczy po prostu dodać odpowiedni szablon?

Plik *lists/views.py* (ch31l005):

```
class NewListView(CreateView):
    form_class = ItemForm
    template_name = 'home.html'
```

Teraz zajmiemy się pozostaymi niepowodzeniami. Jak możesz zobaczyć, występują one w funkcji `form_valid()` widoku. To jest funkcja, którą nadpisujemy w celu dostarczenia wymaganej przez nas funkcjonalności w ogólnym widoku opartym na klasie. Jak sama nazwa wskazuje, funkcja jest wywoływana, gdy widok wykryje istnienie prawidłowego formularza. Możemy skopiować kod ze starej funkcji widoku znajdujący się po wywołaniu `if form.is_valid():`.

Plik *lists/views.py* (ch31l005):

```
class NewListView(CreateView):
    template_name = 'home.html'
    form_class = ItemForm

    def form_valid(self, form):
        list_ = List.objects.create()
        form.save(for_list=list_)
        return redirect(list_)
```

Wszystkie testy zostały zaliczone!

```
$ python3 manage.py test lists
Ran 34 tests in 0.119s
OK
$ python3 manage.py test functional_tests
Ran 4 tests in 15.157s
OK
```

Moglibyśmy zaoszczędzić jeszcze dwa wiersze, próbując zastosować się do reguły DRY i wykorzystać w tym celu główną zaletę widoków opartych na klasach, czyli dziedziczenie.

Plik *lists/views.py* (ch31l007):

```
class NewListView(CreateView, HomePageView):  
  
    def form_valid(self, form):  
        list = List.objects.create()  
        Item.objects.create(text=form.cleaned_data['text'], list=list)  
        return redirect('/lists/%d/' % (list.id,))
```

Testy nadal są zaliczane:

OK



To naprawdę nie jest dobra praktyka programowania zorientowanego obiektowo. Dziedziczenie narzuca związek typu „to jest...”. Prawdopodobnie nie będzie najlepiej, jeśli powiemy, że widok nowej listy „to jest” widok strony głównej... Dlatego też unikaj przedstawionego powyżej rozwiązania.

Z ostatnim krokiem lub bez niego, jak powyższe rozwiązanie można porównać ze starym? Mogę powiedzieć, że nie przedstawia się źle. Wprawdzie występuje pewien powtarzający się kod, ale sam widok nadal pozostaje całkiem prawidłowy. Reasumując, mamy remis — punkt dla ogólnych widoków opartych na klasach i jednocześnie jedna ich wada.

Bardziej złożony widok przeznaczony do obsługi wyświetlania listy i dodawania do niej elementów

Opracowanie tego widoku wymagało *wielu* prób. Muszę w tym miejscu przyznać, że choć testy wskazywały poprawne wykonanie danego zadania, nie okazały się zbyt pomocne w określaniu kolejnych kroków... Praca praktyczna opierała się na metodzie prób i błędów oraz majstrowaniu przy funkcjach, takich jak `get_context_data()`, `get_form_kwargs()` itd.

W trakcie tego zadania pozałem prawdziwą wartość przygotowania oddzielnych testów, z których każdy sprawdzał tylko jedną rzecz. W wyniku tego doświadczenia przeredagowałem materiał w rozdziałach od 10. do 12.

Testy nas prowadzą, przynajmniej przez chwilę

Oto przykładowy sposób wykonania zadania. Zaczynamy od założenia, że chcemy użyć widoku `DetailView` przeznaczonego do wyświetlania szczegółowych informacji o obiekcie.

Plik *lists/views.py*:

```
from django.views.generic import FormView, CreateView, DetailView  
[...]  
  
class ViewAndAddToList(DetailView):  
    model = List
```

Otrzymujemy następujący błąd:

```
[...]  
AttributeError: Generic detail view ViewAndAddToList must be called with either  
an object pk or a slug.  
FAILED (failures=5, errors=6)
```

Komunikat nie jest do końca jasny, ale po przeszukaniu zasobów internetu udaje się ustalić, że w wyrażeniu regularnym konieczne jest użycie „nazwanej” grupy przechwytywania.

Plik *lists/urls.py* (ch31l011):

```
@@ -1,7 +1,7 @@
 from django.conf.urls import patterns, url
-from lists.views import NewListView
+from lists.views import NewListView, ViewAndAddToList

 urlpatterns = patterns('',
-    url(r'^(\d+)/$', 'lists.views.view_list', name='view_list'),
+    url(r'^(?P<pk>\d+)/$', ViewAndAddToList.as_view(), name='view_list'),
     url(r'^new$', NewListView.as_view(), name='new_list'),
 )
```

Kolejny komunikat błędu okazał się bardziej pomocny:

```
[...]
django.template.base.TemplateDoesNotExist: lists/list_detail.html

FAILED (failures=5, errors=6)
```

Rozwiązywanie problemu jest całkiem proste.

Plik *lists/views.py*:

```
class ViewAndAddToList(DetailView):
    model = List
    template_name = 'list.html'
```

Pozostały do rozwiązyania następujące błędy:

```
[...]
ERROR: test_displays_item_form (lists.tests.test_views.ListViewTest)
KeyError: 'form'

FAILED (failures=5, errors=2)
```

Jesteśmy zdani na metodę prób i błędów

Wcześniej określiłem, że widok nie tylko pozwala na wyświetlanie informacji szczegółowych o obiekcie, ale pozwala także na tworzenie nowych. Utworzmy więc połączenie widoków *DetailView* oraz *CreateView*.

Plik *lists/views.py*:

```
class ViewAndAddToList(DetailView, CreateView):
    model = List
    template_name = 'list.html'
    form_class = ExistingListItemForm
```

Otrzymujemy sporo błędów o następującej treści:

```
[...]
TypeError: __init__() missing 1 required positional argument: 'for_list'
```

W komunikatach nadal przewijał się zwrot `KeyError: 'form'`.

Na tym etapie komunikaty błędów przestały być użyteczne i nie bardzo wiedziałem, co dalej zrobić. Pozostała już tylko metoda prób i błędów. Testy informowały mnie tylko, że sprawy idą w dobrym lub złym kierunku.

Pierwsza próba użycia funkcji `get_form_kwargs()` zakończyła się niepowodzeniem, ale ustaliłem, że mógłbym spróbować użyć `get_form()`.

Plik `lists/views.py`:

```
def get_form(self, form_class):
    self.object = self.get_object()
    return form_class(for_list=self.object, data=self.request.POST)

KeyError: 'form'

FAILED (errors=3)
```

Wracamy do gry

Kolejne eksperymenty doprowadziły mnie do zastąpienia widoku `DetailView` widokiem `SingleObjectMixin` (w dokumentacji znajdziesz użyteczne informacje na jego temat):

```
from django.views.generic.detail import SingleObjectMixin
[...]

class ViewAndAddToList(CreateView, SingleObjectMixin):
    [...]
```

Liczba błędów spadła do jedynie dwóch:

```
django.core.exceptions.ImproperlyConfigured: No URL to redirect to. Either
provide a url or define a get_absolute_url method on the Model.
```

W przypadku ostatniego niepowodzenia testy ponownie okazały się użyteczne. Bardzo łatwo można zdefiniować funkcję `get_absolute_url()` w klasie `Item`, aby elementy wskazywały stronę ich listy nadzędnej.

Plik `lists/models.py`:

```
class Item(models.Model):
    [...]

    def get_absolute_url(self):
        return reverse('view_list', args=[self.list.id])
```

Czy to już rozwiązanie ostateczne?

Poniżej przedstawiono ostateczny kod klasy widoku.

Plik `lists/views.py` (ch31l010):

```
class ViewAndAddToList(CreateView, SingleObjectMixin):
    template_name = 'list.html'
    model = List
    form_class = ExistingListItemForm

    def get_form(self, form_class):
        self.object = self.get_object()
        return form_class(for_list=self.object, data=self.request.POST)
```

Porównanie starego i nowego widoku

Dla porównania poniżej przedstawiono starą wersję widoku.

Plik `lists/views.py`:

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ExistingListItemForm(for_list=list_)
    if request.method == 'POST':
        form = ExistingListItemForm(for_list=list_, data=request.POST)
        if form.is_valid():
            form.save()
    return redirect(list_)

return render(request, 'list.html', {'list': list_, "form": form})
```

Cóż, liczba wierszy kodu zmniejszyła się z dziewięciu do siedmiu. Według mnie wersja oparta na funkcji jest nieco łatwiejsza do zrozumienia, ponieważ zawiera mniejszą ilość „magii”, to znaczy jakby to określono w zen Pythona: „wyraźne zdefiniowanie czegoś jest lepsze od niejawnego zdefiniowania”. Mam tutaj na myśli `SingleObjectMixin`. Co takiego? I najważniejsze, rozwiązanie przestaje działać, jeśli zapomnisz o przypisaniu do `self.object` w funkcji `get_form()`. Fuj!

Myślę, że wybór podejścia i jego ocena zależą od programisty.

Najlepsze praktyki w zakresie testów jednostkowych dla ogólnych widoków opartych na klasach?

Kiedy pracowałem nad rozwiązaniem, to zacząłem odczuwać, że moje testy „jednostkowe” czasami działały na zbyt wysokim poziomie. To nie jest zaskoczeniem, ponieważ testy przeznaczone dla widoków wykorzystujących klienta testowego Django powinny być raczej nazywane testami zintegrowanymi.

Wprawdzie testy informowały, że obrany kierunek był prawidłowy lub nieprawidłowy, ale nie zawsze oferowały wystarczające wskazówki, jak należy poprawić błędy.

Czasami zastanawiałem się, czy osiągnąłem znaczący postęp w teście, który przybliżał mnie do właściwej implementacji, na przykład:

```
def test_cbv_gets_correct_object(self):
    our_list = List.objects.create()
    view = ViewAndAddToList()
    view.kwargs = dict(pk=our_list.id)
    self.assertEqual(view.get_object(), our_list)
```

Problem polega na tym, że wymagana jest duża wiedza o wewnętrznym działaniu widoków opartych na klasach w Django, aby móc prawidłowo przygotować konfigurację dla tego rodzaju testów. Ponadto możesz być nieco zakłopotany skomplikowaną hierarchią dziedziczenia.

Jesteśmy w domu — pomocne jest przygotowanie wielu odizolowanych testów widoku wraz z pojedynczymi asercjami

A teraz przechodzimy do ostatniej kwestii wartej wspomnienia w tym dodatku. Opracowanie wielu krótkich testów jednostkowych dla widoków będzie znacznie bardziej pomocne niż mniejszej ich liczby, ale z narracyjnymi seriami asercji.

Spójrz na poniższy monolityczny test:

```
def test_validation_errors_sent_back_to_home_page_template(self):
    response = self.client.post('/lists/new', data={'text': ''})
    self.assertEqual(List.objects.all().count(), 0)
    self.assertEqual(Item.objects.all().count(), 0)
    self.assertTemplateUsed(response, 'home.html')
    expected_error = escape("You can't have an empty list item")
    self.assertContains(response, expected_error)
```

Powyższy test zdecydowanie okaże się mniej użyteczny niż trzy oddzielne:

```
def test_invalid_input_means_nothing_saved_to_db(self):
    self.post_invalid_input()
    self.assertEqual(item.objects.all().count(), 0)

def test_invalid_input_renders_list_template(self):
    response = self.post_invalid_input()
    self.assertTemplateUsed(response, 'list.html')

def test_invalid_input_renders_form_with_errors(self):
    response = self.post_invalid_input()
    self.assertIsInstance(response.context['form'], ExistingListItemForm)
    self.assertContains(response, escape(empty_list_error))
```

Powód jest prosty. W pierwszym przypadku wczesne niepowodzenie oznacza, że nie wszystkie asercje zostaną sprawdzone. Jeżeli widok przypadkowo przeprowadza operację zapisu w bazie danych podczas nieprawidłowego żądania POST, otrzymasz wczesne niepowodzenie i nie dowiesz się, czy został użyty odpowiedni szablon lub wygenerowany formularz. W drugim przypadku znacznie łatwiej wychwycić dokładnie to, co działa lub nie działa.

Lekcje wyciągnięte na podstawie ogólnych widoków opartych na klasach

Ogólne widoki oparte na klasach mogą zrobić wszystko

Być może nie zawsze będzie do końca jasne, co się dzieje, ale za pomocą widoków opartych na klasach można zrobić praktycznie wszystko.

Zawierające pojedynczą asercję testy jednostkowe pomagają w trakcie refaktoryzacji

Gdy każdy test jednostkowy dostarcza informacje o działaniu lub niedziałaniu pojedynczej rzeczy, wtedy znacznie łatwiej można zmienić implementację widoków, używając do tego zupełnie odmiennego, omówionego tutaj paradygmatu.

Przygotowanie serwera za pomocą Ansible

W celu automatyzacji operacji wdrożenia w serwerach nowych wersji kodu wykorzystaliśmy Fabric. Jednak przygotowanie nowego serwera, a także uaktualnienie plików konfiguracyjnych Nginx i Gunicorn zostało pozostawione jako ręczny proces.

Tymczasem jest to rodzaj zadania, którego wykonanie coraz częściej jest zlecone narzędziom typu „zarządzanie konfiguracją” lub „nieustanne wdrożenie”. Chef i Puppet były pierwszymi tego rodzaju narzędziami, w świecie Pythona mamy Salt i Ansible.

Z wymienionych najłatwiej rozpocząć pracę z Ansible. Potrzebne są do tego jedynie dwa pliki instalowane po wydaniu poniższego polecenia:

```
pip install ansible # Niestety tylko Python 2.
```

Plik *inventory.ansible* umieszczony w katalogu *deploy_tools* definiuje serwery, które mogą być używane.

Plik *deploy_tools/inventory.ansible*:

```
[live]
superlists.ottg.eu

[staging]
superlists-staging.ottg.eu

[local]
localhost ansible_ssh_port=6666 ansible_host=127.0.0.1
```

(Sekcja *local* jest opcjonalna. W omawianym tutaj przykładzie to maszyna wirtualna VirtualBox wraz ze skonfigurowanym przekierowaniem portów 22 i 80).

Instalacja pakietów systemowych i Nginx

Kolejnym krokiem jest przygotowanie pliku stosującego składnię YAML i określającego możliwości aplikacji w serwerze.

Plik *deploy_tools/provision.ansible.yaml*:

```
---
```

- hosts: all
 - sudo: yes

vars:

- host: \$inventory_hostname

tasks:

- name: make sure required packages are installed
 - apt: pkg=nginx,git,python3,python3-pip state=present
- name: make sure virtualenv is installed
 - shell: pip3 install virtualenv
- name: allow long hostnames in nginx
 - lineinfile:
 - dest=/etc/nginx/nginx.conf
 - regexp='(\s+)? ?server_names_hash_bucket_size'
 - backrefs=yes
 - line='\1server_names_hash_bucket_size 64;'
- name: add nginx config to sites-available
 - template: src=./nginx.conf.j2
 - dest=/etc/nginx/sites-available/{{ host }}

notify:

- restart nginx

- name: add symlink in nginx sites-enabled
 - file: src=/etc/nginx/sites-available/{{ host }}
 - dest=/etc/nginx/sites-enabled/{{ host }} state=link

notify:

- restart nginx

Sekcja vars definiuje dla wygody zmienną „hosta”, której następnie można używać w różnych nazwach plików, a także przekazywać do samych plików konfiguracyjnych. Wartość pochodzi z \$inventory_hostname, czyli nazwy domeny używanego serwera.

W tej sekcji za pomocą narzędzia apt instalujemy wymagane oprogramowanie, modyfikujemy konfigurację Nginx, aby zezwolić na użycie długich nazw hostów za pomocą wyrażeń regularnych, a następnie zapisujemy plik konfiguracyjny Nginx, posługując się szablonem. Ta zmodyfikowana wersja pliku szablonu została w rozdziale 8. zapisana jako *deploy_tools/nginx.template.conf*. Jednak teraz wykorzystamy inną składnię szablonów — Jinja2, która bardzo przypomina składnię szablonów w Django.

Plik *deploy_tools/nginx.conf.j2*:

```
server {  
    listen 80;  
    server_name {{ host }};  
    location /static {  
  
        alias /home/harry/sites/{{ host }}/static;  
    }  
    location / {  
        proxy_set_header Host $host;  
        proxy_pass http://unix:/tmp/{{ host }}.socket;  
    }  
}
```

Konfiguracja Gunicorn i użycie procedur obsługi do ponownego uruchamiania usług

Oto część druga omawianego tutaj pliku YAML.

Plik `deploy_tools/provision.ansible.yaml`:

```
- name: write gunicorn init script
  template: src=./gunicorn-upstart.conf.j2
            dest=/etc/init/gunicorn-{{ host }}.conf
  notify:
    - restart gunicorn

- name: make sure nginx is running
  service: name=nginx state=running
- name: make sure gunicorn is running
  service: name=gunicorn-{{ host }} state=running

handlers:
- name: restart nginx
  service: name=nginx state=restarted

- name: restart gunicorn
  service: name=gunicorn-{{ host }} state=restarted
```

Raz jeszcze wykorzystamy szablon dla konfiguracji Gunicorn.

Plik `deploy_tools/gunicorn.upstart.conf.j2`:

```
description "Serwer Gunicorn dla {{ host }}"

start on net-device-up
stop on shutdown

respawn

chdir /home/harry/sites/{{ host }}/source
exec ../virtualenv/bin/gunicorn \
  --bind unix:/tmp/{{ host }}.socket \
  --access-logfile ../access.log \
  --error-logfile ../error.log \
  superlists.wsgi:application
```

Następnie mamy dwie „procedury obsługi” przeznaczone do ponownego uruchamiania Nginx i Gunicorn. Ansible to sprytnie działające narzędzie i jeśli wykryje wiele kroków wywołujących te same procedury obsługi, z ich wywołaniem wstrzymuje się aż do ostatniego kroku.

I to tyle. Polecenie pozwalające na użycie powyższego rozwiązania jest następujące:

```
ansible-playbook -i ansible.inventory provision.ansible.yaml --limit=staging
```

Więcej informacji znajdziesz w *dokumentacji*¹ Ansible.

¹ <http://docs.ansible.com/>

Co dalej?

Przedstawiłem tutaj jedynie niewielkie wprowadzenie do możliwości oferowanych przez Ansible. Im bardziej zautomatyzujesz wdrożenia, tym większą masz pewność o ich prawidłowym przeprowadzaniu. Oto kilka kwestii, na które warto zwrócić uwagę.

Przeniesienie wdrożenia z Fabric do Ansible

Jak mogłeś zobaczyć, Ansible okazuje się pomocne na etapie przygotowań serwera, choć w zakresie wdrożenia może nam zaoferować znacznie więcej. Przekonaj się, czy jesteś w stanie rozbudować plik YAML w taki sposób, aby Ansible wykonało wszystkie zadania realizowane dotąd przez skrypt Fabric, w tym także powiadomienie o konieczności ponownego uruchomienia serwera.

Użyj Vagrant do obsługi lokalnych maszyn wirtualnych

Wykonywanie testów względem witryny prowizorycznej daje ostateczną pewność, że rozwiązanie będzie działało w witrynie produkcyjnej. Istnieje także możliwość użycia maszyn wirtualnych w komputerze lokalnym.

Pobierz Vagrant i VirtualBox, a następnie sprawdź, czy za pomocą Vagrant możesz przygotować serwer programistyczny we własnym komputerze i użyć Ansible do wdrożenia w nim kodu. Zmodyfikuj silnik testów funkcjonalnych, aby móc je wykonywać względem lokalnej maszyny wirtualnej.

Plik konfiguracyjny Vagrant okazuje się szczególnie użyteczny podczas pracy w zespole, ponieważ pozwala nowym programistom otrzymać środowisko pracy dokładnie takie samo jak Twoje.

Testowanie migracji bazy danych

Django-migrations i jego poprzednik (South) istnieją już od lat, więc testowanie migracji bazy danych nie należy do niezwykłych zadań. Jednak zdarza się wprowadzenie niebezpiecznych typów migracji, na przykład definiujących nowe ograniczenia spójności dla danych. Kiedy po raz pierwszy uruchomiłem skrypt migracji w serwerze prowizorycznym, zobaczyłem komunikat błędu.

W przypadku ogromnych projektów operujących na danych wrażliwych może wystąpić konieczność podjęcia dodatkowych kroków i przetestowania migracji w bezpiecznym środowisku przed jej zastosowaniem na danych produkcyjnych. Mam nadzieję, że przedstawiony tutaj prosty przykład będzie użytecznym wprowadzeniem.

Innym częstym powodem testowania migracji jest szybkość. Migracje bardzo często wymagają wyłączenia serwera z jego normalnej pracy, a czas przestoju w przypadku ogromnych zbiorów danych może trwać bardzo długo. Dlatego też dobrze jest wcześniej wiedzieć, ile może potrwać przerwa w trakcie pracy serwera.

Próba wdrożenia do serwera prowizorycznego

Poniżej możesz zobaczyć efekt pierwszej próby zastosowania w rozdziale 14. nowych ograniczeń w trakcie sprawdzania poprawności:

```
$ cd deploy_tools  
$ fab deploy:host=elsbeth@superlists-staging.ottg.eu  
[...]  
Running migrations:  
  Applying lists.0005_list_item_unique_together...Traceback (most recent call  
last):  
    File "/usr/local/lib/python3.3/dist-packages/django/db/backends/utils.py",  
    line 61, in execute  
      return self.cursor.execute(sql, params)  
    File  
"/usr/local/lib/python3.3/dist-packages/django/db/backends/sqlite3/base.py",  
line 475, in execute  
    return Database.Cursor.execute(self, query, params)  
sqlite3.IntegrityError: columns list_id, text are not unique  
[...]
```

Pewne dane istniejące w bazie danych nie spełniały nowo zdefiniowanego ograniczenia dotyczącego spójności, stąd generowane przez bazę danych komunikaty błędów w trakcie próby wprowadzenia migracji.

Aby rozwiązać tego rodzaju problem, konieczne jest przygotowanie „danych migracji”. Najpierw zajmiemy się konfiguracją odpowiedniego środowiska lokalnego.

Lokalne przeprowadzenie testu migracji

Do przeprowadzenia testu migracji posłużymy się kopią produkcyjnej bazy danych.



Zachowaj naprawdę ogromną ostrożność podczas użycia w trakcie testów rzeczywistych danych. Dane mogą na przykład zawierać prawdziwe adresy e-mail klientów, a nie chcesz przypadkowo wysłać im ogromnych ilości testowych wiadomości e-mail. Nie pytaj mnie, skąd o tym wiem.

Wprowadzenie problematycznych danych

Zaczynamy od listy zawierającej powielone elementy w witrynie produkcyjnej, jak pokazano na rysunku D.1.

A screenshot of a Mozilla Firefox browser window. The title bar says "Listy rzeczy do zrobienia - Mozilla Firefox". The menu bar includes "Plik", "Edycja", "Widok", "Historia", "Zakładki", "Narzędzia", and "Pomoc". A tab labeled "Listy rzeczy do zrobienia" is open. The address bar shows "superlists.ottg.eu/lists/6/". The main content area displays a list titled "Twoja lista rzeczy do zrobienia". Below the title, there is a text input field containing the word "duplikat". Below the input field, the list items are: "1: lista z powtarzającymi się elementami", "2: duplikat", "3: duplikat", and "4: duplikat".

The screenshot shows a browser window with a to-do list page. The list contains four items, all of which are identical: "duplikat". This illustrates a data inconsistency that might be tested during a migration.

Rysunek D.1. Lista z powtarzającymi się elementami

Skopiowanie danych testowych z witryny produkcyjnej

Skopuj bazę danych z witryny produkcyjnej:

```
$ scp edyta@superlists.ottg.eu:\n/home/edyta/sites/superlists.ottg.eu/database/db.sqlite3 .\n$ mv ..database/db.sqlite3 ..database/db.sqlite3.bak\n$ mv db.sqlite3 ..database/db.sqlite3
```

Potwierdzenie istnienia błędu

Mamy teraz lokalną bazę danych z problematycznymi danymi, w której nie została przeprowadzona migracja. Próba wydania polecenia `migrate` powinna spowodować wyświetlenie komunikatu błędu.

```
$ python3 manage.py migrate --migrate\npython3 manage.py migrate\nOperations to perform:\n[...]\nRunning migrations:\n[...]\n    Applying lists.0005_list_item_unique_together...Traceback (most recent call\nlast):\n[...]\n        return Database.Cursor.execute(self, query, params)\nsqlite3.IntegrityError: columns list_id, text are not unique
```

Wstawienie danych migracji

Dane migracji¹ to specjalny typ migracji modyfikującej dane znajdujące się w bazie danych, ale niezmieniającej schematu. Musimy przygotować takie dane. Zostaną one sprawdzone przed zastosowaniem ograniczenia dotyczącego spójności, aby profilaktycznie usunąć wszelkie duplikaty:

```
$ git rm lists/migrations/0005_list_item_unique_together.py\n$ python3 manage.py makemigrations lists --empty\nMigrations for 'lists':\n    0005_auto_20140414_2325.py:\n$ mv lists/migrations/0005_lists/migrations/0005_remove_duplicates.py*
```

Więcej informacji znajdziesz w wymienionej wcześniej dokumentacji Django poświęconej migracjom danych. Poniżej przedstawiłem kilka polecień pozwalających na zmianę istniejących danych.

Plik `lists/migrations/0005_remove_duplicates.py`:

```
# Kodowanie znaków: utf8.\nfrom django.db import models, migrations\n\ndef find_dupes(apps, schema_editor):\n    List = apps.get_model("lists", "List")\n    for list_ in List.objects.all():\n        items = list_.item_set.all()\n        texts = set()
```

¹ <https://docs.djangoproject.com/en/dev/topics/migrations/#data-migrations>

```

        for ix, item in enumerate(items):
            if item.text in texts:
                item.text = '{} ({})'.format(item.text, ix)
                item.save()
            texts.add(item.text)

class Migration(migrations.Migration):

    dependencies = [
        ('lists', '0004_item_list'),
    ]

    operations = [
        migrations.RunPython(find_dupes),
    ]

```

Ponowne odtworzenie starej migracji

Starą migrację ponownie odtwarzamy za pomocą polecenia `makemigrations`, które gwarantuje, że mamy do czynienia z szóstą migracją mającą wyraźnie zdefiniowaną zależność od piątej:

```

$ python3 manage.py makemigrations
Migrations for 'lists':
  0006_auto_20140415_0018.py:
    - Alter unique_together for item (1 constraints)
$ mv lists/migrations/0006_* lists/migrations/0006_unique_together.py

```

Testowanie razem nowych migracji

Teraz możemy przeprowadzić testy względem danych produkcyjnych:

```

$ cd deploy_tools
$ fab deploy:host=edyta@superlists-staging.ottg.eu
[...]

```

Konieczne jest ponowne uruchomienie zadania Gunicorn:

```
edyta@serwer:$ sudo restart gunicorn-superlists.ottg.eu
```

Można wykonać testy funkcjonalne względem serwera prowizorycznego:

```

$ python3 manage.py test functional_tests --liveserver=superlists-staging.ottg.eu
Creating test database for alias 'default'...
....
```

```
Ran 4 tests in 17.308s
```

```
OK
```

Wydaje się, że wszystko działa. Przeprowadzamy więc migrację w środowisku produkcyjnym:

```
$ fab deploy --host=superlists.ottg.eu
[superlists.ottg.eu] Executing task 'deploy'
[...]
```

I to już koniec. Warto jeszcze przekazać pliki do repozytorium.

Podsumowanie

Ćwiczenie przedstawione w rozdziale miało na celu pokazanie przygotowania danych migracji i przetestowanie migracji wraz z pewną ilością danych produkcyjnych. Jak było do przewidzenia, to tylko kropla w morzu dostępnych opcji testowania migracji. Możesz sobie wyobrazić przygotowanie zautomatyzowanych testów sprawdzających, czy wszystkie dane zostały zachowane, oraz porównujących zawartość bazy danych przed migracją i po niej. Istnieje także możliwość opracowania poszczególnych testów jednostkowych dla funkcji pomocniczych w migracji danych. Więcej uwagi można poświęcić na pomiary czasu przeprowadzania migracji oraz eksperymenty z różnymi sposobami skrócenia tego czasu, na przykład przez podział migracji na więcej lub mniej kroków.

Pamiętaj, że tego rodzaju sytuacje powinny należeć do rzadkości. Z mojego doświadczenia wynika, że nie musiałem testować 99% przeprowadzonych migracji. Kiedy jednak uznasz za konieczne przetestowanie migracji w Twoim projekcie, mam nadzieję, że przedstawione tutaj podpowiedzi pomogą Ci w rozpoczęciu pracy.

Informacje o testowaniu migracji bazy danych

Uważaj na migracje wprowadzające ograniczenia dla danych

99% migracji przebiega bez problemów. Uważaj jednak na sytuacje takie jak ta, w której wprowadzane są nowe ograniczenia dla istniejących kolumn.

Testuj migracje pod kątem szybkości ich wykonywania

Gdy pracujesz nad ogromnym projektem, powinieneś rozważyć sprawdzenie, ile czasu będzie potrzebne na przeprowadzenie migracji. Migracja bazy danych zwykle wiąże się z przestojem serwera, ponieważ w zależności od bazy danych uaktualnienie schematu może spowodować zablokowanie tabeli aż do chwili zakończenia tej operacji. Dobrym pomysłem jest więc użycie witryny prowizorycznej do określenia czasu trwania operacji migracji.

Zachowaj wyjątkową ostrożność podczas użycia danych produkcyjnych

W tym celu bazę danych znajdującą się w witrynie prowizorycznej należy wypełnić taką ilością danych, aby odpowiadała wielkości danych produkcyjnych. Wyjaśnienie sposobu przeprowadzenia tej operacji wykracza poza zakres tematyczny książki, choć mogę powiedzieć jedno: jeżeli zamierzasz utworzyć kopię produkcyjnej bazy danych, a następnie wczytać dane do bazy prowizorycznej, to zachowaj wyjątkową ostrożność. Dane produkcyjne mogą zawierać informacje o rzeczywistych klientach. Osobiście jestem odpowiedzialny za wysłanie kilkuset nieprawidłowych rachunków, gdy zautomatyzowany proces w serwerze prowizorycznym zaczął przetwarzać wczytane do niego dane produkcyjne. To popołudnie na pewno nie należało do przyjemnych.

Co dalej?

W tym dodatku znajdziesz kilka podpowiedzi, czym zająć się w dalszej kolejności, aby zdobyć jeszcze większe umiejętności w zakresie testów, a także wykorzystać posiadane umiejętności podczas pracy ze świetnymi (w chwili pisania książki) technologiami programowania sieciowego.

Mam nadzieję, że wymienione tutaj podpowiedzi zamienią kogoś dnia na pewnego rodzaju post bloga lub też kolejny dodatek w książce. Planuję także przygotować przykładowe fragmenty kodu dla przedstawionych tutaj informacji. Zaglądaj więc do witryny <http://www.obeythetestinggoat.com/> i sprawdź, czy pojawiły się zapowiedziane aktualnienia.

Możesz też spróbować mnie uprzedzić i opublikować post bloga przedstawiający podejmowane przez Ciebie próby w dowolnym z wymienionych obszarów.

Chętnie odpowiem na wszelkie pytania i służę pomocą. Jeżeli uznasz coś za interesujące i utkniesz w pewnym miejscu, proszę, nie wahaj się i napisz do mnie na adres obeythetestinggoat@gmail.com.

Powiadomienia zarówno w witrynie, jak i przez e-mail

Byłoby miło, gdyby użytkownik był powiadamiany, że ktoś współdzieli jego listę.

Można wykorzystać django-notifications do wyświetlenia odpowiedniego komunikatu użytkownikowi w trakcie kolejnego odświeżenia strony. W teście funkcjonalnym będą do tego potrzebne dwie różne przeglądarki internetowe.

Równocześnie (lub zamiast) można wysyłać powiadomienia za pomocą wiadomości e-mail. Zapoznaj się z oferowanymi przez Django możliwościami w zakresie obsługi poczty elektronicznej. Następnie powinieneś przeprowadzić prawdziwe testy z użyciem rzeczywistych wiadomości e-mail, ponieważ wymieniona funkcja ma znaczenie krytyczne. Do pobrania rzeczywistych wiadomości e-mail z testowego konta pocztowego wykorzystaj bibliotekę IMAPClient.

Przejdź na Postgres

SQLite to wspaniała, mała baza danych, ale nie będzie działała zbyt dobrze z więcej niż jednym procesem roboczym obsługującym żądania generowane przez witrynę. Postgres to obecnie ulubiona przez wielu baza danych; dowiedz się, jak można ją zainstalować i skonfigurować.

Trzeba będzie znaleźć miejsce do przechowywania nazw użytkowników i haseł dla serwerów Postgres lokalnego, prowizorycznego i produkcyjnego. Ponieważ ze względów bezpieczeństwa prawdopodobnie nie chcesz umieszczać tych danych w repozytorium kodu źródłowego, znajdź sposoby modyfikacji skryptów wdrożenia, aby wspomniane dane przekazać za pomocą powłoki. Zmienne środowiskowe to jedno z popularnych rozwiązań służących do przechowywania wspomnianych danych.

Poeksperymentuj z zachowaniem działania testów jednostkowych względem SQLite i zobacz, o ile szybciej są wykonywane w przypadku użycia bazy danych Postgres. Skonfiguruj komputer lokalny do użycia SQLite w celach testowych, natomiast w serwerze ciąglej integracji zdecyduj się na Postgres.

Wykonuj testy względem różnych przeglądarek internetowych

Selenium zapewnia obsługę różnego rodzaju przeglądarek internetowych, między innymi Chrome i Internet Explorer. Wypróbuj je obie i zobacz, czy występują jakiekolwiek różnice w zachowaniu testów funkcjonalnych.

Powinieneś sprawdzić także przeglądarki internetowe niewymagające ekranu do działania, na przykład PhantomJS.

Z mojego doświadczenia wynika, że zmiana przeglądarki internetowej powoduje pojawienie się różnego rodzaju stanu wyścigu w testach Selenium. Prawdopodobnie znacznie częściej będziesz musiał zastosować wzorzec interakcja-oczekiwanie, zwłaszcza w przypadku PhantomJS.

Testy pod kątem błędów o kodach 404 i 500

Profesjonalna witryna wymaga dobrze wyglądających stron błędów. Przetestowanie witryny pod kątem błędu o kodzie 404 jest łatwe, ale prawdopodobnie będziesz musiał zgłosić odpowiedni wyjątek widokowi, aby przetestować witrynę pod kątem błędu o kodzie 500.

Witryna administracyjna Django

Wyobraź sobie sytuację, w której użytkownik wysyła do Ciebie wiadomość e-mail, w której narzeka, że jego lista jest anonimowa. Powiedzmy, że implementujemy ręczne rozwiązanie dla takiego zadania. Administrator witryny musi wtedy ręcznie zmienić rekord w witrynie administracyjnej Django.

Ustal, jak przejść do witryny administracyjnej, i rozwiąż problem. Opracuj test funkcjonalny pokazujący zwykłego, niezalogowanego użytkownika tworzącego listę, a następnie zalogowanie administratora, przejście do witryny administracyjnej i przypisanie listy użytkownikowi. Od tego momentu użytkownik będzie mógł zobaczyć listę na swojej stronie „Moje listy”.

Sprawdź narzędzie BDD

BDD oznacza Behaviour-Driven Development (programowanie oparte na zachowaniu). To jest sposób tworzenia testów bardziej czytelnych dla człowieka przez implementacje pewnego rodzaju DSL (ang. *domain-specific language*) dla kodu testów funkcjonalnych. Sprawdź Lettuce (framework BDD dla Pythona) i wykorzystaj go do modyfikacji istniejących testów funkcjonalnych lub też do utworzenia nowych.

Utwórz pewne testy dotyczące bezpieczeństwa

Rozbuduj testy dotyczące logowania, utworzonych list i współdzielenia — co powinieneś jeszcze zrobić, aby mieć pewność, że użytkownicy będą mogli wykonywać tylko te zadania, do których zostali upoważnieni?

Sprawdź aplikację pod kątem eleganckiej degradacji

Co się stanie, jeśli system Persona ulegnie awarii? Czy użytkownikom został przynajmniej wyświetlony odpowiedni komunikat błędu?

- Podpowiedź. Jednym ze sposobów symulacji awarii systemu Persona jest modyfikacja pliku `hosts` (`/etc/hosts` w systemach Linux i `C:\Windows\System32\drivers\etc` w Windows). Pamiętaj o przywróceniu pliku w metodzie `tearDown()` testu!
- Stronę serwera traktuj podobnie jak stronę klienta.

Testowanie buforowania i wydajności

Dowiedz się, jak zainstalować i skonfigurować `memcached`. Dowiedz się, jak używać ab Apache do przeprowadzenia testu wydajności. Jaki jest wynik testu przeprowadzonego bez buforowania oraz z jego uwzględnieniem? Czy możesz przygotować zautomatyzowany test, który zakończy się niepowodzeniem, jeśli buforowanie jest wyłączone? Jak wygląda kwestia unieważniania zawartości bufora? Czy testy mogą pomóc w sprawdzeniu, czy logika unieważniania zawartości bufora działa niezawodnie?

Frameworki JavaScript MVC

Obecnie biblioteki JavaScript pozwalają na implementację wzorca MVC (model-widok-kontroler) pod stronie klienta. Aplikacja list rzeczy do zrobienia to jedno z ulubionych demo tego rodzaju funkcjonalności. Konwersja obecnej witryny na postać aplikacji będącej pojedynczą stroną powinna być całkiem łatwa. W takim przypadku wszystkie operacje na listach są przeprowadzane za pomocą JavaScript.

Wybierz framework — prawdopodobnie Backbone.js lub Angular.js — i zajmij się implementacją. Poszczególne frameworki mają własne preferencje w zakresie tworzenia testów jednostkowych, poznaj jeden z nich i przekonaj się, czy go polubisz.

Async i WebSocket

Przymajemy założenie, że dwóch użytkowników jednocześnie pracuje nad tą samą listą. Czy nie byłoby miło zapewnić aktualnianie listy w czasie rzeczywistym? W takim przypadku, gdy jedna osoba umieści nowy element na liście, druga osoba natychmiast go widzi. Trwałe połączenie między klientem i serwerem za pomocą protokołu WebSocket to podstawa tego rodzaju rozwiązania.

Sprawdź asynchroniczne serwery WWW dla Pythona — Tornado, gevent i Twisted — i przekonaj się, czy możesz je wykorzystać do implementacji dynamicznych powiadomień.

Do przetestowania tego rodzaju rozwiązania będziesz potrzebował dwóch różnych przeglądarek internetowych (podobnie jak w przypadku testu współdzielenia listy). Sprawdź, czy powiadomienia o akcjach pojawiają się w drugiej przeglądarce internetowej bez konieczności odświeżenia strony...

Zacznij używać pakietu PyTest

Pakiet PyTest pozwala na tworzenie testów jednostkowych z mniejszą ilością powtarzającego się kodu. Spróbuj skonwertować niektóre testy jednostkowe na wykorzystujące PyTest. Zapewnienie eleganckiej współpracy pakietu PyTest z Django może wymagać użycia wtyczki.

Szyfrowanie po stronie klienta

Oto zabawny przypadek: co się stanie, jeśli użytkownicy będą obawiali się działalności NSA¹ i zadecydują, że nie ufają dłużej listom przechowywanym w chmurze? Czy potrafisz zbudować oparty na kodzie JavaScript system szyfrowania, w którym użytkownik podaje hasło i tym samym przeprowadza szyfrowanie tekstu elementu listy przed jego wysłaniem do serwera?

Jednym ze sposobów przetestowania będzie utworzenie użytkownika administratora, który przejdzie do widoku administracyjnego w Django, przejrzy utworzone przez użytkowników listy rzeczy do zrobienia i zaznaczy, że w bazie danych są przechowywane w postaci zaszyfrowanej.

Miejsce na Twoje propozycje

Jak sądzisz, co tu powiniensem jeszcze umieścić? Proszę o propozycje!

¹ Agencja Bezpieczeństwa Narodowego USA — przyp. tłum.

Ściąga

Na ogólne życzenie przedstawiona tutaj „ściąga” została oparta na lekko zmodyfikowanych ramkach znajdujących się na końcu poszczególnych rozdziałów. Moim celem jest przypomnienie wybranych aspektów i wskazanie rozdziałów, w których znajdziesz więcej informacji na dany temat. Mam nadzieję, że uznasz tę ściągę za użyteczną.

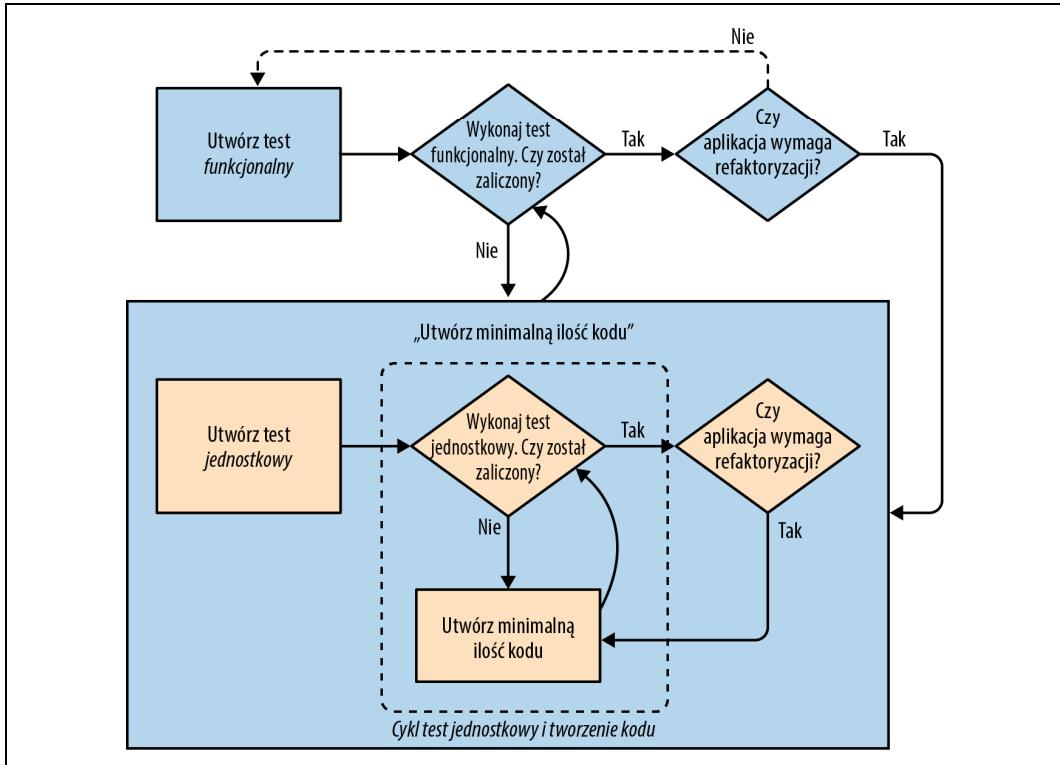
Początkowa konfiguracja projektu

- Pracę rozpocznij od przygotowania *informacji od użytkownika*, a następnie mapuj je na pierwszy *test funkcjonalny*.
- Wybierz framework testów — unittest jest dobrym wyborem, inne, takie jak PyTest i nose, również oferują pewne zalety.
- Wykonaj test funkcjonalny i zobacz pierwsze *oczekiwane niepowodzenie*.
- Wybierz framework sieciowy, na przykład Django, i dowiedz się, jak wykonywać w nim *testy jednostkowe*.
- Utwórz pierwszy test jednostkowy przeznaczony do usunięcia niepowodzenia testu funkcjonalnego i zobacz, jak kończy się niepowodzeniem.
- Przeprowadź *pierwsze przekazanie plików* do systemu VCS, na przykład *Git*.

Rozdziały, w których znajdziesz więcej informacji: 1., 2. i 3.

Podstawowy sposób pracy z użyciem technik TDD

- Podwójna pętla TDD (patrz rysunek F.1).
- Czerwony, zielony, refaktoryzacja.
- Triangulacja.
- Notatnik.
- Do trzech razy sztuka, a później refaktoryzacja.
- Od jednego stanu działającej aplikacji do innego stanu działającej aplikacji.
- YAGNI.



Rysunek F.1. Proces TDD obejmujący testy funkcjonalne i jednostkowe

Rozdziały, w których znajdziesz więcej informacji: 4., 5. i 6.

Wykraczamy poza testy w komputerze programisty

- System testowy przygotuj jak najwcześniej. Upewnij się o prawidłowym współdziałaniu komponentów: serwer WWW, treść statyczna, baza danych.
- Przygotuj środowisko prowizoryczne odpowiadające produkcyjnemu i wykonaj w nim zestaw testów funkcjonalnych.
- Zautomatyzuj środowiska prowizoryczne i produkcyjne:
 - Paas kontra VPS,
 - Fabric,
 - zarządzanie konfiguracją (Chef, Puppet, Salt, Ansible),
 - Vagrant.
- Przeanalizuj aspekty, które mogą być źródłem problemów podczas wdrożenia: bazę danych, pliki statyczne, zależności, dostosowanie ustawień do własnych potrzeb itd.
- Jak najwcześniej przygotuj serwer ciągłej integracji. W ten sposób nie będziesz musiał polegać na własnej dyscyplinie w zakresie wykonywania testów.

Rozdziały, w których znajdziesz więcej informacji: 8., 9., 20. i C.

Najlepsze praktyki dotyczące ogólnego testowania

- Każdy test powinien dotyczyć tylko jednej rzeczy.
- Używaj jednego pliku testu dla pliku kodu źródłowego aplikacji.
- Rozważ przynajmniej miejsce zarezerwowane na test dla każdej funkcji i klasy, niezależnie od tego, jak jest prosta.
- Pamiętaj, aby nie testować stałych.
- Spróbuj testować zachowanie, a nie implementację.
- Spróbuj wykroczyć poza zaplanowaną ścieżkę działania kodu i uwzględnij przypadki skrajne oraz błędy.

Rozdziały, w których znajdziesz więcej informacji: 4., 10. i 11.

Najlepsze praktyki dotyczące testów funkcjonalnych i Selenium

- Używaj wyraźnie zdefiniowanych operacji oczekiwania zamiast niejawnych, a ponadto stosuj wzorzec interakcja-oczekiwanie.
- Unikaj powielania kodu testu — jednym z dobrych rozwiązań są tutaj metody pomocnicze w klasie bazowej oraz wzorzec strony.
- Unikaj podwójnego testowania funkcjonalności. Jeżeli masz test sprawdzający proces wymagający nieco czasu (na przykład logowanie), rozważ jego pominięcie w innych testach. (Musisz być jednak świadomy istnienia nieoczekiwanych interakcji między, wydawałoby się, niepowiązanymi funkcjonalnościami).
- Spójrz na dostępne narzędzia BDD i rozważ ich użycie, aby inaczej przeprowadzić strukturyzację testów funkcjonalnych.

Rozdziały, w których znajdziesz więcej informacji: 17., 20. i 21.

Podejście Outside-In, testy odizolowane kontra zintegrowane oraz imitacje

Przede wszystkim nie zapominaj o powodach, dla których tworzymy testy:

- Aby zapewnić poprawność aplikacji i uniknąć regresji.
- Aby pomóc w tworzeniu przejrzystego i łatwego w obsłudze kodu.
- Aby umożliwić szybką i produktywną pracę.

Mając na uwadze powyższe cele, zastanów się nad różnymi typami testów, a także ich wadami i zaletami.

Testy funkcjonalne

- Dają największą gwarancję, że aplikacja naprawdę działa prawidłowo z punktu widzenia użytkownika.
- Jednak charakteryzują się wolniejszym dostarczaniem informacji zwrotnych.
- Ponadto niekoniecznie pomogą w tworzeniu przejrzystego kodu.

Testy zintegrowane (oparte na ORM lub kliencie testów Django)

- Można je szybko utworzyć.
- Pozostają łatwe do zrozumienia.
- Ostrzegają o wszelkich problemach związanych z integracją.
- Jednak nie zawsze prowadzą do powstania dobrego projektu (to zależy wyłącznie od Ciebie!).
- Zwykle są wykonywane wolniej niż testy odizolowane.

Testy odizolowane (stosujące „imitacje”)

- Przygotowanie tego rodzaju testów wymaga najwięcej pracy.
- Mogą być trudniejsze do odczytu i zrozumienia.
- Jednak są najlepsze, jeśli chcesz opracować dobry projekt aplikacji.
- A ponadto są wykonywane najszybciej.

Jeżeli okaże się, że tworzysz testy wraz z wieloma imitacjami, i uznasz to za męczące, zawsze możesz „wsłuchać się w swoje testy” — brzydkie, oparte na imitacjach testy mogą sugerować możliwość uproszczenia kodu.

Rozdziały, w których znajdziesz więcej informacji: 18., 19. i 22.

Bibliografia

- [dip] Mark Pilgrim, *Dive Into Python*: <http://www.diveintopython.net/>
- [lpthw] Zed A. Shaw, *Learn Python The Hard Way*: <http://learnpythonthehardway.org/>
- [iwp] Al Sweigart, *Invent Your Own Computer Games With Python*: <http://inventwithpython.com/>
- [tddbe] Kent Beck, *TDD. Sztuka tworzenia dobrego kodu*, Helion.
- [refactoring] Martin Fowler, *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*, Helion.
- [seceng] Ross Anderson, *Inżynieria zabezpieczeń*, Helion.
- [jsgoodparts] Douglas Crockford, *JavaScript — mocne strony*, Helion.
- [twoscoops] Daniel Greenfield i Audrey Roy, *Two Scoops of Django*: <http://twoscoopspress.com/products/two-scoops-of-django-1-6>
- [mockfakestub] Emily Bache, *Mocks, Fakes and Stubs*: <https://leanpub.com/mocks-fakes-stubs>
- [GOOSGBT] Steve Freeman i Nat Pryce, *Growing Object-Oriented Software Guided by Tests*, Addison-Wesley.

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
dosłosowanie
 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
 witryny, 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

unicorn-superlists-staging.ottg.eu.conf, 168

unicorn-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
syntezja, 395
system kontroli wersji, VCS, 33
system kontroli wersji Git, 21, 71
szablon, 327, 330
szablon do wyświetlania list, 109
szifrowanie, 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
 bazы 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
 - provizoryczne, 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
 - provizoryczna, 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
 - wyswietlanie
 - 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 autorze

Po wspaniałym dzieciństwie upływającym na zabawach z językiem programowania BASIC we francuskich komputerach 8-bitowych typu Thompson T-07, których klawisze wydawały dźwięki po naciśnięciu, kolejnych kilka lat upłynęło Harry'emu niezbyt szczęśliwie na doradztwie w zakresie ekonomii i zarządzania. Wkrótce jednak ponownie odkrył swoje prawdziwe powołanie i z ogromną przyjemnością przyłączył się do grupy fanatyków programowania ekstremalnego pracujących nad pionierskim, lecz już niestety nieistniejącym arkuszem kalkulacyjnym Resolver One. Obecnie pracuje dla PythonAnywhere LLP i na wszelkiego rodzaju warsztatach i konferencjach propaguje z właściwą sobie pasją i entuzjazmem podejście oparte na stosowaniu technik TDD w programowaniu.

Kolofon

Zwierzę znajdujące się na okładce książki *TDD w praktyce* to koza kaszmirska. Chociaż prawie wszystkie gatunki kóz mogą być hodowane w celu produkcji słynnego i cenionego kaszmiru, to jednak tylko te kaszmirskie są hodowane wyłącznie w takim celu. Dlatego też są uznawane za „lokatę kapitału”. Kozy kaszmirskie należą do gatunku kozy domowej (*Capra hircus*).

Sierść kóz kaszmirschich wykazuje swego rodzaju podwójność, ponieważ składa się z dwóch warstw. Pierwsza to warstwa wewnętrzna (podszytowa), zwana też zimową — jest delikatniejsza i zbudowana z miękkich włosów. Druga to warstwa zewnętrzna (pokrywowa), grubsza i szorstka. Warstwa wewnętrzna sierści pojawia się zimą i ma na celu uzupełnienie zewnętrznej warstwy ochronnej. Karbowane i niezwykle lekkie włosy puchowe warstwy zimowej zapewniają kozom świetną izolację cieplną.

Nazwę „kaszmirska” kozy zawdzięczają Kotlinie Kaszmirskiej położonej w regionie Kaszmiru w Himalajach (Indie), gdzie tkanina o tej samej nazwie jest znana i wytwarzana od tysięcy lat. Zmniejszenie populacji kóz kaszmirschich w dzisiejszym Kaszmirze doprowadziło do zaprzestania eksportu tego włókna z regionu. Obecnie większość wełny pochodzącej od kóz kaszmirschich dostarczana jest z obszarów Afganistanu, Iranu, Mongolii, Indii oraz Chin, które są uznawane za jej największych dostawców.

Sierść kóz kaszmirschich ma różne kolory. Osobniki zarówno męskie, jak i żeńskie posiadają rogi, które chłodzą zwierzęta w lecie oraz ułatwiają im wykonywanie niektórych codziennych czynności życiowych.

Rysunek na okładce pochodzi ze zbiorów Wood's Animate Creation.

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄZKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA
Helion SA