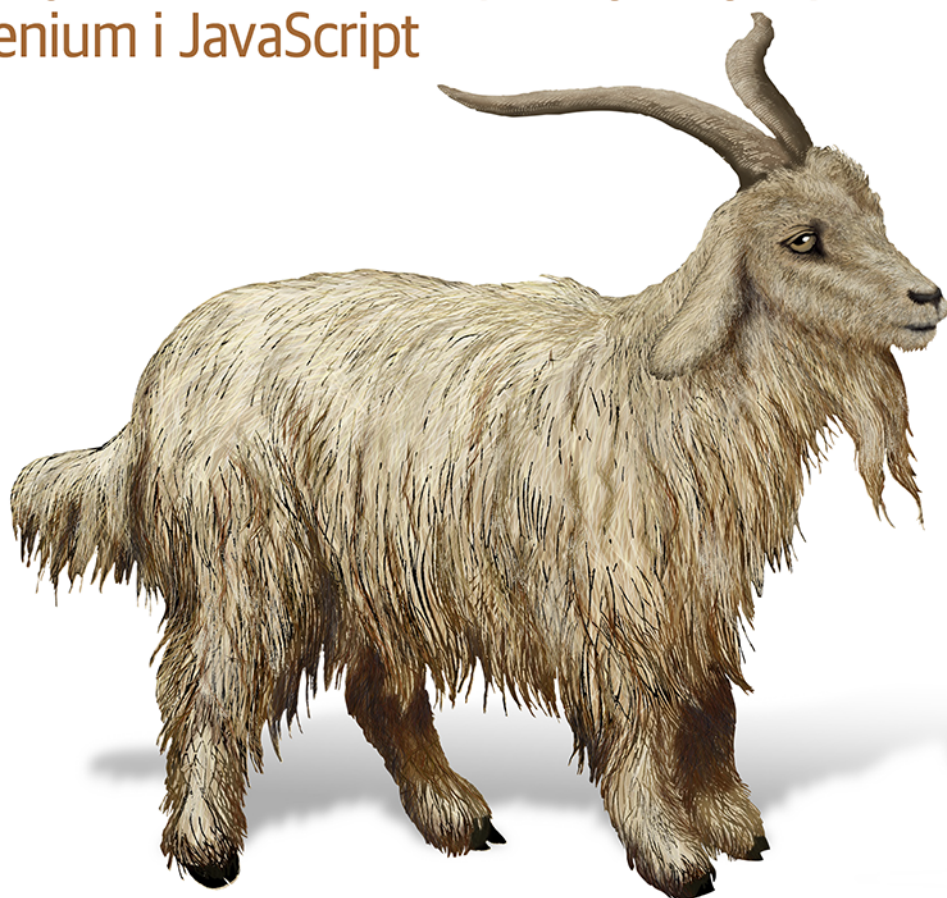


O'REILLY®

Wydanie III

Test-Driven Development w Pythonie

Praktyczne tworzenie aplikacji z Django,
Selenium i JavaScript



Helion 

Harry J. W. Percival

Tytuł oryginału: Test-Driven Development with Python: Obey the Testing Goat:
Using Django, Selenium, and JavaScript, 3rd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-3821-2

© 2026 Helion S.A.

Authorized Polish translation of the English edition of *Test-Driven Development with Python, 3E*
ISBN 9781098148713 © 2026 Harry J.W. 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.

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 jakiegokolwiek postaci jest zabronione.
Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie
książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie
praw autorskich niniejszej publikacji.

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

Autor oraz wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności
za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

helion.pl/user/opinie/tdwp3

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

WWW: helion.pl (księgarnia internetowa, katalog książek)

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

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

Spis treści

Wprowadzenie	17
Wprowadzenie do wydania trzeciego, czyli TDD w erze AI	23
Przygotowania i założenia	27
Podziękowania	36

Część I. Podstawy TDD i Django

1. Konfiguracja Django za pomocą testu funkcjonalnego	41
Słuchaj Testing Goat! Nie rób nic, dopóki nie przygotujesz testu	41
Rozpoczęcie pracy z frameworkiem Django	45
Utworzenie repozytorium Git	47
2. Rozszerzenie testu funkcjonalnego za pomocą modułu unittest	52
Użycie testu funkcjonalnego do przygotowania minimalnej aplikacji	52
Moduł unittest ze standardowej biblioteki Pythona	55
Przekazanie plików do repozytorium	58
3. Testowanie prostej strony głównej za pomocą testów jednostkowych	60
Nasza pierwsza aplikacja Django i test jednostkowy	61
Testy jednostkowe i różnice dzielące je od testów funkcjonalnych	61
Testy jednostkowe w Django	63
MVC w Django, adresy URL i funkcje widoku	64
Testy jednostkowe widoku	65
Wreszcie zaczynamy tworzyć kod aplikacji	66
Cykl test jednostkowy – tworzenie kodu	67
Testy funkcjonalne podpowiadają, że to jeszcze nie koniec	69
Odczyt stosu wywołań	71
urls.py	72

4. Do czego służą te wszystkie testy (i refaktoryzacja)?	76
Programowanie przypomina wyciąganie wiadrem wody ze studni	77
Użycie Selenium do testowania interakcji użytkownika	79
Reguła „nie testuj stałych” i szablony na ratunek	82
Refaktoryzacja w celu użycia szablonu	82
Modyfikacja testów jednostkowych	85
Testuj sposób działania, nie implementację	86
Refaktoryzacja	88
Nieco więcej o stronie głównej	89
Przypomnienie — proces TDD	91
Podwójna pętla TDD	92
5. Zapis danych wejściowych użytkownika — testowanie bazy danych	94
Od formularza internetowego do wykonania żądania POST	95
Testowanie kontraktu między frontendem i backendem	95
Debugowanie testów funkcjonalnych	97
Debugowanie z użyciem <code>time.sleep()</code>	98
Przetwarzanie żądania POST w serwerze	100
Przekazanie zmiennych Pythona do wygenerowania w szablonie	102
Oczekiwane niepowodzenie	103
Usprawnianie komunikatów błędów w testach	104
Do trzech razy sztuka, a później refaktoryzacja	106
Django ORM i nasz pierwszy model	108
Pierwsza migracja bazy danych	110
Zdumiewająco duży postęp w teście	111
Nowa kolumna oznacza nową migrację	112
Zapis w bazie danych informacji z żądania POST	113
Przekierowanie po wykonaniu żądania POST	116
Każy test powinien testować jedną rzecz	118
Wygenerowanie elementów w szablonie	119
Utworzenie produkcyjnej bazy danych za pomocą polecenia <code>migrate</code>	121
Podsumowanie	125
6. Usprawnienie testów funkcjonalnych — gwarancja izolacji i pozbycie się magicznie działających poleceń	127
Gwarancja izolacji testu w testach funkcjonalnych	127
Wykonanie tylko testów jednostkowych	130
Oczekiwanie jawne i niejawne oraz magiczne wywołania <code>time.sleep</code>	131

7. Praca małymi krokami	136
Stawiaj na małe projekty	136
Rozpocznij od niewielkiego projektu	136
YAGNI!	137
REST	138
Implementacja nowego projektu za pomocą TDD	139
Upewnienie się o istnieniu testu regresji	140
Iteracja w kierunku nowego projektu	142
Wykonanie pierwszego samodzielnego kroku: nowy adres URL	143
Rozdzielenie funkcjonalności strony głównej i widoku listy	144
Testy funkcjonalne wykrywają regresję	146
Jak najszybszy powrót do działającej wersji aplikacji	148
Zielony? Refaktoryzacja	150
Kolejny mały krok: oddzielny szablon do wyświetlania list	150
Trzeci mały krok: nowy adres URL pozwalający dodać elementy listy	154
Klasa testowa dla operacji tworzenia nowej listy	155
Adres URL i widok przeznaczony do tworzenia nowej listy	156
Usunięcie zbędnego kodu i dalsze testy	157
Regresja! Formularze korzystają z nowego adresu URL	157
Debugowanie za pomocą narzędzi programistycznych w przeglądarce internetowej	158
Chwytały byka za rogi, czyli dostosowanie modeli	160
Związek klucza zewnętrznego	162
Dostosowanie reszty świata do naszych nowych modeli	163
Każda lista powinna mieć własny adres URL	165
Przechwytywanie parametrów z adresów URL	166
Dostosowanie <code>new_list</code> do nowego świata	168
Testy funkcjonalne wskazują na jeszcze inną regresję	169
Jeszcze jeden adres URL pozwalający dodać element do istniejącej listy	170
Ostatni nowy wpis w pliku <code>urls.py</code>	171
Ostatni nowy widok	171
Bezpośrednie testowanie kontekstu szablonu	172
Ostatnia refaktoryzacja za pomocą polecenia <code>include</code>	175
Czy możesz w to uwierzyć?	176
 8. Upiększanie — jak przetestować układ i style?	 178
Jaką funkcjonalność należy testować w przypadku układu i stylów?	178
Upiększanie za pomocą frameworka CSS	181
Dziedziczenie szablonu w Django	183
Integracja z frameworkiem Bootstrap	184
Wiersze i kolumny	185

Pliki statyczne w Django	188
Zaczynamy używać klasy StaticLiveServerTestCase	189
Użycie komponentów Bootstrapa do poprawy wyglądu witryny	190
Jumbotron	190
Ogromne pola danych wejściowych	190
Nadanie stylu tabeli	191
Opcjonalny tryb ciemny	191
Całkiem przyzwoita strona	192
Analizowanie kodu w HTML-u w celu uzyskania bardziej niezawodnych testów	
kluczowej zawartości w HTML-u	193
Co zostało zatuszowane: polecenie collectstatic i inne katalogi statyczne	197
Kilka tematów, które nie zostały omówione	199

Część II. Wdrożenie w środowisku produkcyjnym

9. Konteneryzacja, czyli Docker	207
Docker, kontenery i wirtualizacja	207
Dlaczego po prostu nie używamy środowiska wirtualnego Pythona?	210
Docker i Twoje CV	210
Jak zwykle zaczynamy od testu	210
Utworzenie katalogu src	213
Instalowanie Dockera	213
Tworzenie obrazu Dockera i uruchamianie kontenera Dockera	215
Pierwsze zetknięcie z plikiem Dockerfile	216
Polecenie docker build	217
Polecenie docker run	218
Instalowanie Django w środowisku wirtualnym obrazu kontenera	218
Zakończone sukcesem uruchomienie kontenera	219
Stosowanie testu funkcjonalnego do sprawdzenia, czy kontener działa	220
Debugowanie problemów sieciowych związanych z kontenerem	221
Debugowanie za pomocą narzędzia curl dostępności serwera WWW	221
Stosowanie polecenia docker exec do uruchamiania kodu	
„wewnątrz” kontenera	222
Mapowanie portów Dockera	223
Wyszukiwanie w internecie informacji o komunikatach błędów	225
Migracje bazy danych	227
Dlaczego nie należy przeprowadzać migracji z poziomu pliku Dockerfile?	229
Montowanie plików w kontenerze	230

10. Przygotowanie aplikacji do wdrożenia	233
Co musimy jeszcze zrobić?	233
Użycie Gunicorna	234
Testy funkcjonalne wychytują problem z plikami statycznymi	235
Udostępnianie plików statycznych za pomocą WhiteNoise	235
Użycie pliku requirements.txt	237
Użycie zmiennych środowiskowych w celu dostosowania ustawień środowiska produkcyjnego	240
Ustawienia DEBUG=True i SECRET_KEY	240
Definiowanie zmiennych środowiskowych w pliku Dockerfile	241
Definiowanie zmiennych środowiskowych z poziomu powłoki Dockera	242
Po wyłączeniu trybu debugowania wymagana jest opcja ALLOWED_HOSTS	242
Polecenie collectstatic jest wymagane po wyłączeniu debugowania	244
Zastosowanie użytkownika bez uprawnień administratora	245
Możliwość konfiguracji ścieżki dostępu do pliku bazy danych	246
Stosowanie identyfikatorów użytkowników do zdefiniowania uprawnień podczas montowania plików w kontenerze i komputerze lokalnym	247
Konfigurowanie mechanizmu rejestrowania danych	248
Celowe wywołanie błędu	248
Ćwiczenie do samodzielnego wykonania: użycie polecenia check w Django	251
Podsumowanie	252
11. Przygotowanie serwera do wdrożenia aplikacji	253
Ręczne przygotowanie serwera do hostingu witryny	253
Wybór hostingu dla witryny	254
Uruchomienie serwera	254
Pobranie nazwy domeny	255
Konfiguracja DNS dla witryn testowej i rzeczywistej	256
Ansible	257
Ansible kontra SSH — komunikacja z serwerem	257
Na początek upewniamy się, czy możemy uzyskać dostęp poprzez SSH	258
Debugowanie problemów z SSH	259
Instalowanie Ansible	261
Sprawdzenie, czy Ansible może komunikować się z serwerem	261
12. Infrastruktura jako kod — zautomatyzowane wdrożenia za pomocą Ansible	264
Pierwsza wersja scenariusza Ansible dla wdrożenia	265
Nawiązanie połączenia SSH z serwerem i wyświetlenie dzienników zdarzeń kontenera	267
Umożliwienie Dockerowi dostępu bez uprawnień użytkownika root	269
Przekazanie obrazu na serwer	271
Ciąg dalszy rozglądania się	274

Zdefiniowanie zmiennych środowiskowych i kluczy tajnych użytkownika	275
Samodzielne sprawdzenie zmiennych środowiskowych w uruchomionym kontenerze	277
Wykonanie testów funkcjonalnych w celu sprawdzenia wdrożenia	279
Debugowanie serwera testowego za pomocą narzędzia curl	279
Montowanie bazy danych na serwerze i przeprowadzanie migracji	281
Działaaa!	283
Wdrożenie do środowiska produkcyjnego	283
Oznaczenie wydania tagiem Gita	284
Powiedz wszystkim!	284
Źródła dalszej nauki	285

Część III. Formularze i weryfikacja

13. Podział testów na wiele plików i praca z ogólnymi metodami pomocniczymi	289
Testy funkcjonalne weryfikacji danych — ochrona przed pustymi elementami	289
Pominięcie testu	290
Podział testów funkcjonalnych na wiele plików	291
Wykonanie pojedynczego pliku testu	294
Nowe narzędzie testów funkcjonalnych —	
ogólna metoda pomocnicza oczekiwania	295
Dokończenie pracy nad testem funkcjonalnym	298
Refaktoryzacja testów jednostkowych na oddzielne pliki	299
14. Weryfikacja na poziomie bazy danych	302
Sprawdzenie warstwy modelu	303
Menedżer kontekstu <code>self.assertRaises()</code>	303
Ograniczenia modelu Django i ich interakcja z bazami danych	304
Sprawdzanie ograniczeń na poziomie bazy danych	304
Testowanie weryfikacji modelu Django	305
Dziwactwo Django — zapis modelu nie wywołuje operacji	
sprawdzenia poprawności	306
Wyświetlanie w widoku błędów z weryfikacji modelu	307
Upewnienie się, że nieprawidłowe dane nie zostaną zapisane w bazie danych	310
Dodanie wcześniejszego polecenia <code>return</code> do testu funkcjonalnego,	
aby umożliwić refaktoryzację w stosunku do wersji zielonej	312
Wzorzec Django: przetwarzanie żądań POST w widoku generującym formularz	312
Refaktoryzacja: przekształcenie funkcjonalności <code>new_item</code> na <code>view_list</code>	313
Egzekwowanie w widoku <code>view_list</code> weryfikacji modelu	317
Refaktoryzacja: usunięcie na stałe zdefiniowanych adresów URL	319
Znacznik szablonu <code>{% url %}</code>	319
Użycie <code>get_absolute_url</code> w przekierowaniach	320

15. Prosty formularz	323
Przeniesienie do formularza logiki odpowiedzialnej	
za sprawdzanie poprawności danych	323
Użycie testu jednostkowego do analizy API formularzy	324
Przejsie do Django ModelForm	326
Testowanie i dostosowanie do własnych potrzeb logiki weryfikacji formularza	327
Próba użycia formularza w widokach	329
Użycie formularza w widoku za pomocą żądania GET	330
Kompromisy klasy ModelForms w Django — frontend jest połączony z bazą danych	332
Duża operacja znajdź i zastąp	332
Wycofanie zmian i przywrócenie aplikacji do stanu działającego	332
Zmiana atrybutu name	334
Zmiana atrybutu id	336
Druga próba użycia formularza w widokach	338
Użycie formularza w widoku obsługującym żądania POST	341
Użycie formularza w celu wyświetlenia błędów w szablonie	341
Przywrócenie aplikacji do stanu działającego	342
Metoda pomocnicza dla wielu krótkich testów	343
Użycie formularza w istniejącym widoku listy	346
Użycie formularza do przekazywania błędów do szablonu	346
Refaktoryzacja widoku w celu pełnego użycia formularza	347
Nieoczekiwana korzyść: bezpłatna weryfikacja po stronie klienta	
za pomocą HTML5	350
Ułga	352
Czy zmarnowaliśmy dużo czasu?	352
Użycie metody save() formularza	353
16. Bardziej skomplikowane formularze	356
Kolejny test funkcjonalny dotyczący powielonych elementów	356
Ochrona przed duplikatami w warstwie modelu	357
Przepisanie testu starego modelu	360
Pewne błędy spójności ujawniają się podczas zapisu	360
Eksperymenty w warstwie widoku sprawdzające, czy są powielone elementy	362
Bardziej skomplikowany formularz do obsługi unikalności elementów	364
Użycie istniejącego formularza w widoku listy	366
Dostosowanie metody save() do potrzeb nowego formularza	367
Testy funkcjonalne wychwytyują problem z klasami Bootstrapa	368
Warunkowe dostosowywanie klas CSS dla nieprawidłowych formularzy	370
Mała dygresja o kolejności API Querystring i przedstawianiu ciągu tekstowego	372

O kompromisach związanych z klasą Django ModelForm i ogólnie frameworkami	374
Przeniesienie logiki prezentacyjnej z powrotem do szablonu	376
Porządkowanie formularzy	380
Powrót do prostych formularzy	381
Podsumowanie informacji o testowaniu	382

Część IV. Bardziej zaawansowane zagadnienia związane z testowaniem

17. Zagłębiaamy się ostrożnie w JavaScript	387
Rozpoczynamy od testów funkcjonalnych	388
Krótką sesję z kodem eksperymentalnym	390
Prosty skrypt osadzony w kodzie	391
Użycie narzędzi programistycznych przeglądarki internetowej	392
Wybór prostego silnika wykonywania testów w JavaScriptcie	394
Ogólne omówienie frameworka Jasmine	394
Przygotowanie środowiska testowego dla JavaScriptu	395
Pierwszy test dymny — bloki describe, it, expect	396
Wykonywanie testów za pomocą przeglądarki internetowej	397
Testowanie z zawartością modelu DOM	399
Tworzenie testu jednostkowego JavaScriptu dla żądanej funkcjonalności	401
Dane testowe, kolejność wykonywania i stan globalny — największe wyzwania podczas testowania JavaScriptu	403
Wywołanie console.log() do wyświetlenia informacji debugowania	403
Korzystanie z funkcji inicjalizującej dla większej kontroli nad czasem wykonania	404
Celowe psucie kodu, żeby zmusić się do utworzenia większej liczby testów	406
Czerwone – zielone – refaktoryzacja — usunięcie na stałe zdefiniowanych selektorów	407
Czy to działa?	410
Testowanie integracji z CSS i Bootstrapem	411
Columbo mówi: poczekaj na zdarzenie onload	413
Testowanie JavaScriptu w cyklu TDD	414
18. Wdrożenie nowego kodu	416
Lista kontrolna dla wdrożenia	416
Lokalne wykonanie pełnego zestawu testów	417
Krótki zestaw testów w Dockerze	417
Wdrożenie w środowisku testowym i uruchomienie testów	418
Wdrożenie w środowisku produkcyjnym	419

A jeśli wystąpi błąd bazy danych?	419
Jak usunąć bazę danych w środowisku testowym?	419
Podsumowanie — git tag i nowe wydanie	420
19. Uwierzytelnianie użytkownika oraz tworzenie i usuwanie kodu eksperymentalnego	421
Uwierzytelnianie bez hasła, za pomocą „magicznych łączy”	422
Nieco większy kod eksperymentalny	422
Utworzenie nowej gałęzi dla spike	423
Interfejs użytkownika logowania	423
Wysyłanie wiadomości e-mail z Django	424
Konfigurowanie serwera poczty w Django	426
Kolejny klucz tajny użytkownika i kolejna zmienna środowiskowa	426
Przechowywanie tokenów w bazie danych	427
Niestandardowe modele uwierzytelniania	428
Dokończenie pracy nad niestandardowym uwierzytelnieniem Django	430
Zmiana rozwiązania eksperymentalnego na zwykłe	433
Opracowanie planu	434
Tworzenie testu funkcjonalnego dla kodu eksperymentalnego	434
Wycofywanie kodu eksperymentalnego	436
Minimalny niestandardowy model użytkownika	438
Testy jako dokumentacja	442
Model tokena do powiązania łącza z unikatowym identyfikatorem	443
20. Stosowanie imitacji do testowania zależności zewnętrznych	447
Zanim zaczniemy — przygotowanie podstawowej infrastruktury	448
Imitacje ręczne — monkeypatching	449
Biblioteka mock Pythona	452
Użycie unittest.patch	453
Niewielki postęp z testem funkcjonalnym	455
Testowanie frameworka komunikatów Django	455
Dodawanie komunikatów do kodu w HTML-u	458
Rozpoczęcie pracy nad adresem URL do logowania się	460
Sprawdzenie, czy wysyłamy użytkownikowi łącze z tokenem	460
Zmiana eksperymentalnej wersji backendu uwierzytelniania na zwykłą	463
Polecenie if oznacza więcej testów	463
Metoda get_user()	466
21. Izolacja testów za pomocą imitacji	470
Używanie w widoku logowania backendu mechanizmu uwierzytelniania	470
Prosty test widoku bez użycia imitacji	472
Problem eksplozji kombinatorycznej	474
Przykład fabryki samochodów	474

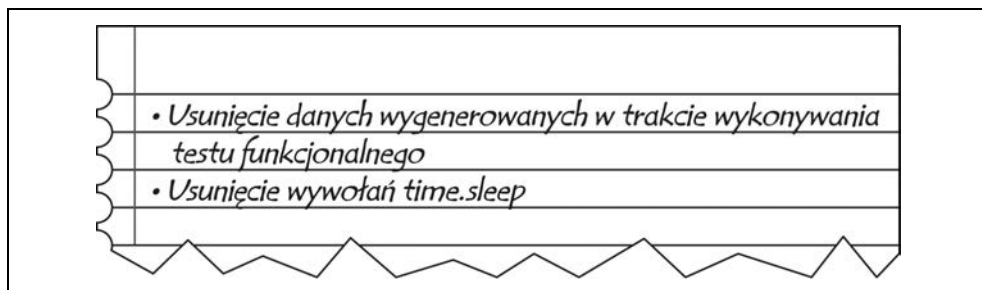
Stosowanie imitacji do oddzielnego testowania elementów systemu	476
Imitacje umożliwiają również testowanie implementacji, gdy ma to znaczenie	478
Zaczynamy od nowa — testowanie implementacji za pomocą imitacji	478
Stosowanie atrybutu <code>mock.return_value</code>	482
Stosowanie atrybutu <code>.return_value</code> podczas przygotowywania testu	485
Wycofanie operacji NIEowania testu	486
Które testy powinny być zachowane?	487
Chwila prawdy — czy test funkcjonalny zostanie zaliczony?	488
Działa w teorii, ale czy w praktyce?	489
Użycie nowej zmiennej środowiskowej i zapisanie jej w pliku <code>.env</code>	489
Dokończenie testu funkcjonalnego — testowanie wylogowania	492
22. Konfiguracja testu i dekorator wyraźnego oczekiwania	496
Pominięcie procesu logowania przez wstępne utworzenie sesji	496
Sprawdzamy rozwiązanie	499
Ostateczna metoda pomocnicza wyraźnego oczekiwania — dekorator <code>wait</code>	501
23. Debugowanie po stronie serwera testowego	506
Dowód znajdziesz w praktyce — użycie Dockera do wychwycenia błędów	506
Sprawdzanie dzienników zdarzeń kontenera Dockera	507
Kolejna zmienna środowiskowa w Dockerze	508
<code>mail.outbox</code> nie będzie działać poza środowiskiem testowym Django	509
Jak testować „rzeczywiste” wysyłanie wiadomości e-mail?	510
Alternatywna metoda definiowania na serwerze zmiennych środowiskowych z tajnymi wartościami	512
Debugowanie z użyciem kodu w SQL-u	513
Zarządzanie konfiguracją testów w rzeczywistych bazach danych	514
Polecenie Django do tworzenia sesji	515
Test funkcjonalny uruchamiający w serwerze narzędzie zarządzania	516
Wykonywanie poleceń za pomocą <code>docker exec</code> i (opcjonalnie) SSH	517
Podsumowanie tworzenia sesji w systemie lokalnym i testowym	518
Testowanie narzędzi zarządzania	520
Porządkowanie testowej bazy danych	521
Podsumowanie	522
24. Kończymy „Moje listy” — podejście outside-in	524
Alternatywa, czyli podejście inside-out	524
Dlaczego preferowane jest podejście outside-in?	525
Test funkcjonalny dla strony „Moje listy”	525
Warstwa zewnętrzna — prezentacja i szablony	528
Przejdźcie o jedną warstwę w dół do funkcji widoku (kontroler)	529

Kolejne zaliczenie — podejście outside-in	530
Szybka restrukturyzacja szablonu	531
Wczesne zakończenie, czyli refaktoryzacja pod kątem zielonego	533
Wyodrębnienie dwóch tagów include w szablonie	533
Projektowanie API za pomocą szablonu	536
Przejdźcie w dół do kolejnej warstwy — co widok przekazuje szablónowi?	539
Kolejne wymaganie z warstwy widoku — nowe listy powinny zapamiętywać swego właściciela	539
Czy przejść do kolejnej warstwy, gdy test kończy się niepowodzeniem?	540
Przejdźcie w dół do warstwy modelu	541
Ostatni krok: uzyskanie z poziomu szablonu dostępu do właściciela za pomocą API .name	543
25. Ciągła integracja	546
Ciągła integracja w nowoczesnych przepływach pracy	547
Wybór usługi ciągłej integracji	547
Przekazanie naszego kodu do serwisu GitLab	548
Utworzenie konta	548
Utworzenie projektu	548
Przekazanie kodu za pomocą git push	549
Początkowa konfiguracja potoku CI	550
Pierwsza kompilacja (i pierwsze niepowodzenie)	551
Próba lokalnego odtworzenia błędu CI	554
Włączanie dzienników debugowania dla Selenium, Firefoksa i Webdrivera	555
Włączenie trybu headless dla przeglądarki Firefox	557
Niestabilne testy — najczęstszy problem w Selenium	558
Wykonanie zrzutów ekranu	559
Zapisywanie danych wyjściowych kompilacji (lub plików debugowania) jako artefaktów	560
W razie wątpliwości spróbuj wydłużyć limit czasu	561
Pomyślne wykonanie zestawu testów Pythona	563
Wykonanie testów JavaScriptu w CI	563
Instalacja Node.js	564
Instalowanie i konfigurowanie Jasmine	564
Dodawanie kroku kompilacji dla kodu w JavaScriptcie	567
Testy są obecnie zaliczone	568
Więcej zadań do wykonania za pomocą serwera ciągłej integracji	570
Zdefiniowanie obrazu Dockera dla CI	570
Buforowanie	570
Zautomatyzowane wdrożenia, czyli ciągłe wdrażanie (CD)	571

26. Token serwisów społecznościowych, wzorzec strony i ćwiczenie dla czytelnika	572
Test funkcjonalny z wieloma użytkownikami i funkcja addCleanup()	572
Wzorzec strony	574
Rozszerzenie testu funkcjonalnego na drugiego użytkownika i stronę „Moje listy”	576
Ćwiczenie dla czytelnika	578
Wyjaśnienie krok po kroku	578
27. Szybkie testy, wolne testy i gorąca lawa	581
Dlaczego testujemy? Nasze wymagania dotyczące skutecznych testów	582
Pewność i poprawność (zapobieganie regresji)	583
Wydajny przepływ pracy	583
Dążenie do lepszego projektu	583
Czy nasze testy jednostkowe były od początku testami integracyjnymi?	
Skąd ta ciepła poświata bijąca od bazy danych?	583
Byliśmy w „idealnym punkcie”	584
Czym jest „prawdziwy” test jednostkowy i dlaczego to ma znaczenie?	584
Wraz z upływem czasu testy integracyjne i funkcjonalne są wykonywane coraz wolniej	584
Nie wykorzystujemy w pełni potencjalnych korzyści z testowania	585
Idealna piramida testów	586
Unikanie piekła imitacji	587
Rzeczywiste rozwiązania są architektoniczne	588
Porty i adaptory, architektura heksagonalna, architektura cebulowa i czysta architektura	589
Architektura Functional Core, Imperative Shell	590
Górnołotna metafora — te architektury są „lepsze”	590
Najtrudniejsze zadanie: trzeba wiedzieć, kiedy przeprowadzić zmianę	591
Podsumowanie	592
Dalsza lektura	592
Niech Cię prowadzi Testing Goat!	594
Bibliografia	596
A. Ściąga	597
B. Co dalej?	600
C. Przykładowe fragmenty kodu	604
Skorowidz	607

Usprawnienie testów funkcjonalnych — gwarancja izolacji i pozbycie się magicznie działających poleceń

Zanim zagłębimy się w temat i rozwiążemy problem z pojedynczą listą globalną, zajmijmy się kilkoma sprawami porządkowymi (rysunek 6.1). Pod koniec poprzedniego rozdziału zauważyliśmy, że różne testy kolidują ze sobą, więc to naprawimy. Nie jestem też zadowolony z wywołań `time.sleep` w kodzie — wydają się niezbyt profesjonalne i dlatego zastąpimy je czymś bardziej niezawodnym.



Rysunek 6.1. Obecna postać naszej osobistej listy rzeczy do zrobienia

Rozwiązanie obu tych kwestii pozwoli przejść w kierunku najlepszych praktyk podczas testowania, a dzięki temu testy staną się bardziej deterministyczne i niezawodne.

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 dane, które mogą zakłócać wyniki kolejnych przeprowadzanych testów.

Kiedy wykonujemy testy *jednostkowe*, oferowany przez Django silnik testów automatycznie tworzy zupełnie nową bazę danych dla testów (to zupełnie inna baza danych niż używana przez aplikację). Tę testową bazę 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 *db.sqlite3*.

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

Ponieważ to jest powszechny problem, Django udostępnia klasę testową *LiveServerTestCase*, gotową do wykonania wymienionych wcześniej zadań. Automatycznie utworzy testową bazę danych (podobnie jak ma to miejsce podczas wykonywania testu jednostkowego) i uruchomi serwer umożliwiający wykonanie testów funkcjonalnych. Wprawdzie to 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*, który wyszukuje wszystkie pliki o nazwach rozpoczynających się od *test_*. Dla zachowania przejrzystości i porządku tworzymy katalog dla testów funkcjonalnych, aby całość bardziej przypominała aplikację. Django wymaga, aby ten katalog był prawidłowym modulem Pythona (wraz z plikiem *__init__.py*):

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

Następnie *przenosimy* testy funkcjonalne z oddzielnego pliku *functional_tests.py* do pliku *tests.py* aplikacji *functional_tests*. Wydajemy polecenie *git mv*, aby Git „zauważył” przeniesienie pliku. To pozwoli zachować pojedynczą historię zmian w repozytorium:

```
$ git mv functional_tests.py functional_tests/tests.py
$ git status # Polecenie 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
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   ├── 0001_initial.py
│   │   ├── 0002_item_text.py
│   │   └── __init__.py
│   ├── models.py
│   ├── templates
│   └── home.html
```

```

|   |   | tests.py
|   |   | views.py
|   |   |
|   |   | manage.py
|   |   | superlists
|   |   |
|   |   | |   | __init__.py
|   |   | |   | asgi.py
|   |   | |   | settings.py
|   |   | |   | urls.py
|   |   | |   | wsgi.py

```

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



Istnieje możliwość łączenia testów funkcjonalnych z testami aplikacji *lists*. Ja wolę je rozdzielać, ponieważ testy funkcjonalne zwykle mają różne zadania wykonywane w poszczególnych aplikacjach. Testy funkcjonalne służą 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
[...]

class NewVisitorTest(LiveServerTestCase):
    def setUp(self):
        [...]

```

Następnie zamiast na stałe definiować przejście do portu 8000 w komputerze lokalnym, klasa *LiveServerTestCase* udostępnia atrybut `live_server_url`:

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

```

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

```

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

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

```

$ python manage.py test functional_tests
Creating test database for alias 'default'...
Found 1 test(s).
System check identified no issues (0 silenced).
.

```

```
-----
Ran 1 test in 10.519s
```

OK

Destroying test database for alias 'default'...



Gdy dzisiaj przeprowadzałem ten test, natknąłem się na wyskakujące okienko z prośbą o uaktualnienie przeglądarki internetowej Firefox. Zatem przypominam, na wypadek jeśli też zobaczysz takie okienko, że w rozdziale 1. znajdziesz ramkę z informacjami o tym, co należy wówczas zrobić.

Testy funkcjonalne są nadal zaliczane i tym samym potwierdzają, że podczas refaktoryzacji niczego nie zepsuliśmy. 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 niepodzielną zmianę:

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

Wykonanie tylko testów jednostkowych

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

```
$ python manage.py test
Creating test database for alias 'default'...
Found 8 test(s).
System check identified no issues (0 silenced).
.....
```

```
-----
Ran 8 tests in 10.859s
```

OK

Destroying test database for alias 'default'...

Aby wykonać tylko testy jednostkowe, trzeba wskazać, że chcemy przeprowadzić jedynie testy aplikacji `lists`:

```
$ python manage.py test lists
Creating test database for alias 'default'...
Found 7 test(s).
System check identified no issues (0 silenced).
.....
```

```
-----
Ran 7 tests in 0.009s
```

OK

Destroying test database for alias 'default'...

Uaktualnienie dotyczące użytecznych poleceń

Wykonanie testów funkcjonalnych

python manage.py test functional_tests

Wykonanie testów jednostkowych

python 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 po zaliczeniu wszystkich testów jednostkowych. Jeżeli masz jakiegokolwiek wątpliwości, warto wykonać oba rodzaje testów!

Oczekiwanie jawne i niejawne oraz magiczne wywołania `time.sleep`

Rozważmy wywołanie `time.sleep` w teście funkcjonalnym:

Plik `functional_tests/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)
time.sleep(1)

self.check_for_row_in_list_table("1. Kupić pawie pióra")
```

To jest przykład tak zwanego jawnego oczekiwania i stanowi przeciwieństwo niejawnego oczekiwania: w pewnych sytuacjach Selenium próbuje „automatycznie” czekać na użytkownika, gdy uważa, że strona jest wczytywana. Udostępnia nawet metodę `implicitly_wait()`, pozwalającą kontrolować czas oczekiwania na element, który prawdopodobnie jeszcze nie znajduje się na stronie.

W pierwszym wydaniu książki mogłem całkowicie polegać na oczekiwaniu niejawnym. Problem polega na tym, że to oczekiwanie jest zawsze nieco niestabilne, a ponadto wraz z wydaniem Selenium 4 zostało domyślnie wyłączone. Jednocześnie ogólna opinia zespołu Selenium jest taka, że oczekiwanie niejawne to po prostu zły pomysł i należy go unikać¹.

Dlatego kod zamieszczony w tym wydaniu od samego początku korzysta z oczekiwania jawnego. Jednak w tym przypadku problem polega na tym, że wywołania `time.sleep` powodują inne problemy.

Obecnie czekamy sekundę, ale skąd wiadomo, że to jest wystarczające? W przypadku większości testów przeprowadzanych w komputerze lokalnym sekunda to zdecydowanie za długo i bardzo spowolni obliczenia w teście funkcjonalnym. Wartość 0,1 sekundy byłaby wystarczająca.

¹ Zobacz <https://www.selenium.dev/documentation/webdriver/waits/>.

Jednak problem polega na tym, że jeśli zdefiniujemy tak małą wartość, co jakiś czas będzie zgłaszana pozorna awaria, ponieważ z jakiegoś powodu akurat wtedy komputer działał nieco za wolno. Nawet w przypadku sekundy istnieje ryzyko wystąpienia losowych awarii, które nie wskazują na rzeczywisty problem, a mylące wyniki w testach są prawdziwą udręką².



Nieoczekiwane błędy typu `NoSuchElementException` i `StaleElementException` często wskazują na potrzebę zastosowania oczekiwania jawnego.

Zatem zastąpmy problematyczne wywołania narzędziem, które będzie czekać tak długo, jak to będzie potrzebne, aż do przekroczenia całkiem długiego limitu czasu, aby wychwycić wszelkie błędy. Nazwę metody `check_for_row_in_list_table()` zmienimy na `wait_for_row_in_list_table()` i zdefiniujemy w niej logikę odpytywania i ponawiania operacji:

Plik `functional_tests/tests.py` (ch06l004):

```
[...]
from selenium.common.exceptions import WebDriverException
import time

MAX_WAIT = 5 ❶

class NewVisitorTest(LiveServerTestCase):
    def setUp(self):
        [...]
    def tearDown(self):
        [...]

    def wait_for_row_in_list_table(self, row_text):
        start_time = time.time()
        while True: ❷
            try:
                table = self.browser.find_element(By.ID, "id_list_table") ❸
                rows = table.find_elements(By.TAG_NAME, "tr")
                self.assertIn(row_text, [row.text for row in rows])
                return ❹
            except (AssertionError, WebDriverException): ❺
                if time.time() - start_time > MAX_WAIT: ❻
                    raise ❼
                time.sleep(0.5) ❼
```

- ❶ Za pomocą stałej `MAX_WAIT` określamy maksymalny czas, jaki jesteśmy gotowi oczekiwać na wynik operacji. Pięć sekund powinno wystarczyć na wychwycenie wszelkich zakłóceń bądź przypadkowych spowolnień.
- ❷ Tutaj została zdefiniowana pętla działająca w nieskończoność, o ile nie zostanie wybrane jedno z dwóch wyjść.
- ❸ W tym miejscu mamy trzy wiersze asercji z poprzedniej wersji metody.
- ❹ Jeśli uda się je spełnić, to działanie funkcji się kończy i opuszczamy pętlę.

² Więcej informacji na ten temat znajdziesz w artykule Martina Fowlera na stronie <https://martinfowler.com/articles/nonDeterminism.html>.

- ❾ Natomiast w przypadku zgłoszenia wyjątku czekamy chwilę i ponawiamy operację. Chcemy wychwycić dwa typy wyjątków. Pierwszy to `WebDriverException`, gdy strona nie została wczytana i Selenium nie może znaleźć elementu tabeli na stronie. Drugi to `AssertionError`, gdy tabela istnieje, ale prawdopodobnie w postaci sprzed odświeżenia strony, więc nie zawiera jeszcze naszego wiersza.
- ❿ A to jest drugie wyjście. Po dotarciu do tego miejsca można przyjąć założenie, że kod zgłaszał wyjątki w trakcie każdej wcześniejszej próby przeprowadzenia operacji, aż do przekroczenia limitu czasu. Tym razem ponownie zgłaszamy wyjątek i pozwalamy przekazać go do testu. Ten wyjątek prawdopodobnie trafi do komunikatu błędu i pomoże wyjaśnić przyczynę niezaliczenia testu.

Czy uważasz, że ten kod jest dość brzydki i nieco utrudnia zrozumienie przyjętego sposobu działania? Zgadzam się. W dalszej części książki (rozdział 13.) przeprowadzimy refaktoryzację ogólnej metody pomocniczej `wait_for()`, aby oddzielić logikę pomiaru czasu i asercji testowych. Ale poczekamy, aż to będzie potrzebne w wielu miejscach.



Jeśli już korzystałeś z Selenium, to możesz wiedzieć o istnieniu kilku funkcji pomocniczych do obsługi oczekiwania³. Nie jestem ich wielkim fanem, choć tak naprawdę nie ma żadnego obiektywnego powodu. W trakcie lektury książki opracujemy kilka narzędzi pomocniczych do obsługi oczekiwania, które według mnie pozwolą utworzyć czytelny kod. Jednak w wolnym czasie oczywiście warto rzucić okiem na autorskie funkcje Selenium i sprawdzić, czy Ci odpowiadają.

Kolejnym krokiem jest zmiana nazw wywoływanych metod i usunięcie magicznych wywołań `time.sleep`:

Plik `functional_tests/tests.py` (ch06l005):

```
[...]
# 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.wait_for_row_in_list_table("1. Kupić pawie pióra")

# Na stronie nadal znajduje się pole tekstowe zachęcające do podania kolejnego zadania.
# Edyta wpisała "Użyć pawich piór do zrobienia przynęty"
# (Edyta jest niezwykle skrupulatna).
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.wait_for_row_in_list_table("2. Użyć pawich piór do zrobienia przynęty")
self.wait_for_row_in_list_table("1. Kupić pawie pióra")
[...]
```

³ Zobacz <https://www.selenium.dev/documentation/webdriver/waits/#explicit-waits>.

Zobaczmy, jaki będzie wynik wykonania testów:

```
$ python manage.py test
Creating test database for alias 'default'...
Found 8 test(s).
System check identified no issues (0 silenced).
```

```
.....
```

```
-----
Ran 8 tests in 4.552s
```

```
OK
```

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

Hura! Testy ponownie są zaliczone. Zwróć również uwagę na skrócenie o kilka sekund czasu ich wykonania. Wprawdzie oszczędność czasu może wydawać się niewielka, ale pamiętaj, że zaoszczędzone sekundy się sumują.

Aby sprawdzić, czy zrobiliśmy to dobrze, celowo zepsujmy test na kilka sposobów i sprawdźmy, czy występują jakieś błędy. Najpierw spróbujmy wyszukać nieistniejący tekst i sprawdźmy, czy otrzymujemy oczekiwany błąd:

Plik *functional_tests/tests.py* (ch06l006):

```
def wait_for_row_in_list_table(self, row_text):
    [...]
    rows = table.find_elements(By.TAG_NAME, "tr")
    self.assertIn("foo", [row.text for row in rows])
    return
```

Wynikiem jest niezaliczenie testu i wygenerowanie czytelnego komunikatu błędu, który wyjaśnia przyczynę niepowodzenia:

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



Czy znudziło Cię czekanie pięć sekund na niepowodzenie testu? To jedna z wad oczekiwania jawnego. Mamy więc trudny kompromis między oczekiwaniem wystarczająco długim, aby uniknąć drobnych błędów, i jednocześnie na tyle długim, że obserwowanie spodziewanych awarii jest boleśnie powolne. Dobrym pomysłem może być więc zdefiniowanie konfigurowanej wartości `MAX_WAIT`, aby testy działały szybko w środowisku programistycznym, ale bardziej konserwatywnie na serwerach ciągłej integracji (ang. *continuous integration*, CI). Wprowadzenie do ciągłej integracji znajdziesz w rozdziale 25.

Wróćmy do poprzedniego stanu i popsujmy coś innego:

Plik *functional_tests/tests.py* (ch06l007):

```
try:
    table = self.browser.find_element(By.ID, "id_nothing")
    rows = table.find_elements(By.TAG_NAME, "tr")
    self.assertIn(row_text, [row.text for row in rows])
    return
[...]
```

W tym przypadku oczywiście pojawiają się błędy, gdy strona nie zawiera szukanego elementu:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate  
element: [id="id_nothing"]; For documentation on this error, [...]
```

Wszystko wydaje się w porządku. Przywróćmy kod do stanu, w jakim powinien się znajdować, i przeprowadźmy ostatni test:

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

Doskonale. Skoro ta krótka przerwa jest już za nami, możemy zabrać się za przystosowanie aplikacji do obsługi wielu list. Nie zapomnij najpierw przekazać zmian do repozytorium.

Najlepsze praktyki z zakresu testowania, które zastosowaliśmy w tym rozdziale

Gwarancja izolacji testu w testach funkcjonalnych

Poszczególne testy nie powinny na siebie wpływać. Zatem po zakończeniu każdego testu należy zerować wszelki trwały stan. Silnik testów Django pomaga w tym poprzez utworzenie testowej bazy danych, której zawartość jest usuwana między poszczególnymi testami.

Unikanie „magicznych” wywołań `time.sleep`

Gdy musimy czekać na wczytanie czegoś, pojawia się pokusa użycia wywołania `time.sleep`. Jednak problem polega na tym, że czas oczekiwania zawsze jest trochę ryzykowny — albo zbyt krótki i podatny na pozorne błędy, albo zbyt długi, co spowolni testy. Lepszym rozwiązaniem będzie zastosowanie pętli ponawiania, która wykonuje zapytania do aplikacji i jak najszybciej przechodzi do kolejnego etapu.

Unikanie polegania na oczekiwaniach niejawnych w Selenium

Wprawdzie teoretycznie Selenium wykonuje oczekiwania niejawne, ale implementacja różni się w zależności od przeglądarki internetowej i nie zawsze jest niezawodna. „Jawne jest lepsze niż niejawne”, jak głosi Zen Pythona⁴, i dlatego należy preferować oczekiwania jawne.

⁴ `python -c "import this"`

A

- adres URL, 142
 - do logowania się, 460
 - dla listy, 142–144, 154–157, 165, 170
 - przechwytywanie parametrów, 166
 - usuwanie, 319
- Anaconda, 35
- analiza
 - API formularzy, 324
 - kodu w HTML-u, 193
 - pliku cookie, 498
 - stosu wywołań, 71
- analyzer składni lxml, 194
- Ansible, 257, 263
 - instalowanie, 261
 - kommunikacja z serwerem, 258, 261
 - scenariusz, playbook, 261
 - wdrażanie aplikacji, 265
 - wyeksportowanie obrazu kontenera, 271
 - zadania, tasks, 265
 - zautomatyzowane wdrożenia, 264
- API
 - projektowanie za pomocą szablonu, 536
 - Querystring, 372
 - REST, 138
- architektura
 - cebulowa, 589
 - czysta, 589
 - Functional Core, 590
 - heksagonalna, 589
 - Imperative Shell, 590
- argument for_list=, 367
- argumenty wariacyjne, 503
- arkusze stylów, 179, 181
- artefakty, 560, 562

- asercja assertAlmostEqual(), 180
- asercje, 383
- Async, 602
- atak typu CSRF, 99
- atrybut
 - .return_value, 478, 485
 - action=, 437
 - class, 370, 378
 - errors, 328
 - FunctionalTest.test_server, 511
 - id, 336
 - instance, 354
 - mail.outbox, 435, 447, 452
 - method="POST", 95–97, 195
 - mock.return_value, 482, 494
 - name=, 95, 96, 334
 - placeholder, 324
 - widgets, 332
- automatyczne wdrażanie, 264, 285

B

- backend, 95
 - mechanizm uwierzytelniania, 470
- baza danych, 109
 - konfiguracja ścieżki dostępu, 246
 - migracja, 110, 227
 - obiekt sesji, 497
 - ograniczenia, 306
 - połączenie z frontendem, 332
 - Postgres, 600
 - produkcyjna, 121
 - przechowywanie tokenów, 427
 - sprawdzanie ograniczeń, 304
 - sprawdzanie poprawności danych, 310
 - SQLite, 122, 600

- baza danych
 - testowa, 519, 521
 - testowanie, 94
 - usuwanie w środowisku testowym, 419
 - w obrazie kontenera, 230
 - zapis żądania POST, 113

BDD, Behaviour-Driven Development, 55

BDUF, Big Design Up Front, 136

bezpieczeństwo serwerów, 261

biblioteka

- mock, 452

- pytest, 483

- WhiteNoise, 235

błąd

- AssertionError, 133, 358, 483

- bazy danych, 112, 419

- IntegrityError, 304, 307, 322, 361, 362

- IntegrityErrors, 179

- ValidationError, 306, 362

- ValueError, 327

- z kodem stanu 200, 308

- z kodem stanu 302, 308

- z kodem stanu 400, 242

- z kodem stanu 404, 71, 187

- z kodem stanu 500, 249, 322

błędy spójności, 360

Bootstrap, 178, 181

- dokumentacja, 307

- dostosowanie za pomocą Sassa, 199

- integracja z frameworkiem, 184

- klasa jumbotron, 190

- komponenty, 190

- nadanie stylu tabeli, 191

- opcjonalny tryb ciemny, 191

- wiersze i kolumny, 185

C

CD, Continuous Deployment, 571

CDN, content delivery network, 236

chmura, 271

CI, continuous integration, 134, 546–571

ciągła integracja, CI, 134, 546–571

- buforowanie, 570

- najlepsze praktyki, 571

- pełna kompilacja, 568

- początkowa konfiguracja potoku, 550

- próba odtworzenia błędu, 554

- w przepływach pracy, 547

- wybór usługi, 547

- wykonanie testów JavaScriptu, 563

- zdefiniowanie obrazu dockera, 570

ciągłe wdrażanie, CD, 571

- zautomatyzowane wdrożenia, 571

Colima, 214

CRUD, create, read, update and delete, 332

CSRF, cross-site request forgery, 99

CSS, 179, 181

- uszkodzone arkusze, 235

- warunkowe dostosowywanie klas, 370

cykl test jednostkowy – tworzenie kodu, 67, 75,

109, 161, 172

D

debugowanie, 227

- Dockera, 268

- dostępności serwera WWW, 221

- opcja ALLOWED_HOSTS, 242

- po stronie serwera, 506

- polecenie collectstatic, 244

- połączenia SSH, 260

- problemów z siecią, 232, 259

- problemów z SSH, 259

- problemów z uwierzytelnieniem, 259

- problemów związanych z kontenerem, 221

- serwera testowego, 279

- testów funkcjonalnych, 97

- wyświetlanie informacji, 403

- z użyciem kodu w SQL-u, 513

- z użyciem time.sleep(), 98

- za pomocą narzędzi programistycznych, 158

dekorator, 504

- @property, 544

- @wait, 504

- mock.patch(), 454, 469

- wait, 501, 522

- wyraźnego oczekiwania, 496

DevTools, 158

Django, 28, 39, 233

- adresy URL, 64

- dziedziczenie szablonu, 183

- funkcje widoku, 64

- informacje o błędzie CSRF, 99

- instalacja, 33

- instalacja w środowisku wirtualnym, 218

- konfiguracja, 41

- ModelForm, 326

- ograniczenia modelu, 304
 - ORM, 108
 - plik cookie, 498
 - pliki statyczne, 188
 - serwer poczty, 426
 - sesje, 497
 - skomplikowane formularze, 356
 - sprawdzanie poprawności modeli, 306
 - sprawdzenie poprawności przy zapisie, 306
 - testowanie weryfikacji modelu, 305
 - testy jednostkowe, 63
 - tworzenie sesji, 515
 - uruchomienie serwera, 75
 - ustawienie LOGGING, 507
 - uwierzytelnienie, 427, 472
 - uwierzytelnianie niestandardowe, 430
 - użycie klienta testowego, 73
 - widok
 - obsługa żądań POST, 312
 - wyświetlanie formularza, 312
 - witryna administracyjna, 601
 - wysyłanie wiadomości e-mail, 424
 - wzorzec MVC, 64
 - DNS, 263
 - konfiguracja, 256
 - Docker, 207–232, 207
 - dzienniki zdarzeń, 274
 - Hub, 271
 - instalowanie, 213
 - mapowanie portów, 223
 - niebezpieczeństwa wdrażania, 204
 - pominięcie sudo, 269
 - sprawdzanie dzienników zdarzeń, 507
 - testowanie instalacji, 215
 - tworzenie obrazu, 215
 - uruchamianie kontenera, 215
 - uruchamianie na platformie Mac, 214
 - wychwytywanie błędów, 506
 - zatrzymanie kontenera, 220
 - dodanie
 - adresu URL, 154
 - elementów listy, 154
 - elementu do listy, 138, 170
 - komunikatów do strony, 458
 - DOM, document object model, 195
 - domena
 - konfiguracja, 256
 - rejestracja nazwy, 255, 263
 - dostosowanie modeli, 160
 - DRY, don't repeat yourself, 105, 106
 - dziedziczenie, 532
 - szablonu, 183
 - dzienniki
 - debugowania
 - dla Firefoksa, 555
 - dla Selenium, 555
 - dla Webdrivera, 555
 - zdarzeń
 - kontenera Dockera, 267, 268, 274, 507
- ## F
- Firefox, 28, 557
 - aktualizacja przeglądarki, 44
 - formularz, 95, 97
 - atrybut action, 314
 - kontrolki, 307
 - metoda save(), 367, 381
 - nowy adres URL, 157
 - obsługa unikalności elementów, 364
 - porządkowanie, 380
 - przekazywanie błędów do szablonu, 346
 - sprawdzanie poprawności danych, 323
 - użycie metody save(), 353
 - użycie w widoku, 329, 338, 341
 - w widoku listy, 346, 366
 - wyświetlanie błędów w szablonie, 341
 - zmiana z ModelForm na zwykłe, 381
 - framework
 - Bootstrap, 178, 181
 - Django, 28, 455
 - Jasmine, 394
 - pytest, 394
 - React, 602
 - Vue.js, 602
 - frameworki frontendu, 415
 - frontend, 95, 332
 - funkcja
 - .is_authenticated(), 543
 - addCleanup(), 572, 573
 - afterEach(), 400
 - any(), 81, 90, 104
 - application(), 234
 - as_p(), 350
 - assertIn(), 104
 - assertTemplateUsed(), 85, 88
 - asserttrue(), 90

funkcja

- `authenticate()`, 431, 482
- `beforeEach()`, 400
- `fake_send_mail()`, 450
- `find_element()`, 80, 81
- `find_elements()`, 80, 81
- `fn()`, 298, 503
- `get()`, 467
- `get_absolute_url()`, 320
- `handle()`, 515
- `home_page()`, 100
- `initialize()`, 405–412
- `is_valid()`, 328
- pomocnicza `html.escape()`, 310
- pomocnicza `patch()`, 453
- `redirect()`, 320
- `render()`, 83
- `save()`, 109
- `self.wait_for()`, 502
- `send_keys()`, 80
- `send_mail()`, 426, 449–451, 454
- `wait_for()`, 499

funkcje

- anonimowe, 391
- `lambda`, 297
- strzałki, 391
- widoku, 65, 145

G

generowanie elementów w szablonie, 119

Git, 29

- oznaczenie wydania tagiem, 284, 420
- przekazanie plików do repozytorium, 69, 176, 184, 193, 238, 245, 283, 319, 329, 352, 355, 362, 374, 443, 500, 536, 543
- przekazywanie modelu Git, 112
- sprawdzanie postępu pracy, 605
- umieszczenie kodu w repozytorium, 58
- utworzenie repozytorium, 47

GitHub, 93

GitLab, 548

grupa przechwytywania, 167

Gunicorn, 234

- instalacja w kontenerze, 234

H

hosting witryny, 253

HTML, 28, 193

I

IDE, 33

- PyCharm, 33

- Visual Studio Code, 33

identyfikator

- klienta, 498

- sesji, 497

- uniwersalnie unikalny, UUID, 445

- użytkownika, UID, 247, 431

imitacje, 447, 449, 457, 468, 494, 587

- izolowanie testów, 470, 494

- oddzielne testowanie elementów, 476

- ręczne, 449

- testowanie implementacji, 478

- unikanie metod magicznych, 482

- weryfikowanie szczegółów implementacji, 494

- właściwość `call_args`, 480

implementacja układu graficznego, 199

informacje o komunikatach błędów, 225

infrastruktura jako kod, IaC, 264

instalacja

- Ansible, 261

- Django, 33, 218

- Dockera, 213

- Jasmine, 564

- Node.js, 564

- Selenium, 33

- WhiteNoise, 236

interfejs użytkownika logowania, 423

J

Jasmine, 394

- działanie, 397

- instalowanie i konfigurowanie, 564

- silnik testów, 398

JavaScript, 28, 387–415

- kolejność wykonywania kodu, 404

- osadzanie skryptu, 391

- test jednostkowy funkcjonalności, 401

JSON, 500

K

kacze typowanie, duck typing, 544

kaskadowe arkusze stylów, CSS, 179, 181

katalog roboczy, 47

katalogi statyczne, 197

klasa

- AnonymousUser, 543
- FunctionalTest, 292
- HttpRequest, 65
- ItemValidationTest, 388
- jumbotron, 190
- Keys, 80
- LiveServerTestCase, 57
- Meta, 328, 359
- Model, 110
- ModelForm, 326, 327, 359, 374
- ModelForms, 332
- StaticLiveServerTestCase, 189
- TestCase, 121

klasy formularzy, 374

klient testowy Django, 73

klucz

- podstawowy, 111
- SECRET_KEY, 240
- sessionid, 498
- tajny użytkownika, 275, 426
- zewnętrzny, 162

kod eksperymentalny, 422, 446

komentarze, 54

kompilacja w GitLabie, 551

kompozycja, 532

komunikacja

- z bazą danych, 519
- z serwerem, 257, 258, 261

komunikat błędu, 34, 97, 173, 196, 212, 224, 225, 235, 250, 328, 373, 445, 539

- AssertionError, 46
- ModuleNotFoundError
- No module named, 46
- o niepowodzeniu testu, 181
- poprawianie, 104
- textInput is null, 402
- ukrywanie, 388
- wyświetlanie, 307

konfiguracja

- deklaratywna, 267
- Django, 41
- DNS, 256
- domeny, 256
- gotowości produkcyjnej, 252
- idempotentna, 267
- mechanizmu rejestrowania danych, 249
- projektu, 597

- serwera poczty, 426
- środowiska wirtualnego, 32
- testu, 496, 504, 522
- testu w formacie JSON, 500, 505

konstrukcja try-except, 309, 347

kontener, 207, 209

- montowanie plików, 230
- plik bazy danych, 230

kontroler, 529

L

liczby rzymskie, 47

Linux, 30, 254

lista składana, list comprehension, 81

listy

- własne adresy URL, 165

logowanie, 470

M

macOS, 29

magiczne

- łącza, 422
- metody assert_called, 482

mail.outbox, 509

małe kroki

- nowy adres URL, 143, 154
- szablon do wyświetlania list, 150

mapowanie

- adresów URL, 72
- obiektowo-relacyjne, ORM, 108
- portów Dockera, 223

maszyna wirtualna, virtual machine, 208

menedżer kontekstu self.assertRaises(), 303

metoda pomocnicza

- self.wait_for, 298, 299
- wait_for, 297
- wait_for_row_in_list_table, 299

metody

- testowe, 383
- zwinne, 136

migracja bazy danych, 110, 227

- błąd IntegrityError, 361
- z poziomu pliku Dockerfile, 229

model

- DOM, 399, 401
- MVC, model – widok – kontroler, 64, 541
- niestandardowy użytkownika, 438
- tokena, 443

moduł

- accounts, 425
 - ansible.builtin.file, 282
 - contrib.auth, 482
 - docker.container_exec, 282
 - subprocess, 518
 - unittest, 52, 55, 394
 - unittest.mock, 468
- monkeypatching, 449, 468
- montowanie
- bazy danych, 281
 - plików, 232, 247
- myślenie życzeniowe, 538, 545

N

narzędzia

- programistyczne, 159
- programistyczne przeglądarki internetowej, 392
- zarządzania, 520

narzędzie

- Anaconda, 35
 - Ansible, 257
 - bower, 199
 - coverage.py, 602
 - curl, 221, 279, 280
 - Forgejo, 569
 - nerdctl, 213
 - npm, 199
 - pip, 29
 - Podman, 213
 - pytest, 58
 - virtualenv, 29
 - Woodpecker, 569
- nieoczekiwane niepowodzenie, 126
- nieszczelne abstrakcje, leaky abstractions, 24
- niewielkie widoki, 355
- Node.js
- instalowanie, 564

O

- obrazy kontenerów, 271
- oczekiwane niepowodzenie, 103, 149
- oczekiwanie
- jawne, 131, 132
 - niejawne, 131
- ograniczenia bazy danych, 306

- ograniczenie NOT NULL, 304
- operacja znajdź i zastąp, 332
- operacje weryfikacji modelu, 306
- operator `:=`, 211
- ORM, object-relational mapper, 108
- osadzanie skryptu, 391

P

- PaaS, platform as a service, 254, 263
- pakiet pytest, 602
- piramida testów, 586, 599
- plik
- .dockeignore, 231
 - .env, 490
 - .gitignore, 48, 49, 231, 245
 - gitlab-ci.yml, 550, 553, 558, 560, 567
 - .tar, 272
 - /etc/passwd, 247
 - container.db.sqlite3, 514
 - db.sqlite3, 122, 245, 281
 - Dockerfile, 216, 218, 226, 232, 234, 236, 239, 245, 246
 - functional_tests.py, 43, 46, 53, 56, 79, 90, 98, 106, 107
 - functional_tests/tests, 210
 - functional_tests/tests.py, 129, 132, 134, 140, 147, 153, 179, 180, 189
 - home.html, 97
 - infra/debug-ci.py, 554
 - infra/deploy-playbook.yml, 262, 265, 269, 271, 273, 275, 280, 282, 512
 - infra/Dockerfile.ci, 554
 - list/templates/home.html, 152
 - lists/models.py, 110, 111, 162, 163
 - lists/templates/base.html, 184, 185, 190, 191, 488
 - lists/templates/home.html, 89, 96, 102, 121, 124, 149, 159, 180, 184
 - lists/templates/list.html, 151, 173, 174, 184
 - lists/tests.py, 63, 70, 73, 86, 96, 100, 103, 113–116, 119, 142, 148, 155, 158, 160, 164, 166, 170, 173, 174, 195, 196
 - lists/urls.py, 176
 - lists/views.py, 66, 67, 83, 101, 103, 116–118, 121, 143–146, 152, 156, 157, 164, 167, 168, 172
 - manage.py, 45, 47
 - models.py, 111

- requirements.txt, 237, 238
- settings.py, 45, 233, 240
- SpecRunner.html, 397
- src/accounts/authentication.py, 431, 464–467
- src/accounts/models.py, 427–430, 440–445
- src/accounts/tests/test_authentication.py, 463, 466
- src/accounts/tests/test_models.py, 439–442, 444
- src/accounts/tests/test_views.py, 448, 450, 453–457, 460, 472, 473, 482, 485, 486
- src/accounts/urls.py, 425, 432, 493
- src/accounts/views.py, 425, 431, 448–452, 454, 458–462, 472, 473, 477, 481, 484–486
- src/functional_tests/base.py, 292, 296, 336, 337, 499–503, 511, 527, 559
- src/functional_tests/container_commands.py, 517, 521
- src/functional_tests/list_page.py, 574, 575, 577
- src/functional_tests/management/command s/create_session.py, 515
- src/functional_tests/my_lists_page.py, 577
- src/functional_tests/test_layout_and_styling.py, 293, 528
- src/functional_tests/test_list_item_validation.py, 293, 295, 312, 316, 337, 351, 356, 388, 389
- src/functional_tests/test_login.py, 434, 492, 500, 511
- src/functional_tests/test_my_lists.py, 497, 500, 516, 525, 527
- src/functional_tests/test_sharing.py, 572, 575–577
- src/functional_tests/test_simple_list_creation.py, 293, 337
- src/functional_tests/tests.py, 289–291
- src/lists/forms.py, 324–329, 354, 365–368, 370, 374, 377, 381
- src/lists/models.py, 321, 358, 359, 373, 542, 543
- src/lists/static/lists.js, 405, 409–412
- src/lists/static/tests/Spec.js, 396–401, 407, 408, 412
- src/lists/static/tests/SpecRunner.html, 411
- src/lists/templates/base.html, 307, 331, 335, 337, 339, 342, 345, 378, 391, 410, 412, 424, 455, 458, 492, 528, 529, 533
- src/lists/templates/home.html, 535
- src/lists/templates/includes/form.html, 534
- src/lists/templates/includes/scripts.html, 535
- src/lists/templates/list.html, 315, 535
- src/lists/tests/test_forms.py, 324, 327–329, 353, 364, 367, 380
- src/lists/tests/test_models.py, 303, 320, 358–361, 372, 373, 541
- src/lists/tests/test_views.py, 308, 311, 313, 317, 334, 343–345, 362, 364, 366, 376, 379, 383, 437, 438, 540
- src/lists/views.py, 308–310, 315, 317, 320, 321, 330, 333, 338, 340, 341, 346–348, 354, 367, 540, 543
- src/superlists/settings.py, 236, 250, 426, 427, 440, 441, 470, 490, 507
- src/superlists/urls.py, 425
- superlists/settings.py, 84, 122, 197
- superlists/urls.py, 72, 143, 156, 166, 171, 175
- superlists/wsgi.py, 234
- test_models.py, 300
- test_views.py, 300

pliki

- .pyc, 49
- blokad, lockfiles, 238, 239
- ciasteczek, cookies, 240, 497
- CSS, 198
- debugowania, 560
- statyczne, 188, 235, 252
- udostępnianie, 235
- w kontenerze, 230

podejście

- inside-out, 524
- outside-in, 524, 525, 530, 538, 543, 599
- outside-in w TDD, 545

polecenie

- ./src/manage.py runserver, 387
- ansible-playbook, 262, 266–269, 281, 418
- assert, 55
- build && run, 221
- build, 219
- check --deploy, 252
- check, 251
- collectstatic, 197, 244
- create_session, 518
- curl -iv localhost, 280
- curl -iv, 224
- curl, 232
- dbshell, 304

- polecenie
 - diff, 148, 160
 - django-admin.py startproject superlists ., 45
 - docker build && docker run, 232, 234
 - docker build, 217, 232, 273
 - docker exec env, 278
 - docker exec -it container-id-or-name bash, 222
 - docker exec, 222, 223, 232, 282, 517
 - docker image inspect superlists, 268
 - docker inspect superlists, 268
 - docker inspect, 278, 279
 - docker logs, 277, 281
 - docker ps, 220, 268, 277, 280
 - docker run -e, 557
 - docker run, 218, 232, 271
 - docker stop, 220
 - export, 213
 - git add, 59
 - git commit -am, 69
 - git commit, 50
 - git diff -w, 58
 - git diff, 69, 125
 - git init, 48
 - git push gitlab, 549
 - git status, 50, 58, 69, 125
 - git switch -c passwordless-spike, 423
 - grep -r id_new_item, 337
 - include, 175, 535
 - makemigrations, 110, 428, 430
 - manage.py migrate, 179
 - manage.py runserver, 350
 - manage.py test functional_test, 294
 - manage.py test, 130
 - migrate, 121, 124
 - nslookup, 259
 - ping, 259
 - pip freeze, 237
 - pip install -r requirements.txt, 238
 - pip install whitenoise, 236
 - pip install, 218, 237
 - push -f, 420
 - python functional_tests.py, 69, 75, 79, 85
 - python manage.py check --deploy, 252
 - python manage.py makemigrations, 162
 - python manage.py migrate, 123
 - python manage.py runserver, 45, 75
 - python manage.py startapp lists, 61
 - python manage.py test functional_tests, 129, 141
 - python manage.py test lists, 130, 162, 163
 - python manage.py test, 63, 65, 73, 75, 83
 - python src/manage.py makemigrations, 361, 374, 442
 - python src/manage.py migrate, 361
 - python src/manage.py runserver, 489, 490
 - python src/manage.py shell, 498
 - python src/manage.py test accounts, 441, 442, 449–452, 457, 458, 461, 465
 - return, 312
 - rm db.sqlite3, 124
 - run, 219
 - SELECT *, 513
 - slurp, 276
 - sudo chmod g+rw container.db.sqlite3, 247
 - sudo, 269, 270
 - with, 304
- porty, 212
- Postgres, 600
- PostgreSQL, 230
- prezentacja, 528
- proces TDD, 91, 139, 598
- programistyczny serwer Django, 233
- programowanie
 - „od dołu do góry”, 438
 - funkcyjne, 298
 - sterowane testami, TDD, 39
 - z wykorzystaniem myślenia życzeniowego, 538
- projektowanie API, 536
- protokół
 - HTTP, 497
 - SMTP, 426
 - SSH, 257
 - WebSocket, 602
- prototypowanie typu spike, 390
- przechwytywanie parametrów z adresów URL, 166
- przeglądarka internetowa
 - cechy, 415
 - Firefox, 28
 - konsola debugowania, 403
 - narzędzia programistyczne, 392
 - silnik testów Jasmine, 398
 - włączenie trybu headless, 557
 - wykonywanie testów, 397
 - zmienna globalna document, 399

- przekazanie
 - listy do szablonu, 174
 - zmiennych, 102
- przekierowanie po żądaniu POST, 116
- przetwarzanie żądania POST, 100
- puste elementy, 289
- PyCharm, 33
- Python 3, 27

R

- refaktoryzacja, 82, 88, 106, 126, 150, 171, 291, 301
 - funkcjonalności `add_item`, 316
 - funkcjonalności `new_item`, 313
 - funkcjonalności `view_list`, 313
 - pod kątem testów zielonych, 533
 - przenoszenie funkcjonalności, 376
 - testów jednostkowych, 299
 - testu funkcjonalnego, 337
 - usunięcie adresów URL, 319
 - widoku, 347
 - za pomocą polecenia `include`, 175
- regresja, 126, 146, 169
- reguła DRY, 105, 106
- rejestrwanie danych, 248, 252, 523
- REST, representational state transfer, 138
- rozdzielanie funkcjonalności, 144

S

- selektor
 - `#id_text`, 407
 - `.invalid-feedback`, 407
- selektory
 - CSS, 194, 195, 391
 - usuwanie, 407
- Selenium, 599
 - instalacja, 33
 - jako pętla zewnętrzna, 415
 - niestabilne testy, 558
 - testowanie interakcji użytkownika, 79
- serwer
 - bazy danych, 230
 - ciągłej integracji, 570, 571, 595
 - Django, 233
 - Gunicorn, 234
 - montowanie bazy danych, 281
 - poczty, 426
 - produkcyjny, 522

- programistyczny Django, 45
- przetwarzanie żądania POST, 100
- uruchomienie, 254
- sesje, 498
 - Django, 497
 - tworzenie, 518
- składnia
 - `{{ ... }}`, 102, 269
 - przypisania `[form] =`, 195
- słowo kluczowe
 - `assert`, 55, 483
 - `if`, 101
 - `let`, 399
 - `with`, 304
- SMTP, Simple Mail Transfer Protocol, 426
- sprawdzanie
 - duplikatów, 357, 362
 - dzienników zdarzeń, 507
 - modelu, 306
 - menedżer kontekstu, 303
 - ograniczenia bazy danych, 304
 - ograniczenia Django, 304
 - testowanie weryfikacji modelu, 305
 - wyświetlanie błędów w widoku, 307
 - testu regresji, 140
- sprawdzenie
 - łącza z tokenem, 460
 - modelu, 303
 - ustawienia LOGGING, 507
- SQLite, 600
- SSH, 257, 263, 517
 - połączenie z serwerem, 267
- stan wyścigu, 147
- stosowanie imitacji, 447
- struktura
 - katalogów, 396
 - katalogów aplikacji, 182
- styl tabeli, 191
- style, 178
- system kontroli wersji, VCS
 - Git, 29, 47
- szablon, 82, 83, 96, 528
 - dostęp do właściciela, 543
 - dziedziczenie, 183
 - projektowanie API, 536
 - szybka restrukturyzacja, 531
 - testowanie, 102
 - wyodrębnienie tagów `include`, 533
 - wyświetlanie błędów, 341

Ś

ścieżka dostępu do pliku bazy danych, 246
środowisko
 produkcyjne, 240, 283
 testowe dla JavaScriptu, 395
 uruchomieniowe Colima, 214
 wirtualne Pythona, 32, 210
 aktywowanie, 32
 dezaktywowanie, 32

T

tabele baz danych, 111
TDD, test-driven development, 39, 77
 cykl test jednostkowy – tworzenie kodu, 91
 implementacja nowego projektu, 139
 informacje od użytkownika, 59
 kot refaktoryzacji, 177
 niebezpieczeństwa wdrażania, 204
 nieoczekiwane niepowodzenie, 126
 oczekiwane niepowodzenie, 59
 podejście outside-in, 524, 545
 podwójna pętla, 92
 proces TDD, 92
 refaktoryzacja, 91, 126
 regresja, 126
 technika małych kroków, 177
 techniki, 597
 Testing Goat, 177
 testy funkcjonalne, 91, 93
 testy jednostkowe, 91, 93
 triangulacja, 126
 YAGNI, 177
test
 akceptacji, *Patrz* testy funkcjonalne
 czarnej skrzynki, 53
 dymny, 87, 396
 E2E, 53, *Patrz* testy funkcjonalne
 integracji, 109
Testing Goat, 41, 365, 594
testowa baza danych, 519, 521
testowanie, 594
 alternatywna strategia, 114
 backendu, 95
 backendu uwierzytelniania, 474
 bazy danych, 94
 działania formularzy, 323
 formularza, 327
 frameworka komunikatów Django, 455
 frontendu, 95
 implementacji, 478
 instalacji Dockera, 215
 integracji z Bootstrapem, 411
 integracji z CSS, 411
 interakcji użytkownika, 79
 izolacja testów, 470
 JavaScriptu w cyklu TDD, 414
 jednej rzeczy, 118, 355
 konfiguracja testu, 496
 kontekstu szablonu, 172
 kontenera Dockera, 513
 metody testowe i asercje, 383
 najlepsze praktyki, 135, 598
 narzędzi zarządzania, 520
 oczekiwane niepowodzenie, 55, 103
 piramida testów, 586
 poczty e-mail, 510
 pomyślne wykonanie zestawu testów, 563
 regresji, 140
 sposobu działania, 86
 strony głównej, 60
 szybkość wykonania, 584
 tworzenie lepszego kodu, 583
 układu graficznego, 199
 układu i stylu, 178
 ustalanie liczby testów, 476
 usuwanie testów, 487
 w JavaScriptcie, 415
 w środowisku produkcyjnym, 591
 warstwy modeli, 474
 warstwy widoków, 474
 widoku bez użycia imitacji, 472
 wydajny przepływ pracy, 583
 wykonanie testów JavaScriptu, 563
 wykorzystanie w pełni korzyści, 585
 wylogowania, 492
 wymagania, 582
 z zawartością modelu DOM, 399
 za pomocą przeglądarki internetowej, 397
 zaawansowane, 385
 zależności zewnętrznych, 447
 zapobieganie regresji, 583
 zautomatyzowane, 583

testy

- akceptacyjne, 587
- dla kontenera Dockera, 211
- funkcjonalne, 62, 581, 586, 587
 - adresu e-mail użytkownika, 488
 - debugowanie, 97
 - dla kodu eksperymentalnego, 434, 446
 - dla strony, 525
 - dotyczące powielonych elementów, 356
 - działanie, 43
 - gwarancja izolacji testu, 127, 135
 - klas Bootstrapa, 368
 - metody pomocnicze, 299
 - moduł unittest, 52, 55
 - najlepsze praktyki, 599
 - obsługa żądań POST, 315
 - ogólna metoda pomocnicza oczekiwania, 295
 - podział na pliki, 291
 - problem z plikami statycznymi, 235
 - przygotowanie minimalnej aplikacji, 52
 - sprawdzanie wdrożenia, 279
 - sprawdzenie działania kontenera, 220
 - unikanie oczekiwań niejawnych, 135
 - uruchamianie narzędzia zarządzania, 516
 - usprawnienie, 127
 - użytkownika i strony, 576
 - weryfikacji danych, 289
 - wykonanie, 75, 131
 - wykrywanie regresji, 169, 146
 - wylogowania, 492
 - wywołanie time.sleep, 131
 - z wieloma użytkownikami, 572
- integracyjne, 583, 586–588
- jako dokumentacja, 442
- JavaScriptu, 567, 568
- jednostkowe, 62, 581, 583, 586, 587
 - a baza danych, 109
 - a testy funkcjonalne, 61
 - a testy integracji, 109
 - analiza API formularzy, 324
 - dla kodu w JavaScriptcie, 393, 413
 - funkcjonalności JavaScriptu, 401
 - modyfikacja, 85
 - na warstwie modeli, 303
 - prawdziwe, 584
 - refaktoryzacja, 299
 - reguła „nie testuj stałych”, 82

- reguła DRY, 105
- Selenium, 413
- w Django, 63
- widoku strony głównej, 63, 65
- wykonanie, 75, 131

token

- serwisów społecznościowych, 572
- w bazie danych, 427
- w wiadomości e-mail, 462

triangulacja, 105, 126

tryb DEBUG, 240

tworzenie

- katalogu, 45
- klasy, 108
- kodu aplikacji, 66
- kodu eksperymentalnego, 421, 446
- listy, 155
- nowej gałęzi dla spike, 423
- obrazu Dockera, 215
- produkcyjnej bazy danych, 121
- repozytorium Git, 47
- sesji, 515, 518
- większej liczby testów, 406

U

UID, universally unique identifier, 427

układ, 178

umieszczanie obrazu na serwerze, 271

uprawnienia, 247

uruchamianie

- kodu „wewnątrz” kontenera, 222
- kontenera Dockera, 215, 219
- testów, 418

usługa ciągłej integracji, 547

usuwanie

- adresów URL, 319
- bazy danych, 419
- kodu, 157
- kodu eksperymentalnego, 421, 436, 446
- selektorów, 407

UUID, universally unique ID, 445

uwierzytelnianie, 259, 470

- eksperymentalna wersja kodu, 463
- niestandardowe, 428, 430
- użytkownika, 421, 427
- za pomocą „magicznych łączy”, 422

V

VCS, version control system, 47
Visual Studio Code, 33
VPS, virtual private server, 254, 263

W

warstwa

- modelu, 541
- widoku, 539
- zewnętrzna, 528

wczesne zakończenie, 533

wdrożenie, 233

- błąd bazy danych, 419
- do środowiska produkcyjnego, 283
- lista kontrolna, 416
- lokalny zestaw testów, 417
- niebezpieczeństwa, 202
- nowego kodu, 416
- podsumowanie, 420
- procedura, 203
- przygotowanie serwera, 253
- w środowisku produkcyjnym, 419
- w środowisku testowym, 418
- zautomatyzowane, 264
- zestaw testów w Dockerze, 417

WebSocket, 602

weryfikacja, 287

danych

- na poziomie formularzy, 303
- na poziomie modelu, 303
- po stronie klienta, 303
- po stronie serwera, 303

modelu

- w widoku `view_list`, 317
- na poziomie bazy danych, 302, 322
- za pomocą HTML5, 350

WhiteNoise, 235

wiadomości e-mail, 424, 428

widok, 529

- logowania, 470
- nowej listy, 156
- obsługa żądań POST, 312, 341
- przypisywanie list właścicielowi, 539
- sprawdzanie duplikatów, 362
- `view_list`
 - weryfikacja modelu, 317

wymagania szablonu, 539

wyświetlanie formularza, 312, 341

widżet, 325

Windows, 30

wirtualizacja, 207, 209

WSGI, Web Server Gateway Interface, 234

WSL, Subsystem for Linux, 258

wstępne utworzenie sesji, 496

wybór

- frameworka testów, 415
- narzędzi testowych, 394

wyjątek typu

- `NoSuchElementException`, 132
- `StaleElementException`, 132
- `WebDriverException`, 133

wykonanie pojedynczego pliku testu, 294

wysłanie wiadomości e-mail, 424 428

wyświetlanie

- błędów w szablonie, 341
- informacji debugowania, 403
- komunikatu błędu, 307
- listy, 138

wywołanie

- `.full_clean()`, 348
- `assertContains()`, 186
- `assertIn()`, 437
- `checkVisibility()`, 413
- `console.log()`, 403
- `form.is_valid()`, 328
- `form.save()`, 353
- `include()`, 175
- `is_valid()`, 364
- `ItemForm()`, 367
- `List.objects.first()`, 170
- `lookup()`, 512
- `lookup('password')`, 276
- `redirect()`, 172
- `self.fail()`, 560
- `time.sleep()`, 98, 100
- `time.sleep`, 125, 131, 133
- `wait_for()`, 388

wzorzec

- projektowy MVC, 64
- strony, 574

Y

YAGNI, you aint gonna need it, 137

Z

zależności

- programistyczne, 238
- przechodnie, 238

zapach kodu, 113, 118

zarządzanie

- konfiguracją testów, 514
- zawartością bazy danych, 519

zatrzymanie kontenera Dockera, 220

zdarzenie onload, 413

zmiana

- atrybutu id, 336
- atrybutu name, 334
- wyglądu aplikacji, 178

zmienna

- globalna document, 399
- środowiskowa
 - DISPLAY, 557
 - DJANGO_DEBUG_FALSE, 241
 - DJANGO_SECRET_KEY, 275
 - DJANGO_SETTINGS_MODULE, 515
 - DOCKER_HOST, 272
 - EMAIL_PASSWORD, 427, 490, 508, 512
 - MOZ_HEADLESS, 557, 565
 - SECRET_KEY, 275, 512, 519
 - TEST_SERVER, 210, 213, 290

zmienne środowiskowe

- definiowanie, 275
- definiowanie na serwerze, 512
- samodzielne sprawdzenie, 277
- ustawienia środowiska produkcyjnego, 240
- w pliku Dockerfile, 241
- zdefiniowane z poziomu powłoki, 242

znacznik

- <body>, 392
- <div>, 368, 369, 399
- <form>, 95, 96, 531
- <input type="hidden">, 100
- <input>, 95–97, 392
- <script>, 391, 392, 410, 531, 535
- <textarea>, 327
- szablonu
 - {% block table %}, 531
 - {% csrf_token %}, 100, 125
 - {% for ... endfor %}, 124, 125
 - {% for message in messages %}, 456
 - {% if %}, 342, 489
 - {% include ... with key=value... %}, 535
 - {% static %}, 199
 - {% url %}, 319
 - {% url ... as %}, 535
 - {{ error }}, 331
 - {{ forloop.counter }}, 123
 - {{ form }}, 331
 - {{ form.errors }}, 345
 - {{ form.errors.text }}, 342
 - {{ form.text }}, 330, 332, 378

zrzuty ekranu, 559

Ż

żądanie

- GET, 138, 330, 346
- POST, 95, 100, 116, 148, 159
- PUT, 138

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

Najpierw pisz testy — i wdrażaj pewnie.

TDD w Pythonie od A do Z

Programowanie sterowane testami (TDD) to jedna z najważniejszych kompetencji współczesnego programisty. W świecie, w którym jakość kodu, szybkość wdrożeń i niezawodność systemów decydują o sukcesie projektów, TDD przestało być niszową techniką — stało się standardem w profesjonalnych zespołach deweloperskich. Python, jako jeden z najpopularniejszych języków programowania na świecie, zyskuje coraz więcej zastosowań w tworzeniu aplikacji webowych, a Django pozostaje wiodącym frameworkiem w tym ekosystemie. Połączenie TDD z nowoczesnymi narzędziami, takimi jak Docker, Ansible czy Selenium, odpowiada na realne potrzeby rynku IT, na którym DevOps i automatyzacja wdrożeń są dziś absolutną koniecznością.

Trzecie wydanie tego uznanego przewodnika prowadzi czytelnika przez kompletny proces tworzenia aplikacji webowej w Pythonie 3.14 i Django 5 — od pierwszego testu aż po wdrożenie produkcyjne. Autor krok po kroku pokazuje, jak definiować testy przed napisaniem kodu, konteneryzować aplikację za pomocą Dockera, automatyzować wdrożenia przy użyciu Ansible, a także testować interakcje przeglądarkowe za pomocą Selenium. Omawia również obsługę plików statycznych, konfigurację środowiska produkcyjnego, mechanizmy uwierzytelniania bez hasła i izolację testów za pomocą obiektów imitacji. Całość utrzymana jest w duchu małych, ale pewnych kroków — zgodnie z filozofią TDD.

Testowanie ma niezwykle istotne znaczenie w pracy programisty. Harry spisał się doskonale — przyciągnął naszą uwagę, wyjaśniając praktyki stosowane podczas testowania.

Michael Foord, programista Pythona, autor modułu unittest

Ta książka to znacznie więcej niż wprowadzenie do programowania sterowanego testami w Pythonie. To pełny kurs przedstawiający najlepsze praktyki od początku do końca.

Kenneth Reitz, członek Python Software Foundation

W książce:

- Pełny przepływ pracy TDD: od pisania testów, przez implementację, po refaktoryzację
- Konteneryzacja aplikacji Django z użyciem Dockera i przygotowanie jej do wdrożenia produkcyjnego
- Automatyzacja wdrożeń na serwerze za pomocą narzędzia Ansible (infrastruktura jako kod)
- Testowanie interakcji przeglądarkowych przy użyciu Selenium i testy jednostkowe logiki biznesowej
- Izolacja zależności zewnętrznych i własnych modułów za pomocą obiektów imitacji (unittest.mock)
- Konfiguracja ciągłej integracji (CI) i automatyczne uruchamianie testów
- Tworzenie frontendu z kodem JavaScript pisanym zgodnie z podejściem TDD

Harry J.W. Percival jest zagorzałym zwolennikiem podejścia TDD i doświadczonym programistą Pythona. Swoją wiedzę dzieli się na całym świecie poprzez prelekcje i warsztaty. Pracuje dla Kraken Technologies, gdzie tworzy oprogramowanie wspierające globalną transformację energetyczną w kierunku odnawialnych źródeł energii. Jest autorem kolejnych wydań cenionego przewodnika po TDD w Pythonie.

**Helion**

**helion.pl**

**HELION S.A.**
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-289-3821-2



Cena: 139,00 zł