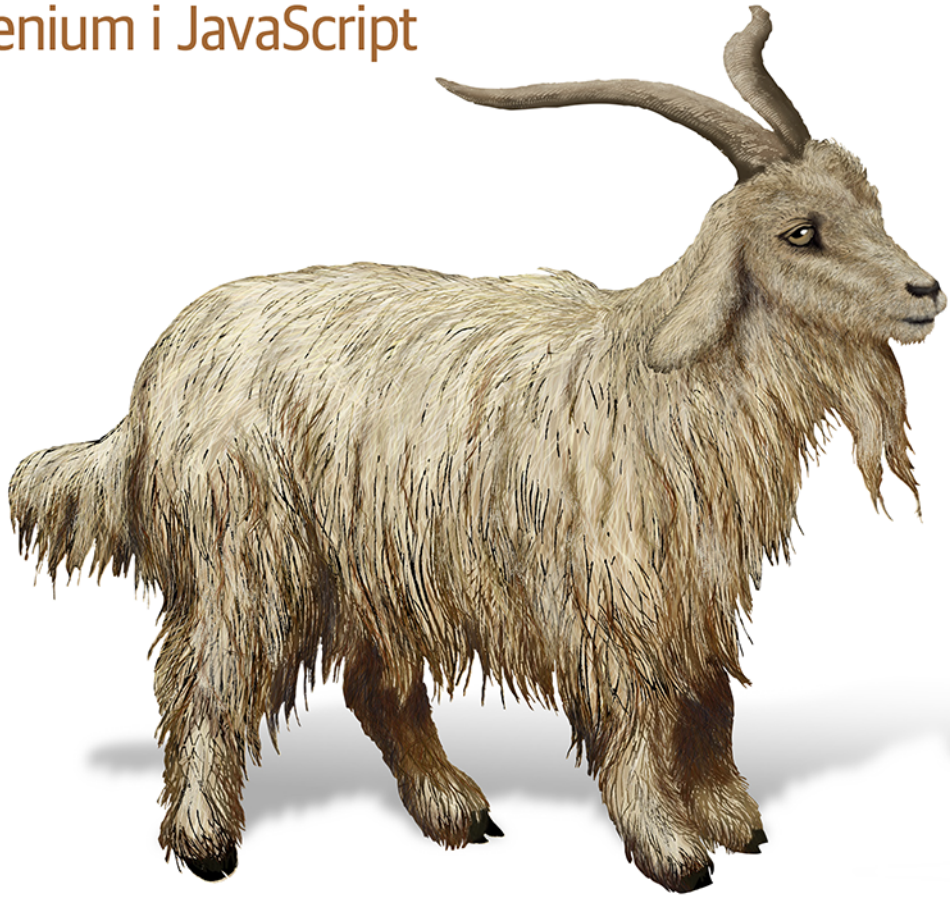
O'REILLY®

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/tddwp3
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.

Spis treści

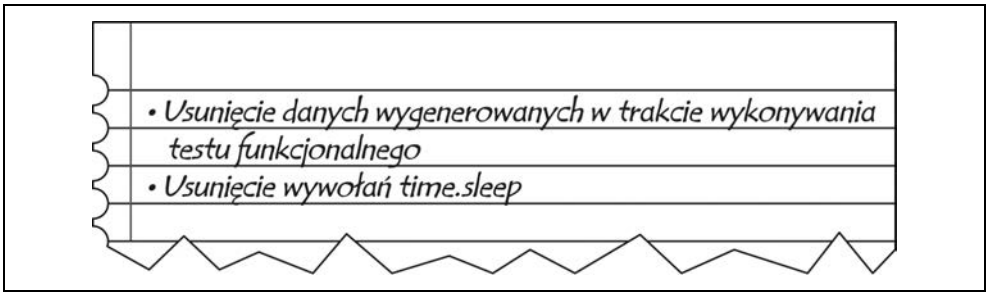
Część II. Wdrożenie w środowisku produkcyjnym

Część III. Formularze i weryfikacja

Część IV. Bardziej zaawansowane zagadnienia związane z testowaniem

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 modułem 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 wykonania testów funkcjonalnych będziemy używać silnika testów Django.

Na tym etapie jesteśmy gotowi do wykonania testów funkcjonalnych za pomocą silnika testów Django. Odbywa się 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 jakiekolwiek 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 unikać[1].

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 myląće wyniki w testach są prawdziwą udręką[2].

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  ❻
                time.sleep(0.5)  ❺
```

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*.

❺ 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.

❻ A to jest drugie wyjście. Po dotarciu do tego miejsca można przyjąć złożenie, 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 oczekiwania[3]. 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 Pythona[4], i dlatego należy preferować oczekiwania jawne.

[4] `python -c "import this"`

Skorowidz

C

D

F

G

H

I

J

K

L

M

N

O

P

R

S

Ś

T

U

V

W

Y

Z

Ż

Notatki

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 wiedzą 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
KOD KORZYŚCI
Sięgnij po więcej!
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
ISBN 978-83-289-3821-2
9 788328 938212
Cena: 139,00 zł