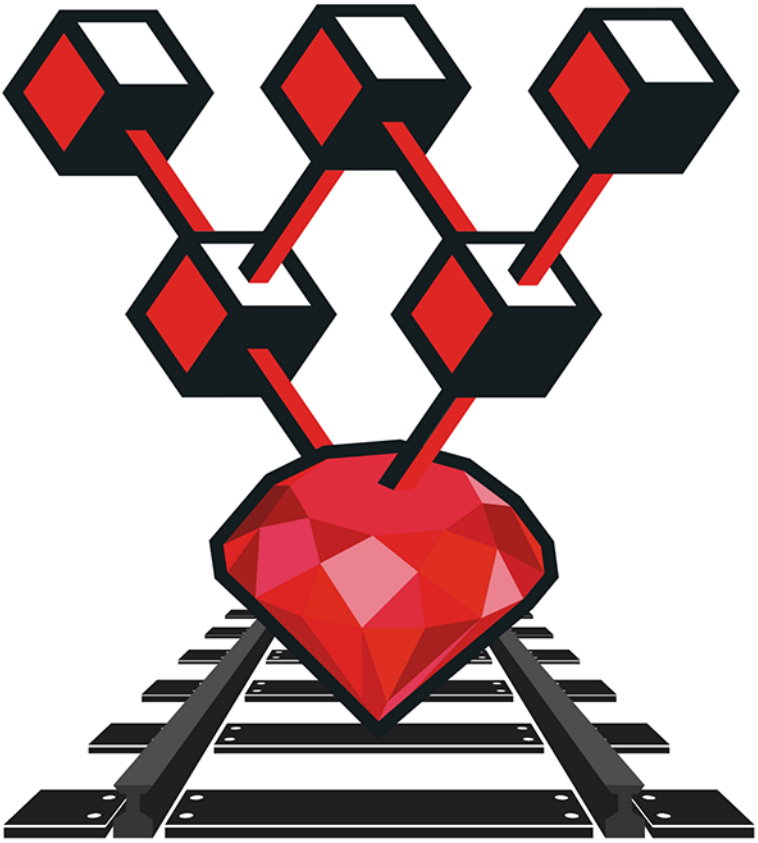
Michał Sobczak

PROGRAMOWANIE W JĘZYKU RUBY

Mikrousługi i konteneryzacja



Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

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

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

ISBN: 978-83-283-5241-4

Printed in Poland.

Spis treści

Rozdział 3.

Zapewnienie jakości

Quality is not an act, it is a habit.

Arystoteles

W tym rozdziale:

- Metodyki
- Wykrywanie defektów
- Testowanie Rack
- Testowanie Rails

Wprowadzenie

Nieodłącznym elementem tworzenia oprogramowania jest zapewnienie, aby funkcjonowało ono poprawnie i bez istotnych defektów. Pomimo szeregu narzędzi wspomagających unikanie i usuwanie błędów te nadal występują. Dzieje się tak, ponieważ różne są definicje błędu.

> *Najdokładniej planowanym i najobficiej finansowanym oprogramowaniem w historii były programy zbudowane dla obsługi lotów księżycowych Apollo. Przedsięwzięcie to obejmowało najlepszych programistów w kraju, zorganizowanych w dwa współzawodniczące zespoły. Przy sprawdzaniu oprogramowania wykorzystano cały istniejący w tej dziedzinie zasób umiejętności. W sumie oprogramowanie systemu Apollo kosztowało ok. 660 mln dol. Mimo to każde poważniejsze niebezpieczeństwo w trakcie lotów, od fałszywych alarmów do prawdziwych nieszczęść, było bezpośrednio spowodowane błędami w oprogramowaniu. [20]*

Teoria i praktyka

W tym rozdziale omawiam elementy teorii pozwalające zrozumieć, na czym problem polega i skąd się bierze. Właściwe rozpoznanie przyczyn pozwala efektywnie tworzyć oprogramowanie o mniejszej ilości defektów. Poza elementami teoretycznymi przedstawiam konkretne rozwiązania programistyczne na przykładzie testów automatycznych.

Metodyki

Podstawy teoretyczne

Definicja błędu

Wyróżniamy co najmniej kilka definicji błędu, zwanego też usterką lub defektem. Do katastrofy lądownika bezzałogowego na Wenus przyczynił się taki oto błąd w programie napisanym w języku FORTRAN:

```
DO 3 I = 1.3
```

Zamiast jako pętla zostało to zinterpretowane jako przypisanie zmiennej `D03I` wartości `1.3`. Zamiast przecinka użyta została kropka.

> *Błąd oprogramowania występuje wówczas, gdy oprogramowanie nie spełnia sensownych oczekiwań użytkownika. Awaria oprogramowania jest to wystąpienie błędu oprogramowania. [20]*

Specyfikacja

Powszechnie przyjęło się, że błąd jest to sytuacja, gdy oprogramowanie nie zachowuje się zgodnie ze specyfikacją. Oczywiście przy założeniu, że sama specyfikacja i jej zrozumienie przez każdego z zainteresowanych są poprawne.

Czynniki zewnętrzne

Innego rodzaju definicja wskazuje sytuację trudniejszą w ocenie, mianowicie gdy oprogramowanie działa zgodnie ze specyfikacją, ale wystąpiły szczególne okoliczności, które sprawiają, że nie potrafi ono dostarczyć oczekiwanych rezultatów. Przy założeniu, że program i jego moduły powinny być odporne na nieoczekiwane warunki zewnętrzne, taka sytuacja rzeczywiście ma charakter usterki.

Dokumentacja

Kolejnym elementem, który nosi znamiona błędu, jest niezgodna ze stanem rzeczywistym dokumentacja techniczna lub użytkownika. Sam program może być zgodny z pierwotnymi założeniami i specyfikacją. Błędem jest wtedy sama dokumentacja.

Oczekiwania

Ostatnim powszechnie stosowanym opisem dla błędu jest brak realizacji oczekiwań użytkowników. Występuje wtedy zapewne zgodność ze specyfikacją projektu, odporność na otoczenie, a dokumentacja odzwierciedla stan faktyczny funkcjonowania programu. Problemem są tutaj nieprecyzyjnie wyartykułowane wymagania użytkownika podczas zamawiania programu lub w momencie uzgadniania specyfikacji funkcjonalnej. Użytkownik końcowy lub zamawiający mogą stwierdzić, że program działa wadliwie, ponieważ mają co do niego inne oczekiwania. Proces wytwarzania zawodzi wówczas na całej linii.

Definicja niezawodności

Kolejny termin wart przeanalizowania to niezawodność. Kiedy już wiemy, jak zdefiniować błąd, choć co prawda na kilka różnych sposobów, chcemy dowiedzieć się, jak mierzyć odporność oprogramowania na usterki.

> *Niezawodność oprogramowania jest prawdopodobieństwem, że oprogramowanie działa bez awarii w określonym przedziale czasu, opatrzonym wagą odpowiadającą kosztom poniesionym przez użytkownika z powodu każdej zaistniałej awarii. [20]*

Przyczyny błędów

Proces budowy oprogramowania jest bardzo złożony przede wszystkim z racji występowania w nim czynnika ludzkiego, który jest zawodny. Możemy zatem uogólnić i przytoczyć następują definicję dotyczącą przyczyn występowania usterek:

> *Główną przyczyną błędów oprogramowania są błędy w tłumaczeniu informacji. [20]*

Badając temat, możemy stwierdzić, że występują dwa rodzaje modeli procesu tłumaczenia. Pierwszy, makroskopowy, zakłada, że projekt programistyczny jest to dziedzina interdyscyplinarna, wymagająca integracji szeregu czynników, w której problemem w głównej mierze jest komunikacja. Drugi zaś, mikroskopowy, opiera się na stwierdzeniu, że niemal każda informacja występująca w takim projekcie jest przetwarzana przez człowieka i to jego niedoskonałości powodują rozbieżności, braki i konflikty.

Model makroskopowy

Proces, czy też model, obejmuje swoim zasięgiem wyartykułowanie potrzeb przez zamawiającego, następnie opracowanie założeń i specyfikacji, powstanie kodu i dokumentacji, a także uzyskanie informacji zwrotnej od użytkowników.

(1) Najpierw przedstawiane są wymagania użytkownika, którego jednocześnie możemy nazwać zamawiającym lub sponsorem projektu programistycznego. Na tym etapie głównym problemem jest brak możliwości poprawnego i pełnego przedstawienia potrzeb lub ich błędna interpretacja.

(2) Kolejny etap to tłumaczenie wymagań użytkownika na język założeń. Jak nietrudno zgadnąć, błędy popełnione już w poprzednim kroku będą w sposób kumulacyjny dotyczyły kolejnych kroków. Tak też jest w tym przypadku, a dodatkowo sam proces materializacji wymagań w opisie założeń jest źródłem dodatkowych błędów.

(3) Etap następny to przekształcenie założeń na specyfikację wysokopoziomową. Jest to dokumentacja o dużym poziomie abstrakcji, nie zawierająca szczegółów implementacyjnych.

(4) Specyfikacja funkcjonalna o wysokim poziomie szczegółowości to kolejny etap w procesie. Informacje tutaj zawarte są istotne z punktu widzenia włączenia wymagań i założeń poprzez język specyfikacji ogólnej do tego, nad czym będzie pracował zespół produkcyjny.

(5) Kolejny etap to programowanie, czyli tłumaczenie specyfikacji na kod źródłowy. Jest to etap, w którym błędy są obecne i stanowią niezbywalną część procesu. Są też relatywnie proste w obsłudze.

Model mikroskopowy

Człowiek jako uczestnik procesu twórczego angażuje swoje zdolności poznawcze, przyjmując, zapamiętując, wyszukując i przekazując informacje. Istotą zrozumienia problematyki źródeł błędów jest fakt, że człowiek jest po prostu omylny, a swoje działania opiera na subiektywnych doświadczeniach, które nie zawsze pasują do każdej napotkanej sytuacji podczas procesu tworzenia oprogramowania. Rolą modeli i poszczególnych procesów dotyczących zapewnienia jakości jest wprowadzenie odpowiednich regulacji, tak aby czynnik ludzki dawał z siebie to co najlepsze w kreatywnym kontekście, jednocześnie nie będąc problematycznym w testowaniu, które jest ewidentnie procesem destruktywnym.

TDD

Test Driven Development polega na takiej organizacji pracy z kodem aplikacji, że w pierwszej kolejności staramy się stworzyć specyfikację funkcjonalną i idące za tym zestawy testowe. Kod testów biegnie przynajmniej równolegle, idealnie zaś jest, gdy wyprzedza właściwy kod aplikacyjny. Weźmy za przykład poniższy zestaw testowy (`Cucumber`), definiujący operacje związane z fakturowaniem w aplikacji księgowej:

```
Feature: Invoices
   Scenario: List invoices
      Given I am logged in as a user
      Given I am on "/invoices" page
      Then page should have notice "Faktury"

   Scenario: New invoice
      Given I am logged in as a user
      And There is a contractor called "PPHU"
      And There is a tax rate of "23" called "23% - naliczony" with account "264-2-1"
         at side "WN" for year of "2014"
      And There is a tax type called "Podlega opodatkowaniu" with code "PO"
      And There is a invoice template called "Testowy" at "Fakturowanie"
      And document type "Faktura zakupu" is linked with "Testowy" template
      And I am on "/invoices" page
      When I press "Nowa Faktura"
      Then page should have notice "Nowa Faktura"
      When I fill in "invoice[name]" with "Testowa faktura FV"
      And I select "invoice_document_type_id" with "Faktura zakupu"
      And I press "Zapisz"
      Then page should have notice "Utworzono nową fakturę"
      And page should have notice "Dokument wprowadzany do systemu"
      And page should have notice "Pozycje Faktury"
      When I fill in "invoice_invoice_entries_attributes_0_net" with "100"
      And I select "invoice_invoice_entries_attributes_0_tax_rate_id" with "23% - naliczony"
      And I select "invoice_invoice_entries_attributes_0_tax_type_id" with "Podlega opodatkowaniu"
      And I press "Zapisz"
      Then page should have notice "Faktura została zaktualizowana"
      When I click on "add_element_div"
      And I fill in "invoice_invoice_entries_attributes_1_net" with "200"
```

```
    And I select "invoice_invoice_entries_attributes_1_tax_rate_id" with "23% - naliczony"
    And I select "invoice_invoice_entries_attributes_1_tax_type_id" with "Podlega opodatkowaniu"
    And I press "Zapisz"
    Then page should have notice "Faktura została zaktualizowana"
    When I ensure the confirm box returns OK
    And I press "Zatwierdź"
    Then page should have notice "Pomyślnie zatwierdzono fakturę, utworzono dokument w księgowości"
    And page should have notice "Dokument w księgowości"
    And page should have notice "Zatwierdzony"
    And page should not have notice "Zapisz"
    And page should not have notice "Zatwierdź"

Scenario: Check link between invoicing module and main ledger
    Given I am logged in as a user
    And I ensure the confirm box returns OK
    And I am on "/invoices" page
    When I select "invoices_grid_document_state_id" with "Zatwierdzony"
    And I press "Szukaj"
    Given I ensure the confirm box returns OK
    When next to "Testowa faktura FV" I press "Edytuj"
    Then page should have notice "Dokument w księgowości"

Scenario: Create and move to main ledger
    Given there is invoice called "Testowa faktura FV"
    When I click link "Dokument w księgowości"
    Then page should have notice "Faktura w fakturowaniu"
    And page should have notice "Rejestr VAT (tylko do odczytu)"
    And page should have notice "Dokument wprowadzany do systemu"
```

Przykład składa się z weryfikacji listy dokumentów, wystawienia nowej faktury, przeniesienia jej do modułu księgowego oraz sprawdzenia poprawności tej operacji. Należy samodzielnie ocenić, czy na podstawie tak zdefiniowanego testu jesteśmy w stanie opracować kod aplikacji.

TIP

Testing in Production oznacza wyjście poza strefę komfortu testowanej aplikacji bądź grupy aplikacji. Sprawdzanie poprawności działania w systemie produkcyjnym wiąże się przede wszystkim ze zwiększonym ryzykiem, odpowiedzialnością, potencjalnym wpływem na działanie u klientów itd. Możemy jednak przyjąć, że testowanie tego rodzaju to odwzorowanie zachowań klientów, wtedy rzeczywiście działanie to nie będzie bardzo intruzywne i ujawni zdecydowanie nową kategorię problemów w porównaniu z podejściem tradycyjnym, które zakłada testowanie w środowisku zamkniętym i odseparowanym.

Eksploracja

Eksploracja, czyli *Chaos Testing*, oferuje cały zbiór praktyk, którymi możemy się posługiwać, aby sprawdzać odporność systemu na zaburzenia jego funkcjonowania. Przykładowo restartujemy bazę danych lub serwer komunikatów. Wytwarzamy w systemie wadliwe komunikaty, sprawdzając, jak zachowają się aplikacje je przetwarzające. Obciążamy system, aby wystąpiły opóźnienia na każdej z możliwych warstw uruchomieniowych. Ostatecznie wyłączamy system,

a potem sprawdzamy, co się stanie, kiedy wróci on do działania. CT to koncepcja, która pozwala na bardzo kreatywne podejście do procesu testowania, daleko wychodzące poza regularne ramy.

Wykrywanie defektów

Właściwością człowieka jest błądzić, głupiego — w błędzie trwać.

Cyceron

Nie należy nigdy zakładać, że tworzone oprogramowanie będzie pozbawione błędów. Jako twórcy programów możemy wykrywać, poprawiać i tolerować błędy. Z pewnością najtrudniejsze, ale i najbardziej efektywne jest unikanie usterek na etapie gromadzenia wymagań i ich przekładania na język specyfikacji.

Projektowanie oprogramowania powinno domyślnie obejmować mechanizmy obsługi usterek. Optymalnym podejściem jest eliminowanie potencjalnych przyszłych problemów na etapie analizy i wczesnego projektowania. Wyróżniamy dwa podejścia w obsłudze wykrywania defektów, a mianowicie bierne i czynne.

Bierne wykrywanie defektów

Komponenty programu powinny być względnie niezależne i zachowywać wzajemną podejrzliwość co do założeń swojego funkcjonowania. Anomalie w działaniu chcemy wykrywać jak najwcześniej, tak aby ich skutki nie nabrały charakteru awarii, lecz jedynie zwykłego błędu. Wobec tego powinniśmy przeprowadzać wiele walidacji, czyli sprawdzeń poprawności wykorzystania poszczególnych fragmentów programu. Stosowanie zasady ograniczonego zaufania do zewnętrznych zasobów, danych, kodu, sprzętu też jest bardzo dobrą praktyką w tym zakresie. Dodatkowo, jeśli dany system składa się z szeregu komponentów działających równolegle i wymieniających się komunikatami, nie możemy być nigdy pewni co do jakości i czasu takiej transmisji.

Czynne wykrywanie defektów

Innym podejściem do wykrywania defektów jest mechanizm, który cyklicznie analizuje stan programu i generowanych przez niego danych. Opiera się zatem na predefiniowanych zakresach, których przekroczenie oznacza konieczność wykonania korekty. Możemy taki mechanizm nazwać monitorem lub systemem kontrolnym. Stosowanie tego podejścia zda egzamin szczególnie w przypadku utrzymywania oprogramowania wytworzonego w poprzednich latach i którego działanie nie jest do końca poznane i zrozumiałe dla operatora.

Mikrousługi zakładają dużą dekompozycję struktury aplikacji. Bazy danych dedykowane są poszczególnym komponentom systemu, a ich liczba przekłada się na ilość problemów w zarządzaniu ich strukturą i zmianami oraz panowaniu nad jakością znajdujących się w nich danych. Pytanie zatem nie brzmi, czy baza ma usterki, lecz ile ich jest i jak szybko możemy je zidentyfikować.

Poniższy przykład to program służący do analizy tabel bazy danych. Stosuje on podstawowe funkcje statystyczne.

Plik 01_db_quality.rb:

```
#*******************************************************************************
# Biblioteki
#*******************************************************************************
require 'rubygems'
require 'json'
require 'active_record'
require 'active_support'
require 'active_support/core_ext'
require 'json'
#*******************************************************************************
# Parametryzacja
#*******************************************************************************
PARAM_DB_NAME  = ARGV[0]
PARAM_DB_USER  = ARGV[1]
PARAM_DB_PASS  = ARGV[2]
#*******************************************************************************
# Połączenie
#*******************************************************************************
conn = ActiveRecord::Base.establish_connection(:adapter=>"postgresql",
                                               :encoding=>"unicode",
                                               :pool=>1,
                                               :database=>"#{PARAM_DB_NAME}",
                                               :username=>"#{PARAM_DB_USER}",
                                               :password=>"#{PARAM_DB_PASS}",
                                               :host=>"172.28.128.11")
if conn.nil? then raise '!: NO -AR- CONNECTION' end
#*******************************************************************************
# Dynamiczne mapowanie klas ActiveRecord
#*******************************************************************************
tables = conn.connection.tables
tables.each do |table|
  class_name = table.classify
  begin
  eval <<DYNAMIC
  class #{class_name} < ActiveRecord::Base
    self.table_name = "#{table}"
    self.inheritance_column = :_type_disabled
  end
DYNAMIC
rescue
end # begin/rescue
end # tables

def top_bottom(t)
  val = ""
  if !t[0].to_s.blank? then
    val = t[0].to_s.delete("\n").to_s[0..19].gsub('"', "'")
  end # if
  return { "#{val}": t[1].to_s }
end # top_bottom
```

```
#*****************************************************************************
# Statystyka
#*****************************************************************************
ActiveSupport::Deprecation.silenced = true
Time::DATE_FORMATS[:no_timezone] = "%Y-%m-%d %H:%M:%S"
$stdout.sync = true
json = {}
tables.each do |table|
  if table == "files" then next end
  tbl = {}
  fields = []
  cnt = table.classify.constantize.count
  tbl[:count] = cnt
  limit = 29
  table.classify.constantize.new.attributes.map { |x|
    attrs = {}
    field = x[0]
    if !field.blank? then
      last_nn_val = table.classify.constantize
        .select(field).last[field].to_s[0..limit] rescue 'N\A'
      null_count  = table.classify.constantize
        .where("#{field} IS NULL").count rescue 'N/A'
      min_val     = table.classify.constantize
        .where("#{field} IS NOT NULL").minimum("#{field}").to_s[0..limit] rescue 'N/A'
      max_val     = table.classify.constantize
        .where("#{field} IS NOT NULL").maximum("#{field}").to_s[0..limit] rescue 'N/A'
      avg_val     = table.classify.constantize
        .where("#{field} IS NOT NULL").average("#{field}").to_s[0..limit] rescue 'N/A'
      top10       = table.classify.constantize
        .group("#{field}").order("COUNT(#{field}) DESC").limit(10).count
      bottom10    = table.classify.constantize
        .group("#{field}").order("COUNT(#{field}) ASC").limit(10).count
      distinct    = table.classify.constantize.select("DISTINCT #{field}").count
      attrs[:name] = x[0]
      attrs[:uniq] = distinct
      attrs[:last] = last_nn_val
      attrs[:null] = "#{null_count} (#{((null_count.to_f/cnt.to_f)*100) rescue '-'}%)"
      attrs[:min]  = min_val
      attrs[:max]  = max_val
      attrs[:avg]  = avg_val
      tops = []
      bottoms = []
      top10.each do |t|
        tops << top_bottom(t)
      end # top10
      bottom10.each do |t|
        bottoms << top_bottom(t)
      end # bottom10
      attrs[:tops] = tops
      attrs[:bottoms] = bottoms
    else
      raise "!: FIELD IS BLANK"
    end # if
    fields << attrs
  } # fields
  tbl[:fields] = fields
```

```
  json[table] = tbl
end # tables

puts json.to_json
```

Powyższy przykład wykorzystuje funkcje `COUNT`, `MIN`, `MAX`, `AVG` i `DISTINCT` jako podstawowy zestaw informacji na każdej z kolumn tabeli. Cała struktura pobierana jest za pośrednictwem biblioteki `ActiveRecord`, a modele są tworzone dynamicznie za pomocą biblioteki `ActiveSupport`. Prezentowane są wartości ostatnie, najpopularniejsze oraz najmniej popularne.

Oto fragment przykładowego wyniku w formacie JSON:

```
"name": "content",
"uniq": 939,
"last": "4",
"null": "0 (0.0%)",
"min": "10",
```

Tolerancja na błędy

W ramach każdego projektu możemy zdefiniować pewną kategorię problemów, których obecność nie będzie powodowała przestojów całego systemu, lecz jedynie drobne niedogodności. Warto zatem zbudować bazę wiedzy dotyczącą priorytetyzacji poszczególnych kategorii usterek i taką dokumentacją zasilić zespoły odpowiedzialne za utrzymanie systemu w ruchu oraz grupy kontaktujące się z użytkownikami końcowymi. Należy bowiem przyjąć, że pewne sytuacje krańcowe występują, a koszt i czas ich korekty przewyższa potencjalne zyski w tym zakresie.

Izolacja defektów

Od strony projektowania oprogramowania możemy wskazać programowanie obiektowe i strukturalne jako fundamenty, na których zbudujemy niezależne względem siebie jednostki, komunikujące się na zasadzie ograniczonego zaufania. To podejście pozwala unikać pewnych problemów.

Testowanie Rack

Przegląd rozwiązań

Aplikacje wykonane z wykorzystaniem interfejsu Rack przeważnie są niewielkich rozmiarów i służą do ograniczonych zastosowań. Można je spotkać jako element architektury mikrousług. Oferują w większości przypadków jedynie interfejs HTTP, zatem testowanie opiera się głównie na tej właśnie warstwie abstrakcji.

Przedstawiam tutaj przykłady testowania aplikacji z interfejsem Rack i jego wariantem w postaci platformy Sinatra. Zaczniemy jednak od przykładu opartego na jednej spośród aplikacji z proponowanego zestawu mikrousług, lb.

Mikrousługi: lb

Komponent ten odpowiedzialny jest za rozdzielanie ruchu pomiędzy poszczególne instancje aplikacji z interfejsem Rack uruchomionych na portach z wysokiego zakresu dynamicznie przydzielanych przez orkiestratora.

config.ru

Definiujemy tutaj, że chcemy załadować plik z głównym programem oraz wywołać metodę `run`, która inicjalizuje cały przepływ żądań. Aplikacja w tej formie jest uruchamiana za pośrednictwem polecenia `rackup`.

```
load 'app_proxy.rb'
run AppProxy.new
```

app_proxy.rb

Zawartość tego pliku jest identyczna jak w przykładzie podanym w części dotyczącej mikrousług, komponentu lb. Zamiast jednak znaleźć się w pliku `config.ru` tutaj jest to wydzielone w celu uzyskania możliwości załączania tego pliku do zestawu testowego.

test/test.rb

Jest to plik z testem przeprowadzonym z wykorzystaniem biblioteki `MiniTest`. Załączamy `minitest/autorun` i `rack/test`. Ładujemy plik z programem `app_proxy.rb`. Wczytywane są metody pomocnicze `Rack::Test`. Test definiuje metodę `app`, która udostępnia instancję naszej aplikacji do poszczególnych przypadków testowych.

```
require 'rubygems'
require 'bundler/setup'
require 'rack/test'
require 'minitest/autorun'
load 'app_proxy.rb'
class RackProxyTestSuite < MiniTest::Test
  include Rack::Test::Methods
  def app()
    AppProxy.new
  end # app
  def test_success
    get "/", nil, { "SERVER_NAME" => "rack.home.lab" }
    assert_equal (last_response.status).to_i, 200
  end # test_success
  def test_no_backend_available
    begin
      get "/", nil, { "SERVER_NAME" => "no-backend.home.lab" }
      assert_equal (last_response.status).to_i, 200
    rescue Exception => e
      assert_equal "!: no backend available", e.to_s
    end # begin/rescue
  end # test_no_backend_available
end # RackProxyTestSuite
```

Zestaw dostarcza dwa przypadki testowe. Pierwszy z nich, `test_success`, oczekuje poprawnego wywołania całej ścieżki, począwszy od przetwarzania nazwy domenowej, ekstrakcji subdomeny, sprawdzania serwerów DNS oraz wyszukiwania dostępnych instancji aplikacji docelowej.

Drugi przypadek testowy, wywołując żądanie typu `get`, ustawia jednocześnie adres domenowy, pod którym nie jest znana żadna instancja aplikacji w klastrze i w katalogu `Consul` (wykorzystamy bowiem Nomad jako orkiestrator). Przechwytując wyjątek, sprawdzamy zwrotny tekst z aplikacji, oczekując, że będzie on określał właśnie takie zachowanie.

Uruchomienie

Aby wykonać zestaw testowy, należy wykonać następujące polecenie:

```
[sobczam@home lb]$ ruby test/test.rb
Run options: --seed 16224

# Running:

..

Finished in 0.056790s, 35.2174 runs/s, 35.2174 assertions/s.
2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

Uzyskujemy informacje o czasie wykonywania, wydajności testów oraz ilości przypadków testowych z podziałem na poprawnie i błędnie wykonane.

Rack

Poprzedni przykład, dotyczący aplikacji lb, odnosił się do biblioteki `rack/proxy` i nie był zbyt reprezentatywny dla większości przypadków standardowych aplikacji opartych na interfejsie Rack. W tej sekcji omówię aplikację wzorcową oraz kilka różnych zestawów testowych przygotowanych z wykorzystaniem biblioteki `RSpec` (`gem install rspec`).

application.rb

Jest to standardowa aplikacja oparta na interfejsie Rack, która definiuje metodę `call`. Obsługuje żądania HTTP, mając dostęp do obiektu środowiskowego `env`, który dostarcza informacji o szczegółach żądania.

```
class Application
  def call(env)
    handle_request(env["REQUEST_METHOD"], env["PATH_INFO"])
  end # call
  private
  def handle_request(method, path)
    if method == "GET"
      get(path)
    else
      method_not_allowed(method)
    end
  end # handle_request
  def get(path)
    [200, { "Content-Type" => "text/html" }, ["You have requested the path #{path}, using GET"]]
```

```
  end # get
  def method_not_allowed(method)
    [405, {}, ["Method not allowed: #{method}"]]
  end # method_not_allowed
end # Application
```

Obsługa żądania polega na zwróceniu w ścieżce przetwarzania tablicy z odpowiedzią w postaci kodu odpowiedzi, nagłówków oraz samej treści. W przykładzie użyte są kody 200 i 405.

test_0_spec.rb

Pierwszy przypadek testowy operuje na surowych obiektach i wywołaniach. Tworzymy instancję aplikacji wywołaniem `Application.new`. Definiujemy metodę żądania jako `GET` oraz ścieżkę wywołania.

```
load 'application.rb'
describe Application do
  context "get to /some/url" do
    it "returns the body" do
      app = Application.new
      env = { "REQUEST_METHOD" => "GET", "PATH_INFO" => "/some/url" }
      response = app.call(env)
      body = response[2][0]
      expect(body).to eq "You have requested the path /some/url, using GET"
    end # it
  end # context
end # Application
```

Odpowiedzi oczekujemy po wywołaniu metody `call`, do której przekazujemy konfigurację środowiskową `env`. Treść odpowiedzi znajduje się w `response[2][0]`. Przypadek testowy zakłada, że treścią odpowiedzi będzie `You have requested the path /some/url, using GET`.

test_1_spec.rb

Poprzedni przykład operował na żądaniu i odpowiedzi wprost. W tym przypadku wykorzystujemy metodę `let`, która pozwoli nam wprowadzić modularność tych wywołań.

```
load 'application.rb'
describe Application do
  context "get to /ruby/url" do
    let(:app)      { Application.new }
    let(:env)      { { "REQUEST_METHOD" => "GET", "PATH_INFO" => "/some/url" } }
    let(:response) { app.call(env) }
    let(:status)   { response[0] }
    let(:body)     { response[2][0] }
    it "returns the status 200" do
      expect(status).to eq 200
    end # it
    it "returns the body" do
      expect(body).to eq "You have requested the path /some/url, using GET"
    end # it
  end # context
end # Application
```

Logika samego testu pozostaje niezmienna. Dochodzi jeden przypadek testowy, który weryfikuje kod HTTP odpowiedzi. Oczekujemy, że będzie to 200. Mamy zatem w sumie dwa przypadki testowe w zestawie.

test_2_spec.rb

Jeśli chcemy, aby kod był bardziej modularny, warto stosować metody pomocnicze z `Rack::Test::Methods`. Nie musimy dzięki temu budować żądań i przetwarzać odpowiedzi w sposób całkowicie manualny.

```
load 'application.rb'
require "rack/test"
describe Application do
  include Rack::Test::Methods
  context "get to /some/url" do
    let(:app) { Application.new }
    it "returns the status 200" do
      get "/some/url"
      expect(last_response.status).to eq 200
    end # it
    it "returns the body" do
      get "/some/url"
      expect(last_response.body).to eq "You have requested the path /some/url, using GET"
    end # it
  end # context
end # Application
```

Inicjalizujemy `app` za pomocą `Application.new`. Same przypadki testowe wykorzystują `get` i asercje, które określają, co faktycznie chcemy sprawdzać.

test_3_spec.rb

Kolejnym przykładem jest jeszcze bardziej kompaktowa wersja, która również wykorzystuje `Rack::Test::Methods` i `let`. Wprowadza to jednak ułatwienie polegające na tym, że w kolejnych przypadkach testowych mamy już dostępną odpowiedź.

```
load 'application.rb'
require "rack/test"
describe Application do
  include Rack::Test::Methods
  context "get to /some/url" do
    let(:app)      { Application.new }
    let(:response) { get "/some/url" }
    it { expect(response.status).to eq 200 }
    it { expect(response.body).to include "/some/url, using GET" }
  end # context
end # Application
```

Cały zestaw testowy operuje na tym samym żądaniu. Można zatem tę część uwspólnić. Jest to zdecydowane ułatwienie.

test_4_spec.rb

Przykład ten różni się od poprzedniego wprowadzeniem konfiguracji RSpec przed samym zestawem testowym. Konfiguracja jest wspólna dla całego zestawu. Reszta pozostaje bez zmian.

```
load 'application.rb'
require "rack/test"
RSpec.configure do |config|
  config.include Rack::Test::Methods
end # configure
describe Application do
  context "get to /some/url" do
    let(:app)      { Application.new }
    let(:response) { get "/some/url" }
    it { expect(response.status).to eq 200 }
    it { expect(response.body).to include "/some/url, using GET" }
  end # context
end # Application
```

test_5_spec.rb

Rozwinięciem jednego z poprzednich przykładów jest poniższy, który różnicuje wykonywane żądania, raz na /some/url, innym razem na /. Oczekujemy kodu 200 dla pierwszego przypadku i 405 dla drugiego.

```
load 'application.rb'
require "rack/test"
describe Application do
  let(:app) { Application.new }
  context "get to /some/url" do
    let(:response) { get "/some/url" }
    it { expect(response.status).to eq 200 }
    it { expect(response.body).to include "/some/url, using GET" }
  end # context
  context "post to /" do
    let(:response) { post "/" }
    it { expect(response.status).to eq 405 }
    it { expect(response.body).to eq "Method not allowed: POST" }
  end # context
end # describe
```

Jesteśmy dzięki temu w stanie weryfikować poprawność nie tylko treści odpowiedzi, ale także całego cyklu życia obiektów i ich statusów odzwierciedlonych właśnie w kodach HTTP.

Sinatra

Jednym z przykładów praktycznego zastosowania interfejsu Rack (poza Rails) jest platforma Sinatra. Bardziej jest to jednak DSL przeznaczony do tworzenia aplikacji serwerowych opartych na interfejsie Rack. W poniższym przykładzie stosujemy najprostszą z możliwych aplikacji.

hello_world.rb

Aplikacja, na której podstawie wykonamy kilka zestawów testowych, znajduje się poniżej. Jest to bodaj najprostszy z możliwych przykładów.

```
require 'sinatra'
get '/' do
  "Hello World #{params[:name]}".strip
end
```

Załączamy bibliotekę (`gem install sinatra`) za pomocą `require` oraz definiujemy `endpoint` w postaci /, z którym należy się komunikować za pośrednictwem `GET`. Metoda `get` odpowiada tekstem `Hello World` i parametrem `name` z żądania.

test_1_unit.rb

Pierwszy przypadek testowy wykorzystuje bibliotekę `test/unit`. Zestaw testowy dziedziczy po klasie `Test::Unit::TestCase`. Załączamy metody pomocnicze znajdujące się w `Rack::Test::Methods`.

```
ENV['RACK_ENV'] = 'test'
require_relative 'hello_world'
require 'test/unit'
require 'rack/test'
class HelloWorldTest < Test::Unit::TestCase
  include Rack::Test::Methods
  def app
    Sinatra::Application
  end # app
  def test_it_says_hello_world
    get '/'
    assert last_response.ok?
    assert_equal 'Hello World', last_response.body
  end # test_it_says_hello_world
  def test_it_says_hello_to_a_person
    get '/', :name => 'Michal'
    assert last_response.body.include?('Michal')
  end # test_it_says_hello_to_a_person
end # HelloWorldTest
```

Zestaw testowy składa się z dwóch przypadków. Oba wykonują żądania typu `GET` na ścieżkę /. Pierwszy test sprawdza obecność tekstu `Hello World` w odpowiedzi, drugi natomiast przekazuje parametr `name` z tekstem, który sprawdzamy, czy znajduje się w odpowiedzi.

test_2_spec.rb

Do testowania również w tym wypadku możemy zastosować `RSpec` w połączeniu z pomocniczymi metodami z `rack/test`. Przypadek testowy to żądanie na /, gdzie oczekujemy `be_ok`, czyli kodu 200. Pożądaną treścią odpowiedzi jest `Hello World`.

```
ENV['RACK_ENV'] = 'test'
require_relative 'hello_world'  # <-- your Sinatra app
require 'rspec'
require 'rack/test'
describe 'The HelloWorld App' do
  include Rack::Test::Methods
```

```
  def app
    Sinatra::Application
  end # app
  it "says hello" do
    get '/'
    expect(last_response).to be_ok
    expect(last_response.body).to eq('Hello World')
  end # it
end # describe
```

Plik ze zdefiniowaną aplikacją załączamy za pomocą `require_relative`.

test_3_capybara.rb

Ostatni z przykładów można uznać za najciekawszy, gdyż wykorzystuje biblioteki `Capybara`, `Selenium` i `Webdriver`. Należy zainstalować wszystkie wymienione komponenty — pierwszy i drugi za pomocą `gem`, ostatni natomiast to kwestia pobrania pliku ze strony i umieszczenia go w PATH.

```
ENV['RACK_ENV'] = 'test'
require_relative 'hello_world'
require 'capybara'
require 'capybara/dsl'
require 'test/unit'
class HelloWorldTest < Test::Unit::TestCase
  include Capybara::DSL
  Capybara.default_driver = :selenium
  def setup
    Capybara.app = Sinatra::Application.new
  end # setup
  def test_it_works
    visit '/'
    assert page.has_content?('Hello World')
  end # test_it_works
end # HelloWorldTest
```

`Capybara` definiuje DSL, który wprowadza pomocnicze metody, takie jak np. `visit`. Metoda ta inicjalizuje otwarcie przeglądarki internetowej i wywołuje stronę /. Test sprawdza obecność tekstu `Hello World` na wynikowej stronie.

Testowanie Rails

Scenariusze

Poprzedni podrozdział zawiera przykłady testów interfejsu Rack, w których zostały zastosowane `MiniTest`, `Test::Unit`, `RSpec` i `Capybara`. Samo wykorzystanie narzędzi nie definiuje, jakie metodyki stosujemy w przypadkach testowych. Jeśli rozpoczniemy produkcję kodu od zestawów testowych, będzie to TDD. Jeśli zaś będziemy pisać testy równolegle do kodu lub post factum będzie to dobra praktyka i pierwszy krok ku niezawodności.

Jeżeli jednak zastosujemy przykłady podane dalej, będziemy bliżej metodyki BDD, która charakteryzuje się ścisłym funkcjonalnym powiązaniem scenariuszy testowych z testowanym systemem. Nie sprawdzamy tylko i wyłącznie warstwy jednostkowej czy funkcjonalnej. Proces weryfikacji nabiera tutaj cech testów integracyjnych.

Narzędziem, które wspomaga takie podejście, jest `Cucumber` dla Rails. Przewiduje on scenariusze napisane językiem zbliżonym do naturalnego. Sformułowania zawarte w przypadkach testowych mapowane są na odpowiednie metody realizujące konkretne akcje w testowanym systemie. Mogą to być kroki związane z otworzeniem strony w HTTP, sterowaniem jej zawartością oraz weryfikowaniem rezultatów. Może to być również szereg operacji na bazie danych lub każda inna operacja, którą da się zdefiniować z poziomu kodu testu.

Wzorcową aplikacją, na której będziemy wykonywać zestawy testowe, jest UI z podrozdziału dotyczącego kompletnego projektu w Rails. Aplikacja ta służy do przeglądania zawartości danych w bazie w formie tabel z wykorzystaniem mechanizmów dynamizujących działanie.

Instalacja

W pliku `Gemfile` wskazujemy, że chcemy zainstalować bibliotekę `cucumber-rails` oraz `database_cleaner`:

```
group :test do
  gem 'cucumber-rails', require: false
  # database_cleaner is not required, but highly recommended
  gem 'database_cleaner'
end
```

Następnie wykonujemy polecenia `bundle install` i `rails generate cucumber:install`. Proces testowy uruchamiamy poleceniem `rake cucumber`. Na tym etapie musimy zweryfikować konfigurację połączenia do bazy danych, z racji tego, że proces używa dedykowanej bazy testowej, która będzie za każdym uruchomieniem czyszczona z całej zawartości.

Oto przykładowy plik `config/database.yml`:

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  username: logs
  password: abc123!
  host: 172.28.128.11
  port: 5432
  schema_search_path: myapp,sharedapp,public
development:
  <<: *default
  database: logs
test: &test
  <<: *default
  database: ui_app_test
cucumber:
  <<: *test
```

Funkcje pomocnicze

Domyślnie Cucumber wykorzystuje w strukturze projektu Rails katalog features. Przedstawione dalej pliki pochodzą właśnie z tej lokalizacji.

methods.rb

Definiujemy tutaj podstawowe funkcje pomocnicze. Najistotniejszą jest sign_in, która wykonuje operację zalogowania użytkownika do systemu. Aby przykład był bardziej zwięzły, zamiast stosować fixtures przykładowy użytkownik, w przypadku jego braku, tworzony jest w samej metodzie.

```
def set_selenium_window_size(width, height)
  window = Capybara.current_session.driver.browser.manage.window
  window.resize_to(width, height)
end # set_selenium_window_size
def select_something(where, what)
  option_xpath = "//select[@id='#{where}']/option[text()='#{what}']"
  option = page.first(:xpath, option_xpath).text
  select(option, :from => where)
end # select_something
def sign_in
  u = User.first
  if u.blank? then
    u = User.new
    u.email = "michalasobczak@gmail.com"
    u.password = "abc123!"
    u.save!
  end
  visit '/'
  fill_in 'user[email]', :with => 'michalasobczak@gmail.com'
  fill_in 'user[password]', :with => 'abc123!'
  click_button 'Sign in'
end # sign_in
```

Wykorzystujemy metody visit, fill_in i click_button, które odpowiadają za sterowanie zachowaniem strony. Pochodzą z domyślnej biblioteki Capybara, która jest skonfigurowana razem z biblioteką Cucumber. Mamy możliwość użycia innej biblioteki sterującej, np. Selenium. Jak widać, w metodzie select_something możemy używać również selektorów typu xpath.

given.rb

Siłą rozwiązań opartych na Rails jest konwencja, standard. Także w przypadku biblioteki Cucumber mamy z tym podejściem do czynienia. Nomenklatura testu opiera się na słowach kluczowych (de facto metodach) given, when oraz then.

Metoda given zakłada obecność pewnych elementów lub inicjalizuje punkt startowy testu.

```
Given /^I am on "(.*?)" page$/ do |arg1|
  visit arg1
end
Given /I ensure the confirm box returns OK/ do
  page.evaluate_script('window.confirm = function() { return true; }')
end
```

```
Given /^I am logged in as a user$/ do
  sign_in
end
```

Słowa when i then kolejno przyjmują znaczenie odpowiednio akcji i rezultatu. W przykładzie powyżej Given I am on ... page oznacza, że wywołujemy metodę visit, czyli otwieramy stronę pod wskazanym adresem. Metoda Given I am logged in a user wykonuje zalogowanie użytkownika. Kod za to odpowiedzialny został zdefiniowany w pliku methods.rb.

when.rb

Kolejnym krokiem są metody opisujące wykonywanie czynności sterujących stroną i jej treścią. Przykładowo będzie to wybranie przycisku click_button, otwarcie odnośnika click_link itd.

```
When /^I press "(.*?)"$/ do |arg1|
  click_button(arg1)
end
When /^I click on "(.*?)"$/ do |arg1|
  find("##{arg1}").click
end
When /^I fill in "(.`?)" with "(.`?)"$/ do |arg1, arg2|
  fill_in arg1, :with => arg2
end
When /^I fill in "(.`?)" with "(.`?)" using JS$/ do |arg1, arg2|
  page.evaluate_script("$('##{arg1}').val('#{arg2}')")
end
When /^I click link "(.*?)"$/ do |arg1|
  click_link arg1
end
When /^next to "(.`?)" I press "(.`?)"$/ do |arg1, arg2|
  xpath = "//td[normalize-space()='#{arg1}']/..//input[@value='#{arg2}']"
  page.all(:xpath, xpath).first.click
end
When /^I click by xpath "(.*?)"$/ do |xpath|
  page.all(:xpath, xpath).first.click
end
When /^I press key "(.`?)" on "(.`?)"$/ do |key,id|
  key_int = key.to_i
  page.find("##{id}").native.send_keys key.to_sym
end
When /^I select "(.`?)" with "(.`?)"$/ do |arg1, arg2|
  option_xpath = "//select[@id='#{arg1}']/option[text()='#{arg2}']"
  option = find(:xpath, option_xpath).text
  select(option, :from => arg1)
end
When /^I wait for "(.*?)" seconds$/ do |seconds|
  sleep(seconds.to_i)
end
When /^I set today at "(.*?)"$/ do |arg1|
  page.all("input[name='#{arg1}']").first.set(Date.today.to_s)
  page.all("input[name='#{arg1}']").last.set(Date.today.to_s)
end
```

Możemy poza sterowaniem zawartością opisową strony, tj. HTML, wykorzystywać elementy dynamiczne w postaci kodu w języku JavaScript. Jest to nieco bardziej złożony temat niż proste wypełnienie pola na stronie, ale zdecydowanie przydatne, ponieważ praktycznie każda aplikacja będzie taki kod miała. Chcemy, aby przypadki testowe odzwierciedlały faktyczne użycie aplikacji, bez wprowadzania szczególnych ograniczeń.

then.rb

Finalnie, aby móc weryfikować wykonane na stronie czynności, wywołujemy metody `Then`, które w swoich definicjach przyjmują różnego rodzaju asercje. Sprawdzamy przykładowo obecność tekstu na stronie wynikowej.

```
Then(/^page should have notice "(.*?)"$/) do |arg1|
  # puts page.body.inspect
  assert page.has_content?(arg1)
end
Then(/^page should not have notice "(.*?)"$/) do |arg1|
  assert !page.has_content?(arg1)
end
Then /^page should have elements by xpath "(.*)"$/ do |xpath|
  page.all(:xpath, xpath)
end
```

Możemy też oczekiwać, że danego tekstu nie będzie na stronie, w przypadku na przykład testu kasującego treści z aplikacji. Możemy również wykonywać asercje oparte na bezpośrednim sprawdzaniu zawartości bazy danych. Dzięki temu weryfikacja będzie kompleksowa.

Przypadki testowe

Wykorzystując zdefiniowane w poprzedniej sekcji funkcje pomocnicze, wykonujące konkretne czynności związane ze sterowaniem stroną i weryfikacją wyników, możemy przystąpić do budowy zestawów testowych.

users.feature

Jest to przykład najprostszy z możliwych. Loguje użytkownika do systemu i nawiguje do strony `/users`. Sprawdzaniu w zasadzie podlega jedynie brak obecności wyjątków w sprawdzanej ścieżce.

```
Feature: Users
   Scenario: List users
      Given I am logged in as a user
      Given I am on "/users" page
```

messages.feature

To przykład bardziej rozbudowany. Weryfikuje on zalogowanie użytkownika, nawiguje na stronę `/messages` oraz, co najważniejsze, sprawdza wynikowy tekst strony pod kątem konkretnych fraz, tzn. pól nagłówka tabeli `datagrid`.

```
Feature: Messages
   Scenario: List messages
      Given I am logged in as a user
```

```
    Given I am on "/messages" page
       Then page should have notice "Id"
      Then page should have notice "Content"
      Then page should have notice "Processed at"
      Then page should have notice "Created at"
      Then page should have notice "Updated at"
```

W `given`, `when` i `then` podanych zostało więcej możliwych zastosowań, co pozostawiam do samodzielnego sprawdzenia.

Uruchomienie

Aby wywołać zestawy testowe, wykonujemy polecenie `rake cucumber`:

```
sobczam@home ui_app]$ rake cucumber
/home/sobczam/.rvm/rubies/ruby-2.5.1/bin/ruby -S bundle exec cucumber --profile default
Using the default profile...
Feature: Messages

  Scenario: List messages                       # features/messages.feature:3
    Given I am logged in as a user              # features/step_definitions/given.rb:11
    Given I am on "/messages" page              # features/step_definitions/given.rb:1
    Then page should have notice "Id"           # features/step_definitions/then.rb:1
    Then page should have notice "Content"      # features/step_definitions/then.rb:1
    Then page should have notice "Processed at" # features/step_definitions/then.rb:1
    Then page should have notice "Created at"   # features/step_definitions/then.rb:1
    Then page should have notice "Updated at"   # features/step_definitions/then.rb:1

Feature: Users

  Scenario: List users             # features/users.feature:3
    Given I am logged in as a user # features/step_definitions/given.rb:11
    Given I am on "/users" page    # features/step_definitions/given.rb:1

2 scenarios (2 passed)
9 steps (9 passed)
0m0.982s
```

Dla przykładów podanych wcześniej mamy w sumie dwa scenariusze z dziewięcioma krokami. Test zakończył się wynikiem pozytywnym.

Skorowidz

I

J

K

L

M

N

O

P

Poznaj Ruby on Rails od praktycznej strony!

- Odkryj język Ruby i platformę Rails
- Naucz się testować i uruchamiać swój kod
- Poznaj zalety mikrousług i konteneryzacji

Ruby to nowoczesny, wieloparadygmatowy, interpretowany język programowania. Wraz z platformą Rails stanowi jedno z najpopularniejszych rozwiązań służących do szybkiego tworzenia aplikacji sieciowych; wspiera wiele znanych serwisów dostępnych w internecie. Ruby on Rails od lat utrzymuje się w ścisłej czołówce platform klasy MVC — dzięki rozbudowanym funkcjom, wysokiej wydajności oraz łatwości pisania kodu, a także możliwości stosowania dużej liczby rozszerzeń.

Jeśli chcesz się dowiedzieć, jak wykorzystać tę platformę w swoich projektach, jesteś na dobrym tropie! Dzięki tej książce poznasz możliwości i konstrukcje języka Ruby oraz mechanizm działania platformy Rails, a w szczególności interfejs Rack. Dowiesz się, jak zapewniać odpowiednią jakość swoich rozwiązań, nauczysz się je uruchamiać przy użyciu technologii wirtualizacji VMware ESXi oraz konteneryzacji Docker na platformach OpenShift Origin, OKD i Nomad. Prześledziwszy praktyczne przykłady, zdobędziesz wiedzę na temat architektury mikrousług, poznasz też sposoby wykorzystania oprogramowania GitLab w funkcji repozytorium kodu, systemu zgłoszeń, bazy wiedzy i narzędzia CI/CD.

- Mechanizmy języka Ruby i ich praktyczne zastosowanie
- Programowanie strukturalne i obiektowe oraz metaprogramowanie
- Możliwości platformy Rails i ich wykorzystanie w praktyce
- Zastosowanie interfejsu Rack i szyny usługowej RabbitMQ
- Zapewnianie jakości aplikacji — teoria i praktyka
- Uruchamianie aplikacji przy użyciu maszyn wirtualnych i kontenerów
- Zastosowanie systemów orkiestracji kontenerów
- Praca z repozytorium kodu oraz ciągła integracja i dostarczanie (CI/CD)

Naucz się tworzyć wydajne aplikacje sieciowe z Ruby on Rails!

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-5241-4

Cena: 59,00 zł