

Spis treści

1. Streszczenie	2
2. Abstract	3
3. Wprowadzenie historyczne do Behaviour Driven Development	4
Jak zmieniało się myślenie o programowaniu i jak powstała metodyka BDD	
4. Wprowadzenie do Behaviour Driven Development (BDD)	12
Na przykładzie tandemu systemów Rspec i Cucumber	
5. Aplikacje webowe	16
Krótkie wprowadzenie	
6. Charakterystyka języka Ruby	20
oraz platformy Ruby on Rails	
7. Model Iteracyjno-Przyrostowy a metodyka BDD	25
8. Budowa aplikacji według modelu BDD	28
8.0. Cel projektu i wymagania aplikacji	28
8.1. Przygotowanie platformy	31
8.2. Uwierzytelnianie	38
8.3. Rejestracja i testy negatywne	52
8.4. Odzyskiwanie konta użytkownika dzięki Mailerowi	62
8.5. Dodawanie treści przez użytkownika	67
9. Przydatne metody debugowania	72
Bindowanie danych i „migawka” widoku	
10. Prezentacja aplikacji	76
11. Podsumowanie	79

1. Streszczenie

Celem pracy jest przedstawienie praktycznego zastosowania *Programowania Metodą Behawioralną* – jednej z metodyk *Programowania Zwinnego*, reprezentującego alternatywne do *Modelu Kaskadowego*, kompleksowe podejście do zarządzania i rozwijania projektów informatycznych.

Na potrzeby prezentacji tego podejścia w kolejnych rozdziałach przedstawiona zostanie historia ewolucji metodyk programowania, a później, praktyczne wykorzystanie techniki *Behawioralnej* krok po kroku, w budowie demonstracyjnej aplikacji internetowej, przedstawiającej podstawowe funkcjonalności nowoczesnej aplikacji typu Web2.0.

W projekcie wykorzystane zostaną:

- Język programowania: *Ruby* (wersja 1.9.2, implementacja *MRI*)
- Platforma do tworzenia aplikacji webowych: *Ruby on Rails* (wersja 3.1)
- Platforma do projektowania testów akceptacyjnych: *Rspec*
- Język opisu scenariuszy behawioralnych: *Cucumber*
- System uwierzytelniania: *Devise*
- System zarządzania uprawnieniami: *Cancan*

W tworzeniu aplikacji pomagać będą inne narzędzia, nie mają one jednak wpływu na kształt programu, ani na sposób prezentacji metodyki BDD.

Skorzystanie z nich wynika tylko z osobistych preferencji autora niniejszej pracy.

- System wersjonowania oprogramowania: *Git*
- System zarządzania wersjami interpretera języka *Ruby*: *RVM*

2. Abstract

The goal of this thesis is to present a practical use of *Behaviour Driven Development* (*BDD*) methodology – a part of *Agile Software Development*, a group of methodologies alternative to *Waterfall* development model. *BDD* is a complete and complex approach to software development and maintenance.

For presentation purposes the next chapters will tell a short history of software development evolution, then a practical step-by-step usage of the Behaviour Driven methodology, resulting in a construction of a simple web application, demonstrating the most popular features expected from a modern Web2.0 project.

The technologies used are:

- Programming language: *Ruby* (version 1.9.2, *MRI implementation*)
- Web application framework: *Ruby on Rails* (version 3.1)
- Acceptance test framework: *Rspec*
- Behaviour development tool: *Cucumber*
- Authentication framework: *Devise*
- Authorization and permission management tool: *Cancan*

During the course of the development other tools were used as well. Those software packages are irrelevant neither to the *BDD* process nor to the functionality of the completed application.

Their usage is only advocated by the personal preferences of this thesis' author.

- Software versioning system: *Git*
- *Ruby* language interpreter version manager: *RVM*

3. Wprowadzenie historyczne do Behaviour Driven Development

Jak zmieniał się myślenie o programowaniu i jak powstała metodyka *BDD*

Programowanie było jeszcze bardzo młodą dziedziną nauki, kiedy pod koniec lat 50. XX wieku w *Massachusetts Institute of Technology* pojawiły się pierwsze próby wprowadzenia „obiektów” jako jej podstawowego budulca. Cyfrowy obiekt miał reprezentować nowy poziom abstrakcji, stanowić most pomiędzy ludzkim językiem i jego ostrzeganiem świata, a pamięcią komputera, w której wszystko musi mieć swoje miejsce, wartość; a przy tym koncepcja ta miała być instynktownie prosta, jak pojęcie podmiotu w zdaniu. Stworzony w 1958 r. przez Johna McCarthy'ego (nagroda Turinga w 1971 r. za prace nad podstawami sztucznej inteligencji) język *LISP* oferował „atomy” – protoplastów dzisiejszych obiektów – i system ich atrybutów na długo przed tym, gdy dr Alan Kay (który w 2003 r. otrzymał nagrodę Turinga za wkład w prace nad językami programowania, jak również interfejsami graficznymi) w 1966 r. zaproponował termin *object-oriented programming*¹ (ang. programowanie zorientowane obiektowo – choć w języku polskim używa się prostszej nazwy „programowanie obiektowe”). Język *LISP* nie był dla Kaya jedyną inspiracją – na przełomie lat 1960-61 Ivan Edward Sutherland (laureat nagrody Turinga w 1988 r.) rozwijał swój autorski projekt *Sketchpad*, kładąc kamień węgielny pod całą rodzinę programów typu *CAD* (*Computer-aided Design*, ang. Projektowanie Wspomagane Komputerowo). Jednak ten pierwszy w historii program z interfejsem graficznym (którego Kay stał się natychmiast gorącym orędownikiem), napisany został w języku dedykowanym (*Sketchpad* był jedynym użytkownikiem nowego języka); poza tym miał inną wadę – działał na superkomputerze *Lincoln TX-2*, o którym dr Alan Kay powiedział: „Był to ostatni komputer w Ameryce, który miał własny dach”².

Pierwszy język programowania, w którym explicite zdefiniowano obiekty powstał jednak nie na terenie Stanów Zjednoczonych, ale w odległej Norwegii – był to język *Simula I*, autorstwa Kristena Nygaarda i Ole-Johana Dahla (nagroda Turinga przyznana obojgu w 2001 r. za pionierskie prace nad programowaniem obiektowym). Zgodnie z nazwą, rodzina języków *Simula* (później powstał jeszcze *Simula 67*) służyła do przeprowadzania symulacji – nie tylko wspomaganych komputerowo, ale także wykonywanych w całości w pamięci maszyny. Nowatorskie rozwiązania zapewniły norweskim informatykom sławę. W roku 1962 Nygaard został zaproszony do laboratoriów *UNIVAC*, producenta komputerów „biznesowych”, czyli maszyn typu mainframe.

1 http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en - archiwalny zapis korespondencji mailowej między drem Alanem Kay'em a Stefanem Ramem (stan z dnia 12.07.2011)

2 W 1987r, na potrzeby *University Video Communications*, dr Alan Kay przeprowadził prezentację pionierskich systemów komputerowych, które jako pierwsze w latach 60-tych i 70-tych przedstawiły światu interfejs użytkownika; w połowie piątej minuty filmu rozpoczyna się prezentacja programu *Sketchpad* - <http://www.archive.org/details/AlanKeyD1987> (stan z dnia 13.07.2011)

W czasie swojego pobytu w USA prezentował rozwiązania programowania obiektowego, które sam stworzył. Wkrótce *Simula* zaczęła zdobywać uznanie na całym świecie, zaimplementowano ją w amerykańskim komputerze *Burroughs B5500*, i w radzieckim *Ural-16*.

W 1966 r. brytyjski informatyk Sir Charles Antony Richard Hoare (laureat nagrody Turinga w 1980 r.) przedstawił definicję „klasy” – formalnego opisu danego rodzaju obiektu, przy założeniu, że programista może stworzyć podklasę – nowy „gatunek” obiektu, który „dziedziczyć” będzie atrybuty z klasy z której się wywodzi, dokonując na nich własnych zmian, analogicznie jak podczas ewolucji czynią to gatunki potomne w przyrodzie. Rok później powstał *Simula 67* – był pierwszym językiem w historii, który wprowadził klasy i pierwszym wyposażonym w system *garbage collection* (ang. zbieranie śmieci) – proces działający w tle programu, „czyszczący” zwalniane komórki pamięci tak, by łatwo zająć je mogły inne programy. W maju tego roku, w Oslo, przyjęto formalną definicję tego języka, a już w czerwcu Ole-Johan Dahl postulował wprowadzenie paradygmatu, który miał traktować typy prymitywne (liczby, znaki alfanumeryczne, wartości logiczne) jako obiekty określonej klasy, w ten sposób unifikując wszystkie dane programu do postaci obiektu. Jego propozycja została jednak odrzucona.

Dr Alan Kay, zafascynowany biologią, widział przyszłość programistyki w kontynuowaniu prac nad „obiektem” – cyfrowym bytem, opisanym atrybutami, reagującym na komunikaty, wysyłane przez inne byty. Atomy *LISPa* były dosyć prymitywne, obejmowały tylko liczby i łańcuchy znaków (zwane symbolami); program *Sketchpad* traktował geometryczne wierzchołki, krawędzie, a również całe figury, jako obiekty, ale jakkolwiek innowacyjne rozwiązania wprowadzał (jest pierwszym programem wykorzystującym ideę „ekranu dotykowego”, chociaż „dotyk” odbywał się tutaj za pomocą świetlnego rysika, więc system nie reagowałby na dotyk ludzkiej dłoni), w swojej implementacji ograniczony był do konkretnego rozwiązania – projektowania graficznego. *Simula I* i *67*, były krokiem w dobrym kierunku, ale choć języki te jako pierwsze wprowadziły w życie koncepcje, o których Kay mówił na konferencjach otwarcie już od roku, chociaż dawały programistom potężne narzędzie w postaci cyfrowego obiektu i jego klasy, to już w 1969 r. Kay (pracujący wtedy w ośrodku *Xerox* w Palo Alto) poprowadził zespół w pracach nad językiem nowej generacji.

W roku 1972 powstał język *Smalltalk* (wśród członków zespołu Kaya należy wymienić Daniela Ingallsa, który pierwszą wersję nowego języka zaimplementował w ciągu zaledwie kilku dni na podstawie jednej strony notatek, w efekcie wygrywając zakład z Kayem). Na przestrzeni lat 1972-80 powstało kilka wersji rozwojowych tego języka. Zaprezentowany w 1980 r. *Smalltalk-80* z niewielkimi zmianami przetrwał do dziś i wciąż jest wykorzystywany w praktyce. Podstawową ideą tego języka było „traktowanie wszystkiego jako obiekt” – tak, jak to w 1967 r. w Oslo postulował Ole-Johan Dahl. Ujednolicenie wszystkich „podmiotów” biorących udział w konstrukcji programu, poza spełnieniem osobistego marzenia dra Alana Kaya o języku prawdziwie „obiekto-zorientowanym”, miało bardzo konkretny cel – zmienić sposób myślenia programistów.

Pracując jeszcze w *MIT*, dr Kay miał okazję obserwować początki rewolucji w komunikacji człowieka z maszyną. Wśród prekursorów w tej dziedzinie należy wymienić rzecz jasna wspomnianego wcześniej Ivana Sutherlanda, ale przede wszystkim Douglasa Carla Engelbarta (nagroda Turinga w 1997 r.) – wynalazcę myszki komputerowej i *hipertekstu* – systemu wzajemnie powiązanych dokumentów (leksji), które użytkownik nawiguje nie w ustalonej kolejności, a wg. własnego uznania, wybierając odpowiednie łącza – bez tego wynalazku internet, jakim go znamy, nigdy by nie powstał. Mimo wszystko te koncepcje całkowicie odmiennego podejścia do komunikacji z maszyną przez blisko 20 lat pozostawały tylko w sferze ciekawostek. Doświadczenia wyniesione przez dr Alana Kaya z tych obserwacji umocniły go w przekonaniu, że komputery zmieniły się wystarczająco i nadszedł czas by zmienić „drugą stronę” dialogu: użytkownika i jego podejście. Przyświecała mu przy tym myśl kanadyjskiego filozofa i teoretyka komunikacji Herberta Marshalla McLuhana (jako człowiek niezwiązany z amerykańskimi ośrodkami badań i rozwoju informatyki przewidział powstanie sieci komputerowej i unifikację mediów na 30 lat zanim został stworzony internet; jako pierwszy użył też powiedzenia, że świat dąży do tego, by stać się „globalną wioską”), który żartobliwie mówił, że „nie wiemy kto wymyślił wodę, ale na pewno nie były to ryby” – przekaz był dla Kaya jasny, rewolucji obsługi komputera nie można dokonać tylko w gronie naukowców, którzy od lat stykają się z najnowszymi technologiami. Potrzebni byli użytkownicy „z zewnątrz”.

Smalltalk posłużył w serii eksperymentów w drugiej połowie lat 70., w których wzięło udział kilkaset dzieci w wieku lat 12-13 z okolicznych szkół w Palo Alto. W ciągu trwających dwa szkolne semestry zajęć nauczono młodych programistów obsługi komend *Smalltalk* poczynając od tych prostych, odziedziczonych po języku *Logo* – czyli tworzenia grafiki na ekranie za pomocą funkcji – aż po naukę pisania własnych programów. Najważniejszym elementem eksperymentu były zajęcia prowadzone dla uczniów przez uczniów ze starszych klas. Okazało się, że dzieci, które nie miały wcześniej żadnej styczności z informatyką, nie będące „zakładnikami” starych koncepcji programowania, były w stanie pisać kompletne programy użytkowe, jak edytory grafiki i animacji, a ich kod był wyjątkowo koherentny w porównaniu z tym jak wyglądałby on w języku proceduralnym. Zespół *Xeroxa* z Palo Alto dowiódł tym samym, że paradygmat „obiektu” jest naturalny, instynktownie rozumiany nawet przez dzieci, mające zaledwie kilkumiesięczne doświadczenie w pracy z komputerem.

Języki z rodziny *Smalltalk*, pierwsze w historii „zorientowane obiektowo”, stały się inspiracją dla wielu późniejszych języków wysokiego poziomu, jak chociażby *Java*, dodatkowo część języków przejęła rozwiązania składni i dynamicznego typowania (typ, czyli klasa zmiennych nie jest deklarowana wprost, przez podanie słowa kluczowego, ale przez przypisanie wartości), są to przede wszystkim języki skryptowe i interpretowane, jak *Perl*, *Python* i *Ruby*.

Nie ma przesady w mówieniu, że programowanie obiektowe zrewolucjonizowało informatykę, chociaż na prawdziwe efekty należało czekać do lat 80., kiedy komputery

osobiste na dobre trafiły do domów indywidualnych użytkowników, a programowaniem mogli zająć się ludzie nie związani z uczelnianymi, bądź wojskowymi ośrodkami badań. Do dziś jedyną alternatywą dla programowania proceduralnego jest programowanie obiektowe, a najpopularniejsze języki³ w użyciu są właśnie obiektowymi. Wyjątkiem jest język C, któremu historia wyznaczyła specjalnie miejsce: jako język niemal natywny dla tworzonych procesorów obsługuje ich najbardziej podstawowe funkcje, jest przez to najszybciej wykonywanym ze wszystkich języków wysokiego poziomu (choć często mówi się o nim, jako o języku poziomu pośredniego), stanowi przez to jedyny słuszny wybór, gdy liczy się szybkość, a nie wygoda wynikająca z użycia obiektów.

Rok 1958 był bardzo ważny w historii informatyki i telekomunikacji nie tylko ze względu na pionierskie prace nad rozwojem programowania w instytucie *MIT*. Amerykanie od początku lat 50. opracowywali narzędzie ułatwiające komunikację i współpracę cywilnych i wojskowych ośrodków badawczych rozsianych po terenie całych Stanów. Pomysł wykorzystania sieci komputerowej, przesyłającej dane podzielone na „pakiety” pochodził od Paula Barana, informatyka polskiego pochodzenia (ur. w 1926 r., w Grodnie – obecnie Białoruś) ówczesnego członka *RAND Corporation*, jednego z pierwszych „think-tanków” w historii, i stanowił odpowiedź na postawione przez Departament wymaganie, aby system taki był w stanie przetrwać atak nuklearny z użyciem wielu głowic, co oznaczało, że musiał być zdecentralizowany. Do roku 1957 prace na ten temat były czysto teoretyczne, gdy nieoczekiwanie świat obiegła wiadomość, że Związek Radziecki umieścił na ziemskiej orbicie pierwszego satelitę w historii – *Sputnika 1*. Podjęto decyzję o natychmiastowym przyspieszeniu badań i już rok później, w 1958 r., administracja prezydenta Dwighta Eisenhowera powołała agencję *ARPA* (*Advanced Research Projects Agency* ang. Agencja Zaawansowanych Projektów Badawczych).

Zadanie stworzenia komputerowej sieci było tylko jednym z wielu, jakie rząd zlecił nowo powstałej agencji, ale mimo wszystko na pierwsze efekty trzeba było czekać zaskakująco długo, bo aż jedenaście lat. Projekt rozwijany był jednocześnie w kilku różnych ośrodkach badawczych, a najlepiej rokujące pomysły tworzyły pewnego rodzaju rdzeń dalszych prac. Pierwsze połączenie dwóch niezależnych ośrodków nastąpiło w Kalifornii w 1969 r. pomiędzy Instytutem Badawczym Stanforda, a kampusem Uniwersytetu Kalifornijskiego w Los Angeles⁴ – jest to dystans około 500 kilometrów. Jeszcze w grudniu tego roku w sieci nazywanej *ARPANETem* połączone były cztery ośrodki; trzy w Kalifornii, jeden w Utah. To właśnie możliwość włączenia istniejących, wewnętrznych sieci jako podsieci do większego systemu teleinformatycznego, zapewniła *ARPANETowi* sukces.

W połowie lat 70. na całym świecie operowało kilka niezależnych sieci komputerowych, jak francuski *CYCLADES* i brytyjska sieć Mark I, a w samych Stanach

3 Wykresy popularności języków programowania używanych w projektach <http://www.langpop.com/> (stan z dnia 12.07.2011)

4 http://www.netvalley.com/cgi-bin/intval/net_history.pl - artykuł o historii internetu (stan z dnia 17.07.2011)

Zjednoczonych: *Tymnet* w Kalifornii, czy *Merit Network* w stanie Michigan. Zrodził się problem jak połączyć istniejące sieci całego świata w jeden wydajny układ. W 1974 r. w efekcie współpracy Roberta Kahna z *DARPA* (Agencja została przemianowana na *Defense Advanced Research Projects Agency* w 1972 r.) i Vintona Cerfa z Uniwersytetu Stanforda, powstała „specyfikacja *RFC-675*”⁵, w tytule której po raz pierwszy pada nazwa „internet”. Najważniejszym punktem dokumentu było postulowanie o umniejszenie roli samej sieci do zaledwie medium, przenoszącego pakiety, całą resztę pracy miały wykonywać przede wszystkim serwery i w mniejszym stopniu maszyny klienckie; był to pomysł zaczerpnięty od twórcy *CYCLADES*a – Louisa Pouzina. W roku 1982 formalnie zdefiniowano i opatentowano protokół *TCP/IP*.

Jeszcze do końca lat 80ych na terenie USA powstawały odrębne sieci naukowe jak *CSNET* (*Computer Science Network*) i *NSFNET* (*National Science Foundation*), ale już z początkiem lat 90. stanowiły wraz z *ARPANET*em jeden nierozróżnialny system, pozbawiony elementów centralnych, tak, jak przewidywały to wymagania Departamentu Obrony sprzed 40 lat. Termin „internet” który wcześniej był zaledwie skrótem od wyrażenia *internetworking* (ang. praca wewnątrz sieci) – zaczął być używany jako nazwa własna nowego medium teleinformatycznego.

Ośrodek *CERN* w Europie kuszony perspektywami połączenia z uczelniami zza oceanu zaadaptował rozwiązania *TCP/IP* na własne potrzeby. To właśnie tam już w 1989 r. Sir Timothy Berners-Lee przedstawił propozycję utworzenia systemu, który umożliwi magazynowanie i przeglądanie danych wewnątrz sieci, wykorzystując do tego programy zwane „przeglądarkami”, działające za zasadzie obsługi *hipertekstu* – pomysłu Douglasa Engelbarta sprzed trzydziestu lat, który jeszcze nie doczekał się realizacji na skalę światową. Rok później do Bernersa-Lee dołączył Belg Robert Cailliau; razem dopracowali projekt, stworzyli pierwszą przeglądarkę, pierwsze strony hipertekstowe i serwer na stacji roboczej *NeXT*, do którego można było się połączyć i czytać znajdujące się na nim dokumenty. Swój prototypowy system nazwali *World Wide Web* (ang. Sieć Ogólnoświatowa), jakby przewidując, że istotnie stanie się on narzędziem używanym na całym świecie; Berners-Lee tworzył go od podstaw z zamierzeniem włączenia *WWW* do Internetu. I tak, jak przewidział w latach 60. Marshall McLuhan, a dziesięć lat po nim jeden z ojców fantastyki naukowej Sir Arthur C. Clarke – powstała teleinformatyczna sieć, dająca ludziom dostęp do wiedzy pochodzącej z całego świata na wyciągnięcie ręki. Optymizm jej twórców tylko raz zderzył się z rzeczywistością – zakładali oni, że rozbudowanie systemu do funkcjonalności, w której użytkownicy sami będą mogli zarządzać jego treścią, to kwestia sześciu miesięcy od czasu zaprezentowania pierwszej wersji. Minęło 10 lat zanim świat internetu wkroczył w epokę *Web 2.0* – epokę blogów, forum, filmu, muzyki i telewizji w internecie. Statyczne internetowe „witryny” i „strony” miały zostać zastąpione funkcjonalnymi aplikacjami, hipertekstowe dokumenty – interaktywnymi prezentacjami z dźwiękiem i animacją.

Czasy gdy *MIT*, czy jakakolwiek inna instytucja, wytyczały kierunek i dyktowały

5 <http://tools.ietf.org/html/rfc675> - treść specyfikacji RFC 675 (stan z dnia 17.07.2011)

trendy w rozwijaniu nowych metodyk pracy, czy technologii, minęły bezpowrotnie. *Web2.0* uczynił z internautów współautorów nowego medium; każdy mógł publikować swoje pomysły i opinie, ale nowe czasy wymagały nowego „przywództwa”. To właśnie wtedy, w 2001 roku, w Snowbird, w stanie Utah, odbyła się konferencja, w której udział wzięli entuzjaści nowych technologii. Jej głównym zadaniem było wypracowanie porozumienia w sprawie metodyki pracy alternatywnej do używanego od lat 70ych „modelu kaskadowego”. Opracowano *Manifest Zwinnego Wytwarzania Oprogramowania*, zwany potocznie *Manifestem Agile*, od jego angielskiej nazwy *Manifesto for Agile Software Development*. Jego główną ideą było przedkładanie⁶:

- **Ludzi i ich wzajemne interakcje (współdziałanie)** ponad procedury i narzędzia.
- **Działające oprogramowanie** nad wyczerpującą dokumentację.
- **Współpracę z klientem** nad negocjację umów.
- **Reagowanie na zmiany** nad realizowanie planu.

Najciekawszym, moim zdaniem, aspektem *Manifestu*, jest fakt, że był pierwszym formalnym dokumentem opisującym metody, którymi przez całe lata nieoficjalnie posługiwali się najwięksi programiści, jak wspomniany wcześniej współtwórca *Smalltalka* – Dan Ingalls, czy Donald Knuth, autor wielotomowej „Sztuki Programowania” i twórca systemu składania tekstu *TEX*. W gruncie rzeczy wszystkie metody, objęte wspólną nazwą *Zwinnych*, używane były wcześniej, *Manifest* wpisał je tylko w nowe ramy, zidentyfikował je. Na przykład metodyki typu *test-first* znane były już pod koniec lat 90ych, odwracały one „standardową” i – można by pomyśleć – naturalną kolejność pisania oprogramowania, zakładając, że środowisko testowe programu nie powstaje w odpowiedzi na potrzebę sprawdzenia poprawności jego działania, lecz to kod programu powstaje tak, by być w pełni zgodnym z uprzednio narzuconymi ramami testu. W związku z rosnącą rolą testów w tym okresie powstawały również pierwsze automatyczne systemy ich wykonywania na kodzie, wraz z nimi – terminologia określająca m.in. testy jednostkowe i integracyjne. Pierwsze z nich, jak sama nazwa wskazuje, dotyczą testowania wyodrębnionego, atomowego elementu w kodzie – najczęściej pojedynczej metody konkretnej klasy obiektu. Testy integracyjne mają zakres szerszy i sprawdzają współpracę między kilkoma obiektami.

W 2003 r. jeden z sygnatariuszy *Manifestu*, Kent Beck, uporządkował metodyki *test-first* i zaproponował rozwiązanie o nazwie *TDD (Test-Driven Development, ang. dosł. Programowanie Testowo-Sterowane)*. W tym samym roku Dan North zaproponował metodykę będącą de facto rozwinięciem *TDD* – Programowanie Metodą Behawioralną (*BDD – Behaviour-Driven Development*). Dan North uznał, że *TDD* nie jest dostatecznie intuicyjne⁷, nie istnieją wskazówki co testować, jak szczegółowo i kiedy uznać, że test

6 http://pl.wikipedia.org/wiki/Manifest_Agile - opis manifestu Agile na stronie Wikipedii (stan z dnia 12.07.2011)

7 <http://dannorth.net/introducing-bdd/> - wpis na blogu Dana Northa wprowadzający czytelników do nowej metodyki

zakończył się pełnym sukcesem. *BDD* miało rozwiązać ten problem – jego głównym założeniem jest tworzenie oprogramowania w oparciu nie o suche techniczne testy, ale „scenariusze” przewidujące możliwie jak najdokładniej wszelkie zachowania użytkownika aplikacji, zwłaszcza jego pomyłki i próby uzyskania niedozwolonego dostępu. Takie postępowanie wykracza poza ramy technicznego i technologicznego zarządzania aplikacją – duży nacisk kładzie na komunikację między osobami odpowiedzialnymi za projektowanie funkcjonalności (kierownikami projektu), a tymi odpowiadającymi za implementację (programistami). *BDD* w istocie jest filozofią, której celem jest kierowanie całością rozwoju programu, umożliwienie zleceniodawcom współpracy ze zleceniobiorcą przy całkowitym odrzuceniu sztywnych ram dokumentacji – jest więc kompleksowym rozwiązaniem, wpisującym się w wizję wytwarzania oprogramowania *Zwinnego*. Postuluje, wobec tego, o utworzenie języka pośredniego, między językiem biznesowym (dokumentacji, którą całkowicie odrzucono) i technicznym (który dla filozofii nie jest istotny).

W tym miejscu należy przywołać ponownie osobę Alana Kaya, naukowca odpowiedzialnego nie tylko za podstawy Programowania Obiektowego i pionierskie prace nad graficznymi interfejsami użytkownika. Kay znany jest również w kulturze popularnej jako autor powiedzenia: „Jedyny sposób by przewidzieć przyszłość, to ją wymyślić”, chociaż słowa te wypowiedziane zostały w odniesieniu do całej dziedziny nauki jaką jest informatyka, znakomicie wpisują się w ideę metodyki *BDD* – najlepszy sposób na przewidzenie zachowań użytkownika, odpowiedzi programu, ale również wymagań jakie w przyszłości użytkownik postawi aplikacji, to stworzenie scenariuszy dla każdej możliwości.

Pierwszym frameworkiem Dana Northa był stworzony dla języka Java *JBehave*, prace rozpoczął już w 2003 r.⁸. Mimo sukcesu (*JBehave* rozwijany jest do dzisiaj) North musiał uznać, że język Java nie jest najwygodniejszym do implementacji rozwiązań modelu behawioralnego – dowodem na to jest fakt, że wkrótce porzucił projekt, ale nie filozofię *BDD*. W 2005 r. zainspirowany projektem *RSpec*, platformą *BDD* w języku Ruby autorstwa Dave'a Astelsa, pracował już nad *RBehave*⁹. Chcąc uczynić z *RSpec* kompleksowe rozwiązanie połączył siły z Astelsem, Davidem Chelimskim i Aslakiem Hellesøyem – zgodnie z jego planami *RBehave* przestał istnieć jako osobny projekt i został wcielony do *Rspec*, powoli stającym się standardem w zakresie środowisk testowych platformy *Ruby on Rails* – mimo iż ten popularny framework domyślnie wyposażony jest we własne narzędzia testujące.

Norweg Hellesøy podjął później prace nad maksymalnym uproszczeniem procesu pisania testów i stworzeniem narzędzi, które umożliwiłyby tworzenie bardzo szczegółowych scenariuszy zachowania programu osobom, które nie mają wykształcenia

rozwijania aplikacji, opartej o testy behawioralne (stan z dnia 12.07.2011)

8 <http://edgibbs.com/2007/12/02/jbehave-and-rspec-history/> - Wpis na blogu Eda Gibbisa, będący odpowiedzią na błędną informację w książce „XUnit Test Patterns”, jakoby platformy do testowania behawioralnego aplikacji *Ruby on Rails* pojawiły się wcześniej niż analogiczne rozwiązania dla języka Java

9 <http://dannorth.net/2007/06/17/introducing-rbehave/> - wpis na blogu Dana Northa informujący o rozpoczęciu prac nad frameworkiem *RBehave*.

programistycznego. Tak ziścił się pomysł Northa o zaprojektowaniu języka pośredniego między biznesowym a implementacyjnym; powstał *Cucumber* (ang. ogórek), platforma, której głównym przeznaczeniem było generowanie odpowiedniej struktury testów, w odpowiedzi na scenariusze pisane w języku o nazwie *Gherkin* (ang. korniszon), który łądząco przypomina potoczny język ludzki, nie tylko angielski – reagować ma on na słowa kluczowe 40 języków, wspieranych przez *Cucumber*.

Metodyka *BDD*, choć zaproponowana w 2003 r. jest nadal bardzo młoda, dopiero od 2010 r. można mówić o dojrzewaniu tej idei, edytory tekstowe zaczynają wspierać rozwiązania *RSpec* i *Cucumber*, powstają też narzędzia dedykowane¹⁰. Technologia stabilizuje się, przestaje tak dynamicznie zmieniać, co owocuje powstaniem wyczerpujących dokumentacji i publikacji¹¹ poświęconych zarówno metodyce jak i narzędziom. Nie ma już żadnych technologicznych barier, które uniemożliwiałyby tworzenie testów i scenariuszy osobom nie biorącym udziału w pisaniu kodu.

Nie można jednak mówić o kolejnej nadchodzącej globalnej rewolucji, przede wszystkim dlatego, że *BDD* reprezentuje tylko jeden z wielu sposobów wdrożenia idei Programowania Zwinnego, a samo w sobie jest rozwinięciem pomysłu *TDD*. Inaczej niż w połowie lat 60ych XX wieku nie istnieje obecnie jeden ośrodek decyzyjny nadający kierunek całej nauce programowania. Z drugiej jednak strony trudno nazywać nowe techniki odłamami „lokalnymi” w dobie internetu, zwłaszcza w dziedzinie, która z tym medium jest tak silnie i naturalnie związana. Przyjęcie danej metodyki jest więc dyktowane tylko osobistymi upodobaniami zarządzających projektem.

¹⁰ <http://gherkineditor.codeplex.com/> - strona domowa edytora do tworzenia scenariuszy behawioralnych w języku *Gherkin*

¹¹ <http://pragprog.com/book/achbd/the-rspec-book> - strona książki „the RSpec Book” w witrynie wydawcy

4. Wprowadzenie do Behaviour Driven Development (BDD) Na przykładzie tandemu systemów *Rspec* i *Cucumber*

By móc mówić o metodyce BDD należy najpierw zrozumieć i zdefiniować jej podstawowe terminy, określające trzy kroki konstrukcji całego testu behawioralnego:

- Nadrzedną strukturą testu jest **funkcjonalność** (ang. *feature*). Określa ona ogólne zadanie postawione przed testowanymi elementami. Funkcjonalność powinno się móc nazwać jednym słowem bądź wyrażeniem, jak „uwierzytelnianie”, bądź „zarządzanie obrazami w profilu”.
- Wewnątrz funkcjonalności definiujemy **scenariusze** (ang. *scenario*). Powinny one przewidywać każdą praktycznie możliwą sytuację użycia funkcjonalności, zarówno poprawną, jak i wynikającą z ludzkiego błędu, czy chęci wymuszenia nieuprawnionego dostępu do zasobów aplikacji (np. atak typu *SQL injection*).
- Scenariusze składają się z **kroków** (ang. *step*). Każdy krok powinien być w miarę możliwości atomowy (tzn. odnosić się do jednej logicznie wyodrębnionej cechy aplikacji w danym momencie). Nieoficjalnie te trzy kroki (czy bardziej trzy „zestawy” kroków) określa się mianem *Given-When-Then* (ang. mając [mając dane lub mając do dyspozycji – przyp. autora]; kiedy [podejmuję akcję – przyp. autora]; wtedy [spodziewaj się odpowiedzi – przyp. autora]), połączenia słów kluczowych, używanych do ich opisu w językach domenowych BDD, jak *Gherkin*. Gdybyśmy chcieli tym krokom nadać bardziej oficjalne nazwy, odwzorowujące ich obowiązki, moglibyśmy rozpisać je tak:
 - **Warunki (początkowe)** – określają z jakiej perspektywy podchodzimy do scenariusza, kto jest podmiotem (nazywanym raczej „aktorem”) operacji i jakie ma uprawnienia. Jaki jest stan początkowy aplikacji, jakie dane są dostępne przed rozpoczęciem działania. Aktorem scenariusza może być użytkownik, bądź administrator aplikacji, ale również dowolny obiekt w programie, próbujący uzyskać dostęp do innego obiektu.
 - **Akcja** – jakie działanie jest podejmowane przez aktora w scenariuszu. Akcja może być bardzo szczegółowa, jak np. naciśnięcie konkretnego klawisza w konsoli bankomatu, może być też ogólna, jak np. próba dokonania dowolnego zapisu w bazie danych.
 - **Reakcja** – jaka jest spodziewana odpowiedź systemu na podjętą akcję, jakie działania temu towarzyszą: czy odpowiedź jest sygnalizowana potwierdzeniem, milczącą akceptacją działania aktora, zwróceniem informacji o błędzie, czy wreszcie zwykłym cofnięciem akcji.

Jakakolwiek rozbieżność między odpowiedzią spodziewaną, a otrzymaną w wyniku przeprowadzonej symulacji scenariusza, jest oznaczana jako test niezaliczony. Tylko w wypadku stwierdzenia błędu w formule scenariusza możliwe jest jego poprawienie. Zgodnie z ideą BDD scenariusz niejako zastępuje dokumentację, nie powinien więc być modyfikowany w trakcie tworzenia i rozwijania aplikacji, konieczna jest zatem zmiana (refaktoryzacja) kodu źródłowego, na którym symulacje scenariusza bazują, tak, aby przy założonych warunkach początkowych, określona akcja zakończyła się odpowiednią reakcją systemu.

Ogólny schemat postępowania podczas wytwarzania oprogramowania zgodnego zadaną funkcjonalnością behawioralną przedstawia się za pomocą poniższego grafu:

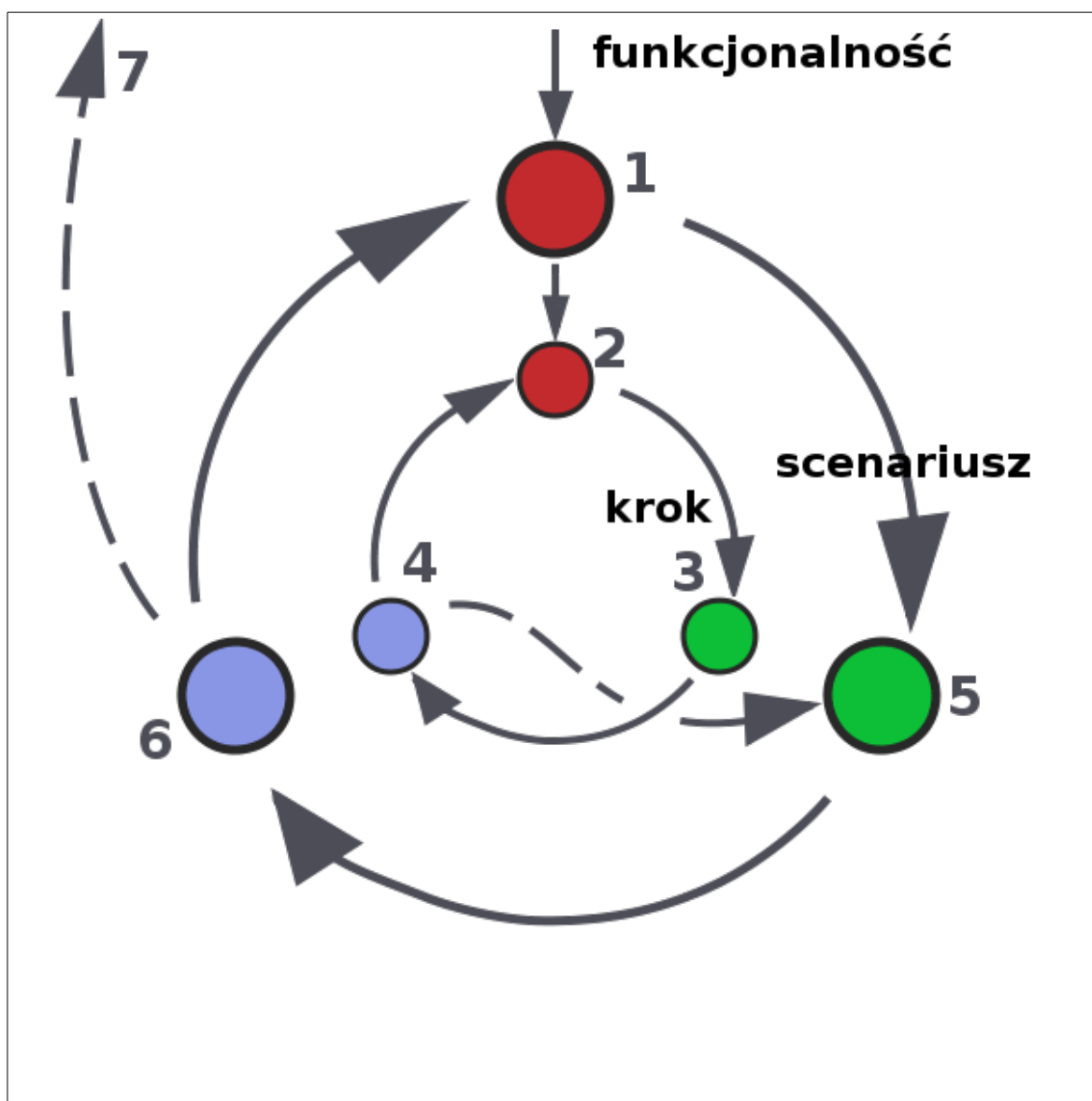


Fig. 1: Schemat refaktoryzacji kodu zgodnie z opisem funkcjonalności w technologii BDD.

Testujemy całość zadanej funkcjonalności, poprzez sprawdzanie poszczególnych scenariuszy:

1. Scenariusz poddajemy testowi; **jeżeli test wypadnie negatywnie** – a tak będzie w przypadku gdy kod aplikacji nie obsługuje, lub jest niezgodny z chociaż jednym krokiem tego scenariusza – rozpoczynamy testowanie poszczególnych kroków.
2. Zakładamy, że **aplikacja nie jest zgodna z danym krokiem scenariusza**. Przystępujemy do refaktoryzacji kodu.
3. Kod należy modyfikować tak długo, aż dany **krok scenariusza zostanie spełniony, a test wypadnie pomyślnie**.
4. Jeżeli nie jest to ostatni krok scenariusza, **kontynuujemy refaktoryzację w następnym kroku**.
5. Jeżeli był to ostatni krok scenariusza, to **całość jest zgodna z założeniami, a test wypadnie pomyślnie dla całego scenariusza**.
6. Jeżeli nie był to ostatni scenariusz funkcjonalności **kontynuujemy refaktoryzację w następnym scenariuszu**.
7. Jeżeli był to ostatni krok scenariusza, praca nad funkcjonalnością jest **zakończona**.

Testy zwyczajowo przeprowadzane są w testowym środowisku uruchomieniowym, co oznacza, że wszystkie zmiany i operacje są tymczasowe, bazy danych i pamięć programu są czyszczone po zakończeniu wszystkich symulacji, tak, by były przygotowane do przeprowadzenia kolejnych symulacji w takich samych, „sterylnych” warunkach. W przypadku aplikacji internetowej napisanej z użyciem frameworka *Ruby on Rails* mamy domyślnie do dyspozycji trzy środowiska uruchomieniowe: testowe (ang. *test*), rozwojowe (ang. *development*) i produkcyjne (ang. *production*). Rozumienie terminologii i stosowanie się do odpowiedniej składni jest wymogiem, do którego dostosować się muszą wszyscy uczestniczący w tworzeniu scenariuszy, to główny powód dla którego praktyka stojąca za metodologią BDD została uproszczona do niezbędnego minimum.

W przypadku zastosowania platformy *Ruby on Rails* i systemów *Rspec* i *Cucumber* istnieje oczywiście konieczność stosowania składni języka *Gherkin*. Pełną dokumentację i wprowadzenie można przeczytać na stronie <http://cukes.info/>. Spośród wielu przykładów (w różnych językach) poniżej zaprezentowano trzy części składowe przypadku zastosowania szablonu scenariusza w języku angielskim¹².

```
# language: en
Feature: Addition
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers
```

Fig. 2: Pierwsza część definicji przykładowej właściwości w języku *Gherkin*.

¹² <https://github.com/cucumber/cucumber/blob/master/examples/i18n/en/features/addition.feature> - zawartość pliku przykładowego *addition.feature* w serwisie github.

Powyżej przedstawiono definicję testowanej właściwości (funkcjonalności), na którą składa się wybór języka, nazwa funkcjonalności i krótki opis tłumaczący jej sens.

Następny fragment kodu przedstawia szablon scenariusza – konstrukcję, która umożliwia wykorzystanie raz napisanego testu akceptacyjnego w kilku różnych przypadkach z różnymi danymi wejściowymi:

```
Scenario Outline: Add two numbers
  Given I have entered <input_1> into the calculator
  And I have entered <input_2> into the calculator
  When I press <button>
  Then the result should be <output> on the screen
```

Fig. 3: Opis szablonu scenariusza w języku Gherkin.

Po zdefiniowaniu szablonu należy przekazać do niego serię zmiennych, na których może operować:

```
Examples:
| input_1 | input_2 | button | output |
| 20      | 30      | add    | 50      |
| 2       | 5       | add    | 7       |
| 0       | 40      | add    | 40      |
```

Fig. 4: Blok opisu serii danych dla testu akceptacyjnego.

Podane słowa kluczowe są słowami domyślnej, angielskiej lokalizacji. Listę odpowiadających im słów kluczowych w innej lokalizacji obsługiwanej przez *Gherkin* można uzyskać wpisując w terminalu systemu operacyjnego komendę:

```
cucumber --i18n <dwuliterowy_kod_języka>
```

Na przykład:

```
$ cucumber --i18n pl
| feature          | "Właściwość" |
| background       | "Założenia"  |
| scenario         | "Scenariusz" |
| scenario_outline | "Szablon scenariusza" |
| examples         | "Przykłady"  |
| given            | "* ", "Zakładając ", "Mając " |
| when             | "* ", "Jeżeli ", "Jeśli " |
| then             | "* ", "Wtedy " |
| and              | "* ", "Oraz ", "I " |
| but              | "* ", "Ale " |
| given (code)     | "Zakładając", "Mając" |
| when (code)      | "Jeżeli", "Jeśli" |
| then (code)      | "Wtedy" |
| and (code)       | "Oraz", "I" |
| but (code)       | "Ale" |
```

5. Aplikacje webowe, Krótkie wprowadzenie

Zaprezentowana w tej pracy metodologia behawioralna zostanie opatrzona przykładem w postaci opisu konstrukcji prostej aplikacji webowej. W erze Web2.0 aplikacja jest zupełnie innym tworem niż statyczna „witryna” internetowa. Mimo udostępnienia webowych baz danych poprzez wprowadzenie i ciągły rozwój takich technologii jak *PHP*, *asp.net*, *Python Django* i *Ruby on Rails*, wciąż najważniejszym narzędziem w budowaniu czy to statycznych dokumentów, czy rozbudowanych, interaktywnych aplikacji, jest język znaczników *HTML*, oparty na omawianej wcześniej technologii hipertekstowej; podstawowym protokołem komunikacji serwera z maszyną kliencką jest HTTP, a programem klienckim jest przeglądarka internetowa. Bazą wyjściową wszystkich tych technologii i frameworków webowych jest relacyjna baza danych typu SQL, stanowiąca serce aplikacji.

Wraz z ewolucją dokumentów do aplikacji rosło zagrożenie ze strony internetu dla przeglądarki, a przez nią – dla systemu operacyjnego użytkownika. Zagrożenie istniało też po drugiej stronie: w postaci użytkowników internetu szukających nieuprawnionego dostępu do danych poufnych – hakerów. Pierwszym „naturalnym odruchem” w ewolucji aplikacji webowych było stworzenie narzędzi do autoryzacji i autentykacji użytkownika. Protokół HTTP udostępnia dwa podstawowe rodzaje „żądań” jakie przeglądarka może wysyłać do serwera: GET - służące do pobierania danych z serwera wysyłane jest w celu pobrania każdego składnika dokumentu HTML, w tym pobrania samego dokumentu i zawartych w nim obrazów, oraz POST – używane kiedy klientka przeglądarka wysyła dane (w postaci parametrów HTTP) do serwera.

Teoretycznie żądanie POST nie podlega żadnym ograniczeniom; jest to łańcuch znaków wysyłany do serwera, który otrzymane dane parsuje i wykonuje na nich zadane operacje. Ta swoboda w dobieraniu parametrów oznacza jednak, że w praktyce poświęca się POSTowi wiele uwagi i programuje się serwer tak, by rozpoznawał i odrzucał szkodliwe parametry-tokeny, zanim przejdzie do wykonywania jakichkolwiek akcji na bazie danych. W ten sposób serwer broni się przed atakami iniekcyjnymi, wśród nich przed bardzo popularnymi, acz prymitywnymi atakami *SQL Injection*, które polegają na podaniu jako parametru fragmentu zapytania do bazy SQL do której serwer ma dostęp. Jeżeli warstwa aplikacyjna serwera nie potrafi rozpoznać zagrożenia i odrzucić takiego parametru, możliwe jest pośrednie wydawanie poleceń bazie danych, uzyskiwanie dostępu do danych wrażliwych, jak również usuwanie danych i modyfikowanie całych struktur bazy.

Wraz z rozwojem technologii przeglądarkowych pojawił się język Javascript, w skrócie zwany po prostu JS. Został stworzony jako *ECMAScript* przez programistę

Brendana Eich pod koniec 1995 r. przeszedł drobne zmiany i jako nowa implementacja, pod nową nazwą, został wprowadzony przez firmę Netscape w ich przeglądarkach w roku 1996. Również wtedy opracowano pierwszy standard *DOM* (ang. *Document Object Model*), technologii która umożliwia językowi Javascript manipulowanie obiektami składowymi (elementami) dokumentu HTML w przeglądarce użytkownika w czasie rzeczywistym.

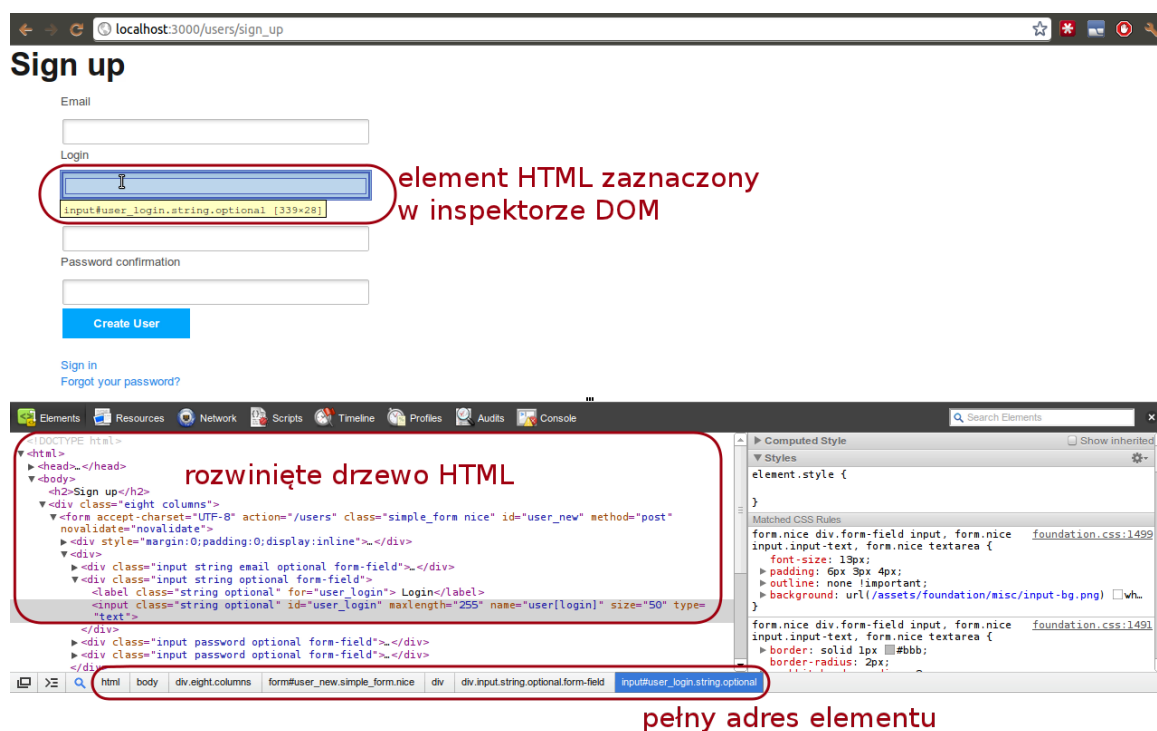


Fig. 5: Struktura dokumentu HTML widziana z perspektywy inspektora DOM (w przeglądarce Google Chrome)

Javascript to interpretowany język obiektowy, o dynamicznym typowaniu, który jako „zasób” dokumentu HTML importowany jest przez przeglądarkę w postaci pełnych skryptów w odpowiedzi na żądanie GET. Umożliwia autorom aplikacji tworzenie bardziej dynamicznych efektów i zachowań dokumentu HTML, uzyskując dostęp do przeglądarki klienckiej i traktując ją jako swoje środowisko. Może też być dodatkowym narzędziem do komunikacji klienta z serwerem – co prawda jego działanie również opiera się na żądaniach GET i POST, ale żądania te mogą być wysyłane w odpowiedzi na niemal dowolną akcję użytkownika przeglądarki, a nie tylko wywołanie hipertekstowego linku, jak to ma miejsce w statycznym dokumencie HTML.

W 2000 r. JavaScript jako narzędzie web-developerskie pojawił się w nowym wcieleniu: technologia *AJAX* sprawiła, że *JavaScript* mógł wysyłać żądania do serwera i odbierać jego odpowiedzi asynchronicznie, czyli niezależnie od żądań POST/GET wysyłanych przez użytkownika z poziomu głównego dokumentu HTML. W praktyce

każdy element dokumentu może komunikować się z serwerem dwustronnie, bez potrzeby korzystania z HTML-owych formularzy gdy coś wysyła i bez potrzeby odświeżania całego dokumentu, gdy coś odbiera.

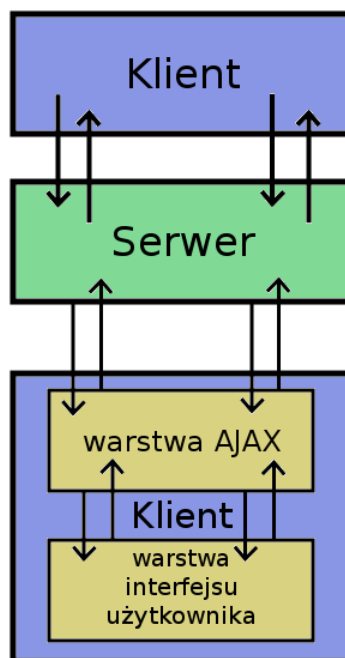


Fig. 6: Schemat komunikacji z użyciem technologii AJAX.

Pojawienie się *AJAXa* umożliwiło autorom budowanie aplikacji w pełni interaktywnych, zadziwiających dynamiką i ilością elementów które można obsłużyć „pozostając na tej samej stronie”, czy – jak nazwalibyśmy to już zgodnie z terminologią ery Web2.0 – „pozostając w tym samym widoku”. Jest oczywiście druga strona medalu takiego rozwiązania – nowy sposób pisania i obsługiwania aplikacji wymusił budowę szybszych i silniejszych serwerów, ponieważ ogromnie wzrosła ilość żądań wysyłanych przez takie aplikacje. Logicznym następstwem zwiększenia ilości połączeń klient-serwer jest wzrost zagrożenia jakie niesie ze sobą niezabezpieczenie tej komunikacji.

Kiedy w 1995 roku powstał język *PHP*, pierwsze narzędzie do tworzenia aplikacji internetowych godnych miana Web2.0, historia webowych narzędzi dopiero się rozpoczynała. W 2002 roku Microsoft wprowadził na internetowy rynek swoje rozwiązanie: *ASP.NET*, część większego frameworka *.NET* (wymawiane: „dot-net”), kompletne narzędzie programistyczne, oparte na uzupełniających się wzajemnie językach, sprawdzonym i zaufanym *Visual Basic* oraz nowoczesnym *C#* (czytane „C sharp”).

Dla niniejszej pracy najważniejszą technologią jest powstała w 2004 roku platforma *Ruby on Rails*, autorstwa duńskiego programisty Davida Heinemeiera Hanssona. Hansson jako podstawę swojego rozwiązania wykorzystał interpretowany język obiektowy *Ruby*.

Hansson wcale nie próbował stworzyć nowej technologii, a zaledwie napisać jedną aplikację na własny użytek; doszedł jednak do wniosku, że zastosowanie *Ruby* sprawiło, że aplikację rozwija się bardzo szybko i przyjemnie, twierdził, że pisanie kolejnych narzędzi ułatwiających dalszą pracę po prostu sprawiało radość¹³, przejrzystość kodu umożliwiała łatwe tworzenie dokumentacji. Z tego powodu Hansson zdecydował się na udostępnienie swojego rozwiązania w internecie pod nazwą *Ruby on Rails*.

13 <http://bigthink.com/ideas/21596> - Wywiad z Davidem Heinemeierem Hanssonem

6. Charakterystyka języka *Ruby* oraz platformy *Ruby on Rails*

Stworzony w 1995 roku przez japońskiego programistę, Yukihiro Matsumoto, język programowania *Ruby*, to interpretowany, dynamicznie typowany, w pełni obiektowy język skryptowy. Oparty na doświadczeniach programisty wyniesionych z pracy z innymi językami, jak np. *LISP*, *Perl* czy *Smalltalk*, posiada kilka unikalnych cech, które sprawiają że dla niektórych programistów *Ruby* stanowi osobną klasę w kategorii ergonomii rozwijania kodu:

Czytelna i przyjazna składnia – (ang. *syntax*) już na pierwszy rzut oka wyróżnia język *Ruby*, zwłaszcza wśród języków opierających swoją składnię na *C* (jak *C++*, *C#*, *Java* i *ActionScript*). Wyraźnie widać, że zaproponowane rozwiązanie ma przede wszystkim służyć programiście w swobodnym wyrażaniu swoich myśli:

```
exit unless "restaurant".include? "aura"
```

Wystarczy przeczytać na głos powyższy kod źródłowy, żeby – mając najbardziej podstawową znajomość języka angielskiego – zrozumieć jego znaczenie dla programu: algorytm zakończy działanie, chyba że w wyrazie „restaurant” znajduje się ciąg znaków „aura”. Wyrażenie warunkowe *unless* jest odwrotnością warunku *if*, dostępnego praktycznie w każdym języku wysokiego poziomu. Nawet początkujący programista powinien potrafić napisać dowolne wyrażenie warunkowe i jego zaprzeczenie, korzystając z praw DeMorgana. *Ruby* udostępnia jednak wyrażenie *unless* jako dogodność dla użytkownika, jego znaczenie dla realizacji kodu jest identyczne z zaprzeczeniem *if*, więc może wydawać się zbyteczne, ale w języku *Ruby* wygoda programisty jest stawiana na pierwszym miejscu.

Filozofia programowania, w której najważniejszy jest użytkownik języka, czyli autor programu jest obecna i ma wpływ na wszystkie rozwiązania przyjęte w *Ruby*.

Iteratory i bloki kodu – pozwalają na łatwe pisanie powtarzalnego i rekurencyjnego kodu. Tylko z pozoru są podobne do pętli znanych z innych języków, są jednak metodami, które przyjmują fragment kodu, otoczony nawiasami klamrowymi lub słowami kluczowymi *do* / *end*, jako argument:

```
10.times { puts "a"}
```

Wypisuje znak „a” dziesięciokrotnie, ponownie zwraca na siebie uwagę czytelna i komfortowa nazwa metody *times* sugerująca wykonanie czynności wielokrotnie.

Kod wywołujący metodę iteratora może do niej przekazać dane i tak:

```
10.times { |x| puts x }
```

...będzie przy każdym iteracyjnym kroku przekazywać licznik do bloku, jako zmienną *x*, każdy krok iteracyjny natomiast wypisuje od nowej linii wartość tej zmiennej, oczywiście zaczynając od zera.

W zależności od klasy obiektu dostępny jest odpowiedni zestaw iteratorów, dla liczb całkowitych i ich przedziałów¹⁴ dostępne są prostsze wyliczenia podobne do pętli *for* z innych języków, dla łańcuchów znaków dostępne są metody iterujące po każdym znaku, bądź każdym bajcie, tablice i tablice asocjacyjne (hasze) można przeszukiwać po każdym elemencie, po parach klucz/wartość, tylko po elementach, które spełniają podany w bloku kodu warunek, niektóre iteratory wykonują kod na pożądanym elementach, inne zwracają wynikową tablicę itd. Praktycznie dowolne powtarzalne zadanie, jakie chcielibyśmy wykonać przy pomocy pętli, w języku *Ruby* można zapisać prościej i krócej za pomocą odpowiedniego iteratora.

Naukę języka *Ruby* warto zacząć od lektury książki „*Ruby. Programowanie*”¹⁵, której autorem – obok Davida Flanagana, wykładowcy programowania – jest twórca języka, Yukihiro Matsumoto. Książka ukazała się w Polsce pod koniec 2008 r., tłumaczona przez wydawnictwo Helion. Opisuje ona przede wszystkim wersję języka 1.8, zawiera jednak dodatkowe informacje i przypisy dotyczące wersji 1.9, która wtedy była jeszcze niestabilną wersją rozwojową. Publikacja może wydawać się przestarzałą, ale zdaniem autora niniejszej pracy jest pozycją obowiązkową, ze względu na przystępny język i niewielką (w porównaniu z innymi książkami dostępnymi na rynku) objętość. Inną ważną pozycją może być opublikowana w sieci (dostępna za darmo) książka „*Why's (Poignant) Guide to Ruby*”¹⁶, autorstwa niezależnego programisty *Ruby*, nietuzinkowej postaci publicznej, mężczyzny znanego tylko pod pseudonimem *Why The Lucky Stiff*.

Język *Ruby* jest popularny wśród blogerów i samodzielnych twórców i autorów (freelancerów), w sieci można znaleźć odpowiedź na praktycznie każde, szczególnie zadane pytanie. Stroną, którą autor tej pracy szczególnie poleca, jest witryna *ApiDock* (<http://http://apidock.com/>), łącząca w sobie zalety przejrzystej dokumentacji różnych wersji języka *Ruby* (i opartych na nim technologii *Ruby on Rails* i *Rspec*) oraz internetowego forum, na którym użytkownicy dzielą się doświadczeniami dotyczącymi korzystania z konkretnych metod i rozwiązań.

14 Przedział liczbowy bądź znakowy w języku *Ruby* reprezentuje klasa *Range*, która może przechowywać liczby całkowite bądź znaki w ich „naturalnej” kolejności, np. (12..24) to wszystkie liczby całkowite od 12 do 24 włącznie, natomiast („a”..”n”) przechowuje wszystkie małe litery alfabetu w podanym zakresie

15 <http://helion.pl/ksiazki/ruby-programowanie-david-flanagan-yukihiro-matsumoto,rubypr.htm> - link do strony

16 <http://mislav.uniqpath.com/poignant-guide/> - mirror internetowego wydania książki *Why The Lucky Stiff*

System zarządzania pakietami *RubyGems* – integrujący się z powłoką systemu operacyjnego program ułatwiający instalowanie, usuwanie, uaktualnianie i przede wszystkim wykorzystywanie z poziomu kodu, bądź z poziomu systemu operacyjnego pakietów *Ruby*, zwanych *gemami* (ang. *gem* – klejnot). Pod szerokim pojęciem pakietu rozumiemy przede wszystkim rozliczne biblioteki i wtyczki, które można dołączyć i wykorzystać we własnym kodzie, ale niektóre pakiety są kompletnymi programami, interfejsami typu API, a nawet frameworkami – jak w przypadku *Ruby on Rails*, które integrują się z systemem operacyjnym i mogą pracować jako samodzielne aplikacje.

Ruby on Rails – platforma ułatwiająca pisanie aplikacji webowych przy pomocy języka *Ruby*, jest całym systemem wzajemnie powiązanych *gemów*. Poza własnym pakietem, zapewniającym integrację ze środowiskiem i powłoką, najważniejszymi jej częściami są:

- *Rack* – api serwera sieciowego, które od czasu powstania stało się standardowym rozwiązaniem wykorzystywanym w pisaniu programów serwerowych i klienckich (jest np. podstawą innego frameworka aplikacji webowych o nazwie *Sinatra*),
- Trzy biblioteki ułatwiające pracę z silnikami bazodanowymi *SQLite*, *MySQL* i *PostgreSQL* (platforma *Rails* wspiera te trzy silniki domyślnie, ale istnieją biblioteki umożliwiające pracę z innymi bazami, jak np. *Oracle*).
- *ActiveRecord* – *gem* udostępniający funkcjonalność ORM (ang. *Object-Relational Mapping* – mapowanie obiektowo-relacyjne), czyli mapowanie relacji z bazy danych do postaci obiektu języka *Ruby*; dzięki temu zaciągnięta z bazy danych „krotka¹⁷” – bądź ich tablica – wraz z całą zawartością, czyli wartością każdej kolumny, jest przez program traktowana jako obiekt odpowiedniej klasy, udostępniający metody wspólne dla wszystkich obiektów bazodanowych (np. czytanie danych, porównywanie, nadpisywanie) jak i metody napisane implícite dla danego rodzaju krotki (np. wiersz zawierający dane użytkownika, zaciągnięty z bazy danych i zmapowany wewnątrz aplikacji jako obiekt klasy *User*, może nabyć metody odpowiedzialne za logowanie i wylogowywanie danego użytkownika do aplikacji i zapisywanie niepoufnych danych w sesji przeglądarki).
- *Rake* – program automatyzujący procesy, będący w dużym uproszczeniu wersją programu *Make* dla języka *Ruby*. Jego najważniejszą funkcją jest wykonywanie predefiniowanych zadań (zwanych *Raketask*-ami) w przestrzeni roboczej aplikacji. Najczęściej kojarzony jest z wykonywaniem operacji na bazie danych, które nie są wynikiem działań użytkowników aplikacji (np. wykonywanie okresowych kopii zapasowych).

Poza cechami technologicznymi, istotna jest też architektura systemu oparta o schemat MVC (ang. *Model-View-Controller* – model-widok-kontroler). Ponieważ

¹⁷ krotką w żargonie określa się wiersz bądź tablicę wierszy zawierających dane, będących odpowiedzią na zapytanie wysłane do bazy

szczegółowy opis wszystkich cech tej architektury wykracza poza temat pracy, skupimy się na najważniejszych, które zakładają wyizolowanie logiki domenowej od części interfejsowej użytkownika, co – przy założeniu że każdy z trzech tytułowych elementów wykorzystywany jest zgodnie z przeznaczeniem (czego *Ruby on Rails* odgórnie nie narzuca) – pozwala na rozwijanie każdej części niezależnie, ułatwia proces testowania i usuwania błędów (debugowania), jak również stanowi podstawę tworzenia całych rodzin aplikacji, których rdzeń jest wspólny, a zmiany dokonuje się tylko w logice biznesowej.

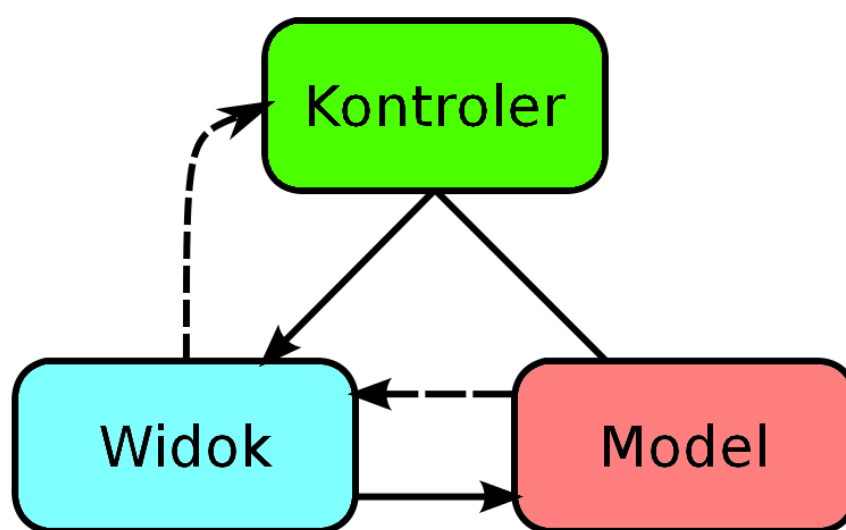


Fig. 7: Schemat architektury MVC

W zależności od implementacji architektury obowiązki jej elementów mogą rozkładać się bardzo różnie, dla *Ruby on Rails* przedstawiają się następująco:

Model – to klasa obiektu reprezentującego relacyjną krotkę bazodanowej. Każdy model w aplikacji *Rails* dziedziczy z klasy podstawowej udostępnionej przez omówiony wcześniej interfejs *ORM ActiveRecord*. Dzięki dziedziczeniu mamy dostęp do wszystkich najważniejszych metod związanych z manipulowaniem krotką w bazie danych za pomocą jej obiektu, ale również możemy opisać całą logikę jaką obiekt musi posiadać wewnątrz aplikacji.

Widok – reprezentacja interfejsu użytkownika. W aplikacji *Ruby on Rails* są to dokumenty html i ich szablony w formacie *eRuby* (ang. *embed Ruby* – zagnieżdżony *Ruby*) rozpoznawane po rozszerzeniu `*html.erb` bądź starszym `*.rhtml`. Pliki *eRuby* pozwalają na dopełnianie dokumentu html logiką języka *Ruby*, analogicznie do szablonów wykorzystywanych w technologiach *PHP*, *ASP* i *JSP*.

Widok w momencie wczytywania ma dostęp do zmiennych zdefiniowanych dla niego przez kontroler.

Kontroler – klasa zarządzająca „ruchem” w aplikacji, obsługuje żądania GET/POST wysyłane przez kliencką przeglądarkę. Spośród wymienionych elementów jest najważniejszym pośrednikiem między klientem a serwerem. Ze względów wydajnościowych, jak i ergonomii rozwijania aplikacji, każda funkcjonalność, dająca się logicznie wydzielić od pozostałych, powinna być reprezentowana w aplikacji przez własny kontroler. Standardem jest grupowanie metod obsługujących wszystkie zapytania manipulujące konkretnym modelem (często określane skrótem *CRUD*; ang. *Create-Read-Update-Destroy* – twórz-czytaj-nadpisuj-usuń), w obrębie jednego kontrolera. Jest to zgodne ze wzorcem architektury *REST* (ang. *Representational State Transfer* – transfer stanu reprezentacyjnego), której głównym aktorem jest „zasób” (ang. *resource*), reprezentujący obecny stan aplikacji w interfejsie użytkownika. Biorąc pod uwagę bezstanową naturę protokołu HTTP, zasób staje się ważnym medium biorącym udział w przenoszeniu przeglądowego stanu aplikacji z widoku do widoku, bądź w utrwalaniu stanu w bazie danych.

Jak opisano powyżej *Ruby on Rails* integruje się z powłoką środowiska w którym pracuje. Efektem takiej integracji jest udostępnianie zestawu komendy zwanych generatorami, które ułatwiają żmudne procesy zarządzania elementami składowymi aplikacji, takie jak budowanie nowego modelu czy kontrolera, Część gemów i wtyczek, przeznaczonych do pracy z aplikacją Rails dodaje swoje własne generatory.

7. Model Iteracyjno-Przyrostowy a metodyka BDD

Metodyka behawioralna, jako jedna z metodyk potomnych Zwinnego Modelu Projektowania, wykazuje analogie do innych metod używanych w ramach Projektowania Zwinnego. Przede wszystkim dobrze daje się zastosować w Zunifikowanym Procesie Projektowania¹⁸ (ang. *Unified Process*, w polskich publikacjach często stosuje się skrót *UP* pochodzący od angielskiego terminu) oraz w metodyce *Scrum*¹⁹, najczęściej stosowanymi procesami Modelu Iteracyjno-Przyrostowego (ang. *Iterative and incremental development*) a jej elementy można sprowadzić do szeroko stosowanych notacji języka UML jak Przypadki Użycia, czy Diagramy Interakcji / Komunikacji.

Ideą Modelu przyrostowego jest projektowanie i rozwijanie aplikacji w autonomicznych krokach, zwanych „iteracjami”, z których każdy ma za zadanie wprowadzić określoną na jego początku funkcjonalność w ścisłych ramach czasowych. Powstanie stabilnej wersji programu jest efektem przeprowadzenia określonej z góry ilości iteracji.

Każda z iteracji musi spełniać następujące wymagania:

- równy czas trwania pojedynczej iteracji; zazwyczaj jest to okres od jednego tygodnia do miesiąca.
- jasno określone wymagania i cele wprowadzanych zmian funkcjonalności i poprawek
- rozpoczęta iteracja jest tematem zamkniętym; wprowadzanie zmian – za wyjątkiem całkowitego zaniechania wykonania jednego lub części ze składających się na nią zadań – jest niemożliwe
- musi być kompletna; planowanie, budowa i poparcie testami danej funkcjonalności muszą zamknąć się w tej samej iteracji
- efektem jest stabilne wdrożenie; po przeprowadzeniu iteracji klient otrzymuje działającą aplikację, wzbogaconą o założone funkcjonalności

...oraz najważniejsze:

- termin wdrożenia jest nieprzekraczalny; w przypadku napotkania problemów jedyną możliwością jest zaniechanie części zmian wymaganych przez iterację i ewentualne przełożenie ich na iterację następną, wobec czego wdrożenie będzie

¹⁸ http://en.wikipedia.org/wiki/Unified_process - link do angielskiego artykułu o Zunifikowanym Procesie Projektowania

¹⁹ [http://en.wikipedia.org/wiki/Scrum_\(development\)](http://en.wikipedia.org/wiki/Scrum_(development)) - link do angielskiego artykułu o metodologii Scrum

uboższe o pominiętą zmianę, ale termin zostanie dotrzymany.

Metody *UP* (*Unified Process*) powstały w odpowiedzi na wprowadzony jeszcze w latach 60. Model Kaskadowy, który zakładał przystąpienie do pracy tylko po uprzednim skompletowaniu dokumentacji i rozplanowaniu wszystkich przypadków użycia. Analizy projektowania potwierdzały, że Model Kaskadowy z czasem przestał być wydajny. Trudno określić co dokładnie przesądziło o jego niepowodzeniu, ale na pewno wpływ na to miało pojawienie się nowych, dynamicznych narzędzi programistycznych, języków obiektowych i związanych z nimi metodyk projektowania obiektowego, które nie tylko radziły sobie z rozwijaniem programu w oparciu o ciągle zmieniające się wymagania, ale nawet zakładały, że taki właśnie sposób współpracy z klientem – formalniej: interesariuszem. Dodatkowo tysiące przeprowadzonych badań wykazały, że duża część funkcjonalności, zaprojektowanych w oparciu o Model Kaskadowy, jest nieużywana. Testy mówią o 45% funkcjonalności, które nie są wykorzystywane nigdy i o 19%, które wykorzystywane są rzadko.

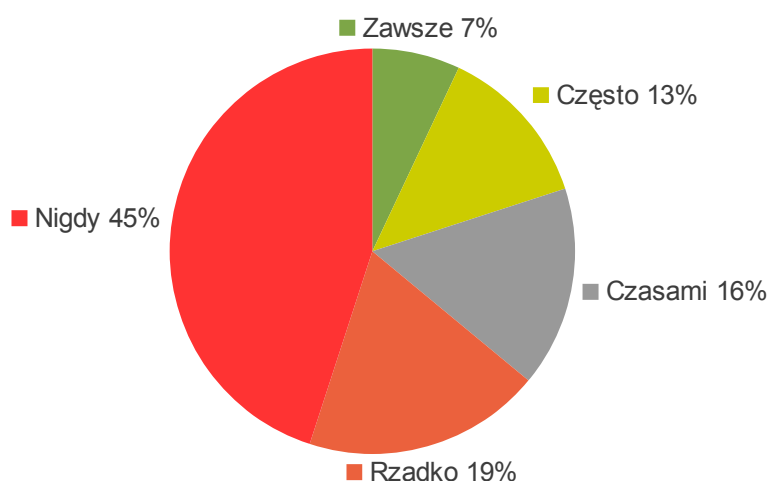


Fig. 8: Rzeczywiste użycie funkcjonalności zdefiniowanych w modelu kaskadowym (za: Craig Larman[2])

Prosty wniosek jest taki, że nie jest możliwe określenie wszystkich wymagań systemu zanim rozpoczną się prace i zanim chociaż jego część zostanie oddana interesariuszom i użytkownikom – co za tym idzie, czas wykorzystany na projektowanie i budowanie tych funkcjonalności można wykorzystać inaczej.

Ważnym elementem Modelu Iteracyjno-Przyrostowego jest rozróżnienie poszczególnych faz iteracyjnych, z których najłatwiejsza do wydzielenia, a zarazem najbardziej potrzebna jest faza „opracowywania”, podczas której określa się wymagania postawione projektowi, cel do którego projekt ma dążyć oraz serię kilku pierwszych iteracji, których efektem ma być dość podstawowa, ale funkcjonalna wersja programu. Ta grupa iteracji zwanych iteracjami podstawowymi bądź początkowymi, to jedyny

formalny przypadek, kiedy iteracje odbiegają od zasady, mówiącej, że każda iteracja kończy się wdrożeniem. Iteracje początkowe, zanim zespół przystąpi do budowania zaplanowanej aplikacji, muszą jeszcze określić technologię i wymagania sprzętowe, wyznaczając tym samym punkt wyjścia dla wszystkich następnych iteracji, są więc iteracjami najważniejszymi z punktu widzenia kształtu projektu zarządzanego zwinnie.

Nawiązując do tej teorii w następnych rozdziałach zaprojektowana i zbudowana zostanie aplikacja korzystająca z uproszczonego Modelu Iteracyjno-Przyrostowego, wykorzystując technologie związane z metodyką BDD: *Cucumber* i *RSpec*.

8. Budowa aplikacji według modelu *BDD*

Iteracja 0:

8.0. Cel projektu i wymagania aplikacji

Opis aplikacji, nawet w iteracjach początkowych, powinien być tworzony zgodnie z paradygmatem zwinnego projektowania, co oznacza, że musi być dość zwięzły, konkretny co do celów projektu, ale bardzo ogólny co do środków, a nawet nie mówiący o nich nic.

Celem projektu jest utworzenie demonstracyjnej aplikacji webowej, której głównym elementem ma być forum internetowe. Konieczne jest utworzenie modułu uwierzytelniania i autoryzowania użytkowników.

Zadania:

1. *Zidentyfikować potrzebne technologie*
2. *Przygotować system operacyjny serwera pod obsługę aplikacji*
3. *Zadbać o niezmiennosć technologii będących podstawą działania aplikacji*

Ten ogólny opis projektu jest jak najbardziej dozwolony, daje już asumpt do wybrania odpowiedniej technologii. Ponieważ wszystkie zainteresowane strony projektu: interesariusze, menadżerzy projektu i programiści zdają sobie doskonale sprawę, że to dopiero pierwszy krok. Wykonawcom projektu logika i doświadczenie każe spodziewać się kolejnych bardzo „standardowych” funkcjonalności zleczanych przez interesariusza, jak możliwość edycji i usuwania postów, załączania plików do postów, formatowania wypowiedzi za pomocą kodu HTML – wszystko to zostanie rozpisane na kilka początkowych iteracji projektu i dołączone do pierwszego funkcjonalnego wdrożenia.

Iteracja zerowa ma za zadanie wyłonić odpowiednią technologię i narzędzia spełniające postawione wymagania. Wszystkie prace przygotowawcze w metodykach zwinnych mają służyć jak najszybszemu przejściu do programowania aplikacji. O kształcie dodawanych funkcjonalności i technologii w nich wykorzystanych decyduje się na bieżąco, co oznacza, że wybór dominującej technologii i sposobu jej wersjonowania ma ogromne znaczenie. W mojej aplikacji korzystam, jak wielokrotnie wspominałem, z frameworka *Ruby on Rails* (w skrócie *RoR*) konkretnie w wersji 3.1. W wersji tej standardem jest zintegrowanie ze środowiskiem *Rails* interpretera języka *Coffescript*. Jest to język transkompilowany do czystej postaci języka JavaScript. Powstał w celu zwiększenia czytelności kodu i możliwości przetwarzania obiektów. Ponieważ efektem wynikowym kompilowania plików *Coffee* jest poprawny JavaScript, jest to technologia w pełni zgodna z najczęściej wykorzystywanymi bibliotekami Javascript jak *jQuery* czy

Backbone. Inną nowością wprowadzoną w wersji 3.1. platformy *RoR*, jest domyślne wykorzystywanie *Sass*, domenowego języka prezentacji dokumentów HTML. *Sass* jest dla CSS tym, czym *Coffeescript* dla *JavaScriptu* – metajęzykiem kompilowanym do innego języka, stworzonym w celu zwiększenia czytelności i użyteczności technologii zaufanej i rozwijanej od wielu lat.

Uzbrojeni w taką wiedzę możemy być spokojni, że wybór dynamicznie rozwijającej się platformy opartej o nowoczesny język *Ruby*, w pełni kompatybilnej ze standardowymi technologiami, a nawet rozszerzającej ich możliwości, jest wyborem trafionym i nie przysporzy problemów w następnych iteracjach.

Należy też pamiętać o tym, że siła platformy *Rails* wynika ze współpracy wielu różnych technologii, z których każda może być rozwijana w innym tempie. Najlepszym rozwiązaniem zapewniającym stabilność wersji wszystkich użytych gemów jest wspieranie się programami *RVM* i *Bundler*. *RVM* (*Ruby Version Manager*) umożliwia instalację wielu wersji interpretera języka *Ruby* i używania wybranej w zależności od projektu. Każda osobno zdefiniowana wersja ma też własny system *Rubygems*, a co za tym idzie – własny zestaw dodatkowych narzędzi. Na poziomie pojedynczego projektu natomiast, to właśnie wymieniony wcześniej *Bundler* dba o korzystanie z określonych w pliku konfiguracyjnym odpowiednich wersji gemów. W przypadku nowszych wersji aplikacji *Ruby on Rails* (od wersji 3.0 w górę) korzystanie z tego systemu jest wymagane, musi być więc on dostępny dla danej wersji interpretera przed wywołaniem generatorów *Rails*.

Ostatecznie dobór najbardziej potrzebnych, startowych technologii, przedstawia się następująco:

- *Interpreter języka Ruby - MRI Ruby 1.9.2*
- *System zarządzania wersjami interpretera języka Ruby - RVM 1.6.31*
- *System zarządzania dodatkami języka Ruby - RubyGems 1.8.6*
- *Platforma do tworzenia aplikacji webowych - Ruby on Rails 3.1*
- *System zarządzania dodatkami projektów Ruby - Bundler 1.0.21*
- *Platforma testów behawioralnych - RSpec 2.7.0*
- *Generator testów behawioralnych - Cucumber 1.1.1*
- *Biblioteka skryptów Javascript - jQuery 1.3.2*

Ważnym elementem projektu jest też dobór systemu wersjonowania oprogramowania. O wyborze technologii zadecyduje przede wszystkim model repozytorium: scentralizowany lub rozproszony. W ostatnich latach dużą popularność zdobywają systemy rozproszone, a zwłaszcza program *Git*, autorstwa Linusa Torvaldsa. Rozproszenie repozytorium zwiększa bezpieczeństwo projektu poprzez pozbycie się problemu jednego, centralnego repozytorium, którego utrata bądź awaria, oznaczać może całkowity upadek projektu; ze względu na to repozytoria centralne rutynowo zabezpiecza się kopiami zapasowymi. W przypadku repozytorium rozproszonego każdy pracujący z repozytorium przechowuje lokalną historię całego projektu, na podstawie której można

odtworzyć mniejszą, bądź większą część oprogramowania w razie wypadku. Inną ważną zaletą jest możliwość pracy i kontynuowania wersjonowania własnych postępów na poziomie lokalnym nawet w sytuacji utraty połączenia sieciowego ze współpracownikami. Pracę można normalnie wykonywać na maszynie lokalnej, a lokalnymi zmianami wymienić się z resztą zespołu w dowolnej chwili gdy połączenie zostanie nawiązane. System *Git* posiada tę właściwość, że zamiast zapisywać kolejne wersje plików, zapisuje skompresowaną historię zmian, wraz ze wszystkimi rozgałęzieniami repozytorium, dzięki czemu jego repozytoria są mniej obszerne, a zaciąganie i wysyłanie zmian trwa krócej niż w przypadku innego popularnego systemu *SubVersion*, głównego konkurenta *Git*. Na koniec należy wspomnieć o serwisie *Github*²⁰, który umożliwia darmowe tworzenie publicznych repozytoriów, które umieszcza się w sieci, udostępnia do tego wiele niezwykle przydatnych narzędzi. Takie rozwiązanie ogromnie ułatwia zespołową pracę nad małymi, jawnymi projektami.

Wobec tego do wersjonowania niniejszego projektu będę używał programu *Git*. W przypadku powiększenia zespołu wprowadzenie nowego członka w historię rozwoju aplikacji jest bardzo ułatwione właśnie dzięki aplikacji *Github*, mogę też dzielić się problemami i rozwiązaniami ze społecznością użytkowników serwisu, jak i spoza niego. Tak długo jak repozytorium jest publiczne, dostęp do jego odczytu ma każdy.

W związku z postawionymi zadaniami, wynikiem przeprowadzenia tej iteracji powinien być serwer z zainstalowanym programem *RVM* (wraz z odpowiednią wersją interpretera *Ruby* i gemem *Bundler* dla tej wersji) oraz systemem wersjonującym *Git*.

20 www.github.com

Iteracja 1:

8.1. Przygotowanie platformy

Zadania:

1. Przygotować platformę języka Ruby pod aplikację Ruby on Rails w wersji 3.1
2. Przygotować repozytorium GIT
3. Zadbać o niezmiennosć wersji użytych technologii (zamrożenie za pomocą systemu „Bundler”)

Mając do dyspozycji przygotowany system operacyjny serwera, z działającym silnikiem bazodanowym MySQL, należy zadbać odpowiednią wersję oprogramowania. Do języka Ruby zaleca się stosować system zarządzania wersjami interpretera Ruby – RVM. Instrukcje instalacji dostępne są na stronie produktu (<http://beginrescueend.com/>), w przypadku powłoki shell wystarczy użyć skryptu:

```
$ bash -s stable < <(curl -s  
https://raw.github.com/wayneeseguin/rvm/master/binscripts/rvm-installer)
```

Po instalacji potrzebne jest ściągnięcie i przekompilowanie odpowiedniej wersji interpretera. RVM robi to automatycznie po wywołaniu polecenia:

```
$ rvm install ruby-1.9.2-p290-patched
```

System operacyjny musi tymczasowo zostać przełączony na korzystanie z tej wersji interpretera:

```
$ rvm use ruby-1.9.2-p290-patched
```

Warto od razu w katalogu projektu przygotować plik `.rvmrc`, w którym wstawiamy linijkę `rvm use ruby-1.9.2-p290-patched` dzięki czemu wchodząc do katalogu aplikacji zawsze przełączymy się na odpowiednią wersję interpretera i otrzymamy dostęp do odpowiednich gemów. Następną rzeczą jest instalacja platformy *Ruby on Rails* i systemu Bundler, dołączenie ich do repozytorium dodatków (gemów) związanego z konkretną wersją interpretera:

```
$ gem install rails -v=3.1.0  
$ gem install bundler
```

Jak opisano wcześniej platforma *Rails* integruje się z systemem operacyjnym, udostępniając komendy zwane generatorami, jeden z nich zajmuje się automatycznym procesem przygotowania nowej aplikacji *Ruby on Rails*:

```
$ rails new mgr
```

Generator stworzy pustą aplikację przygotowaną pod działanie najnowszej, zainstalowanej wersji frameworka.

Następnym ważnym krokiem jest zadbanie o repozytorium gemów, wykorzystywanych przez daną aplikację. Do tego służy wygodny *Bundler*, a jego punktem odniesienia, zawierającym informacje o bibliotekach które ma dla danego projektu obsługiwać, jest plik *Gemfile* w głównym katalogu aplikacji. W przypadku świeżego projektu jego zawartość przedstawia się następująco:

```
source 'http://rubygems.org'

gem 'rails', '3.1.0'

# Bundle edge Rails instead:
# gem 'rails', :git => 'git://github.com/rails/rails.git'

gem 'sqlite3'

# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails', "~> 3.1.0"
  gem 'coffee-rails', "~> 3.1.0"
  gem 'uglifier'
end

gem 'jquery-rails'

# Use unicorn as the web server
# gem 'unicorn'

# Deploy with Capistrano
# gem 'capistrano'

# To use debugger
# gem 'ruby-debug19', :require => 'ruby-debug'

group :test do
  # Pretty printed test output
  gem 'turn', :require => false
end
```

Fig. 9: Początkowa zawartość pliku *Gemfile*

Definiowanie odpowiedniego gema jest banalne i zawiera się w jednej linii, co widać na przykładzie:

```
gem 'rails', '3.1.0'
```

Linijka zawiera informację o nazwie biblioteki z której korzystamy i konkretnej wersji – jeżeli chcemy taką wymusić. W przypadku dodania nowego wpisu bez podawania wersji oprogramowania, uruchomienie *Bundlera* (polecenie `bundle install` wydane w katalogu głównym aplikacji) automatycznie ściągnie najnowszą wersję gema, doda ją do repozytorium używanego obecnie interpretera i podłączy do niej aplikację. Dodatkowo definiować można grupy użycia gemów, dzięki czemu biblioteki służące do testowania możemy wykorzystywać tylko w środowiskach deweloperskich i testowych, przez to wyraźnie „odchudzając” środowisko produkcyjne.

Lista Gemfile zawiera tylko „docelowe” gemy; jeśli te technologie wymagają wcześniej instalacji innych gemów, od których zależą, *Bundler* zajmie się tym samoczynnie. Można wyświetlić listę aktualnie używanych gemów wraz z ich wersjami wywołując komendę `bundle list` wewnątrz katalogu projektu.

Bundler porówna listę obecnie używanych gemów z tym plikiem, zsynchronizuje projekt, a jednym z efektów jego działania będzie stworzenie w katalogu głównym, bądź odświeżenie, pliku `Gemfile.lock` – który zawiera dokładniejsze informacje o gemach i ich zależnościach. To właśnie ten plik jest dla serwera istotniejszy i służy jako referencja co i gdzie znaleźć.

Na potrzeby naszego projektu dodaje się *Bundlerowi* nowe gemy, pozbywając się wszystkich wykomentowanych linii i domyślnego adaptera bazodanowego `'sqlite3'`, w zamian za adapter `'mysql2'`, większość pozostałych gemów jest nowa, wyjątek stanowią przede wszystkim gemy zdefiniowane w grupie `:assets`, które są wykorzystywane do tworzenia połączenia aplikacji z jej statycznymi zasobami (tzw. *Asset Pipeline*), jak grafika, dźwięk i style kaskadowe w plikach `*.scss`. Plik Gemfile po modyfikacjach wygląda następująco:

```
source 'http://rubygems.org'

gem 'rails', '3.1.0'
gem 'jquery-rails'
gem 'mysql2'

gem 'simple_form', '1.5.2'
gem 'haml-rails', '0.3.4'
gem 'pry', '0.9.7.3'
gem 'devise', '1.4.9'
gem 'cancan', '1.6.7'
gem 'zurb-foundation', '2.0.2'
```

```

group :development, :test do
  gem 'rspec-rails' , '2.7.0'
  gem 'factory_girl_rails' , '1.3.0'
  gem 'cucumber-rails' , '1.0.2'
  gem 'capybara' , '1.1.1'
end

group :assets do
  gem 'sass-rails', " ~> 3.1.0"
  gem 'coffee-rails', "~> 3.1.0"
  gem 'uglifier'
end

group :test do
  gem 'turn', :require => false
end

```

Podawanie konkretnych wersji nie jest wymagane, podałem je powyżej, na wypadek gdyby ktoś chciał prześledzić całą budowę aplikacji w przyszłości, gdy wiele z tych gemów będzie miało nowsze wersje, niekoniecznie kompatybilne.

Uzasadniając wybór gemów, należy przedstawić za jakie elementy działania aplikacji odpowiadają:

- `simple_form` – umożliwia prostsze i bardziej eleganckie tworzenie formularzy *HTML* w widokach aplikacji, jest w pełni kompatybilny z systemem zasobów (ang. *resources*) platformy Ruby on Rails.
- `haml-rails` – umożliwia korzystanie z języka znaczników *Haml* w miejsce *HTML*. Zdaniem autora *Haml* jest czytelniejszy, zwłaszcza kiedy kiedy kod *HTML* wymieszany jest z zagnieżdżonym kodem *Ruby* (znaczniki *erb*).
- `pry` – alternatywa dla programu *IRB* (interaktywna powłoka interpretera *Ruby*), która pozwala na dokładniejsze testowanie kodu i udostępnia czytelniejszy wydruk wyniku działań interpretera.
- `devise` – biblioteka upraszczająca zarządzanie procesem uwierzytelniania użytkownika, udostępnia własny generator i dość ogólne widoki zajmujące się rejestrowaniem, logowaniem i przypominaniem hasła użytkownika.
- `cancan` – biblioteka upraszczająca zarządzanie procesem autoryzacji, napisana z myślą o współpracy z gemem *Devise*. Cały system zarządzania uprawnieniami można dzięki niej sprowadzić do jednej prostej klasy.
- `zurb-foundation` – biblioteka udostępniająca zestaw wizualnych stylów, ułatwiający tym samym tworzenie zadowalającej wizualnie kompozycji widoku.

Pozostałe gemy, wymienione w grupie przeznaczonej tylko do rozwoju i testów, są niezbędnymi elementami (choć istnieje możliwość skorzystania z innych technologii w ich miejsce) wykorzystywanymi w metodyce *BDD*:

- `rspec-rails` – to implementacja technologii *RSpec* dla środowiska *Ruby on Rails*.
- `factory_girl_rails` – gem umożliwiający tworzenie testowych obiektów wykorzystywanych w testach automatycznych aplikacji. za pomocą tzw. Fabryk, obiektów, które potrafią tworzyć instancje modeli wg podanego schematu, z danymi potrzebnymi do wykonania konkretnego testu
- `cucumber-rails` – implementacja systemu *Cucumber* dla platformy *Ruby on Rails*
- `capybara` – „symulator” przeglądarki internetowej, zdolny wysyłać żądania HTTP do serwera, i przetwarzać odpowiedzi, dzięki czemu może bardzo szybko sprawdzać zgodność odpowiedzi serwera z oczekiwaniami w teście

Mając tak uporządkowany plik `Gemfile` możemy przystąpić do instalacji i podłączenia wymaganych gemów poprzez polecenie `bundle install`. Następnie trzeba sprawdzić, czy gemy te nie wymagają dodatkowych kroków w celu lepszej integracji ze środowiskiem. W tym wypadku wszystkie gemy wymagają tylko skorzystania z wygodnych generatorów Rails. Polecenie `rails g` wyświetli listę dostępnych generatorów, wśród których zobaczyć można:

(...)

Cucumber:

```
cucumber:install
cucumber:feature
```

Devise:

```
devise
devise:form_for
devise:install
devise:shared_views
devise:simple_form_for
devise:views
```

(...)

FactoryGirl:

```
factory_girl:model
```

Foundation:

```
foundation:install
foundation:layout
```

Jquery:

```
jquery:install
```

(...)

Rspec:

```
rspec:install
```

```
SimpleForm:
  simple_form:install

(...)
```

Należy po kolei odpalić generatory dla tych gemów (korzystając z polecenia `rails g cucumber:install` i analogicznie wywołanie `rails g nazwa-gema:install` dla wszystkich pozostałych).

Ważnym krokiem będzie skonfigurowanie bazy danych w pliku `config/database.yml`, jest to niezwykle prosta konfiguracja, w przypadku korzystania z serwera *MySQL* musimy mieć znajomość do konta użytkownika uprawnionego do modyfikowania schematu bazy (prawa administratora bazy). Na potrzeby tego projektu konfiguracja wygląda następująco:

```
development:
  adapter: mysql2
  encoding: utf8
  database: mgr_dev
  username: nazwa_użytkownika_bazy_daych_mysql
  password: hasło_użytkownika_bazy_daych_mysql

test: &test
  adapter: mysql2
  encoding: utf8
  database: mgr_test
  username: nazwa_użytkownika_bazy_daych_mysql
  password: hasło_użytkownika_bazy_daych_mysql

cucumber:
  <<: *test
```

rys. 10: Zawartość pliku `config/database.yml`

Polecenie `rake db:create` utworzy w bazie danych odpowiednie bazy.

Ostatnim ruchem powinno być utworzenie strony startowej (root) w aplikacji, np. poprzez wykonanie `rails g controller home index`, (co oznacza wydanie polecenia programowi rails by wygenerował kontroler o nazwie home i utworzył dla niego automatycznie widok index, można podać).

By ten widok był uznawany za widok root, należy o tym poinformować *Railsowy* router, poprzez dodanie:

```
root :to => 'home#index'
```

w pliku `config/routes.rb`, a na koniec usunięcie pliku `public/index.html`. Dopiero brak tego pliku spowoduje, że w odpowiedzi na pytanie o stronę root, serwer Railsów odwoła się do wprowadzonej przed chwilą linii.

Jeżeli aplikacja bezbłędnie uruchamia serwer (polecenie `rails s` wywołane

w głównym katalogu aplikacji uruchamia domyślny serwer HTTP aplikacji, dla świeżej instalacji platformy *Ruby on Rails* jest to program *Webrick*, możliwa jest też instalacja – jako gem zainstalowany za pomocą *Bundlera* – serwerów *Mongrel* bądź *Thin*), należy utworzyć pierwszą rewizję kodu w wybranym systemie wersjonowania (w przypadku użycia systemu wersjonowania Git wszystkie informacje o tym jak tego dokonać, można znaleźć na stronie <http://help.github.com/create-a-repo/>). Zgodnie z dobrymi praktykami programowania pierwsza rewizja kodu powinna być pustą, działającą aplikacją na przygotowanym koncie użytkownika systemu operacyjnego.

Iteracja 2:

8.2. Uwierzytelnianie

Zadania:

1. *Stworzyć system uwierzytelniania użytkownika za pomocą gema Devise*

Podstawowym elementem – a na obecnym poziomie rozwoju aplikacji webowych wymaganym – bezpieczeństwa aplikacji webowej jest identyfikacja użytkowników. W przypadku kiedy umożliwiamy użytkownikom umieszczanie różnego typu treści w bazie danych zagrożenie ze strony szkodliwej zawartości wzrasta, zwłaszcza jeśli użytkownik może w tekście zawierać linki lub fragmenty kodu – aplikacja staje się podatna na ataki typu *injection*. Zabezpieczanie się przed atakami tego typu nie jest wyjątkowo trudne, należy do „kanonu” bezpieczeństwa aplikacji. Dodatkowo zapewnia administratorowi możliwość zidentyfikowania użytkownika, który ataku dokonał (lub w inny sposób naruszył regulamin użytkowania aplikacji). Wreszcie umieszczanie danych bez podawania autora, od których dane te pochodzą, jest nieodpowiedzialne i nie napawa współużytkowników serwisu zaufaniem. Wobec tego chodzi nie tylko o identyfikację użytkownika, ale o jego uwierzytelnianie, a logicznym rozwinięciem jest uwierzytelniania jest autoryzacja użytkowników. Obie te funkcjonalności obsłużymy łatwo za pomocą gemów Devise oraz Cancan.

Ze względu na to, że są to gotowe rozwiązania, nie mamy możliwości dokładnego prześledzenia procesu wytwarzania kodu zgodnie z etapami testów behawioralnych, możemy jednak w prosty sposób utworzyć test sprawdzający zgodność z założeniami.

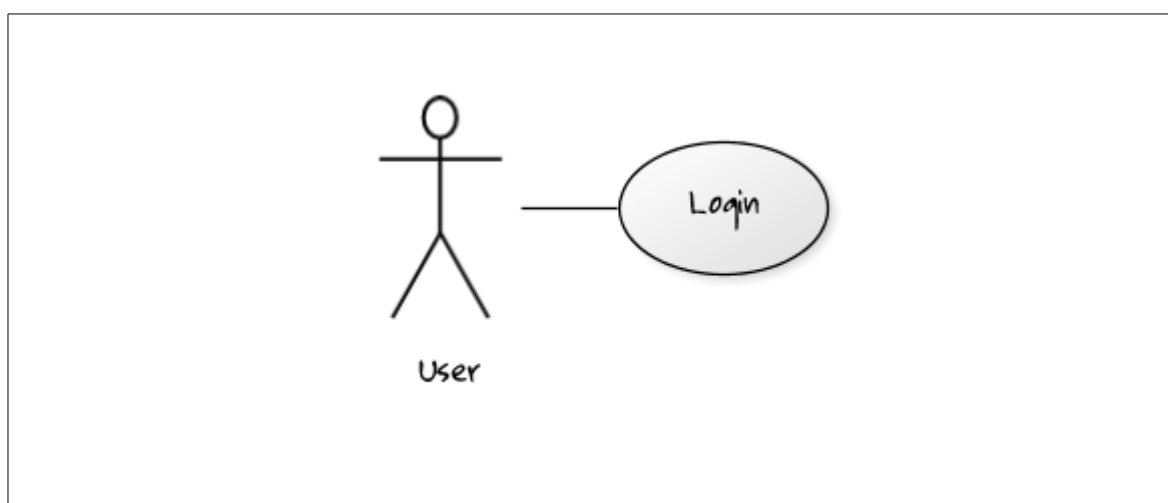


Fig. 11: Pojedynczy przypadek użycia prostego systemu logowania

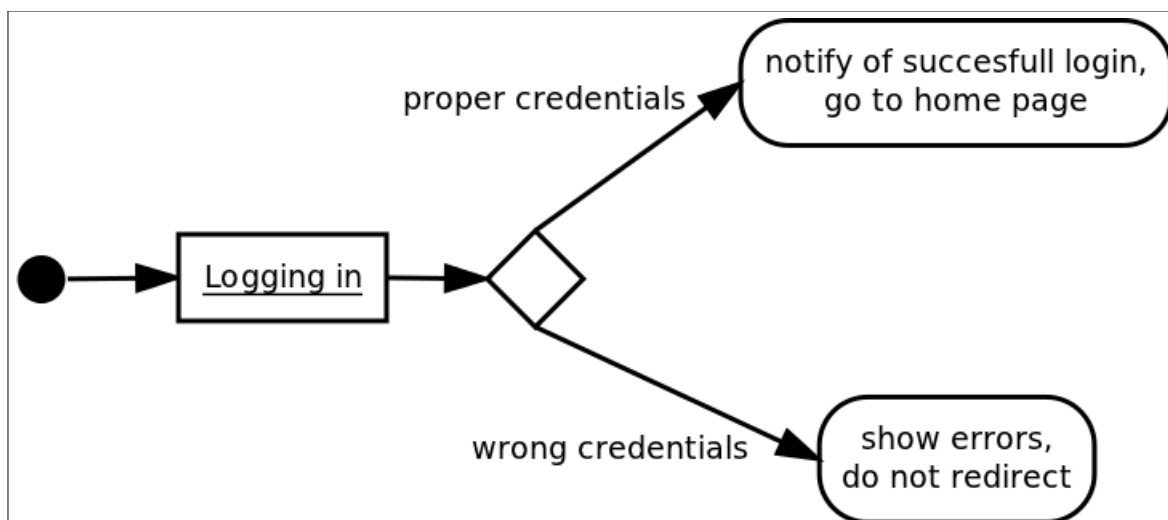


Fig. 12: Prosty diagram przepływu funkcjonalności logowania

Zasada działania jest prosta, ale dodatkowo chcielibyśmy wprowadzić możliwość logowania się za pomocą nazwy użytkownika lub adresu e-mail, tak by system przyjmował dowolną z tych wartości i działał poprawnie.

Następnie działania należy wykonywać w tej kolejności:

1. *Utworzyć model użytkownika*
2. *Utworzyć generator testowych obiektów (tzw. fabrykę) dla modelu użytkownika*
3. *Utworzyć test funkcjonalności*
4. *Refaktoryzować kod aż do całkowitej akceptacji*

Modele w aplikacji Rails buduje się korzystając z domyślnych generatorów, ale w naszym przypadku chcemy skorzystać z gema Devise, który udostępnia własny generator, automatycznie zajmując się stworzeniem nie tylko samego modelu, ale też przypisania mu odpowiednich akcji w kontrolerze i spreparowaniem odpowiednich ścieżek do tych akcji. Zajmie się też indeksowaniem kluczy w bazie danych, haszowaniem hasła, a do wszystkiego tego dołączy jeszcze możliwość odzyskiwania hasła poprzez e-mail. Słowem – naprawdę nie wypada nie skorzystać.

```
rails g devise user
```

`rails g` wywołuje jeden z podstawowych skryptów platformy *Rails* – skrypt generatora, którego argumentami są: nazwa generatora i jego parametry. Powyższa komenda użyje generatora *Devise*, by utworzyć model o nazwie *User*, dodatkowo stworzy jeszcze migrację, czyli plik konstruujący odpowiednie zapytanie do bazy danych, w celu utworzenia na niej tabeli odpowiadającej modelowi. Jednak domyślnie utworzona migracja nie odpowiada w pełni naszym założeniom, ponieważ udostępnia tylko i wyłącznie identyfikację użytkownika za pomocą adresu e-mail. W trosce o ergonomię serwisu chcielibyśmy dać użytkownikom możliwość logowania za pomocą *loginu*. Musimy

zmodyfikować skrypt migracji do następującej postaci (dodaną liniijkę wyróżniono poprzez pogrubienie):

```
class DeviseCreateUsers < ActiveRecord::Migration
  def self.up
    create_table(:users) do |t|
      t.database_authenticatable :null => false
      t.recoverable
      t.rememberable
      t.trackable
      t.timestamps
    end

    add_index :users, :email, :unique => true
    add_index :users, :login, :unique => true
    add_index :users, :reset_password_token, :unique => true
  end

  def self.down
    drop_table :users
  end
end
```

Fig. 13: Opisana migracja w pliku `db/migrate/20111230120140_devise_create_users.rb` (pliki migracji zawsze zawierają dane o czasie, w którym zostały utworzone).

Migrowanie bazy danych to zadanie dla narzędzia *Rake*:

```
rake db:migrate
```

To polecenie porównuje obecny stan bazy danych z plikami migracji zgromadzonymi w katalogu aplikacji i upewnia się, że baza danych odpowiada ostatnim zmianom. Nie jest narzędziem specjalnie wyrozumiałym i bardzo łatwo można sobie zaszkodzić, jeśli nie przestrzega się zasady, że raz stworzone migracje, o ile nie są migracjami końcowymi, nie mogą być modyfikowane. W skrócie – raz wprowadzoną migrację „poprawia się” tylko za pomocą innej, nowej migracji.

Migracja nie tworzy elementów w testowej bazie danych. Dzieje się tak dlatego, że wszystkie dane w tej bazie traktowane są jak jednorazowego użytku. Niszczenie obiektów, a nawet całych tablic nie jest wykluczone w środowisku testowym, może prowadzić bowiem do odkrywania przyczyn błędów w aplikacji. Dodatkowo, choć wszystkie narzędzia działają w tym kierunku, nie zawsze mamy pewność, że po zakończeniu sesji testowej baza danych wróci do pierwotnego, „sterylnego” stanu. Aplikacja *Rails* dysponuje zadaniem *Rake* właśnie na taką okazję; należy też z niego skorzystać przed pierwszym odpaleniem testu.

```
rake db:test:prepare
```


Fabrykowanie danych testowych

`Factory_girl` to gem zajmujący się tworzeniem „fabryk” obiektów testowych. Fabryki tworzą instancje modeli aplikacji z konkretnymi, testowymi danymi, które można wykorzystywać w określonych scenariuszach. Zarówno standardowe testy automatyczne (dostępne domyślnie na platformie *Rails*), jak i testy behawioralne *RSpec* i *Cucumber*, polegają na istnieniu fabryki danego modelu. Można skorzystać z generatora:

```
rails g factory_girl:user
```

Lub samemu utworzyć plik w katalogu aplikacji: `spec/factories/user.rb` i wypełnić go następująco:

```
Factory.define :user do |usr|
  usr.sequence(:login) { |n| "user_#{n}" }
  usr.sequence(:email) { |n| "user_#{n}@example.com" }
  usr.password "jfhsdk2332"
  usr.password_confirmation {|u| u.password}
end
```

Fig. 14: Definicja fabryki użytkowników w pliku `spec/factories/user.rb`

Powyższy kod definiuje prostą „fabrykę użytkowników”. Ilekroć podczas wykonywania testów w środowisku testowym, które potrzebują pełnowartościowego obiektu danej klasy (instancji modelu), można łatwo odwołać się do danej fabryki, która zbuduje tymczasowy obiekt. Metoda `sequence` pozwala dodatkowo utworzyć prosty klucz wg którego wypełniane będą atrybuty kolejnych instancji modelu, jeżeli do testu użyjemy więcej niż jednej. W naszym wypadku pierwszy użytkownik będzie miał login „user_#1”, kolejny: „user_#2”. Ponadto utworzenie użytkownika wymaga potwierdzenia hasła. Można oczywiście wewnątrz fabryki wpisać je dwukrotnie ręcznie, ale znacznie bardziej eleganckim rozwiązaniem jest użyć do wypełnienia atrybutu `password_confirmation` wartości pola `password`, stąd przekazanie bloku `{|u| u.password}`. W bazie danych hasło nie jest unikatowe, więc nie ma potrzeby wypełniania go ze pomocą sekwencji. Wpisany ręcznie łańcuch znaków jest wystarczający dla naszych celów – tym bardziej, że w naszych testach będziemy przede wszystkim nadpisywali domyślne dane fabryki.

Pliki właściwości. Język Gherkin w systemie Cucumber

Teraz w końcu możemy wypróbować system *Cucumber* i jego język domenowy *Gherkin*. *Cucumber*, poza generatorem instalacyjnym, udostępnia też polecenie do tworzenia testów funkcjonalności:

```
rake g cucumber:feature authenticate
```

Generator stworzy plik `features/manage_authenticates.feature` jak również wypełni odpowiednimi plikami katalog `step_definitions`, za pierwszym razem utworzyć może również pliki konfiguracyjne środowiska (`features/support/env.rb`), ścieżek (`features/support/paths.rb`) i selektorów (`features/support/selectors.rb`). Pliki `*.feature` interesują nas najbardziej. To właśnie pliki testu behawioralnego, napisane w języku *Gherkin* – najwyższa powłoka systemu *Cucumber*. Pliki definicji kroków to skrypty Ruby, wykonujące testy za pomocą zdefiniowanego systemu (w naszym wypadku jest to program *RSpec*). Warto zwłaszcza zwrócić uwagę na zawartość pliku `features/step_definitions/web_steps.rb` – dość kontrowersyjnego skryptu, który w nowszych wersjach systemu *Cucumber* został całkowicie usunięty. Jego zadaniem jest interpretowanie słów kluczowych *Gherkina* dotyczących stanu przeglądarki, lub wyglądu strony. Takie testy nie mogą być przeprowadzane bez udziału symulatora przeglądarki, programu, który potrafi kalkulować jak wyglądać będzie odpowiedź serwera na dane żądanie; w naszej aplikacji tę niezwykle ważną rolę spełnia system *Capybara* i w wypadku gdy *Cucumber*, wykonując testy, będzie musiał sprawdzić odpowiedź serwera, zawartość formularza, czy dowolny inny warunek związany z funkcjonowaniem przeglądarki, przekaże odpowiednie testowe obiekty nie do skryptów definicji kroków obsługiwanych przez *RSpec*, ale do tych wywołujących symulator *Capybary*.

Zdaniem autora pracy generator `rake g cucumber:feature` powinno się uruchamiać najwyżej raz, na początku budowania systemu testowego. Kiedy zbudowana zostanie struktura plików, można bezpiecznie kontynuować dalej tworząc pliki `*.feature` ręcznie. Nie jest chyba wielkim zaskoczeniem, że takie nazwy plików: `features/manage_authenticates.feature` nie cieszą się popularnością, przedrostek `manage_` jest całkowicie zbędny. Przemianuję plik na `features/authenticate.feature` a następnie opiszę funkcjonalność uwierzytelniania tak, jak mógłby to zrobić klient, albo lider projektu. Dla celów demonstracyjnych użyję polskiego języka.

UWAGA: Jeśli podczas wykonywania któregośkolwiek z testów pojawi się błąd `invalid multibyte char (US-ASCII)` (`SyntaxError`), oznacza to, że w składni *Ruby* użyto znaków spoza zbioru znaków *ASCII*. Jedynym rozwiązaniem, które pozwoli wykonywać testy pisane zestawem znaków UTF-8 jest umieszczenie wiersza

```
#encoding utf-8
```

na początku każdego pliku testu kroków *RSpec* (plików znajdujących się w katalogu `features/step_definitions/`), które z takich znaków korzystają.

```

# language: pl
Właściwość: Autoryzacja
  W celu zapewnienia elementarnego bezpieczeństwa
  Jako użytkownik
  Chcę mieć możliwość uwierzytelniania i autoryzacji moich poczynań

  Scenariusz: Poprawne Logowanie
    Zakładając że mamy danych użytkowników
      | email | login | password |
      | jkb.zuchowski@gmail.com | ellmo | hasło1 |
    Oraz że jestem na stronie "root"
    Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
    kredencjałów "ellmo" oraz "hasło1"
    Wtedy trafię na stronę "root"
    Oraz będę widział informację "Zalogowałeś się jako ellmo"

```

Fig. 15: Opis funkcjonalności w pliku *features/authenticate.feature*

Słowa kluczowe języka Gherkin w polskiej lokalizacji oznaczono kolorami. Odpowiadają one komendom opisywanym w rozdziale: 4. *Wprowadzenie do Behaviour Driven Development (BDD)*

Możemy zacząć się przyzwyczajać, że praca z testami automatycznymi w dużym stopniu polega na wypełnianiu warunków jednej z powłok testu – w tym wypadku mówimy o poprawnym zdefiniowaniu testu behawioralnego, wykonywaniu testów i reagowaniu na ich wyniki. Jakim zakończy się test, który nie ma żadnych zdefiniowanych kroków w katalogu *features/step_definitions/* składa się tylko i wyłącznie z najwyższej warstwy²¹, z warstwy w postaci pliku definicji funkcjonalności *Cucumber*?

Zgodnie ze schematem opisanym w rozdziale 4. następnym etapem po utworzeniu warstwy pierwszej jest wykonanie testów i postępowanie według zaleceń systemu, który zwróci rezultat próby wykonania każdego z opisanych kroków.

Ponieważ jednak nie zdefiniowano czym każdy krok ma się zająć, ani czym jest opisana powyżej strona „root”, system poinformuje, że nie może wykonać testu.

²¹ pierwszą warstwą testu behawioralnego Cucumber nazywamy warstwę opisu funkcjonalności w pliku w katalogu *features/* rozszerzeniu *.feature* (plik w języku Gherkin), drugą warstwą jest opis kroku RSpec, napisany w języku Ruby, umieszczony w katalogu *features/stepdefinitions/*

```
$ bundle exec cucumber
```

Właściwość: Autoryzacja

W celu zapewnienia elementarnego bezpieczeństwa

Jako użytkownik

Chcę mieć możliwość uwierzytelniania i autoryzacji moich poczynañ

Scenariusz: Poprawne Logowanie

Zakładając że mamy danych użytkowników

email	login	password
jkb.zuchowski@gmail.com	ellmo	haslo1

Undefined step: "że mamy danych użytkowników"

Oraz że jestem na stronie "root"

Undefined step: "że jestem na stronie "root""

Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "ellmo" oraz "haslo1"

Undefined step: "niezalogowany użytkownik loguje się z widoku za
pomocą kredencjałów "ellmo" oraz "haslo1""

Wtedy trafię na stronę "root"

Undefined step: "trafię na stronę "root""

Oraz będę widział informację "Zalogowałeś się jako ellmo"

Undefined step: "będę widział informację "Zalogowałeś się jako
ellmo""

1 scenario (1 undefined)

5 steps (5 undefined)

0m0.724s

(zawartość wydruku edytowana dla większej czytelności)

Powyżej przedstawiono tylko pierwszą część wydruku. Pokazuje ona wyraźnie, że test nie może zostać wykonany ponieważ analiza składniowa (proces nazywany w żargonie z j. ang. *parsowaniem*) jego kroków nie może odnaleźć odpowiednich metod w definicjach pliku *RSpec.Cucumber* wciela się w rolę drogowskazu, który kieruje wątek programu do konkretnych testów, sam zaś za testy nie odpowiada. W sytuacji powyżej *Cucumber* nie potrafi skierować wątków do odpowiednich kroków testowych, ponieważ te zwyczajnie nie istnieją. Dalsza część wydruku wygląda następująco:

You can implement step definitions for undefined steps with these snippets:

```
Zakładając /^że mamy danych użytkowników$/ do |table|
  # table is a Cucumber::Ast::Table
  pending # express the regexp above with the code you wish you had
end
```

```
Zakładając /^że jestem na stronie "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

```
Jeżeli /^niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "([^"]*)" oraz "([^"]*)"$/ do |arg1, arg2|
  pending # express the regexp above with the code you wish you had
end
```

```

Wtedy /^trafię na stronę "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Wtedy /^będę widział informację "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

```

System pokazuje schemat składni kroków *RSpec* w języku *Ruby*, które należy opisać w plikach w katalogu `features/web_steps/`. *Cucumber* wie, że testy nie istnieją, dlatego pomaga skonstruować chociaż ich załączki. Nie potrafi rozpoznać logiki, jaka stoi czy też stać powinna za danymi testami, ale wie czego szuka i gdzie; odpowiedzi – zwracane przez system zarówno w początkowej fazie planowania, podczas rozwijania funkcjonalności, jak i usuwania błędów – są bezcenne. Zmienne umieszczone między kreskami pionowymi (ang. *pipe*) sugerują, że *Cucumber* do ciała danego kroku będzie przysyłał argumenty, które należy obsłużyć.

Zgodnie z zaleceniem w pierwszej linijce drugiej części wydruku, metody należy skopiować i umieścić w pliku definicji kroków *RSpec*, w tym przypadku w pliku `features/step_definitions/authenticate_steps.rb`. Nazwa pliku nie jest istotna, ale powinna być jak najbardziej deskryptywna. Ważne jest by plik znalazł się w tym katalogu:

```

#encoding: utf-8

Zakładając /^że mamy danych użytkowników$/ do |table|
  # table is a Cucumber::Ast::Table
  pending # express the regexp above with the code you wish you had
end

Zakładając /^że jestem na stronie "(.+)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Jeżeli /^niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "(.*)" oraz "(.+)"$/ do |arg1, arg2|
  pending # express the regexp above with the code you wish you had
end

Wtedy /^trafię na stronę "(.+)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Wtedy /^będę widział informację "(.+)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

```

Fig. 16: Treść pliku `features/step_definitions/authenticate.rb` (uproszczono zapis wyrażeń regularnych wewnątrz cudzysłowów)

Metoda `pending` mówi platformie *RSpec*, że ten test istnieje, ale nie jest w pełni zdefiniowany i powinien być traktowany jako załączek, fragment niekompletnego testu. *RSpec* rozpoznaje takie kroki i w trakcie wykonywania testu wyraźnie zaznacza w którym miejscu w kodzie znajduje się nieobsłużony test.

Po przepisaniu kodu zaproponowanego przez *Cucumber* do pliku uruchamiamy test:

```
$ bundle exec cucumber

Using the default profile...
# language: pl
Właściwość: Autoryzacja
(...)

Scenariusz: Poprawne Logowanie
  Zakładając że mamy danych użytkowników
    | email | login | password |
    | jkb.zuchowski@gmail.com | ellmo | haslo1 |
    TODO (Cucumber::Pending)
    ./features/step_definitions/authenticate_steps.rb:5:in `/^że
mamy danych użytkowników$/
    features/authenticate.feature:8:in `Zakładając że mamy danych
użytkowników'
    Oraz że jestem na stronie "root"
    Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "ellmo" oraz "haslo1"
    Wtedy trafię na stronę "root"
    Oraz będę widział informację "Zalogowałeś się jako ellmo"

1 scenario (1 pending)
5 steps (4 skipped, 1 pending)
0m0.235s
```

Cucumber reaguje inaczej, kiedy widzi, że utworzono odpowiednie kroki *RSpec* odpowiadające jego krokom wewnątrz scenariusza behawioralnego, nawet jeśli mają status nierozstrzygniętych. Nie wyświetla już informacji o tym, że tych kroków brakuje. Test nie zakończył się niepowodzeniem, ale został pominięty. Jeśli *Cucumber* trafi na choćby jeden nierozstrzygnięty krok, pomija wszystkie następne wewnątrz testu scenariusza, wyświetlając dokładną informację w jakim pliku i w którym jego wierszu znajduje się nierozstrzygnięty krok.

Pierwszy krok testu przedstawia bardzo prostą tabelę, zawierającą dane użytkownika (tworzenie takich tabel, rozdzielonych kreską pionową pozwala na utworzenie kilku różnych obiektów testowych z fabryki w jednym kroku), które muszą być użyte do utworzenia odpowiednich wierszy w testowej bazie danych, za pomocą fabryki utworzonej wcześniej (Fig 13.). *Cucumber* przesyła dane w postaci tabeli do kroku *RSpec*, który musi wywołać odpowiednią metodę modułu `factory_girl`, umieszczającą dane testowego użytkownika w bazie danych na samym początku testu funkcjonalności.

Nie wolno zapomnieć o fakcie, że przed każdą sesją testów baza danych jest

„sterylna”, nie zawiera żadnych danych – zawiera tylko odpowiednie tabele. Użytkownicy muszą więc być dodani do testowej bazy za każdym razem zanim wykonamy jakiegokolwiek testy z ich udziałem:

```
Zakładając /^że mamy danych użytkowników$/ do |table|
  table.hashes.each do |hash|
    Factory(:user, hash)
  end
end
```

Fig. 17: Przeredagowana pierwsza metoda kroku RSpec, utworzonej wcześniej (Fig. 15.) w pliku `features/step_definitions/authenticate.rb`

Ta metoda jest wywoływana za każdym razem gdy *Cucumber* wykona krok założeń, określony tekstem „że mamy danych użytkowników”, jako argument przyjmuje tablicę i pozwala po niej iterować. `Factory(:user, hash)` – tworzy testowy obiekt za pomocą zdefiniowanej wcześniej fabryki `User`, jako drugi parametr przyjmuje tablicę asocjacyjną z elementami, które mogą zastąpić domyślne atrybuty fabryki.

Tym razem wykonanie `bundle exec cucumber` zwróci wynik:

```
Scenariusz: Poprawne Logowanie
  Zakładając że mamy danych użytkowników
    | email | login | password |
    | jkb.zuchowski@gmail.com | ellmo | haslo1 |
  Oraz że jestem na stronie "root"
  TODO (Cucumber::Pending)
    ./features/step_definitions/authenticate_steps.rb:11:in `(...)'
    features/authenticate.feature:11:in `Oraz że jestem na stronie
"root"'
  Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "ellmo" oraz "haslo1"
  Wtedy trafię na stronę "root"
  Oraz będę widział informację "Zalogowałeś się jako ellmo"

1 scenario (1 pending)
5 steps (3 skipped, 1 pending, 1 passed)
```

Test funkcjonalności nadal traktowany jest jako nierozstrzygnięty, ponieważ przynajmniej jeden z jego kroków zawiera metodę `pending`. Widać jednak, że pierwszy krok scenariusza wykonywany jest poprawnie.

Ciekawostka: nazwa systemu *Cucumber* (ang. ogórek) pochodzi właśnie od zielonego koloru, oznaczającego że wszystkie testy zakończyły się powodzeniem.

Wiemy już teraz jak zmienia się status testu funkcjonalności w zależności od podejmowanych działań i uzupełniania testów, możemy przystąpić do utworzenia pełnego testu, bez nierozstrzygniętych kroków.

```

#encoding: utf-8

Zakładając /^że mamy danych użytkowników$/ do |table|
  table.hashes.each do |hash|
    Factory(:user, hash)
  end
end

Zakładając /^że jestem na stronie "(.+)"/ do |page_name|
  steps "Given I am on #{page_name}"
end

Jeżeli /^niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "(.+) " oraz "(.+)"/ do |login, pass|
  visit new_user_session_path
  fill_in "user_login", :with => login
  fill_in "user_password", :with => pass
  click_button "Sign in"
end

Wtedy /^trafię na stronę "(.+)"/ do |page_name|
  assert_equal path_to(page_name), current_path
end

Wtedy /^będę widział informację "(.+)"/ do |msg|
  page.should have_content(msg)
end

```

Fig. 18: Plik `features/step_definitions/authenticate.rb` z uzupełnionymi metodami kroków.

Na uwagę zasługuje drugi krok, który przechwytuje tytuł strony i kieruje go na inny krok *RSpec* – krok zdefiniowany wewnątrz pliku `web_steps.rb`:

```

Given /^(?:|I )am on (.+)/ do |page_name|
  visit path_to(page_name)
end

```

Fig. 19: metoda zdefiniowana w domyślnie utworzonym pliku `features/step_definitions/web_steps.rb`

`web_steps.rb` obsługuje typowe kroki scenariuszy odwołujące się do stanu przeglądarki i przekazuje je do wskazanego symulatora przeglądarki – w naszym przypadku *Capybary*. Metoda `visit` symuluje odwiedzenie danego widoku strony ze wszystkimi konsekwencjami. Ścieżka do widoku musi być uprzednio zdefiniowana i znana *Capybarze*, w innym wypadku uruchomienie tego testu zakończy się niepowodzeniem:


```

Oraz że jestem na stronie "root"
# features/step_definitions/authenticate_steps.rb:10
  Can't find mapping from "root" to a path.
  Now, go and add a mapping in features/support/paths.rb
(RuntimeError)
  ./features/support/paths.rb:30:in `rescue in path_to'
  ./features/support/paths.rb:24:in `path_to'
  ./features/step_definitions/web_steps.rb:45:in `/^(...)'
  features/authenticate.feature:11:in `Oraz że jestem na stronie
  "root"'

```

Wydruk edytowany dla czytelności

Cucumber informuje na bieżąco jakie kroki testu zakończyły się niepowodzeniem, lub które z nich wywołały błąd w kodzie *Ruby*. W tym przypadku wewnętrzny router *Cucumbera* nie może znaleźć strony o nazwie „root” i podaje w jakim pliku należy dokonać zmiany. Zaglądamy więc do pliku `features/support/paths.rb` do metody `path_to`:

```

def path_to(page_name)
  case page_name
  when /^the home\s?page$/
    '/'
  when /the new authenticate page/
    new_authenticate_path
  else
    begin
      page_name =~ /^the (.*) page$/
      path_components = $1.split(/\s+/)
      self.send(path_components.push('path').join('_').to_sym)
    rescue NoMethodError, ArgumentError
      raise "Can't find mapping from \"#{page_name}\" to a path.\n" +
        "Now, go and add a mapping in #{__FILE__}"
    end
  end
end

```

Fig. 20: Metoda `path_to` w pliku `features/support/paths.rb`

Metoda nie obsługuje widoku „root”, przewiduje, że test stroną startową będzie nazywać „the home page”, co w przypadku terminologii używanej do definiowania ścieżek aplikacji *Rails* jest dużą niekonsekwencją (w pliku `config/routes.rb` aplikacji widok główny definiowany jest jako „root”). Możemy oczywiście zamienić wyrażenie „root” na „the home page” w testach funkcjonalności, ale autor pracy uważa, że powinno się trzymać jednego terminu. Razi też użyte niżej wyrażenie:

```

page_name =~ /^the (.*) page$/
path_components = $1.split(/\s+/)

```

Utworzone w ten sposób wyrażenie regularne zareaguje tylko w momencie, kiedy użyjemy angielskiego określenia „the (dowolna_nazwa) page” do nazwania strony

w scenariuszu. W przypadku użycia języka polskiego do opisu funkcjonalności, staje się to poważnym ograniczeniem.

Wystarczy zmienić dwa wiersze kodu, by – po pierwsze router *Cucumbera* bezbłędnie rozpoznawał nazwę „root”, po drugie by łatwiej było definiować inne nazwy widoków w późniejszych scenariuszach w polskiej lokalizacji:

```
def path_to(page_name)
  case page_name
  when /^root$/
    '/'
  (...)
  else
    begin
      page_name =~ /^(.*)$/
    (...)
    end
  end
end
```

Fig. 21: Zmiany w metodzie `path_to` w pliku `features/support/paths.rb`

To rozwiąże pierwszy „czerwony” krok testu zakończony niepowodzeniem. Ponowne uruchomienie testu *Cucumber* nie zaliczy testu:

```
Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "ellmo" oraz "hasło1" #
features/step_definitions/authenticate_steps.rb:14
cannot fill in, no text field, text area or password field with
id, name, or label 'user_login' found (Capybara::ElementNotFound)
(eval):2:in `fill_in'
```

Błąd zgłosił moduł *Capybara*. Jest to wyjątek klasy `ElementNotFound`, oznaczający, że symulator przeglądarki miał wykonać operację na elemencie dokumentu HTML, ale nie mógł danego elementu znaleźć. Są dwa główne powody, dla których tak się dzieje:

- Krok testu *RSpec* odwołuje się do elementu za pomocą niewłaściwego atrybutu HTML (domyślnie *Capybara* szuka elementu według atrybutów `id`, `label` i `name`)
- Element nie został umieszczony w widoku

Przyjrzyjmy się testowi:

```
Jeżeli /^niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "(.+)" oraz "(.+)"$/ do |login, pass|
  visit new_user_session_path
  fill_in "user_login", :with => login
  fill_in "user_password", :with => pass
  click_button "Sign in"
end
```

rys. 22: `features/step_definitions/authenticate_steps.rb`

Wykonywanie tego kroku kończy się wysłaniem formularza, w odpowiedzi na który użytkownik ma zostać poprawnie zalogowany. Tak się jednak nie dzieje. Można się o tym przekonać uruchamiając serwer środowiska rozwojowego (komenda `rails s`) i wchodząc poprzez przeglądarkę na domyślny adres aplikacji w środowisku deweloperskim (`http://localhost:3000`). Przejdźmy na adres `/users/sign_in`. Wyświetlając kod strony (za pomocą wbudowanych inspektorów, lub opcji przeglądania kodu źródłowego w przeglądarce), zobaczymy, że formularz nie zawiera pola `user_login`, więc *Capybara* nie może go odpowiednio wypełnić.

Pojedynczy widok, z którym mamy problemy w scenariuszu można też obejrzeć za pomocą „migawki”. Robienie migawek widoków jest opisane w rozdziale 9. *Przydatne metody debugowania*.

Instalowanie gema *Devise* w poprzedniej iteracji utworzyło w katalogach aplikacji zarówno kopię kontrolera jak i widoków związanych z uwierzytelnianiem użytkownika.

W pierwszej kolejności zajrzemy do pliku z widokiem logowania: `app/views/devise/session/new.html.haml`, który zawiera wspomniany formularz logowania. Uprościmy widok (korzystając z pomocniczych metod formatujących, zwanych z j. ang. *helpers*, udostępnianych przez gem `simple_form`) i zamienimy pole `email` na pole `login`:

```
%h2 Sign in
%p.notice= notice
%p.alert= alert
.eight.columns
  = simple_form_for(resource, :as => resource_name, :url =>
    session_path(resource_name)) do |f|
    .inputs
      = devise_error_messages!
      = f.input :login, :required => false, :autofocus => true
      = f.input :password, :required => false
      = f.input :remember_me, :as => :boolean if
devise_mapping.rememberable?
    .actions
      = f.button :submit, "Sign in"

  = render :partial => "devise/shared/links"
```

rys. 23: `app/views/devise/session/new.html.haml`

To nie wystarczy; jak się przekonamy *Cucumber* przejdzie wspomniany test, ale już następnego nie uda mu się zaliczyć:

```
Wtedy trafię na stronę "root"
<"/"> expected but was
<"/users/sign_in">. (MiniTest::Assertion)
./features/step_definitions/authenticate_steps.rb:22:in `(...)'
features/authenticate.feature:13:in `Wtedy trafię na stronę'
```

```
"root"
```

Niepowodzenie nie dotyczy tego kroku – jest on tylko testem poprawności poprzednich kroków, więc prawdziwym winowajcą znowu jest metoda logowania. Po wysłaniu formularza z poprawnymi danymi użytkownik nie został zalogowany, dlatego aplikacja zamiast utworzyć sesję i przekierować przeglądarkę na stronę startową, została w widoku formularza oczekując aż użytkownik poprawi błędy – to bardzo czytelny dowód na to, że system *Capybara* bardzo dokładnie symuluje działanie przeglądarki.

Skąd bierze się problem? Jak spełnić warunki testu? Rozwiązaniem jest poprawna konfiguracja gema *devise*, który domyślnie reaguje tylko na próby logowania za pomocą adresu e-mail. Odpowiada za to jedna z domyślnych metod *Devise*:

```
module ClassMethods
  Devise::Models.config(self, :pepper, :stretches)

  def find_for_database_authentication(conditions)
    find_for_authentication(conditions)
  end
end
```

Fig. 24: Definicja metody bezpośrednio w plikach gemu
(~/rvm/gems/ruby-1.9.2-p290-patched/gems/devise-1.4.9/lib/devise/models/database_authenticatable.rb)

By dostosować ją do własnych potrzeb, należy przesłonić ją w klasie modelu użytkownika (`app/models/user.rb`), nie zapominając o udostępnieniu atrybutu `login` za pomocą metody `attr_accessible`:

```
class User < ActiveRecord::Base
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  attr_accessible :email, :login, :password,
                 :password_confirmation, :remember_me

  def self.find_for_database_authentication(conditions={})
    self.find_by_login conditions[:login] or
    self.find_by_email conditions[:login]
  end
end
```

Fig. 25: Zmiany w pliku `app/models/user.rb`

Ten zabieg sam w sobie nie wystarczy, ponieważ *devise* nie przetworzy parametru `login` wysłanego z formularza, dopóki nie pozwolimy na to, modyfikując wiesz kodu w pliku inicjującym `config/initializers/devise.rb`, w którym należy odnaleźć następujący wiersz:

```
# config.authentication_keys = [ :email ]
```

Odkomentujemy tę linię, zmienimy wartość na `login` i dodatkowo przyjrzymy się innym opcjom obsługi tego parametru:

```
config.authentication_keys = [ :login ]
config.case_insensitive_keys = [ :login ]
config.strip_whitespace_keys = [ :login ]
```

Fig. 26: Zmiany w pliku `config/initializers/devise.rb`

To zapewni poprawne ukończenie kolejnego kroku testu. Uruchomienie polecenia `bundle exec cucumber` poinformuje że niepowodzeniem kończy się już tylko ostatni, bardzo prosty krok scenariusza:

```
Zakładając że mamy danych użytkowników
  | email | login | password |
  | jkb.zuchowski@gmail.com | ellmo | haslo1 |
Oraz że jestem na stronie "root"
Jeżeli niezalogowany użytkownik loguje się z widoku za pomocą
kredencjałów "ellmo" oraz "haslo1"
Wtedy trafię na stronę "root"
Oraz będę widział informację "Zalogowałeś się jako ellmo"
(Rspec::Expectations::ExpectationNotMetError)

1 scenario (1 failed)
5 steps (1 failed, 4 passed)
```

Capybara sprawdza, czy, po poprawnym zalogowaniu i przekierowaniu do strony startowej, widoczne jest powiadomienie – tekst w dokumencie HTML, znajdujący się w dowolnym miejscu w drzewie DOM. Jesteśmy tylko o krok od spełnienia ostatniego kroku w całym teście funkcjonalności. Wystarczy tylko zmienić widok strony startowej, znajdujący się w pliku `app/views/home/index.html.haml`:

```
%h1 Home#index

-unless current_user
  = link_to "Sign in", :new_user_session
-else
  %p
    = "Zalogowałeś się jako #{current_user.login}"
    = link_to "Sign out", :destroy_user_session, :method => :delete
```

Po tych wszystkich zmianach wykonanie `bundle exec cucumber` zaliczy cały test funkcjonalności:

```
Scenariusz: Poprawne Logowanie
(...)
Wtedy trafię na stronę "root"
Oraz będę widział informację "Zalogowałeś się jako ellmo"

1 scenario (1 passed)
5 steps (5 passed)
```

Iteracja 3

8.3. Rejestracja i testy negatywne

Zadania:

1. Utworzyć i testować system rejestrowania
2. Obsłużyć błędy użytkownika podczas korzystania z funkcjonalności logowania i rejestracji

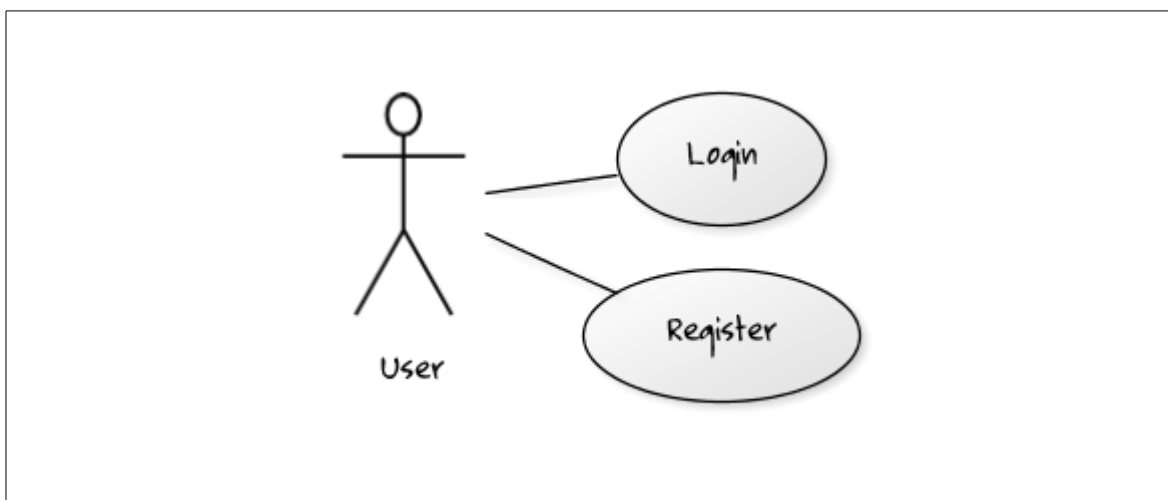


Fig. 27: Funkcjonalność rejestracji to nowy przypadek użycia dla użytkowników.

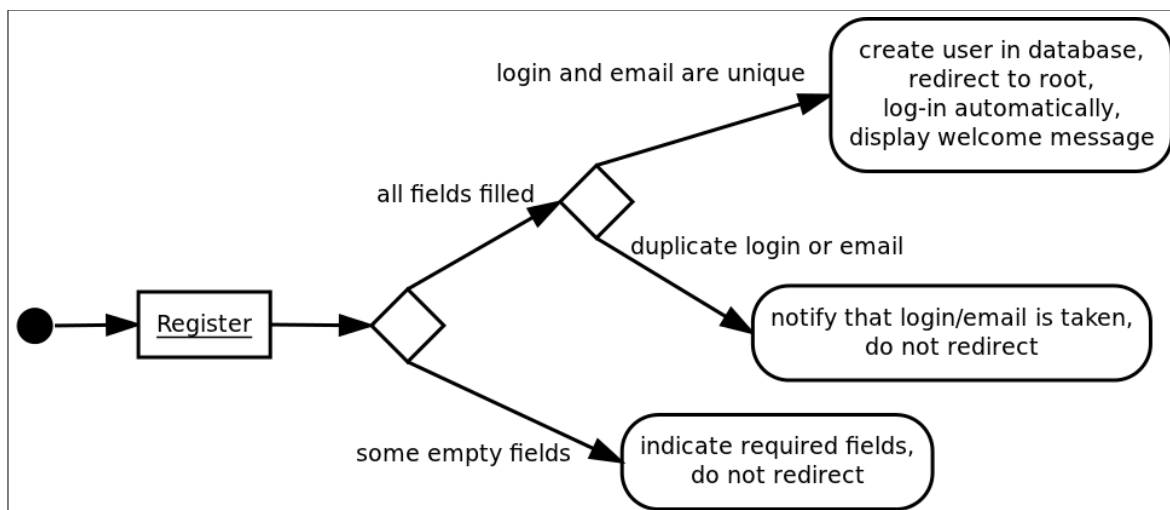


Fig. 28: Schemat komunikacji podczas rejestracji nowego użytkownika

W poprzedniej iteracji omówiliśmy zautomatyzowany system uwierzytelniania użytkowników *Devise* i bardzo szczegółowo opisaliśmy wszystkie kroki refaktorowania kodu pod kątem testów *Cucumbera*. Nie mówiliśmy jednak o poziomie bezpieczeństwa, jaki on oferuje, czemu zapobiega i jak dalej wykorzystywać jego możliwości pod kątem utrzymania, lub zwiększenia poziomu bezpieczeństwa.

Utworzony przez generator *Devise* model użytkownika domyślnie zawiera pole hasła haszowane metodą SHA-256 („*Secure Hash Algorithm*” o skrócie 256-bitowym), co oznacza, że nawet w przypadku nieuprawnionego dostępu do bazy danych, intruz odczyta tylko poprawnie zahaszowane hasła. Ma to też wpływ na sposób odzyskiwania dostępu do konta przez zarejestrowanych użytkowników, ponieważ nie można hasła zwyczajnie przypomnieć – system nie ma fizycznych możliwości odtworzenia oryginalnego hasła – można co najwyżej nadpisać je innym, tymczasowym, a następnie poinformować użytkownika. W tym celu wykorzystamy zautomatyzowane systemy powiadomień, tzw. *Mailery*.

Przyjrzyjmy się fragmentowi migracji użytkownika, którą stworzył *Devise*:

```
t.database_authenticatable :null => false
t.recoverable
```

Te dwie linijki odpowiadają za dodawanie do tabeli użytkowników kolumn, które pozwalają na uwierzytelnianie i odzyskiwanie konta. Na pierwszy rzut oka są dość enigmatyczne, ale wiemy, że dodają kolumnę na hasła i kolumnę na jednorazowe tokeny, służące jako jednorazowe uwierzytelnienie użytkownika, który chce zresetować hasło.

Funkcjonalność rejestracji zawiera się w kontekście pierwszego planu behawioralnego, przemianowano zatem zawierający definicję funkcjonalności uwierzytelniania plik `features/step_definitions/authenticate.feature` na `features/step_definitions/login_register.feature` (powinno się analogicznie przemianować również plik kroków w katalogu `features/step_definitions/`, chociaż ten plik zostanie poprawnie dołączony do testów niezależnie od nazwy). Pracując z więcej niż jednym scenariuszem wewnątrz funkcjonalności, można wykorzystywać przełącznik `@wip` (od ang. *Work In Progress* – praca w toku), słowo kluczowe dodawane linijkę przed scenariuszem, aby system mógł wykonywać testy *Cucumber* tylko dla scenariuszy roboczych.

Domyślnie *Cucumber* obsługuje maksymalnie trzy scenariusze robocze, ale modyfikowanie tych własności jest bardzo łatwe, wystarczy odnaleźć plik `config/cucumber.yml` w katalogu konfiguracji aplikacji i modyfikować linijkę

```
wip: --tags @wip:3 --wip features
```

Zdaniem autora pracy bardziej elastycznym rozwiązaniem jest utworzenie kilku poziomów profili roboczych, dzięki czemu możemy osobno uruchamiać testy dla scenariuszy oznaczonych np. jako `@wip`, `@wip2` i `@wip3`. Wystarczy dodać takie linijki:


```
wip: --tags @wip:3 --wip features
wip2: --tags @wip2:3 --wip features
wip3: --tags @wip3:3 --wip features
```

Następnie należy uzupełnić ścieżki w pliku `features/support/paths.rb`, w sposób w pełni zgodny ze schematami routingu aplikacji *Ruby on Rails*.

```
def path_to(page_name)
  case page_name
  when "root"
    root_path
  when "sign_in"
    new_user_session_path
  when "sign_up"
    new_user_registration_path
  (...)
  end
end
```

Fig. 29: Fragment pliku `features/support/paths.rb`

Dzięki takim ustawieniom nasz test jest odporny na zmiany „miękkich” adresów *url* danych widoków, ponieważ do nawigacji wykorzystuje zmienne aplikacji, zamiast bezpośrednich adresów.

Mając już trochę doświadczenia w pracy z *Cucumberem* można pokusić się nie tylko o dopisanie nowego scenariusza, ale też o refaktoryzowanie poprzedniego. Uzyska się większą przejrzystość struktury poprzez oddzielenie testów dotyczących nawigacji do formularza od testów jego przesyłania i przetwarzania. By scenariusze były kompletne i mogły służyć również jako dokumentacja, utworzymy osobne scenariusze na dowód, że system poprawnie zaloguje użytkownika niezależnie od tego, czy wykorzystał do identyfikacji adres e-mail, czy osobisty login. Jednak budowanie dwóch bliźniaczo podobnych scenariuszy jest nielogiczne, a przede wszystkim kłóci się z podstawową zasadą programowania zwinnego: *DRY* (*Don't Repeat Yourself* – ang. nie powtarzaj się). To dobry moment, by zastosować wygodne rozwiązanie, zwane „szablonem scenariusza” (ang. *Scenario Outline*).

```
# language: pl
Właściwość: Uwierzytelnianie i Rejestracja
W celu zapewnienia elementarnego bezpieczeństwa
Jako użytkownik
Chcę mieć możliwość uwierzytelniania i autoryzacji moich poczynań

@wip
Scenariusz: Widoczność formularza logowania
  Zakładając że jestem na stronie "root"
  Oraz że widzę odnośnik "Sign in"
  Jeżeli kliknę odnośnik "Sign in"
  Wtedy trafię na stronę "sign_in"
  Oraz będę widział formularz logowania

@wip
Szablon scenariusza: logowanie emailem LUB loginem
  Zakładając że mamy danych użytkowników
    | email | login | password |
    | jkb.zuchowski@gmail.com | ellmo | hasło123 |
  Oraz że jestem na stronie "sign_in"
  Jeżeli wypełnię formularz logowania danymi
    | user_login | user_password |
    | <login_or_email> | <pass> |
  Oraz kliknę przycisk "Sign in"
  Wtedy trafię na stronę "root"
  Oraz będę widział informację "Zalogowałeś się jako ellmo"

  Przykłady:
    | login_or_email | pass |
    | ellmo | hasło123 |
    | jkb.zuchowski@gmail.com | hasło123 |

@wip2
Scenariusz: Widoczność formularza rejestracji
  Zakładając że jestem na stronie "root"
  Oraz że widzę odnośnik "Sign up"
  Jeżeli kliknę odnośnik "Sign up"
  Wtedy trafię na stronę "sign_up"
  Oraz będę widział formularz rejestracji

@wip3
Scenariusz: Poprawne rejestrowanie
  Zakładając że jestem na stronie "sign_up"
  Jeżeli wypełnię formularz rejestracji danymi
    | email | login | password |
    | jkb.zuchowski@gmail.com | ellmo | hasło123 |
  Oraz kliknę przycisk "Register"
  Wtedy trafię na stronę "root"
  Oraz w bazie danych znajdzie się użytkownik "ellmo"
  Oraz będę widział informację "Zalogowałeś się jako ellmo"
```

Fig. 30: Zmodyfikowany plik `features/step_definitions/login_register.feature`.

Scenariusze oznaczono stosownie przełącznikami @wip – dla dwóch pierwszych, które powstały z rozbicia napisanego w poprzedniej iteracji działającego testu i które muszą być jak najszybciej poprawione, oraz @wip2 i @wip3 dla nowych scenariuszy.

Żeby testować tylko scenariusze z odpowiednim przełącznikiem, wykorzystamy

polecenie: `bundle exec cucumber -p wip`

Wynik komendy wskaże, że *Cucumber* nie wie jak rozumieć część kroków, co jest oczywiste, zważywszy na to, że dokonano poważnych zmian i przemianowano dane kroki. Kiedy naprawimy definicje kroków, okaże się, że jeden krok może być wykorzystywany przez wiele scenariuszy. Na przykład poniższe kroki są uniwersalne:

```

Zakładając że widzę odnośnik "Sign in"
Jeżeli kliknę odnośnik "Sign in"
Jeżeli kliknę przycisk "Sign in"

```

A przede wszystkim są bardzo proste, „atomowe” – nie daje się danego kroku rozbić na mniejsze części, ani opisać go zwięźle.

Odpowiednie metody kroków *RSpec* dla tych kroków umieszczono w pliku `features/step_definitions/common_steps.rb`:

```

Zakładając /^że widzę odnośnik "(.)"/ do |link|
  page.should have_link(link)
end

Jeżeli /^kliknę odnośnik "(.)"/ do |link|
  click_link(link)
end

Jeżeli /^kliknę przycisk "(.)"/ do |button|
  click_button(button)
end

```

Fig. 31: Plik `features/step_definitions/common_steps.rb`

Brakuje tylko ostatniego kroku, by pierwszy scenariusz kończył się pomyślnie, ponieważ *Cucumber* nie potrafi zinterpretować „Wtedy będę widział formularz logowania”. Ponieważ jest to krok specyficzny dla scenariusza logowania należy zdefiniować ten krok *RSpec* w odpowiednim pliku definicji:

```

Wtedy /^będę widział formularz logowania$/ do
  page.should have_selector("form#user_new")
end

```

Fig. 32: Obsługa warunku widoczności formularza w pliku `features/step_definitions/login_register_steps.rb`

W drugim scenariuszu mamy po raz pierwszy do czynienia z ciekawą konstrukcją szablonu scenariusza:

```

Szablon scenariusza: tytuł
Zakładając że ...
Jeżeli ...
Wtedy ...

Przykłady:
| parametr1 | parametr2 |
| wartość1 | wartość2 |

```

rys. 33: schemat użycia słów kluczowych i składni szablonu scenariusza

Szablon ma konstrukcję podobną do tablicy, która pozwalała wysyłać dane do

kroków *RSpec*, tylko tym razem dla każdego elementu w kolekcji wykonywany jest cały scenariusz.

Aby oba scenariusze danego schematu kończyły się pomyślnie, wystarczy napisać następującą definicję, reprezentującą ostatni z wymaganych kroków *RSpec*:

```
Jeżeli /^wypełnię formularz logowania danymi$/ do |table|
  table.hashes.each_with_index do |hash, index|
    fill_in("user_login", :with => hash[:user_login] )
    fill_in("user_password", :with => hash[:user_password])
  end
end
```

Fig. 34: Obsługa nowego kroku wypełniania formularza logowania w pliku *features/step_definitions/login_register_steps.rb*

Zwróćmy uwagę, że wydruk wyniku testu, będzie odbiegał (również kolorystycznie, jeżeli użyto terminala zdolnego wyświetlać kolory, np. *XTerm*) od naszych oczekiwań:

```
Szablon scenariusza: logowanie emailem LUB loginem
Zakładając że mamy danych użytkowników
| email | login | password |
| jkb.zuchowski@gmail.com | ellmo | haslo123 |
Oraz że jestem na stronie "sign_in"
Jeżeli wypełnię formularz logowania danymi
| user_login | user_password |
| <login_or_email> | <pass> |
Oraz kliknę przycisk "Sign in"
Wtedy trafię na stronę "root"
Oraz będę widział informację "Zalogowałeś się jako ellmo"

Przykłady:
| login_or_email | pass |
| ellmo | haslo123 |
| jkb.zuchowski@gmail.com | haslo123 |
```

W tym wydruku jest logika – sam schemat nie może zawieść, zawieść mogą konkretne scenariusze z konkretnymi danymi, podpięte do schematu.

Funkcjonalności rejestracji będą wymagały od nas dopisania zaledwie trzech kroków *RSpec* i drobnych modyfikacji domyślnych widoków *Devise*.

```
Jeżeli /^wypełnię formularz rejestracji danymi$/ do |table|
  table.hashes.each_with_index do |hash, index|
    fill_in("user_email", :with => hash[:email])
    fill_in("user_login", :with => hash[:login])
    fill_in("user_password", :with => hash[:password])
    fill_in("user_password_confirmation", :with => hash[:password])
  end
end
```

```

Wtedy /^będę widział formularz rejestracji$/ do
  page.should have_selector("form#user_new")
end

Wtedy /^w bazie danych znajdzie się użytkownik "(.+)"/ do |login|
  User.find_by_login(login).should_not be_nil
end

```

Fig. 35: Obsługa nowego kroku wypełniania formularza logowania w pliku `features/step_definitions/login_register_steps.rb`

```


# Home#index -unless current_user %span [ = link_to "Sign in", :new_user_session | = link_to "Sign up", :new_user_registration ] -else %p = "Zalogowałeś się jako #{current_user.login}" = link_to "Sign out", :destroy_user_session, :method => :delete


```

Fig. 36: Plik `app/views/home/index.html.haml` (wyróżniono zmiany w porównaniu do poprzedniej wersji pliku).

```


## Sign up .eight.columns = simple_form_for(resource, :as => resource_name, :url => registration_path(resource_name), :html => {:class => "nice"}) do |f| %div = devise_error_messages! = f.input :email, :wrapper_html => {:class => "form-field"} = f.input :login, :wrapper_html => {:class => "form-field"} = f.input :password, :wrapper_html => {:class => "form-field"} = f.input :password_confirmation, :wrapper_html => {:class => "form-field"} %div = f.button :submit, :value => "Register" = render :partial => "devise/shared/links"


```

rys. 37: Plik `app/views/devise/registrations/new.html.haml` (wyróżniono zmiany w porównaniu do poprzedniej wersji pliku).

Konieczne jest usuwanie przełączników @wip z nagłówków scenariuszy, które są spełnione i nad którymi zakończyliśmy pracę.

Testy są coraz czytelniejsze, tworzenie nowych scenariuszy idzie tym sprawniej, im więcej kroków dzielią z innymi scenariuszami w kontekście tej samej funkcjonalności.

Dysponujemy już działającym systemem logowania i rejestracji, ale w chwili obecnej przewidujemy tylko scenariusze poprawnego wykorzystania funkcjonalności, zaś nasze schematy komunikacji (zarówno dla logowania, jak i dla rejestracji) wyraźnie wskazują, że powinniśmy obsługiwać błędy użytkownika, tak by pomóc mu przy zwykłej pomyłce, jak również i zabezpieczyć się przed próbą uzyskania nieuprawnionego dostępu. Testy obsługi błędów nazywamy testami „negatywnymi”, lub „odwrotnymi”.

Utworzony scenariusz błędnego logowania w dużej mierze wykorzystuje istniejące kroki scenariusza poprawnego, różni się tylko kredencjałami i wynikiem akcji:

```
@wip
Szablon scenariusza: błędne logowanie
Zakładając że mamy danych użytkowników
  | email | login | password |
  | jkb.zuchowski@gmail.com | ellmo | haslo123 |
Oraz że jestem na stronie "sign_in"
Jeżeli wypełnię formularz logowania danymi
  | user_login | user_password |
  | <login_or_email> | <pass> |
Oraz kliknę przycisk "Sign in"
Wtedy trafię na stronę "sign_in"
Oraz będę widział informację o błędach

Przykłady:
  | login_or_email | pass |
  | niepoprawny | haslo123 |
  | ellmo | niepoprawne |
```

Fig. 38: Scenariusz błędnego logowania umieszczony w pliku `features/login_register.feature`

Wobec tego brakuje już tylko jednego kroku:

```
Wtedy /^będę widział informację o błędach$/ do
  page.should have_selector(".alert")
end
```

Fig. 39: Obsługa nowego kroku dodana do pliku `features/step_definitions/common_steps.rb`

Jednak jeśli napiszemy analogiczny szablon scenariusza dla niepoprawnych logowań, okaże się, że system nie zachowa się zgodnie z oczekiwaniem:

```
@wip
Szablon scenariusza: Niepoprawne rejestrowanie
Zakładając że mamy danych użytkowników
| email | login | password |
| jkb.zuchowski@gmail.com | ellmo | hasło123 |
Oraz że liczba użytkowników wynosi "1"
Oraz że jestem na stronie "sign_up"
Jeżeli wypełnię formularz rejestracji danymi
| email | login | password |
| <email> | <login> | <password> |
Oraz kliknę przycisk "Register"
Wtedy trafię na stronę "sign_up"
Oraz w bazie danych nie znajdzie się nowy użytkownik
Oraz będę widział informację o błędach

Przykłady:
| email | login | password |
| jkb.zuchowski@gmail.com | ellmo | hasło123 |
| jkb.zuchowski@gmail.com | | hasło123 |
| jkb.zuchowski@gmail.com | | hasło123 |
| niepoprawny_adres | ellmo | hasło123 |
| jkb.zuchowski@gmail.com | ellmo | hasło123 |
```

Fig. 40: Szablon scenariusza niepoprawnego logowania w pliku `features/login_register.feature`

```
Zakładając /^że liczba użytkowników wynosi "(.+)"/ do |number|
  @user_count = User.count
  @user_count.should == number.to_i
end
```

Fig. 41: Obsługa nowego kroku sprawdzania początkowej liczby użytkowników dodana do pliku `features/step_definitions/common_steps.rb`

Zmienna instancji `@user_count` musi być ustawiona poprawnie, by mogła być wykorzystana w kroku poniżej.

```
Wtedy /^w bazie danych nie znajdzie się nowy użytkownik$/ do
  @user_count.should == User.count
end
```

rys. 42: Plik `features/step_definitions/common_steps.rb`

System *Devise* przy próbie niepoprawnej rejestracji przekierowuje na inny adres. Niezależnie od tego, czy proponowane rozwiązanie jest słuszne czy nie, jest niezgodne z naszym scenariuszem i niekonsekwentne w porównaniu z nieudaną próbą logowania, testowaną w poprzednich scenariuszach, gdzie serwer nie przekierowuje na inny adres. Zatem metoda w kontrolerze *Devise* musi ulec zmianie:


```
# POST /resource
def create
  build_resource

  if resource.save
    (...)
  else
    clean_up_passwords(resource)
    redirect_to(new_user_registration_path,
               :alert => resource.errors.full_messages)
  end
end
```

*Wyróżniona zmiana kodu w akcji create kontrolera rejestracji systemu Devise.
Plik `app/controllers/devise/registrations_controller.rb`*

Ta zmiana zapewni, że przeglądarka zostanie na tym samym adresie i wyświetli błędy formularza wewnątrz elementu HTML o klasie alert. Teraz brakuje tylko wyświetlania tego elementu w widoku:

```
%h2 Sign up
-if notice
  %p.notice= notice
-if alert
  %p.alert= alert
.eight.columns
(...)
```

Obsługa wyświetlania elementów notice i alert w widoku, pochodzącym z pliku `app/views/devise/registrations/new.html.haml`

Na tym etapie powinniśmy mieć łącznie 12 pełnych scenariuszy i 80 kroków.

Iteracja 4

8.4. Odzyskiwanie konta użytkownika dzięki Mailerowi.

Zadania:

1. Utworzyć i testować system wysyłania poczty (ActionMailer)
2. Obsłużyć wysyłanie wiadomości e-mail z jednorazowymi tokenami resetującymi hasło konta użytkownika

Kolejnym ciekawym zagadnieniem jest obsługa systemu automatycznych powiadomień mailowych, które w aplikacji *Ruby on Rails* domyślnie udostępnia system *ActionMailer*.

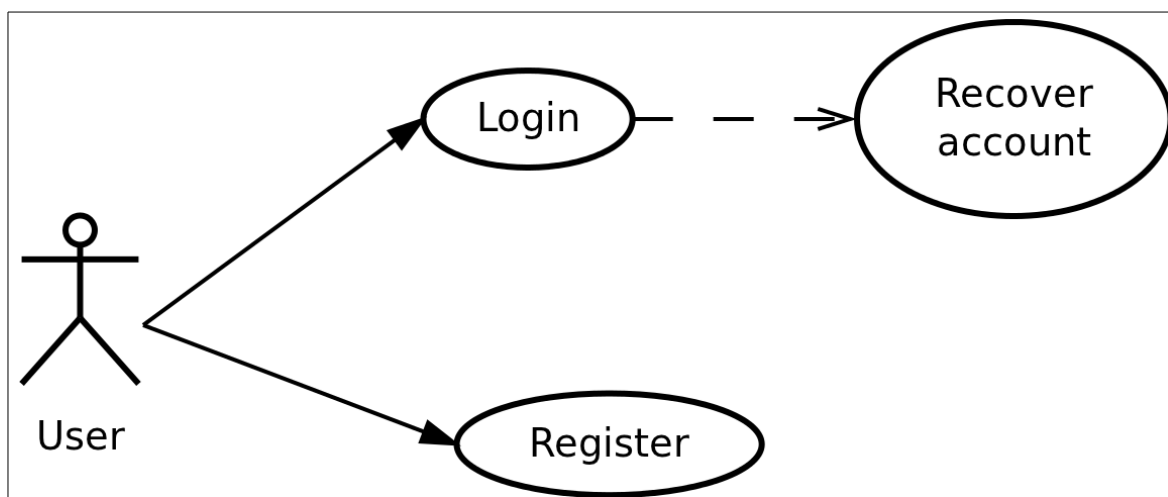


Fig. 43: Przypadek użycia odzyskania hasła powinien zawierać się w kontekście logowania.

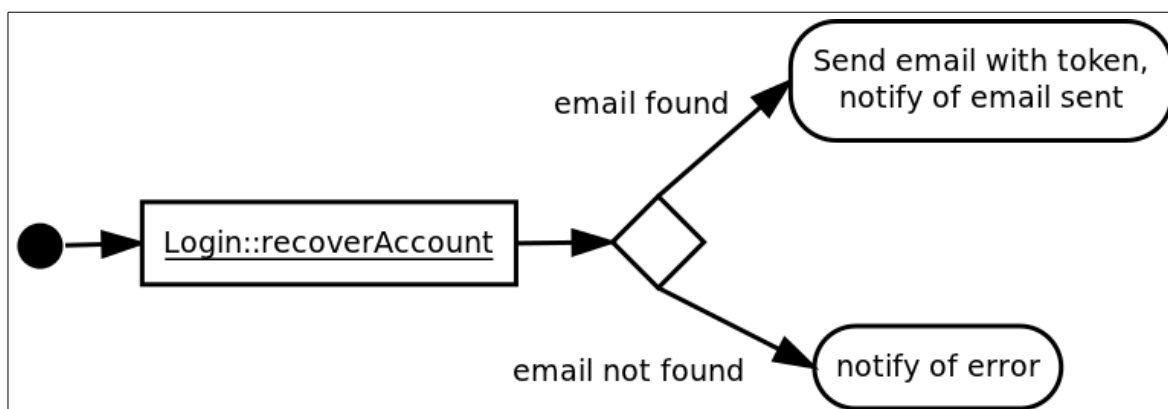


Fig. 44: Schemat komunikacji dla funkcjonalności odzyskiwania hasła.

```

# language: pl
Właściwość: Zautomatyzowana pocztą
  W celu zapewnienia możliwości odzyskiwania konta
  Jako użytkownik
  Chcę mieć możliwość resetowania hasła na wypadek, gdybym je
zapomniał

@wip
Scenariusz: Widoczność formularza odzyskiwania konta
  Zakładając że jestem na stronie "sign_in"
  Oraz że widzę odnośnik "Forgot your password?"
  Jeżeli kliknę odnośnik "Forgot your password?"
  Wtedy trafię na stronę "new_user_password"
  Oraz będę widział formularz odzyskiwania konta

@wip
Scenariusz: Próba odzyskania konta za pomocą poprawnych kredencjałów
  Zakładając że mamy danych użytkowników
    | email | login | password |
    | jkb.zuchowski@gmail.com | ellmo | hasło123 |
  Oraz że jestem na stronie "new_user_password"
  Oraz że mailer jest przygotowany
  Jeżeli wypełnię pole "user_email" tekstem
"jkb.zuchowski@gmail.com"
  Oraz kliknę przycisk "Send me reset password instructions"
  Wtedy zostanie wysłany email

@wip
Scenariusz: Próba odzyskania konta za pomocą niepoprawnych
kredencjałów
  Zakładając że mamy danych użytkowników
    | email | login | password |
    | jkb.zuchowski@gmail.com | ellmo | hasło123 |
  Oraz że jestem na stronie "new_user_password"
  Oraz że mailer jest przygotowany
  Jeżeli wypełnię pole "user_email" tekstem "niepoprawny@gmail.com"
  Oraz kliknę przycisk "Send me reset password instructions"
  Wtedy nie zostanie wysłany email
  Oraz będę widział informację o błędach

```

Fig. 45: Plik `features/mailer.feature`

Jest to dopiero drugi plik `*.feature` w naszym systemie i można zaobserwować, że kroki RSpec są odnajdywane niezależnie od tego, czy ich nazwa jest analogiczna do nawy pliku funkcjonalności. Wobec tego powinniśmy zaprowadzić porządek, utworzyć plik o nazwie np. `features/step_definitions/common_steps.rb` i tam przenieść wszystkie proste kroki, którą są wykorzystywane wielokrotnie w różnych scenariuszach. Jeden z nowych kroków zdefiniowanych dla funkcjonalności *Mailera* (Jeżeli wypełnię pole "..." tekstem "...") również powinien być obsługiwany w tym pliku:

```

Zakładając /^że mamy danych użytkowników$/ do |table|
  (...)
Zakładając /^że liczba użytkowników wynosi "(.+)"/ do |number|
  (...)
Zakładając /^że jestem na stronie "(.+)"/ do |arg1|
  (...)
Zakładając /^że widzę odnośnik "(.+)"/ do |link|
  (...)
Jeżeli /^kliknę odnośnik "(.+)"/ do |link|
  (...)
Jeżeli /^kliknę przycisk "(.+)"/ do |button|
  (...)
Jeżeli /^wypełnię pole "(.+) " tekstem "(.+)"/ do |field, value|
  (...)
Wtedy /^trafię na stronę "(.+)"/ do |page_name|
  (...)
Wtedy /^w bazie danych znajdzie się użytkownik "(.+)"/ do |login|
  (...)
Wtedy /^w bazie danych nie znajdzie się nowy użytkownik$/ do
  (...)
Wtedy /^będę widział informację "(.+)"/ do |msg|
  (...)
Wtedy /^będę widział informację o błędach$/ do
  (...)

```

Fig. 46: Nagłówki metod przeniesionych do pliku *features/step_definitions/common_steps.rb*

W pliku definicji kroków funkcjonalności *ActionMailera* obsłużymy szczególne przypadki:

```

#encoding: utf-8

Zakładając /^że mailer jest przygotowany$/ do
  ActionMailer::Base.delivery_method = :test
  ActionMailer::Base.perform_deliveries = true
  ActionMailer::Base.raise_delivery_errors = true
  ActionMailer::Base.deliveries.clear
end

Wtedy /^będę widział formularz odzyskiwania konta$/ do
  page.should have_selector("form#user_new")
end

Wtedy /^zostanie wysłany email$/ do
  ActionMailer::Base.deliveries.should_not be_empty
end

Wtedy /^nie zostanie wysłany email$/ do
  ActionMailer::Base.deliveries.should be_empty
end

```

Fig. 47: Plik *features/step_definitions/mailer_steps.rb*

W kroku *RSpec* obsługującym przygotowanie *ActionMailera* do pracy ustawiono następujące parametry działania systemu:

```
ActionMailer::Base.delivery_method = :test
```

Ustawienie Mailera w tryb testowy.

```
-
```

```
ActionMailer::Base.perform_deliveries = true
```

Ustawienie sprawiające, że system będzie symulował wysłanie wiadomości, zamiast tylko zapisać ją w pamięci podręcznej.

```
-
```

```
ActionMailer::Base.raise_delivery_errors = true
```

Aktywowanie błędów i wyjątków, zwracanych przez system w razie niepowodzenia.

```
-
```

```
ActionMailer::Base.deliveries.clear
```

Wyczyszczenie dotychczasowej kolejki wiadomości e-mail.

Następnie zgodnie z informacjami drukowanymi przez Cucumber pozbywamy się kolejnych błędów i niezgodności z wytycznymi. Przede wszystkim należy przejrzeć pliki konfiguracyjne środowisk uruchomieniowych `config/environments/test.rb`, `config/environments/development.rb`, w których powinna znaleźć definicja systemu mailingowego. Na potrzeby tych środowisk: testowych i rozwojowego, można spokojnie wykorzystywać własną skrzynkę pocztową, ale dobrą praktyką jest nieumieszczanie danych niejawnych bezpośrednio w tych plikach, niezależnie od środowiska. Najbezpieczniejszym miejscem, jest przechowywanie ich w specjalnej tabeli w bazie danych. W przykładzie poniżej wykorzystano stałe, zdefiniowane w innym, niewersjonowanym pliku `config/environments/secret_data.rb`:

```
module SecretData
  EMAIL = "<prawdziwy_adres_email>"
  PASSW = "<hasło>"
end
```

Fig. 48: Zawartość pliku `config/environments/secret_data.rb`, należy w nim podać prawdziwe kredencjały konta pocztowego, które chcemy wykorzystać w aplikacji.

```

Mgr::Application.configure do
  load Rails.root.join("config/environments/secret_data.rb")
  (...)

  config.action_mailer.default_url_options = { :host =>
    'localhost:3000' }
  config.action_mailer.delivery_method = :smtp
  config.action_mailer.smtp_settings = {
    :address          => "smtp.gmail.com",
    :port             => 587,
    :domain           => 'gmail.com',
    :user_name        => SecretData::EMAIL,
    :password         => SecretData::PASSW,
    :authentication   => 'plain',
    :enable_starttls_auto => true
  }
end

```

Fig. 49: Wspólny fragment plików konfiguracyjnych środowiska testowego i rozwojowego (development), definiujący parametry skrzynki pocztowej, wykorzystywanej przez ActionMailer.

By system mógł znaleźć formularz przypominania hasła i był w pełni zgodny z wytycznymi, musimy jeszcze wykonać rutynowe czynności:

1. Dodać ścieżkę do formularza odzyskiwania hasła do `features/support/path.rb`

```

module NavigationHelpers
  ...
  def path_to(page_name)
    case page_name
    (...)
    when "new_user_password"
      new_user_password_path
    (...)
    end
  end
end

```

rys. 50: zminay w pliku `features/support/path.rb`

2. Zmienić zachowanie kontrolera Devise
(app/controllers/devise/passwords_controller.rb) w odpowiedzi na błędny e-mail.

```
class Devise::PasswordsController < ApplicationController

  def create
    (...)
    if successful_and_sane?(resource)
      (...)
    else
      redirect_to(new_user_password_path, :alert =>
        resource.errors.full_messages)
    end
  end
end
```

rys. 51: 2. Zmiany w pliku app/controllers/devise/passwords_controller.rb

3. Dodać pola notyfikacji i błędu do widoku odzyskiwania hasła
(app/views/devise/passwords/new.html.haml)

```
%h2 Sign up
-if notice
  %p.notice= notice
-if alert
  %p.alert= alert
.eight.columns
(...)
```

Iteracja 5

8.5. Dodawanie treści przez użytkownika.

Zadania:

1. Utworzyć załączek internetowego forum, gdzie zarejestrowani użytkownicy mogą umieszczać wiadomości

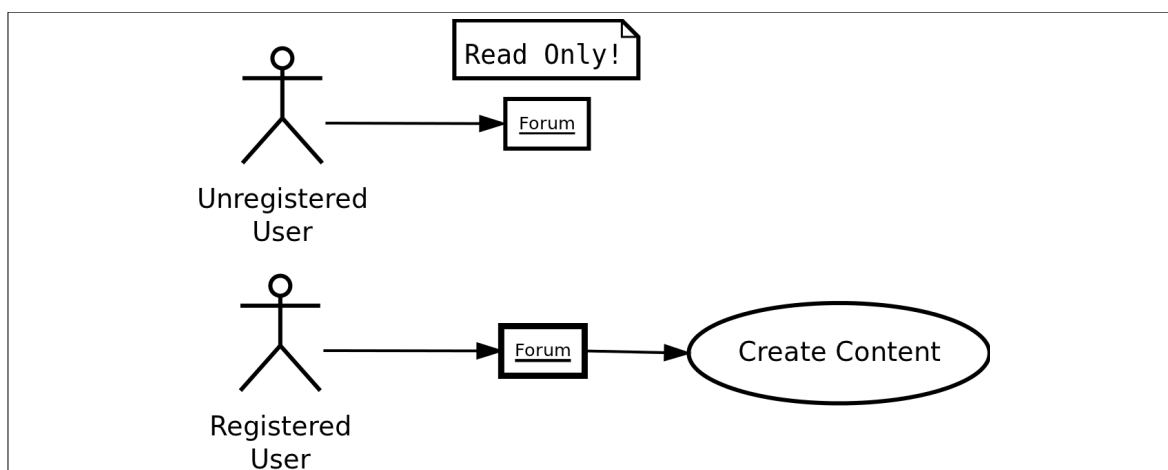


Fig. 52: Schemat funkcjonalności dodawania treści z wyróżnieniem osobnych ról dla użytkowników zalogowanych i niezalogowanych.

To ostatnia z demonstracyjnych funkcjonalności. Inaczej niż poprzednie, ta zostanie stworzona w całości od podstaw. Będzie też pierwszą funkcjonalnością w naszym systemie wprowadzającą potrzebę autoryzacji użytkownika.

W tym celu wykorzystamy zainstalowany na samym początku gem *CanCan*.

```
# language: pl
```

Właściwość: Używanie forum

@wip

Scenariusz: Widoczność forum dla użytkowników

Zakładając że jestem na stronie "root"

Oraz że nie jestem zalogowany

Oraz że widzę odnośnik "Forum"

Jeżeli kliknę odnośnik "Forum"

Wtedy trafię na stronę "forum"

Oraz będę widział listę postów

@wip 2

Scenariusz: dostęp do forum jako zalogowany użytkownik

Zakładając że jestem zalogowany jako użytkownik "ellmo"

Oraz że jestem na stronie "forum"

Oraz że widzę odnośnik "New post"

Jeżeli kliknę odnośnik "New post"

Wtedy trafię na stronę "new_post"

Oraz będę widział formularz dodawania posta


```

@wip 2
Scenariusz: dostęp do forum jako niezalogowany użytkownik
  Zakładając że nie jestem zalogowany
  Oraz że jestem na stronie "forum"
  Wtedy nie będę widział odnośnika "New post"

@wip 3
Scenariusz: dostęp do forum jako zalogowany użytkownik
  Zakładając że jestem zalogowany jako zarejestrowany użytkownik
    "ellmo"
  Oraz że jestem na stronie "new_post"
  Jeżeli wypełnię pole "post_content" tekstem
    "testowy tekst testowy"
  Oraz kliknę przycisk "Post"
  Wtedy trafię na stronę "forum"
  Oraz będę widział post "testowy tekst testowy" na liście postów

```

rys. 53: Funkcjonalność dodawania treści opisana w pliku *features/forum.feature*

Ponieważ aplikacja ma się rozwinąć, na wypadek powstania później dziesiątek widoków obsługiwanych przez różne kontrolery, będziemy potrzebowali prostych sposobów na sprawdzanie stanu uwierzytelnienia użytkownika.

Devise udostępnia kontrolerom globalną zmienną `current_user`, ale testując aplikację przez symulator *Capybara* dotykamy tylko zewnętrznej powłoki serwera, czyli widoku. Używając *Cucumbera* powinniśmy traktować go jak niezwykle dokładnego, ale jednak użytkownika systemu, a nie jego administratora. Symulacje wykonywane za pomocą *Capybary* nie mają dostępu do kontrolerów, nie mogą wykonywać operacji na modelach i zasobach. Z tego powodu kwestię logowania użytkownika musimy rozstrzygnąć za pomocą tego co nam znane – formularza logowania i odnośnika wylogowywania. Tylko w taki sposób mamy całkowitą pewność, jak będzie wyglądał stan sesji przeglądarki.

```

Zakładając /^że nie jestem zalogowany$/ do
  visit '/logout'
end

Zakładając /^że jestem zalogowany jako użytkownik "(.+)"$/ do |login|
  email = 'jkb.zuchowski@gmail.com'
  password = 'haslo123'
  User.new(:login => login, :email => email, :password => password,
    :password_confirmation => password).save!

  visit '/users/sign_in'
  fill_in "user_login", :with => email
  fill_in "user_password", :with => password
  click_button "Sign in"
end

```

Fig. 54: Nowe definicje kroków RSpec dodane do pliku *features/step_definitions/common_steps.rb*

Następnie utworzymy kontroler postów forum, wraz z widokami i ścieżkami w pliku `config/routes.rb` za pomocą generatora „rusztowań” (ang. *scaffold*):

```
rails g scaffold posts user_id:integer content:text
bundle exec rake db:migrate
bundle exec rake db:test:prepare
```

Capybara ma problem z wysyłaniem żądań innych niż GET / POST do serwera. Żądanie wyczyszczenia sesji – czyli wylogowania – jest obsługiwane za pomocą sztucznego żądania DELETE, co powoduje, że symulator nie może odnaleźć odpowiedniej akcji kontrolera *Devise* podążając za linkiem.

Musimy utworzyć ścieżki typu GET dla akcji logowania i wylogowywania, przy okazji stworzymy prostą ścieżkę dla listy postów forum, którą chcemy w systemie mieć pod adresem „/forum” – należy zwrócić uwagę, że kiedy nazwa strony w teście *Cucumber* pokrywa się z nazwą ścieżki w routerze aplikacji *Rails*, nie jest potrzebne jej dodawanie do listy ścieżek w `features/paths.rb`.

```
Mgr::Application.routes.draw do
  root :to => 'home#index'

  get "home/index"

  devise_for :users

  devise_scope :user do
    get "/login" => "devise/sessions#new"
    get "/logout" => "devise/sessions#destroy"
  end

  get "/forum" => "posts#index"
end
```

Fig. 55: Plik `config/routes.rb` z zaznaczonymi zmianami, wymaganymi przez nową funkcjonalność.

Następnie dodamy link do widoku forum na widoku głównym:

```
%h1 Home#index

-unless current_user
  %span
    (...)

-else
  (...)

%p
  = link_to "Forum", forum_path
```

Fig. 56: Plik `app/views/home/index.html.html`

Można teraz przystąpić do definiowania kroków *RSpec* w nowym pliku związanym z funkcjonalnością forum.

```
Wtedy /^będę widział listę postów$/ do
  page.should have_selector("#posts_list")
end

Wtedy /^będę widział formularz dodawania posta$/ do
  page.should have_selector("form#new_post")
end

Wtedy /^będę widział post "(.+) " na liście postów$/ do |post|
  page.should have_content(post)
end
```

Fig. 57: Plik *features/step_definitions/forum_steps.rb*

I kroki bardziej ogólne, które na pewno wykorzystamy w przyszłości w innych scenariuszach:

```
Wtedy /^nie będę widział odnośnika "(.+)"/ do |link|
  page.should_not have_link(link)
end
```

Fig. 58: Plik *features/step_definitions/common_steps.rb*

Kontroler postów po utworzeniu nowego postu będzie nas odsyłał do widoku zawierającego dane tego posta, jako zasobu aplikacji. Nie jest to zgodne z wytycznymi – ponownie sięgniemy do kontrolera, ale tym razem jest to nasz własny kontroler i bardzo łatwo jest znaleźć odpowiednie miejsce i dokonać zmianę:

```
class PostsController < ApplicationController
  (...)
  def create
    @post = Post.new(params[:post])

    respond_to do |format|
      if @post.save
        format.html { redirect_to forum_path,
          notice: 'Post was successfully created.' }
        format.json { render json: @post, status: :created,
          location: @post }
      else
        format.html { render action: "new" }
        format.json { render json: @post.errors,
          status: :unprocessable_entity }
      end
    end
  end
  (...)
end
```

rys. 59: Plik *app/controllers/posts_controller.rb*

Ostatnim krokiem, zanim otrzymamy w pełni przetestowaną, chociaż niezwykle prymitywną funkcjonalność demonstrującą umożliwienie użytkownikom umieszczania zawartości w sieci, jest zaprzęgnięcie do pracy prostego systemu autoryzacji *CanCan*. W obecnej chwili test na brak dostępu do formularza forum dla użytkowników niezalogowanych zakończy się niepowodzeniem, ponieważ w żaden sposób nie rozróżniamy użytkownika zarejestrowanego od zwykłego gościa, nie posiadającego konta w naszej bazie danych. *CanCan* został stworzony, by bezproblemowo współpracować z systemem *Devise*. Generator dostarczany razem z gemem (*cancan:ability*) utworzy plik klasy *Ability*, który określa jakie warunki musi spełnić użytkownik, by uzyskać dostęp do danego zasobu systemu. Domyślnie nie zawiera żadnych warunków, należy więc ją modyfikować:

```
class Ability
  include CanCan::Ability

  def initialize(user)
    user ||= User.new # guest user (not logged in)
    if user.id
      can :manage, :all
    else
      can :read, :all
    end
  end
end
```

rys. 60: Plik *app/models/ability.rb*

Sprawdzenie warunku *user.id* wystarczy, by odróżnić użytkowników zarejestrowanych (a więc tych, których konta znajdują się w bazie), od gości. Aby wymusić na zasobach modelu *Post* autoryzację użytkownika, należy dodać wywołanie następującej metody w kontrolerze *PostsController*:

```
class PostsController < ApplicationController
  load_and_authorize_resource
  (...)
end
```

rys. 61: Plik *app/controllers/posts_controller.rb*

Na koniec upewnimy się, że link dostępu do formularza forum jest niewidoczny w widoku *app/views/posts/index.html.haml* dla niezarejestrowanych użytkowników, i że test sprawdzający widoczność listy postów, będzie mógł ją znaleźć:

```
%h1 Listing posts
```

```
%table{ :id => "posts_list" }
  (...)
  -if can? :manage, Post
    = link_to "Forum", forum_path
```

9. Przydatne metody debugowania

Bindowanie danych i „migawka” widoku.

Bindowanie i robienie „migawki” widoku to dwie niezwykle przydatne metody, ułatwiające proces debugowania metod aplikacji podczas testów. Prezentowana tutaj implementacja jest specyficzna dla zestawu gemów, używanych w aplikacji. Bindowanie to jedna z najważniejszych funkcji klienta konsoli *PRY*, zaś migawka (ang. *snapshot*) to metoda udostępniona przez moduł *Capybary*.

Bindowanie (ang. wiązanie) to połączenie dwóch lub więcej strumieni danych do wspólnego źródła i synchronizacja zmian na danych źródłowych – w tym wypadku będzie to łączenie konsoli interpretera *PRY* do ciała metody na wysokości punktu wstrzymania (ang. breakpoint), dzięki czemu programista może własnoręcznie sprawdzić i ustawiać wartość wszystkich zmiennych, oraz przedefiniować fragmenty kodu (należy pamiętać o tym, że *Ruby* to język całkowicie otwarty, zezwalający na nadpisywanie i uzupełnianie nawet podstawowych metod, również na czas wykonywania – ang. *runtime*).

By poprawnie zbindować proces debugowania z wykonywaną metodą należy umieścić wywołanie metody `binding.pry` wewnątrz metody którą chcemy zbadać, a następnie zwyczajnie doprowadzić do wykonania tej metody, tak, by wynik wykonania został przekazany do strumienia wyjściowego – terminala, stanie się tak zarówno wtedy, gdy uruchomimy serwer aplikacji *Ruby on Rails* w terminalu korzystając ze standardowego polecenia `rails s`, jak i wtedy, gdy wywołamy metodę ręcznie wewnątrz istniejącej instancji konsoli *PRY*. Dla przykładu wykorzystamy pierwszą metodę kontrolera `posts_controller.rb` projektu:

```
class PostsController < ApplicationController
  load_and_authorize_resource

  # GET /posts
  # GET /posts.json
  def index
    @posts = Post.all
    binding.pry
    respond_to do |format|
      format.html # index.html.erb
      format.json { render json: @posts }
    end
  end
  (...)
end
```

Jeżeli standardowy serwer jest uruchomiony i drukuje wyniki swojej pracy w oknie terminala, odwiedzenie widoku, który spowoduje wywołanie tej akcji kontrolera (w tym przypadku `http://localhost:3000/forum`) zatrzyma pracę serwera i otworzy

zbindowaną do aktualnych danych w pamięci serwera konsolę systemu *PRY*:

```
[2012-01-29 15:16:15] INFO WEBrick 1.3.1
[2012-01-29 15:16:15] INFO ruby 1.9.2 (2011-07-09) [i686-linux]
[2012-01-29 15:16:15] INFO WEBrick::HTTPServer#start: pid=5172
port=3000

From: /home/ellmo/rails/mgr/app/controllers/posts_controller.rb
@ line 8 in PostsController#index:

3:
4:  # GET /posts
5:  # GET /posts.json
6:  def index
7:    @posts = Post.all
=> 8:    binding.pry
9:    respond_to do |format|
10:      format.html # index.html.erb
11:      format.json { render json: @posts }
12:    end
13:  end
[1] pry(#<PostsController>)>
```

Fig. 62: Zatrzymanie serwera w momencie wywołania bindowania i uruchomienie konsoli interpretera w odpowiednim kontekście.

Konsola uruchamia się w kontekście, w którym została umieszczona komenda `binding.pry`, oznacza to, że ma dostęp do wszystkich danych, które zdefiniowane były przed wywołaniem bindowania oraz że obiektem lokalnego kontekstu (do którego dostęp uzyskujemy za pomocą słowa kluczowego `self` – analogicznie jak w wielu innych językach, gdzie istnieje słowo kluczowe `this`), jest klasa bądź iterator, wewnątrz którego znajduje się wywołanie. W tym przypadku wywołanie `self` zwróci instancję kontrolera klasy `PostsController`. Można przejrzeć zawartość kolekcji `@posts`, sprawdzić czy spełnia założenia i które obiekty mogą sprawiać problemy. Można też dowolnie zmieniać kolekcję, tak by ręcznie wstawić do niej obiekty, których zachowanie chcemy przetestować po zamknięciu tej sesji bindowania.

Sesję zamykamy metodą `exit`.

Dla serwerów, które nie pracują w terminalu, tylko działają jako demony bądź usługi (serwery *Passenger* i *POW*), utworzono gem `pry-remote`. Działa on analogicznie do bindowania domyślnego interpretera *PRY*, ale zamiast otwierać nową sesję konsoli automatycznie, jedynie zatrzymuje działanie serwera i otwiera proces zdalnego bindowania, do którego można połączyć się z dowolnej konsoli wywołując polecenie `pry-remote` z poziomu powłoki systemu operacyjnego.

Drugą ważną i prostą metodą rozwiązywania problemów, a zwłaszcza niespójności między oczekiwaniami testu dotyczącego widoku strony, a jej stanu faktycznego, jest robienie „migawek”. Jest to funkcjonalność środowisk *Selenium* i *Capybara*, która pozwala zatrzymać stan aktualnie testowanego widoku i wyświetlić go w domyślnej przeglądarce, dzięki czemu można własnoręcznie przekonać się jak wygląda strona, jej struktura, formularze i style kaskadowe.

Żeby obejrzeć migawkę strony, wystarczy wywołać metodę `save_and_open_page` w odpowiednim miejscu, w definicji kroków testu. Dobrym przykładem byłoby umieszczenie takiego wywołania w definicji kroku wypełniania formularza logowania, który opisano w pliku `features/step_definitions/login_register_steps.rb`:

```
#encoding: utf-8

Jeżeli /^wypełnię formularz logowania danymi$/ do |table|
  table.hashes.each_with_index do |hash, index|
    fill_in("user_login", :with => hash[:user_login] )
    fill_in("user_password", :with => hash[:user_password])
    save_and_open_page
  end
end
```

Ten konkretny krok podczas pełnego testu wykonywany był czterokrotnie (raz dla każdego z możliwych scenariuszy: logowanie za pomocą adresu e-mail, za pomocą loginu, logowanie z błędnym loginem i logowanie z błędnym hasłem), jak wynika z szablonów scenariuszy zdefiniowanych w pliku `features/login_register.feature`:

Szablon scenariusza: logowanie emailem LUB loginem

```
(...)
Jeżeli wypełnię formularz logowania danymi
| user_login      | user_password  |
| <login_or_email> | <pass>         |
(...)

```

Przykłady:

```
| login_or_email      | pass          |
| ellmo               | haslo123     |
| jkb.zuchowski@gmail.com | haslo123     |

```

Szablon scenariusza: błędne logowanie

```
(...)
Jeżeli wypełnię formularz logowania danymi
| user_login      | user_password  |
| <login_or_email> | <pass>         |
(...)

```

Przykłady:

```
| login_or_email      | pass          |
| niepoprawny        | haslo123     |
| ellmo              | niepoprawne  |

```

W przypadku uruchomienia pełnego testu, system *Cucumber* otworzy cztery zakładki (bądź okna) w domyślnej przeglądarce, po jednym dla każdego przypadku.

Należy zwrócić uwagę, że „migawki” widoku nie są faktycznymi sesjami przeglądarki na adresie serwera aplikacji, tylko statycznymi plikami, przedstawiają odpowiedź serwera w danej chwili, „zrzuconą” do pliku HTML. Nie potrafią przyjmować żądań i parametrów.

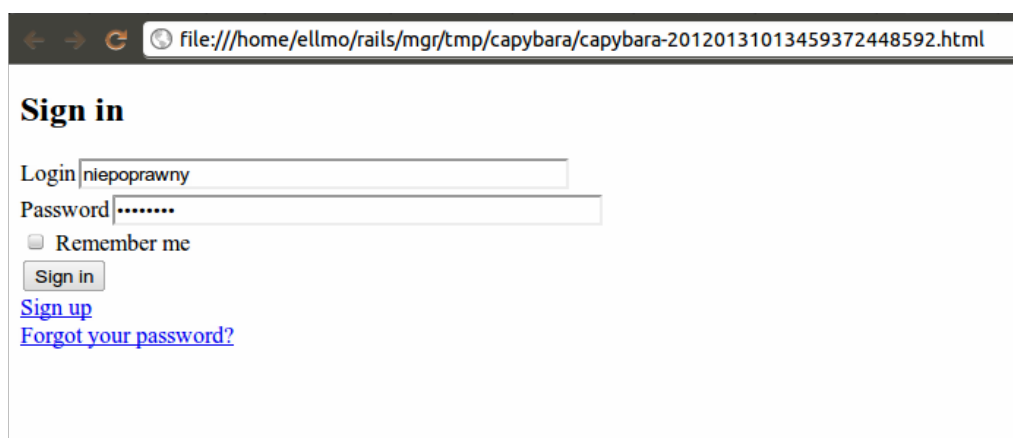


Fig. 63: Zrzut ekranu ukazujący "migawkę" widoku, otwartą w przeglądarce.

10. Prezentacja aplikacji

Utworzona aplikacja ma zaledwie kilka unikatowych widoków, ale reprezentują one najważniejsze aspekty zarówno typowej aplikacji Web2.0 jak i modelu *CRUD* (*Create-Read-Delete-Update* ang. Twórz, Czytaj, Usuń, Modyfikuj) dla zasobu postów. Obiekt użytkownika również jest zasobem, podobnie jak i jego sesja w przeglądarce.

Dostęp do widoku logowania jest możliwy z widoku startowego (ang. *root*).



Fig. 64: Widok główny (*root*) aplikacji.

Widoki logowania i rejestracji są jak najbardziej „standardowe”, w sposób, do którego przywykli internauci.

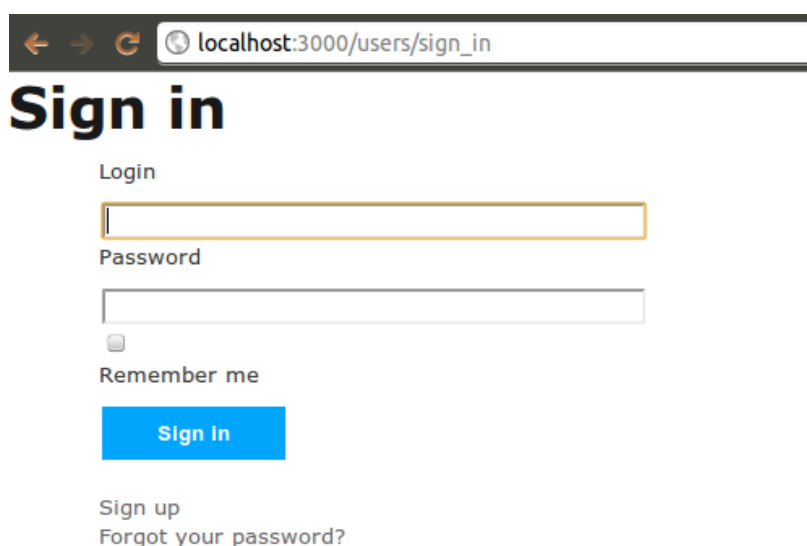
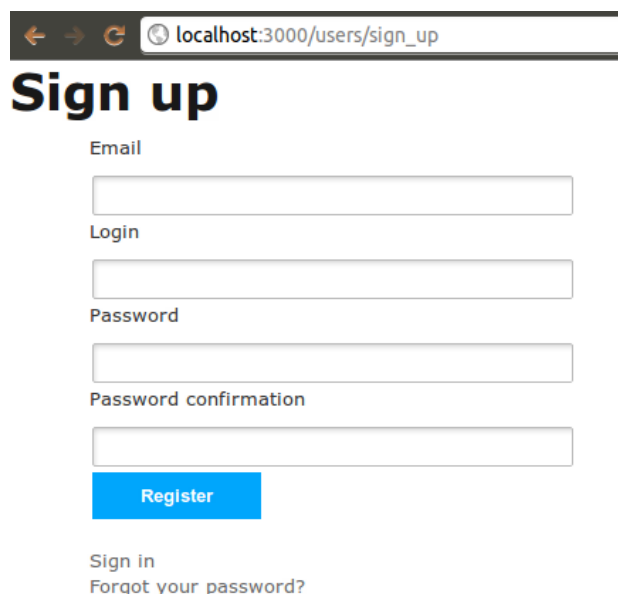


Fig. 65: Widok logowania do aplikacji.

Podobnie sprawa ma się z formularzem rejestracji. Jest przejrzysty i zawiera wystarczającą ilość informacji, żeby zidentyfikować użytkownika i niewystarczającą, by naruszyć jego anonimowość.



← → ↻ 🌐 localhost:3000/users/sign_up

Sign up

Email

Login

Password

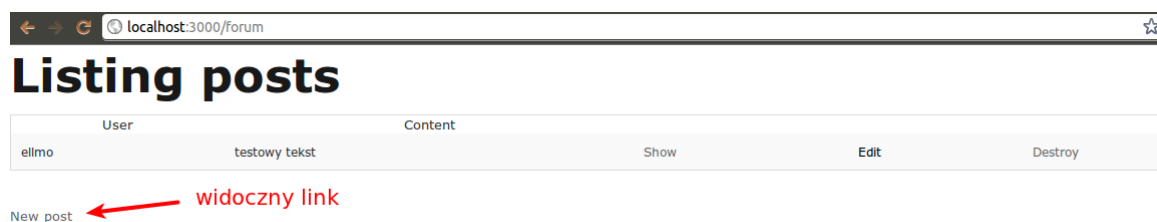
Password confirmation

Register

Sign in
Forgot your password?

Fig. 66: Widok rejestracji nowego użytkownika.

Zalogowany użytkownik może tworzyć nowe posty, ergo – dodawać nową treść do bazy danych wspierających aplikację, funkcjonalność, która jest podstawą wszystkich aplikacji webowych typu Web2.0.



← → ↻ 🌐 localhost:3000/forum ☆

Listing posts

User	Content	
ellmo	testowy tekst	Show Edit Destroy

New post ← widoczny link

Fig. 67: Widok forum (lista postów) dla zalogowanego użytkownika.

Niezalogowany użytkownik (sesja anonimowa) może przeglądać posty innych użytkowników, ale system nie pozwoli mu dodać nowej zawartości, przede wszystkim ukrywając przed nim link:



← → ↻ 🌐 localhost:3000/forum

Listing posts

User	Content
ellmo	testowy tekst

brak linku

Fig. 68: Widok forum dla niezalogowanego użytkownika.

Jeżeli niezalogowany użytkownik spróbuje wejść do widoku dodawania postów „ręcznie”, uzyskując w ten sposób niedozwolony dostęp do ścieżki `/posts/new` (adres formularza dodawania nowych postów), próba ta zostanie wyłapana przez system *CanCan*:

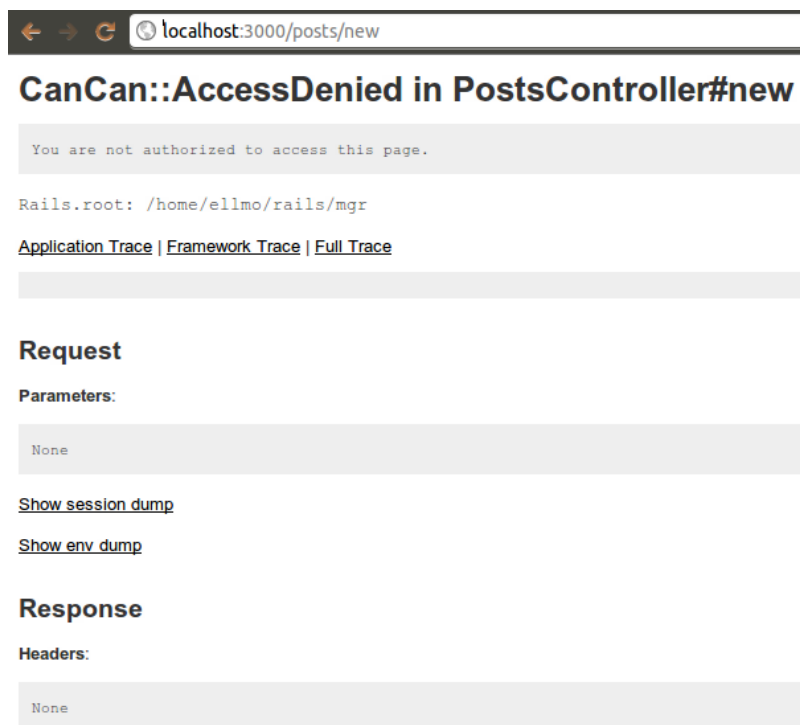


Fig. 69: Niedozwolona próba dostępu do formularza dodawania postów.

W środowisku produkcyjnym powyższy widok zostanie zastąpiony domyślną planszą dla odpowiedzi o statusie 403 (brak dostępu) dla serwera.

11. Podsumowanie

Utworzono od podstaw nieskomplikowaną aplikację internetową, wykorzystując w tym celu otwarte technologie oparte o Model Zwinny i skupione wokół społeczności związanej z językiem programistycznym *Ruby*.

Wykorzystano techniki Rozwoju Sterowanego Behawioralnie: *Cucumber*, *RSpec*, *Capybara* i *Selenium*, by opisać i testować funkcjonalności związane z bardzo różnorodnymi zagadnieniami:

- Uwierzytelnianiem i autoryzacją
- Automatycznym wysyłaniem maili
- Rozwijaniem kontrolerów i widoków aplikacji
- Umożliwianie użytkownikom umieszczania treści w systemie.

Opisano pracę systemu *Capybara* / *Selenium*, symulatora przeglądarki, który można wykorzystywać również poza zakresem prac z Modelem Behawioralnym

Testy behawioralne *Cucumber* stanowiły wytyczne podczas pracy, a po jej zakończeniu stanowią doskonałe źródło dokumentacji, utworzone na ich potrzeby testy *RSpec* zapewniają większą stabilność systemu w przyszłości.

Praca udowadnia, że metodyka *BDD* wpisuje się w Iteracyjno-Przyrostowy model pracy, i że funkcjonalności *Cucumber* mogą zastąpić grafy Przypadków Użycia i Wykresu Komunikacji.

Test całościowy to 19 pełnych scenariuszy, na które składało się 118 kroków.

Spis ilustracji

Fig. 1: Schemat refaktoryzacji kodu zgodnie z opisem funkcjonalności w technologii BDD.	13
Fig. 2: Pierwsza część definicji przykładowej właściwości w języku Gherkin.....	14
Fig. 3: Opis szablonu scenariusza w języku Gherkin.....	15
Fig. 4: Blok opisu serii danych dla testu akceptacyjnego.....	15
Fig. 5: Struktura dokumentu HTML widziana z perspektywy inspektora DOM (w przeglądarce Google Chrome).....	17
Fig. 6: Schemat komunikacji z użyciem technologii AJAX.	18
Fig. 7: Schemat architektury MVC.....	23
Fig. 8: Rzeczywiste użycie funkcjonalności zdefiniowanych w modelu kaskadowym (za: Craig Larman[2]).....	26
Fig. 9: Początkowa zawartość pliku Gemfile.....	32
Fig. 10: Pojedynczy przypadek użycia prostego systemu logowania.....	38
Fig. 11: Prosty diagram przepływu funkcjonalności logowania.....	39
Fig. 12: opisana migracja w pliku db/migrate/20111230120140_devise_create_users.rb (pliki migracji zawsze zawierają dane o czasie, w którym zostały utworzone).....	40
Fig. 13: Definicja fabryki użytkowników w pliku spec/factories/user.rb.....	41
Fig. 14: opis funkcjonalności w pliku features/authenticate.feature.....	43
Fig. 15: Treść pliku features/step_definitions/authenticate.rb (uproszczono zapis wyrażeń regularnych wewnątrz cudzysłowów).....	45
Fig. 16: Przeredagowana pierwsza metoda kroku RSpec, utworzonego wcześniej (Fig. 15.) w pliku features/step_definitions/authenticate.rb.....	47
Fig. 17: Plik features/step_definitions/authenticate.rb z uzupełnionymi metodami kroków.	48
Fig. 18: metoda zdefiniowana w domyślnie utworzonym pliku features/step_definitions/web_steps.rb.....	48

Fig. 19: Metoda <code>path_to</code> w pliku <code>features/support/paths.rb</code>	49
Fig. 20: Zmiany w metodzie <code>path_to</code> w pliku <code>features/support/paths.rb</code>	50
rys. 21: <code>app/views/devise/session/new.html.haml</code>	51
Fig. 22: metoda w <code>~/rvm/gems/ruby-1.9.2-p290-patched/gems/devise-1.4.9/lib/devise/models/database_authenticatable.rb</code>	52
Fig. 23: zmiany w pliku <code>app/models/user.rb</code>	52
Fig. 24: zmiany w pliku <code>config/initializers/devise.rb</code>	53
Fig. 25: Funkcjonalność rejestracji to nowy przypadek użycia dla użytkowników.....	54
Fig. 26: Schemat komunikacji podczas rejestracji nowego użytkownika.....	54
Fig. 27: fragment pliku <code>features/support/paths.rb</code>	56
Fig. 28: Zmodyfikowany plik <code>features/step_definitions/login_register.feature</code>	57
Fig. 29: <code>features/step_definitions/common.rb</code>	58
Fig. 30: obsługa warunku widoczności formularza w pliku <code>features/step_definitions/login_register_steps.rb</code>	58
rys. 31: schemat użycia słów kluczowych i składni szablonu scenariusza.....	58
Fig. 32: obsługa nowego kroku wypełniania formularza logowania w pliku <code>features/step_definitions/login_register_steps.rb</code>	59
Fig. 33: obsługa nowego kroku wypełniania formularza logowania w pliku <code>features/step_definitions/login_register_steps.rb</code>	60
Fig. 34: Plik <code>app/views/home/index.html.haml</code> (wyróżniono zmiany w porównaniu do poprzedniej wersji pliku).....	60
rys. 35: Plik <code>app/views/devise/registrations/new.html.haml</code> (wyróżniono zmiany w porównaniu do poprzedniej wersji pliku).....	60
Fig. 36: scenariusz błędnego logowania umieszczony w pliku <code>features/login_register.feature</code>	61
Fig. 37: obsługa nowego kroku dodana do pliku <code>features/step_definitions/common.rb</code>	61
Fig. 38: Szablon scenariusza niepoprawnego logowania w pliku	

features/login_register.feature.....	62
Fig. 39: obsługa nowego kroku sprawdzania początkowe liczny użytkowników dodana do pliku features/step_definitions/common.rb.....	62
rys. 40: features/step_definitions/common.rb.....	62
Fig. 41: Przypadek użycia odzyskania hasła powinien zawierać się w kontekście logowania.....	64
Fig. 42: Schemat komunikacji dla funkcjonalności odzyskiwania hasła.....	64
Fig. 43: Plik features/mailer.feature.....	65
Fig. 44: Nagłówki metod przeniesionych do pliku features/step_definitions/common_steps.rb.....	66
Fig. 45: Plik features/step_definitions/mailer_steps.rb.....	66
Fig. 46: Zawartość pliku config/environments/secret_data.rb, należy w nim podać prawdziwe kredencjały konta pocztowego, które chcemy wykorzystać w aplikacji.....	67
Fig. 47: Wspólny fragment plików konfiguracyjnych środowiska testowego i rozwojowego (development), definiujący parametry skrzynki pocztowej, wykorzystywanej przez ActionMailer.....	68
rys. 48: zminay w pliku features/support/path.rb.....	68
rys. 49: 2. Zmiany w pliku app/controllers/devise/passwords_controller.rb.....	69
Fig. 50: Schemat funkcjonalności dodawania treści z wyróżnieniem osobnych ról dla użytkowników zalogowanych i niezalogowanych.....	70
rys. 51: Funkcjonalność dodawania treści opisana w pliku features/forum.feature.....	71
Fig. 52: Nowe definicje kroków RSpec dodane do pliku features/step_definitions/common_steps.rb.....	71
Fig. 53: Plik config/routes.rb z zaznaczonymi zmianami, wymaganymi przez nową funkcjonalność.....	72
Fig. 54: Plik app/views/home/index.html.haml.....	72
Fig. 55: Plik features/step_definitions/forum_steps.rb.....	73
Fig. 56: features/step_definitions/common_steps.rb.....	73

rys. 57: Plik app/controllers/posts_controller.rb.....	73
rys. 58: Plik app/models/ability.rb.....	74
rys. 59: Plik app/controllers/posts_controller.rb.....	74
Fig. 60: Zatrzymanie serwera w momencie wywołania bindowania i uruchomienie konsoli interpretera w odpowiednim kontekście.....	76
Fig. 61: Zrzut ekranu ukazujący "migawkę" widoku, otwartą w przeglądarce.....	78
Fig. 62: Widok główny (root) aplikacji.....	79
Fig. 63: Widok logowania do aplikacji.....	79
Fig. 64: Widok rejestracji nowego użytkownika.....	80
Fig. 65: Widok forum (lista postów) dla zalogowanego użytkownika.....	80
Fig. 66: Widok forum dla niezalogowanego użytkownika.....	80
Fig. 67: Niedozwolona próba dostępu do formularza dodawania postów.....	81

Bibliografia

- [1] Peter Seibel „Sztuka kodowania” Helion, Gliwice, 2011
- [2] Craig Larman „UML i wzorce projektowe”, Helion, Gliwice, 2011
- [3] David Flanagan, Yukihiro Matsumoto „Ruby. Programowanie”, Helion, Gliwice, 2009
- [4] Why The Lucky Stiff „Why's (Poignant) Guide to Ruby”
- [5] David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, Dan North „The RSpec book”, The Pragmatic Bookshelf, 2010
- [6] Scott Raymond „Ajax on Rails”, Helion, Gliwice 2008