POLITECHNIKA WROCŁAWSKA

WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka (INF)
SPECJALNOŚĆ: Inżynieria systemów informatycznych (INS)

PRACA DYPLOMOWA MAGISTERSKA

Analiza porównawcza popularnych frameworków webowych.

Comparative analysis of most popular web frameworks.

AUTOR:

inż. Marcin Mantke

PROWADZĄCY PRACĘ:

dr inż. Roman Ptak

OCENA PRACY:

WROCŁAW 2017

Spis treści

Rozdział 1

Wstęp

1.1 Ogólny opis pracy

W momencie pisania pracy istnieje niezliczona ilość frameworków webowych. Prawie codziennie, dla samego języka JavaScript, powstaje jeden nowy (micro) framework. Przyczyn takiego stanu rzeczy jest kilka. Po pierwsze, problemy, z jakimi spotykają się programiści są bardzo zróżnicowane oraz skomplikowane. Wynika to z nacisku biznesu, czyli klientów, na to, aby nowo powstały proukt był innowacyjny. Programista oczywiście posiada narzędzia, które powinny być wystarczające do rozwiązania powierzonego mu problemu, jednakże niekiedy korzystanie z narzuconych przez framework rozwiązań wręcz utrudnia wykonanie powierzonej pracy. Z tego powodu powstają nowe frameworki, które udostępniają zestaw narzędzi ukierunkowany pod rozwiązanie nowego, konkretnego problemu. Druga przyczyna jest w pewien sposób powiązana z pierwszą. Jest to stworzenie frameworku nie w odpowiedzi na potrzebę, ale wygenerowanie potrzeby poprzez stworzenie frameworku, który ułatwia rozwiązanie przykładowego problemu. Kolejnym powodem jest próba odchudzenia istniejących frameworków. Przykładem może być Ruby on Rails, który jest frameworkiem kompletnym, ale niekiedy posiadającym zbyt dużo wbudowanych funkcji, z których trudno jest zrezygnować, a które w danym projekcie nie zostaną wykorzystane. Zwiększa to rozmiar oraz złożoność projektu, co oczywiście jest niekorzystne.

Ilość dostępnych frameworków pokazuje jak ważnym elementem stały się one dla programistów. Niestety, tak dynamiczny rozwój rozwiązań tego typu powoduje spory problem jeśli chodzi o wybór technologii. Ninejsza praca ma na celu zaprezentowanie wybranych frameworków webowych oraz dokonanie porównania ich funkcjonalności oraz wydajności.

Analiza porównawcza ma na celu wyszczególnienie cech frameworków, na które programista powinien zwrócić szczególną uwagę przy doborze frameworku.

Jako że framework dodaje do aplikacji pewną warstwę abstrakcji, czyli kod, naturalny jest narzut wydajnościowy na aplikację. z tego powodu, poza różnymi cechami odnośnie budowy i dostarczanych funkcjonalności, frameworki różnią się również wydajnością.

1.2 Cel pracy

Celem niniejszej pracy jest dokonanie analizy porównawczej wybranych frameworków webowych. Analiza ta ma posłużyć do wyciągnięcia wniosków na temat cech poszczególnych rozwiązań, a także ich wydajności. Dokumentacja zawiera opis implementacji testowej aplikacji przy pomocy każdego z frameworków oraz wnioski z analizy testów.

Aby poprawnie zrealizować cel pracy, stworzono aplikację testową. W celu urzeczywistnienia problemu, który rozwiązuje owa aplikacja, postanowiono stworzyć rozwiązanie umożliwiające zarządzanie inteligentnym domem. Aplikacja ta pełni rolę jednostki zarządzającej podzespołami inteligentnego domu oraz zbierającej dane z czujników. Spełnia ona następujące wymagania funkcjonalne:

- rejestracja i logowanie użytkowników,
- przetwarzanie danych w tle,
- możliwość dodawania, edycji oraz usuwania sensorów,
- możliwość dodawania oraz usuwania danych z sensorów,
- przypisywanie danych do sensorów.

Aplikacja spełnia również następujące wymagania niefunkcjonalne:

- połączenie z bazą danych PostgreSQL,
- zapewnienie skalowalności aplikacji,
- posiadanie testów jednostkowych.

Rozdział 2

Przedstawienie omawianych technologii

2.1 Historia i rozwój technologii webowych

Od lat 90 XX wieku świat obserwuje bardzo dynamiczny rozwój technologii związanych z Internetem. Począwszy od roku 1991, kiedy to naukowcy z instytu badawczego **CERN** (ang. *European Organization for Nuclear Research*) opracowali standard WWW, przed programiastami zaczęła się otwierać nowa gałąź tworzenia aplikacji, którą są aplikacje internetowe. Początkowo aplikacje te były jedynie statycznymi stronami WWW, na których znajdował się jedynie tekst. Wprowadzenie kaskadowych arkuszy styli (*CSS*) w roku 1996 sprawiło, że strony internetowe przybrały graficzną formę. Rok 1997 przyniósł obsługę języka *JavaScript* w przeglądarkach internetowych. Oznaczało to, że strony internetowe, poza statycznymi elementami, zyskały elementy dynamiczne, np. reagujące na akcje użytkownika.

Wraz ze wzrostem dostępu ludzi do Internetu rozwijały się technologie odpowiedzialne za strony internetowe. Za punkt początkowy istnienia nie stron, a aplikacji internetowych, można przyjać rok 1997 i powstanie języka *PHP*. Był to pierwszy interpretowany skryptowy język programowania, który służył do budowania aplikacji internetowych działających w czasie rzeczywistym. Wraz z rozwojem języka PHP oraz innych, podobnych mu języków, np. *Python* i *Ruby*, zmienił się sposób budowania aplikacji. Programiści zaczęli rezygnować ze standardowych klientów w postaci aplikacji desktopowych i przechodzili na tzw. cienkich klientów (ang. *thin client*). Trend ten przyspiesza rozwój oraz różnorod-

ność aplikacji serwerowych posiadających interfejs graficzny w formie strony internetowej, które nazywane są aplikacjami internetowymi [1].

2.2 Wykorzysywane pojęcia i terminy

2.2.1 Aplikacja internetowa

Aplikacja internetowa (webowa) jest aplikacją znajdującą się nie na komputerze użytkownika, lecz na ogólnodostępnym serwerze. Komunikacja pomiędzy, niekiedy rozproszonymi, elementami aplikacji odbywa się poprzez sieć komputerową. Aplikacja webowa swój interfejs graficzny poprzez przeglądarkę internetową bądź np. aplikację mobilną.

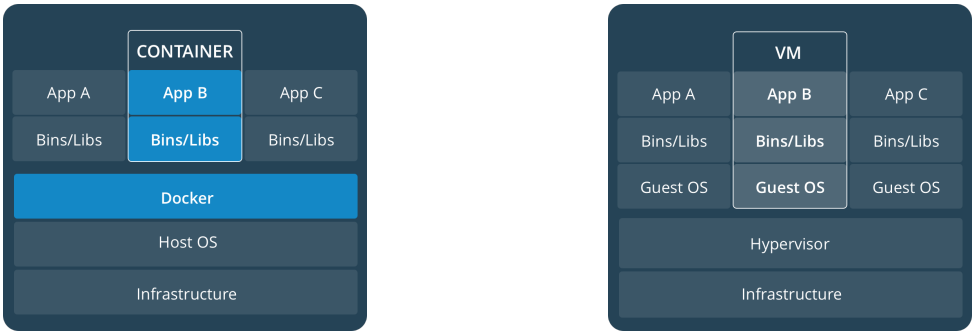
2.2.2 Framework webowy

Aby ułatwić korzystanie z coraz liczniejszych technologii wykorzystywanych w tworzeniu aplikacji internetowych, powstały narzędzia nazywane frameworkami webowymi. Framework jest uniwersalnym środowiskiem programistycznym, które dostarcza niezbędne narzędzia wymagane do stworzenia aplikacji internetowej w wybranym języku programowania [2]. Każdy z frameworków dostarcza pewną abstrakcję, która znajduje się wokół kodu napisanego przez programistę. Przykładem takiej abstrakcji jest system mapowania ścieżki podstrony (np. */users/3*) na konkretną akcję w aplikacji (zwykle akcja *SHOW* dla kontrolera *Users*), czyli *routing*. Twórcy frameworków zauważyli, że w każdej aplikacji webowej są stosowane te same typy rozwiązań, więc w wielu przypadkach wprowadzili dane rozwiązania jako integralne części frameworków. W efekcie programiści mogą korzystać z gotowych, dogłębnie przetestowanych rozwiązań, które znajdują się w 90% aplikacji webowych.

2.2.3 Wirtualizacja, kontener

W celu zapewnienia izolacji środowiska działania aplikacji, stosuję się technikę nazywaną wirtualizacją. Polega ona na stworzeniu wirtualnej maszyny, bądź też kontenera, gdzie uruchamiana jest aplikacja. Rozwiązania te, pomimo osiągania podobnego efektu, różnią się pomiędzy sobą pod wieloma względami.

Korzystając z informacji zawartych w dokumentacji Docker'a [10], można wykazać następujące cechy obu rozwiązań:



Rysunek 2.1: Porównanie kontenerów i wirtualnej maszyny.

- maszyny wirtualne:
  - abstrakcja w warstwie sprzętowej, która zmienia jeden serwer w wiele serwerów,
  - hipernadzorca (ang. *hypervisor*) pozwala na pracę wielu wirtualnych maszyn na jednej maszynie fizycznej,
  - każda maszyna wirtualna posiada pełną kopię systemu operacyjnego, aplikacji, niezbędnych bibliotek i ich zależności, przez co zajmuje dużo miejsca na dysku twardym,
  - maszyny wirtualne zwykle długo się uruchamiają.
- kontenery:
  - abstrakcja w warstwie aplikacji, która łączy kod z jego zależnościami,
  - na jednej maszynie może być uruchomionych wiele kontenerów, które współdzielą kernel,
  - każdy z kontenerów jest osobnym procesem w przestrzeni użytkownika,
  - kontenery zajmują mniej miejsca na dysku i uruchamiają się dużo szybciej, niż maszyny wirtualne.

Powyższe zestawienie należy uzupełnić o informacje znajdujące się w prezentacji Łukasza Piątkowskiego [11]:

- maszyny wirtualne:
  - wirtualizacja sprzętu - degradacja wydajności,
  - własny kernel,

  - pełna maszyna,

- kontenery:

  - *chroot* z większymi możliwościami,
  - wspólny kernel,
  - własne zasoby pamięci, dysku, I/O.

Analizując cechy obu rozwiązań można stwierdzić, że kontenery mają sporą przewagę w ważnych kwestiach (wydajność, szybkość uruchamiania, wymagane zasoby) nad maszynami wirtualnymi, dlatego też stają się coraz bardziej popularne.

Ze strony programisty, otrzymuje on nową instancję wybranego systemu, gdzie zainstalowane są jedynie zależności wymagane przez daną aplikację. Prowadzi to do odizolowania środowiska aplikacji, przez co znacząco minimalizowane jest ryzyko wystąpienia konfliktów pomiędzy aplikacjami, aplikację można w bardzo prosty sposób przenieść na inną maszynę i uruchomić ją jedną komendą.

2.2.4 CRUD

CRUD, to cztery podstawowe funkcje w aplikacjach korzystających z pamięci trwałej, które umożliwiają zarządzanie nią [12]. Akronim ten powstał od pierwszych liter słów **C**reate, **R**ead, **U**pdate i **D**elete. Określa on udostępnienie użytkownikowi bądź aplikacji zestaw operacji (stworzenie, odczytanie, aktualizacja oraz usunięcie) na danym obiekcie.

Rozdział 3

Przegląd wybranych rozwiązań

3.1 Ruby on Rails

Ruby on Rails jest open source'owym frameworkiem webowym. Został stworzony w głównej mierze przez duńskiego programistę Davida Heinemeiera Hanssona. Pierwsza wersja RoR ukazała się w lipcu 2004 roku. Fakt ten pokazuje, że jest to już dojrzały framework, z ugruntowaną pozycją na rynku. Potwierdzeniem tej pozycji jest liczne wsparcie społeczności. Rails'y zostały stworzone przez ponad 4,5 tysiąca osób, istnieje ok. 132 tysiące ogólnodostępnych paczek, które rozszerzają funkcjonalność tego frameworku. Ruby on Rails domyślnie korzysta z architektury MVC, czyli *Model - View - Controller*.

3.1.1 Doktryna Ruby on Rails

Tworzeniu i rozwojowi Ruby on Rails towarzyszy kilka podstawowych zasad. Jedne istnieją od początku, inne ewoluowały na przestrzeni lat. Spisane są one w *The Rails Doctrine* [4]. Na potrzeby pracy zostaną omówione najważniejsze z nich.

Optimize for programmer happiness

Pierwsza reguła odnosi się mocno do języka, z którego wywodzą się Rails'y. Sam fakt umieszczenia nazwy języka Ruby w nazwie frameworku pokazuje jak ważny jest on dla całego projektu. Motywacją stworzenia języka Ruby była chęć dostarczenia programistom radości z pisania kodu. w momencie powstawania Ruby'ego, większość popularnych wtedy języków programowania narzucało sposób pisania bądź stosowało liczne ograniczenia w sposobie pisania kodu. Ruby stał w opozycji do tych zasad, dając programistom

pełną dowolność w sposobie tworzenia kodu.

Jako przykład owego „uwolnienia" programisty podawana jest *Zasada Najmniejszego Zaskoczenia* (ang. The Principle of Least Surprise).

Listing 3.1: Wyjście z interpretera Ruby'ego.

```
$ irb
irb(main):001:0> exit
$ irb
irb(main):001:0> quit
```

Listing 3.2: Wyjście z interpretera Python'a.

```
$ python
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
```

Ruby zaakceptuje oba polecenia opuszczenia interpretera. Python natomiast, pomimo odczytania intencji programisty (wyświetlenie instrukcji), opuści interpreter jedynie po wpisaniu komendy *exit()* lub po użyciu kombinacji klawiszy *Ctrl-D*, co oczywiście nie jest zgodne z oczekiwaniami programisty.

Wzorując się na zasadach, na których powstawał Ruby, Rails'y również miały umilać życie programistom. Jako przykład jest podawana klasa *Inflector*. Zapisane są w niej reguły oraz wyjątki od reguł w języku angielskim, które umożliwiają mapowanie klasy *Comment* na tabelę *Comments*, *Person* na *People* itp. Kolejnym przykładem może być dodatek do Ruby'owej klasy *Array*, który poza dostępnym w Rubym pierwszym elementem tablicy, umożliwia dostęp do kolejnych czterech elementów tablicy, poprzez wyrażenia *Array#second, Array#third* itd.

Oczywiście nie są to kluczowe cechy frameworku, ale stanowczo zaznaczają ważkość celu, którym jest przede wszystkim sprawianie radości z tworzenia oprogramowania.

Convention over Configuration

Twórcy Ruby on Rails starają się kłaść mocny nacisk na prostotę używania ich narzędzia. Zamiast zrzucać na programistów ciągłe podejmowanie decyzji w kwestiach mało istotnych, jak na przykład format klucza obcego w bazie danych, Rails'y narzucają konwencję nazewnictwa. Oczywiście jest możliwość konfiguracji narzuconych przez framework konwencji, lecz jest to raczej rzadko spotykane, a wręcz niewskazane. Zasada ta, poza

zdjęciem odpowiedzialności za część decyzji z barków programisty, przynosi również inne korzyści.

Dzięki „domyślnej" konwencji, możliwe jest stworzenie głębszej abstrakcji, na której operuje framework. Przykładem może być zastosowanie wcześniej wymienionej klasy *Inflector*. Jeśli możliwe jest zmapowanie klasy *Person* na tabelę *People*, to możliwe jest również zmapowanie relacji *has_many: people* w taki sposób, aby wykorzystywana była klasa *Person*. Jest to o tyle wygodne rozwiązanie, że nawet pomimo posiadania wiedzy na temat tworzenia relacji w bazie danych oraz sposobu odzwierciedlania ich w kodzie aplikacji, programista nie musi przejmować się tworzeniem lub konfiguracją tej części aplikacji.

Kolejną zaletą jest znaczące obniżenie progu wejścia dla początkujących programistów. Takim osobom dużo łatwiej jest poznawać framework stopniowo. Począwszy od poziomu, gdzie wszystko automatycznie działa, lecz nie wiadomo dlaczego, aż do momentu gdzie nadal wszystko automatycznie działa, ale bardzo dobrze wiadomo dlaczego. Znaczna część mechanizmów stosowanych we frameworkach webowych *de facto* nie wymaga niestandardowej konfiguracji, nawet jeśli programista posiada szeroką wiedzę w danej dziedzinie. Przekładanie konfiguracji ponad konwencję wymaga od programisty sporego wysiłku, aby rozpocząć pracę z frameworkiem, co w przypadku poznawania nowych technologii stanowczo nie jest zachęcające.

Zasada ta bywa jednak zgubna. Jest tak z powodu błędnego przekonania, że skoro od samego początku wszystkie elementy aplikacji można wygenerować, i od samego początku wszystko działa, to programista nie musi mieć wiedzy na temat tego, co robi. W przypadku bardzo podstawowych zastosowań wiedza programisty rzeczywiście nie musi być szeroka, natomiast bardzo problematyczne jest w takim przypadku wykonanie części aplikacji, która jest niestandardowa bądź niemożliwa do wygenerowania.

The menu is omakase

Rzadko kiedy framework jest monolitem, który nie składa się z modułów. Częściej framework jest zbiorem mniejszych frameworków lub bibliotek, które współpracują ze sobą. Bardzo często wybór owych narzędzi leży w pełni po stronie programisty. Jako analogię takiego wyboru, w *The Ruby on Rails Doctrine* podawany jest problem wyboru dania z menu w restauracji, którą odwiedzamy po raz pierwszy. Jeśli zdamy się na wybór szefa kuchni, możemy założyć że jedzenie będzie dobre, nie wiedząc jeszcze co „dobre" będzie oznaczać [4].

Ruby on Rails rozwiązuje ten problem poprzez dostarczenie zestawu narzędzi, z których programista może korzystać. Zasada działania jest tutaj bardzo zbliżona do reguły *Convention over Configuration*, lecz operuje na wyższym poziomie abstrakcji. Programista dostaje zestaw domyślnych narzędzi, z którego może od samego początku korzystać, bez potrzeby podejmowania decyzji odnośnie wyboru frameworków i bibliotek. Ma on natomiast możliwość zamiany domyślnie wybranych narzędzi na inne, jeśli widzi taką potrzebę.

Zasada ta niesie za sobą korzyści w sferze rozwoju frameworku. Jest tak, ponieważ programiści, używając tych samych narzędzi, są w stanie bardziej dopracować owe narzędzia. Częstym problemem jest nie fakt jak działa dany framework bądź biblioteka w izolacji, lecz jak działa razem z innymi modułami. Jeśli wielu programistów napotyka te same błędy w integracji poszczególnych składowych frameworku, to dużo łatwiej jest twórcom takie błędy poprawić bądź usprawnić połączenie tych modułów.

No one paradigm

Twórcy frameworka Ruby on Rails są przekonani, że nie istnieje jedno idealne rozwiązanie. Często do jednego celu można dojść różnymi drogami. Z tego też powodu, pomimo wyboru architektury MVC, Railsy są bardzo elastyczne jeśli chodzi o dostosowanie się do innych modeli projektowych. Domyślnie RoR nie posiada wbudowanych *Serwisów* bądź *Prezenterów*, lecz framework jest zbudowany w taki sposób, aby użycie większości wzorców projektowych było bezproblemowe.

Przystosowanie frameworku do korzystania z różnych wzorców projektowych poniekąd wymusza na programistach znajomość większej ilości owych wzorców. Jako, że Ruby jest bardzo elastycznym językiem jeśli chodzi o paradygmaty programowania, ponieważ możliwe jest pisanie funkcyjne, Railsy również wspierają owe podejście. Od programisty zależy, czy z takiej możliwości skorzysta. Wprowadzenie takiej uniwersalności niestety niesie za sobą spory nakład pracy ze strony twórców.

3.1.2 Pozostałe cechy

Poza cechami wymienionymi w *The Ruby on Rails Doctrine*, Ruby on Rails kładzie mocny nacisk na inne aspekty. Są to m.in. kwestie takie, jak przestrzeganie reguły *DRY* (ang. *Don't Repeat Yourself*) bądź też wsparcie dla dodatkowych bibliotek rozszerzających

funkcjonalność frameworku.

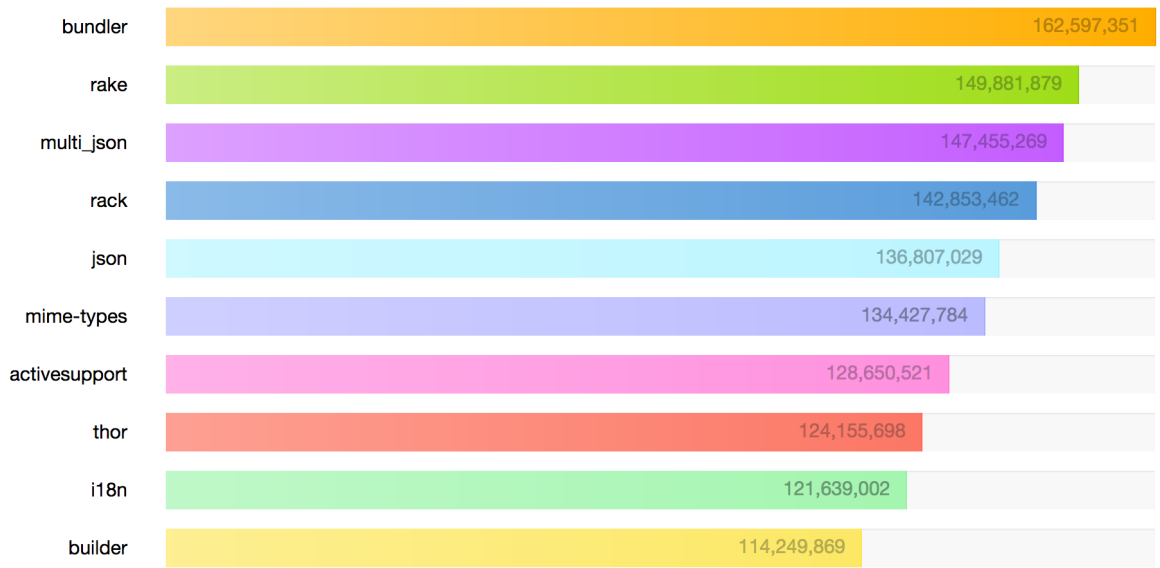
Don't Repeat Yourself

Reguła *DRY* mówi o tym, aby, jak sama nazwa wskazuje, nie powtarzać kodu. Każde powtórzenie logiki biznesowej może być rozwiązane poprzez dodanie warstwy abstrakcji. Natomiast duplikację procesów można rozwiązać poprzez ich automatyzację [9]. Poprzez stosowanie tej reguły, kod aplikacji jest wyraźnie mniejszy, ale przede wszystkim jest on bardziej odporny na błędy i łatwiejszy w utrzymaniu. Jeśli w powtarzającym się fragmencie kodu zostanie popełniony błąd, dzięki zastosowaniu reguły DRY, zmianę trzeba wprowadzić tylko w jednym miejscu, a nie w każdym wystąpieniu powtarzającej się funkcji.

Ruby on Rails wspiera używanie tej reguły poprzez wbudowane w framework elementy, takie jak *Helpery*, *Concerny* lub *Partiale*. Umożliwiają one współdzielenie kodu i używanie ich w każdej z warstw aplikacji.

Biblioteki

Biblioteki tworzone przez społeczność nie są bezpośrednio elementem frameworku, ale samego języka Ruby. Ich nazwa to *gem'y*. Rails'y natomiast mają domyślnie dołączony menedżer bibliotek/pakietów - *bundler*.



Rysunek 3.1: Statystyki pobrań menedżera pakietów bundler.

Jak widać, *bundler* jest najczęściej pobieranym *gem'em* spośród wszystkich, z ilością

pobrań przekraczającą 162,5 miliona. Ogromny wpływ na taki wynik mają oczywiście Rails'y. Strona `https://rubygems.org/`, która jest źródłem większości bibliotek, w statystykach podaje, że dostępnych jest ponad 131,600 paczek. Pokazuje to jak ważną częścią dla twórców Ruby on Rails i całej społeczności jest możliwość tworzenia rozszerzeń oraz ich dostępność.

3.2 Phoenix

Phoenix jest open source'owym frameworkiem webowym napisanym w języku *Elixir*. Inicjatorem tego projektu jest amerykanin Chris McCord. Pierwsza wersja *Phoenix'a* została wydana 28 sierpnia 2015 roku, a więc jest to bardzo młody framework. Pomimo swojego młodego wieku, framework ten zyskał sporą popularność i zainteresowanie społeczności. Jest to spowodowane ideą frameworku, który nie stara się być na siłę inny od istniejących rozwiązań, a zamiast tego stara się wykorzystywać ich najlepsze elementy. Jako przykładowe frameworki, z których czerpie *Phoenix*, wymieniane są *Ruby on Rails* oraz *Django* [6].

Tak, jak większość frameworków webowych, *Phoenix* implementuje wzorzec *Model-View-Controller*.

3.2.1 Podstawowe założenia

Idea działania *Phoenix'a* opiera się na kilku podstawowych założeniach:

- szybkość działania,
- współbieżność,
- wygoda użytkowania,
- niezawodność.

Szybkość działania

Położenie nacisku na szybkość działania aplikacji stworzonej przy pomocy frameworka jest czymś naturalnym. Zwykle jednak szybkość działania osiągana jest kosztem innych aspektów tworzenia aplikacji, jak na przykład szybkością powstawania oprogramowania. Aby osiągnąć zadowalającą szybkość działania, twórcy *Phoenix'a* zdecydowali się stworzyć

framework przy pomocy języka *Elixir*. Wpływ tego języka na framework opisany jest szerzej w sekcji 3.2.3.

Poza samym językiem programowania, na szybkość działania aplikacji wpływ mają następujące rozwiązania:

- router kompilowany jest do bardzo szybkiego w działaniu *pattern matching'u*, czyli dopasowaniu do wzorca. Dzięki temu optymalizacja wydajności wykonywana jest jeszcze zanim zapytanie opuści router,
- template'y są prekompilowane, *Phoenix* nie musi kopiować łańcuchów tekstowych dla każdego wyświetlanego template'u, przez co możliwe jest bardzo wydajne cache'owanie.

Współbieżność

Wraz z rozwojem możliwości sprzętowych, a dokładniej z rosnącą liczbą rdzeni procesorów, możliwe jest coraz większe wykorzystanie współbieżności w aplikacjach internetowych. Większość frameworków opartych jest na językach obiektowych, co niestety generuje problemy w kwestii współbieżności. Jest tak, ponieważ aby skorzystać z mechanizmu współbieżności, programista musi taką funkcjonalność zaimplementować, oczywiście przy pomocy dostępnych w języku narzędzi. Niestety, współbieżność nie jest prostym zagadnieniem, posiada wiele pułapek, przez co programiści, jeśli nie muszą, zwykle z niej nie korzystają.

Współbieżność, podobnie jak szybkość działania, jest cechą, która osiągnięta została poprzez wybór odpowiedniego języka programowania. *Elixir*, jako język funkcyjny stworzony do tworzenia bardzo wielu procesów, w pełni spełnia wymogi stawiane przez twórców *Phoenix'a*. Nie tylko posiada on bardzo wydajne mechanizmy zapewniające współbieżność, ale przede wszystkim bardzo ułatwia korzystanie z owej współbieżności. I owe ułatwienie pracy programisty jest ważnym czynnikiem popularności *Phoenix'a*. Przykład łatwości korzystania z mechanizmu współbieżności znajduje się na listingu 3.3, który pochodzi z pozycji literaturowej [5].

Listing 3.3: Współbieżność w Phoenix'ie

```
company_task = Task.async(fn -> find_company(cid) end)
user_task = Task.async(fn -> find_user(uid) end)
cart_task = Task.async(fn -> find_cart(cart_id) end)
```

```
company = Task.await(company_task)
user = Task.await(user_task)
cart = Task.await(cart_task)
```

Dzięki tak prostym interfejsom, wykonanie kodu z listingu 3.3 zajmie tyle, ile najdłuższe zapytanie do bazy danych, a nie łączny czas zapytań. Dzięki temu dużo lepiej wykorzystywana jest baza danych, co przekłada się na szybkość działania całej aplikacji.

Wygoda użytkowania

Wygoda użytkowania jest wypadkową wielu czynników. Z jednej strony jest to wybrany język programowania, który udostępnia programiście w przystępny sposób wiele zaawansowanych mechanizmów, ma „przyjazną" składnię i jest dość prosty w nauce. Z drugiej strony *Phoenix* w swoich założeniach chce się wzorować na innych frameworkach wykorzystując ich najlepsze elementy. Tak więc programista korzysta z rozwiązań, które mogą być mu znane z innych technologii, jak na przykład *Django* lub *Laravel*, co dodatkowo przyspiesza powstawanie aplikacji webowych.

Niezawodność

Niezawodność jest podstawą każdej aplikacji, nie tylko internetowej. Nawet najładniejszy, współbieżny i responsywny kod jest bezwartościowy, jeśli nie jest niezawodny. Aplikacje pisane w *Erlang'u* zawsze były bardziej niezawodne od innych, głównie poprzez strukturę linkowania procesów oraz komunikację pomiędzy procesami, co pozwala na efektywną pracę *supervisor'a* [5].

Po raz kolejny jedno z głównych założeń frameworka jest głęboko zakorzenione w przeznaczeniu języka programowania. Wysoka niezawodność jest oczywiście celem praktycznie wszystkich aplikacji, ponieważ błędy aplikacji są bardzo kosztowne dla ich właścicieli.

Dla programistów niezawodność fundamentów, na których budują funkcjonalność aplikacji, jest nieoceniona. Dużo łatwiej tworzy się aplikację mając pewność, że abstrakcje nałożone przez framework są wydajne i niezawodne, przez co można skupić się na tworzeniu funkcjonalności.

3.2.2 Elementy składowe

Phoenix jest nadrzędną warstwą w wielowarstwowym systemie zaprojektowanym, aby być modularnym i elastycznym. Pozostałe warstwy zawierają narzędzia takie jak *Plug* (odpowiedzialny za modularność), *Ecto* (odpowiedzialny za interakcję z bazą danych), *Cowboy* (serwer HTTP Erlanga) [6].

Efektem połączenia wszystkich warstw są następujące elementy:

- **endpoint**, który jako pierwszy zajmuje się zapytaniami, odpowiednio je obsługuje i przekazuje do routera,
- **router** - parsuje przychodzące zapytania i przekierowuje je do odpowiednich kontrolerów i akcji,
- **kontroler** - dostarcza on akcje, które obsługują zapytanie (przetwarzanie danych, przekazanie ich do widoków, generowanie HTML przekazywanych do przeglądarki),
- **szablony** - w połączeniu z danymi oraz po pewnym przetworzeniu generują widoki,
- **widoki**, które są warstwą prezentacji, czyli wizualnym efektem działania aplikacji,
- **kanały** - zarządzają websocketami, które umożliwiają komunikację w czasie rzeczywisttym pomiędzy serwerem, a klientem.

3.2.3 Język programowania - Elixir

Podobnie, jak w przypadku *Ruby on Rails*, duży wpływ na jakość użytkowania *Phoenix'a* ma język programowania, w którym jest on stworzony. *Elixir* jest kompilowanym językiem funkcyjnym, który uruchamiany jest na maszynie wirtualnej *Erlanga*. Dodatkowo, jak piszą jego twórcy, jest językiem programowania zaprojektowanym do tworzenia skalowalnych i łatwych w utrzymaniu aplikacji [13].

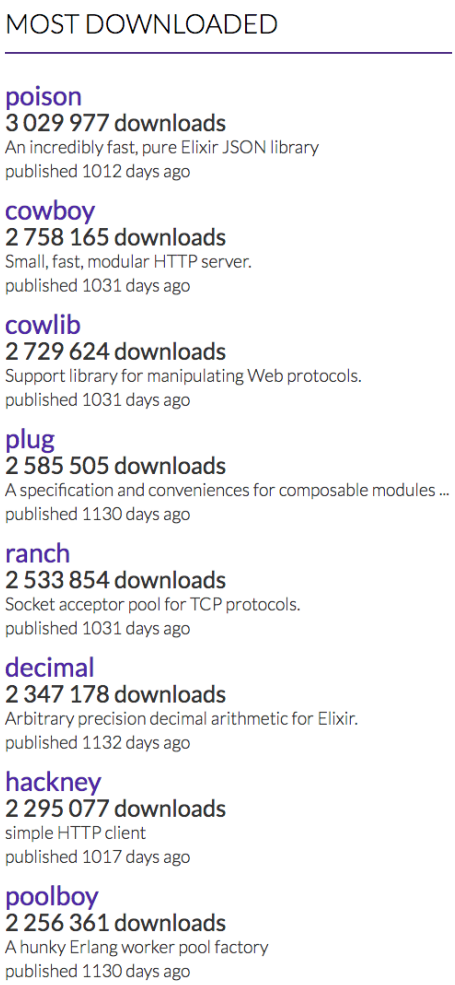
Wybór tego języka programowania umożliwia zrealizowanie jednych z podstawowych zadań *Phoenix'a* - zapewnienie wysokiej wydajności oraz niezawodności.

Dodatkowo twórcy zwracają uwagę na przyjazną składnię, co stanowczo ma wpływ na popularność frameworka. Jest tak, ponieważ programiści zwykle wolą czerpać przyjemność z używania „ładnego" języka, niż uczyć się i używać mało przyjaznego języka programowania. *Elixir*, jako pierwszy funkcyjny język programowania, wspiera makra z rodziny języków *Lisp*, lecz robi to stosując bardziej naturalną składnię [5].

3.2.4 Pozostałe cechy

Biblioteki

Idea centralnego repozytorium paczek jest bardzo popularna wśród większości języków programowania. Ułatwia ono pracę w aplikacją, dodawanie nowych zależności nie jest problematyczne, a błędy dotyczące niezgodności wersji zależności są wyeliminowane. Dla języka *Elixir* domyślnym menadżerem paczek jest *Hex*.



Rysunek 3.2: Statystyki pobrań menedżera pakietów Hex.

W momencie powstawania niniejszej pracy, dostępnych jest ok 4,5 tysiąca bibliotek w oficjalnym repozytorium *Hex*. W porównaniu do innych, będących na rynku dłużej języków i frameworków, jest to niewielka liczba. Biorąc natomiast pod uwagę rosnącą popularność *Phoenix'a*, można założyć iż ta liczba będzie rosnąć. Obecnie istniejące paczki są niekiedy odpowiednikami istniejących już paczek napisanych w innych językach. Niestety, bardzo często paczki te, z racji na krótki okres istnienia, są mocno niedopracowane.

3.3 Express

Express jest minimalistycznym i bardzo elastycznym frameworkiem webowym. Framework ten powstał poprzez dostarczenie wrapperów dla API udostępnianego przez platformę *Node.js*. *Express*, ze względu na swoją budowę, można traktować jako podstawę do budowy innych frameworków. Framework ten jest dostępny jako wolne oprogramowanie pod licencją MIT i komercyjnie wspierany jest przez firmę *StrongLoop*. *Express* nie jest frameworkiem typu *full-stack*, jest on nastawiony na możliwość dowolnej konfiguracji zależności niezbędnych w projekcie.

3.3.1 Podstawowe założenia

Pierwotną ideą powstania *Express'a* była chęć stworzenia frameworka podobnego do istniejącego w języku programowania *Ruby* frameworka *Sinatra*. Z tego względu oba te frameworki współdzielą podejście do rozwiązywania problemów oraz struktury kodu.

Configuration over convention

Głównym założeniem frameworka *Express* jest dostarczenie programistom pełnej elastyczności i możliwości wyboru poszczególnych składowych aplikacji. W skład frameworku nie wchodzą więc biblioteki odpowiedzialne m.in. za:

- komunikację z bazą danych,
- autentykację użytkowników,
- silnik szablonów.

Zastosowanie podejścia *configuration over convention* może być problematyczne w przypadku rozpoczynania swojej przygody z *Express'em*. W sytuacji, kiedy framework nie dostarcza wbudowanych mechanizmów, jak np. wcześniej wymieniony adapter do bazy danych, programista nie znający dostępnych bibliotek może mieć duże problemy z wyborem odpowiedniego mechanizmu.

Organizacja struktury aplikacji

Ze względu na podejście *convention over configuration*, *Express* nie narzuca programistom konkretnej struktury aplikacji. W żaden sposób nie jest wyróżnione podejście,

które najczęściej można spotkać w aplikacjach webowych, czyli *Model-View-Controller*. Programista może wybrać dowolny wzorzec projektowy, bądź też nie zastosować żadnego, jeśli taka jest jego wola.

3.3.2 Podstawa dla innych frameworków

Express bez żadnych dodatkowych bibliotek, czyli od razu po instalacji, nie dostarcza wielu, niekiedy ważnych, funkcjonalności. Programiści korzystający z tego frameworka potrzebowali jednak tych samych narzędzi w przypadku tworzenia aplikacji tego samego typu. W momencie, kiedy za każdym razem musieli dołączać do projektu i konfigurować te same biblioteki, zaczęły powstawać bardziej „kompletne" frameworki, które bazowały na *Expressie*. W ten sposób powstały następujące frameworki:

- LoopBack - stworzony do budowy REST API,
- ItemsAPI - framework łączący funkcjonalność *Elastichsearch'a* i *Express'a*,
- KeystoneJS - *Content Management System*,
- MEAN - framework typu full-stack,
- Feathers - stworzony do budowy REST API,
- Sails.js - framework MVC typu full-stack,
- Hydra-Express - framework stworzony do architektury mikroserwisów.

3.3.3 Biblioteki

Większość języków programowania posiada swój menedżer paczek, nie inaczej jest w przypadku języka *JavaScript*. W tym środowisku największą popularność zyskał *npm*. Jako, że *npm* jest menedżerem paczek języka, a nie frameworka, można tam oczywiście znaleźć paczki niekompatybilne z *Expressem*, lecz jest to cecha wszystkich menedżerów stworzonych dla konkretnego języka programowania.

Na oficjalnej stronie *npm* [17] przeczytać można, że dostępnych jest ponad 470 tysięcy paczek. Z repozytorium *npm* tygodniowo pobieranych jest 2 365 813 490 paczek.

Rozdział 4

Analiza porównawcza

4.1 Opisowe porównanie charakterystyk wybranych frameworków

Naturalnym jest, że frameworki webowe różnią się od siebie pod wieloma względami. Różnice można zauważać od samego początku, czyli idei powstania, poprzez podejście do programisty, na sposobie działania i wydajności kończac. Zwykle przyczyną różnorodności jest sama filozofia frameworka, dążenie do doskonałości. Nie jest możliwe osiągnięcie optymalnych wartości w każdej z dziedzin, które frameworki obejmują, przez co twórcy muszą iść na większe bądź mniejsze kompromisy. Niekiedy owe różnice nie pojawiają się z przymusu, ale z przyjęcia konkretnych założeń.

Są jednak sytuacje, kiedy frameworki są do siebie bardzo podobne. Zwykle są to sytuacje, kiedy framework z jednego języka programowania przenoszony jest do innego języka. Tak było w przypadku opisywanego *Express'a*, który powstał jako odzwierciedlenie frameworka *Sinatra*. Innym przypadkiem występowania podobieństw jest chęć wykorzystania jakiegoś fragmentu frameworka, który uznawany jest za standard i bez którego programista musi posiadać większą wiedzę aby odpowiednio wykorzystywać framework.

W kolejnych podrozdziałach zostaną przedstawione porównania wybranych frameworków na zasadzie *„każdy z każdym"*. Poza podobieństwami i różnicami w budowie oraz idei działania, zamieszczona została subiektywna opinia autora na temat sytuacji, w których jeden framework ma przewagę nad drugim.

4.2 Ruby on Rails vs Phoenix

Jako pierwsze analizie porównawczej zostaną poddane najstarszy i najmłodszy framework spośród opisywanych, czyli *Ruby on Rails* oraz *Phoenix*. Owa różnica „wieku" mocno rzutuje na całokształ tych frameworków. Pierwszy ma już stabilną, wręcz nienaruszalną, pozyję na rynku, bardzo dużą społeczność oraz niezliczoną ilość materiałów dostępnych w Internecie. Drugi natomiast, co oczywiście jest efektem wielu składowych, ma spory potencjał. Oczywiście nie jest to pierwszy, ani jedyny, framework, który owy potencjał posiada. Jednakże w przypadku *Phoenix'a* jest duża szansa, iż będzie on wykorzystany. Przyczynić się do tego może środowisko programistów *Ruby on Rails*, ponieważ już teraz duża ilość programistów albo mocno się *Phoenixem* interesuje, albo zdecydowało się zmienić technologię na *Elixir'a* i *Phoenix'a*.

4.2.1 Filozofia działania

Ruby on Rails oraz *Phoenix* posiadają elementy wspólne jeśli chodzi o sposób działania. Jest to spowodowane faktem, iż z założenia *Phoenix* miał wykorzystywać najlepsze elementy z innych frameworków.

W kwestii filozofii działania, oba frameworki są zgodne w kwestii chęci ułatwienia pracy programistom.

4.2.2 Architektura aplikacji, elementy składowe

4.2.3 Wygoda użytkowania

4.2.4 Dostępność najczęściej wykorzystywanych bibliotek

4.2.5 Subiektywna opinia

4.3 Ruby on Rails vs Express

4.3.1 Filozofia działania

4.3.2 Architektura aplikacji, elementy składowe

4.3.3 Wygoda użytkowania

4.3.4 Dostępność najczęściej wykorzystywanych bibliotek

4.3.5 Subiektywna opinia

4.4 Phoenix vs Express

4.4.1 Filozofia działania

4.4.2 Architektura aplikacji, elementy składowe

4.4.3 Wygoda użytkowania

4.4.4 Dostępność najczęściej wykorzystywanych bibliotek

4.4.5 Subiektywna opinia

4.5 Subiektywna ocena

Rozdział 5

Implementacja

5.1 Ruby on Rails

5.2 Phoenix

5.3 Express

Rozdział 6

Projekt komputerowego środowiska eksperymentalnego

6.1 Plan prowadzenia eksperymentów

6.2 Wykorzystane narzędzia

6.2.1 Docker

6.3 Benchmarki

6.3.1 Ruby

6.3.2 Elixir

6.3.3 JavaScript

Rozdział 7

Analiza wydajnościowa

7.1 Metody mierzenia wydajności aplikacji internetowych

7.2 Wyniki badań

Rozdział 8

Podsumowanie

Literatura

[1] *Historia Internetu*, dostęp pod adresem: `https://pl.wikipedia.org/wiki/Historia_Internetu`, aktualne na dzień 1.04.2017r.

[2] *Software framework*, dostęp pod adresem: `https://en.wikipedia.org/wiki/Software_framework`, aktualne na dzień 1.04.2017r.

[3] *Ruby on Rails Guides*, dostęp pod adresem: `http://guides.rubyonrails.org/`,
aktualne na dzień 14.06.2017r.

[4] *Ruby on Rails Doctrine*, dostęp pod adresem: `http://rubyonrails.org/doctrine/`, aktualne na dzień 1.05.2017r.

[5] McCord R., Tate B., Valim J., *Programming Phoenix*, The Pragmatic Programmers LLC, 2016

[6] *Phoenix Guides*, dostęp pod adresem: `http://www.phoenixframework.org/docs/resources`, aktualne na dzień 6.05.2017r.

[7] Martin, R. *Czysty kod. Podręcznik dobrego programisty*, Gliwice, Wydawnictwo Helion 2010

[8] Diwan, A. *Tools for Testing Website Performance*, dostęp pod adresem: `https://www.sitepoint.com/tools-testing-website-performance/`, aktualne na dzień 18.03.2017r.

[9] *Don't Repeat Yourself*, dostęp pod adresem: `http://deviq.com/don-t-repeat-yourself/`, aktualne na dzień 3.05.2017r.

[10] *What is a Container*, dostęp pod adresem: `https://www.docker.com/what-container`, aktualne na dzień 27.05.2017r.

[11] *Docker. Kontener aplikacyjny nie tylko dla programistów*, dostęp pod adresem: `https://dsg.cs.put.poznan.pl/wiki/_media/workshop/docker-skisr.pdf`, aktualne na dzień 27.05.2017r.

[12] *CRUD*, dostęp pod adresem: `https://pl.wikipedia.org/wiki/CRUD`, aktualne na dzień 27.05.2017r.

[13] *Elixir Language*, dostęp pod adresem: `https://elixir-lang.org/`, aktualne na dzień 16.05.2017r.

[14] Mazaika, K. *Why I'm betting on Elixir*, dostęp pod adresem: `https://medium.com/@kenmazaika/why-im-betting-on-elixir-7c8f847b58`, aktualne na dzień 28.05.2017r.

[15] *Express Guides*, dostęp pod adresem: `https://expressjs.com/`, aktualne na dzień 10.06.2017r.

[16] Mardanm A. *Pro Express.js*, Apress, 2014

[17] *npm Homepage*, dostęp pod adresem: `https://www.npmjs.com/`, aktualne na dzień 10.06.2017r.