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:
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

1.3 Wymagania

1.4 Zarys koncepcji

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

2.2 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.

2.3 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.

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.

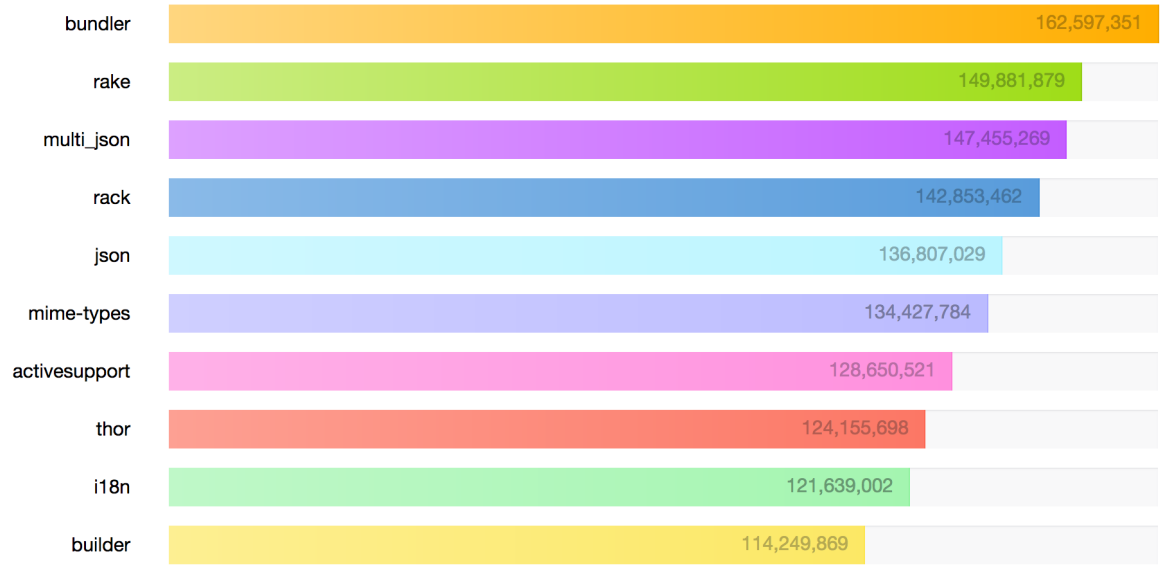
Don't Repeat Yourself

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 [8]. 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

3.3 Express

Rozdział 4

Analiza porównawcza

4.1 Opisowe porównanie charakterystyki wybranych frameworków

4.1.1 Ruby on Rails vs Phoenix

4.1.2 Ruby on Rails vs Express

4.1.3 Phoenix vs Express

4.2 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 Nginx

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

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

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