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

Spis rysunków

Spis tablic

Spis listingów

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.

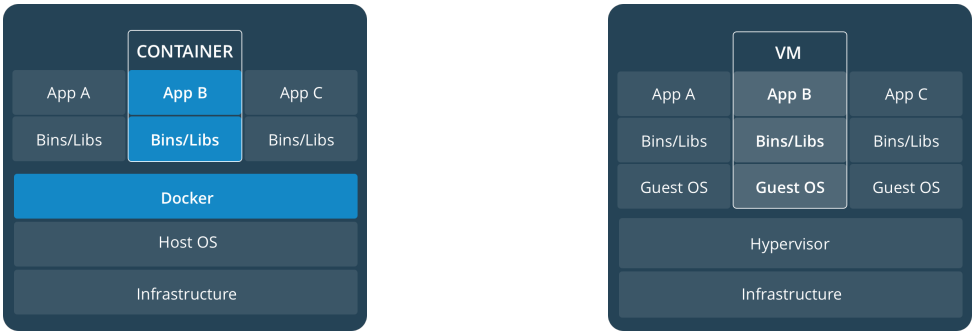
2.2.2 Framework webowy

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.
Źródło: `https://www.docker.com/what-container`

- 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.

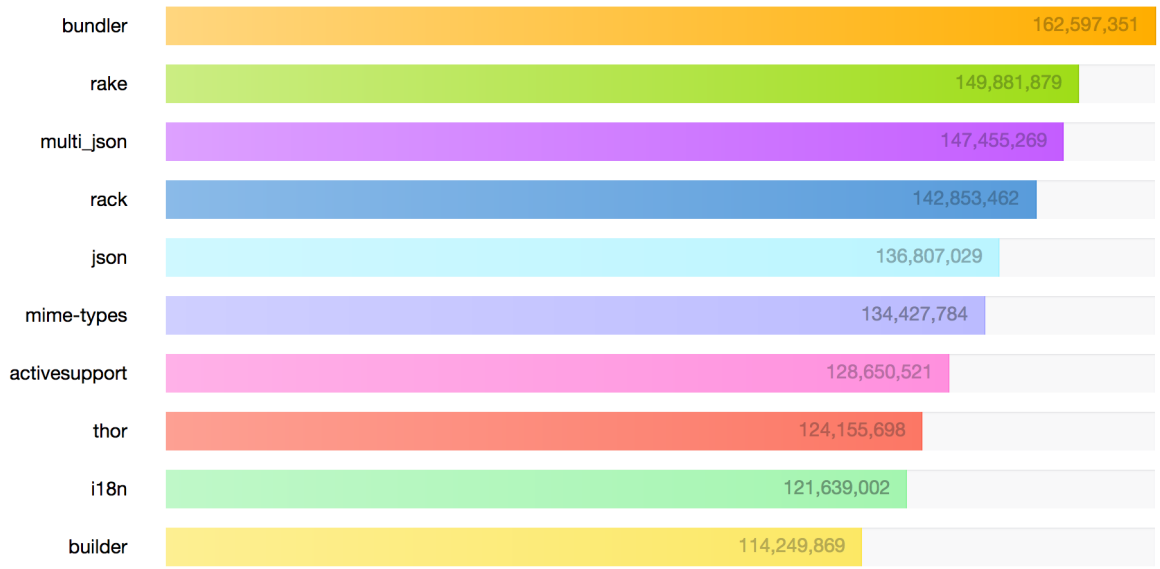
Don't Repeat Yourself

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.

Źródło: `https://rubygems.org/stats`

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].

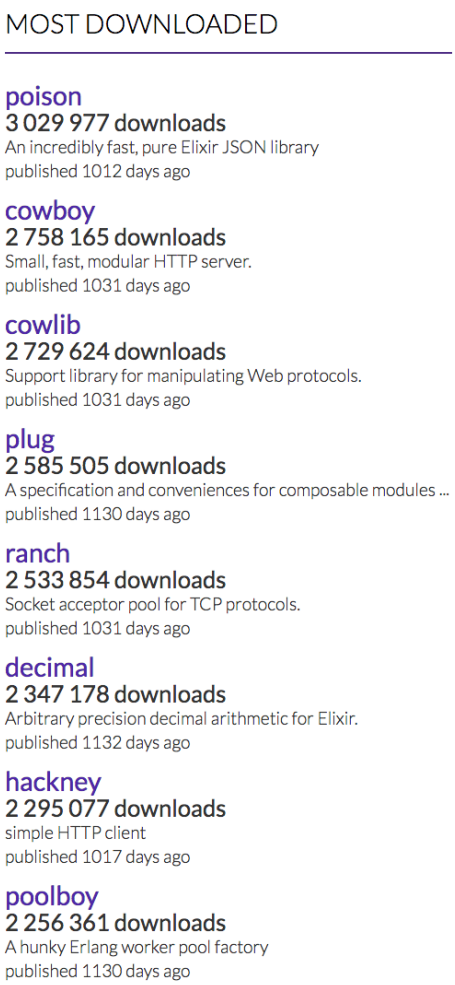
Wybór tego języka programowania umożliwia zrealizowanie jednych z podstawowych zadań *Phoenix'a* - zapewnienie wysokiej wydajności oraz niezawodności.

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.

Źródło: `https://hex.pm/`

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

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.

4.1 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*.

4.1.1 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.

Jeśli chodzi o filozofię działania, oba frameworki są zgodne w kwestii chęci ułatwienia pracy programistom. Zarówno *Ruby on Rails*, jak i *Phoenix*, mocno uwypuklają przyjemność korzystania z języków programowania, na których są oparte. Twórcy *Ruby on Rails* zapoczątkowali trend stawiania na przyjemność użytkowania w świecie frameworków webowych, natomiast twórcy *Phoenix* bardzo dobrze to zaimplementowali w swoim frameworku. Z tego też powodu spora część programistów *Ruby on Rails* interesuje się rozwojem *Phoenix'a*.

Podstawową różnicą w filozofii działania jest kwestia szybkości działania. Twórca *Ruby on Rails* nie poświęcił ani jednego punktu w *„The Ruby on Rails Doctrine"* [4]. Można z tego faktu wyciągnąć wniosek, że szybkość działania **nie jest** kluczowa dla tego frameworka. I powszechna opinia na temat *Rails'ów* to potwierdzi. Nie zmienia to jednak faktu, że wiele bardzo dużych portali opartych jest na *Ruby on Rails*. Przykładami mogą być *GitHub*, *Groupon*, *AirBnB* czy *Kickstarter*. Framework ten jest natomiast idealnym narzędziem do szybkiego tworzenia aplikacji. *Ruby on Rails* na pewno zyskało wielu użytkowników ze względu na szybkość prototypowania, co jest bardzo istotne w dobie *start-up'ów*.

Dla odmiany, *Phoenix* szczyci się swoją wysoką wydajnością. Jest to jeden z filarów tego frameworka. Co ciekawe, pomimo postawienia na szybkość działania, nie ucierpiała

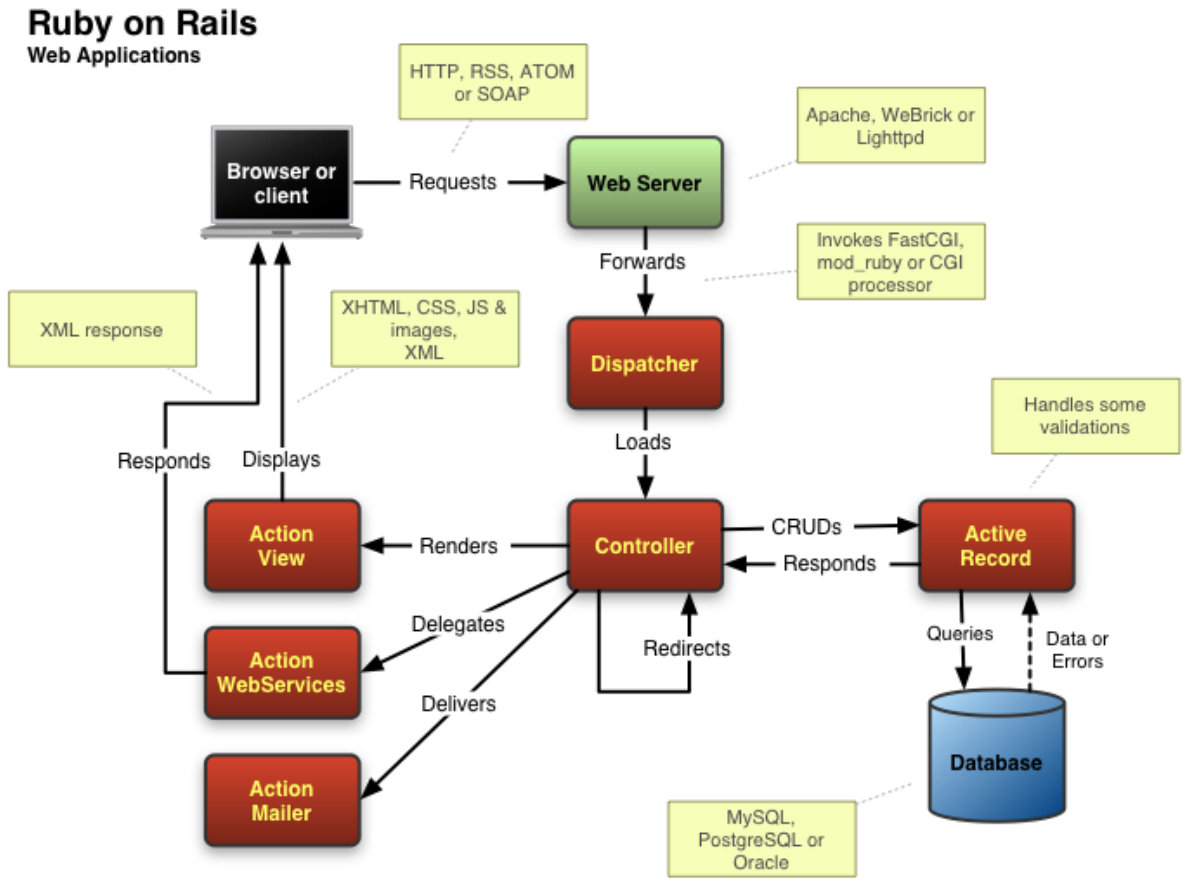
na tym szybkość prototypowania. Powodów takiego stanu rzeczy po raz kolejny można doszukiwać się w języku programowania, na którym oparty jest *Phoenix*. Tworzy to sporą przewagę nad *Ruby on Rails* w przypadkach, gdzie poza szybkością powstania aplikacji równie ważna jest jej wydajność.

4.1.2 Architektura aplikacji, elementy składowe

Zarówno *Ruby on Rails*, jak i *Phoenix* są frameworkami typu full-stack. Oznacza to, że dostarczane są z mechanizmami odpowiedzialnymi za:

- serwer http - przy pomocy którego można uruchomić aplikację w środowisku deweloperskim,
- generowanie struktury projektu (architektura MVC),
- połączenie z bazą danych - adaptery do różnych typów baz danych, ORM,
- router aplikacji - odpowiedzialny za przekierowywanie zapytań do odpowiednich akcji kontrolerów,
- podstawowe mechanizmy bezpieczeństwa (mechanizm sesji, bezpieczne ciasteczka),
- silniki szablonów - Slim, HAML,
- wysyłanie wiadomości e-mail z aplikacji - należy jedynie skonfigurować serwer SMTP z którego ma korzystać aplikacja.

Wykorzystywana w obu frameworkach architektura została pokazana na rysunku 4.1.



Rysunek 4.1: Architektura Ruby on Rails.

Źródło: `http://www.sentex.net/~pkomisar/Ruby/Rails2.png`

4.1.3 Wygoda użytkowania, szybkość powstawania aplikacji

W kwestii wygody użytkowania oba frameworki są na podobnym poziomie. Zarówno *Ruby on Rails* jak i *Phoenix* są stworzone w taki sposób, aby ułatwić prototypowanie aplikacji. Dzięki temu odczucia z użytkowania obu są bardzo przyjemne. Aplikacje można tworzyć relatywnie szybko, frameworki posiadają praktycznie wszystkie niezbędne mechanizmy, które wykorzystywane są w większości aplikacji webowych.

Przykładowe mechanizmy, które były brane pod uwagę:

- przetwarzanie danych w tle - w przypadku *Ruby on Rails* jest to *Sidekiq*, jego odpowiednikiem w *Phoenix'ie* jest *Akira*,
- autentykacja/autoryzacja użytkowników, zarządzanie sesją - *Devise* dla *RoR* oraz *Addict* dla *Phoenix'a*,

- wsparcie dla dołączania skryptów JavaScript w warstwie prezentacji,
- wsparcie dla różnych preprocesorów styli (Sass, Less)

Istnieje jednak pewna różnica, która może mieć wpływ na wygodę programisty, a jest nim rodzaj języka programowania. Dużo bardziej popularne jest podejście obiektowe, na którym oparty jest *Ruby*. *Elixir* natomiast jest językiem funkcyjnym. Paradygmat funkcyjny różni się znacznie od obiektowego, przez co w przypadku braku znajomości tego paradygmatu, używanie *Phoenix'a* może być dużo trudniejsze. Zasada ta działa oczywiście w dwie strony i jeśli ktoś nie zna podejścia obiektowego, korzystanie z *Ruby on Rails* również może być utrudnione.

4.1.4 Podsumowanie

Oba frameworki są do siebie podobne w kwestii używania. Bardzo ciekawe oraz pomocne są materiały, które wyjaśniają dlaczego dany framework powstał oraz jak powinien być używany, aby w pełni wykorzystać jego potencjał. Przewagą *Ruby on Rails* jest na pewno dojrzałość oraz ogromna społeczność. W kwestii, oczywiście subiektywnej, przyjemności z korzystania, oba frameworki są na podobnym poziomie. *Phoenix* zyskuje przewagę w kwestii wydajności. Moim zdaniem w tym frameworku drzemie niesamowity potencjał i mam nadzieję, że na stałe wpisze się w zbiór najbardziej popularnych frameworków webowych.

4.2 Ruby on Rails vs Express

Zestawienie *Ruby on Rails* oraz *Express'a* jest zestawieniem najstarszych frameworków spośród narzędzi wybranych do analizy. Pierwszy powstał w 2004 roku, drugi natomiast swój początek miał w 2010r. Ze względu na swój wiek oba frameworki mają ugruntowaną pozycję na rynku oraz rzesze użytkowników.

4.2.1 Filozofia działania

W przypadku zestawienia *Ruby on Rails* oraz *Express'a* filozofia działania oraz sama idea powstania są kluczowymi aspektami. Wynika to z faktu, iż *Express* powstał z chęci przeniesienia frameworka *Sinatra* z środowiska programistów Rubiego do środowiska *Node.js*.

Framework *Sinatra* powstał natomiast jako kontra do rozbudowanego *Ruby on Rails*. Z tego też powodu *Ruby on Rails* oraz *Express.js* są swoimi przeciwnościami.

Z jednej strony mamy framework typu full-stack i podejście *Convention over Configuration*. Z drugiej - minimalistyczny framework, do którego możemy (ale nie musimy) dołączyć szereg bibliotek, żeby w kwestii funkcjonalności zrównać się z *Ruby on Rails* oraz oczywiście podejście *Configuration over Convention*. Przejście z jednego świata do drugiego bywa dość nieprzyjemne, wymaga dużej czujności programisty oraz większych nakładów pracy. Dlatego też środowiska programistów *Ruby on Rails* oraz *Express'a* rzadko kiedy się przenikają.

Ze względu na ideę powstania każdego z nich, trudno wskazać elementy łączące *Ruby on Rails* oraz *Express'a*. Oba służą do tworzenia aplikacji internetowych, robią to jednak w różny sposób.

4.2.2 Architektura aplikacji, elementy składowe

Różnice w kwestii architektury aplikacji oraz bibliotek, które się składają na framework, są w przypadku *Ruby on Rails* oraz *Express'a* bardzo łatwe do zauważenia. Wynika to ze złożoności samych frameworków, a więc podejścia typu *full-stack* w przypadku *Ruby on Rails* oraz minimalizmu *Express'a*. Pierwszy posiada praktycznie wszystko, co jest wymagane do stworzenia podstawowej aplikacji internetowej. W przypadku *Express'a* niestety wymagany jest dodatkowy wysiłek programisty w celu stworzenia aplikacji. Podstawowym problemem jest brak dołączonych adapterów umożliwiających połączenie z bazą danych. Framework ten nie posiada również mechanizmu umożliwiające mapowanie obiektowo-relacyjne (ang. *ORM* - Object-Relational Mapping).

Jako, że *Express* nie posiada wbudowanego generatora projektu, projekt nie posiada określonej struktury. Z jednej strony daje to swobodę działania programiście, czego niekiedy brakuje w *Ruby on Rails*. Niewątpliwie możliwość wykorzystania innej architektury aplikacji niż MVC jest mile widzianą cechą. Z drugiej zaś strony może być problematyczne w sytuacji, kiedy każdy projekt ma swoją „autorską" strukturę i nie przestrzega żadnego schematu.

4.2.3 Wygoda użytkowania, szybkość powstawania aplikacji

Porównianie wygody użytkowania oraz szybkości powstawania aplikacji w przypadku *Ruby on Rails* oraz *Express'a* w dużym stopniu odnosi się do sekcji 4.2.2. Struktura aplikacji oraz biblioteki dołączone do frameworka są kluczowe dla programistów, szczególnie początkujących. W połączeniu z brakiem domyślnych mechanizmów mapowania obiektowo-relacyjnego, można odnieść wrażenie, iż *Express* nie jest najlepszym frameworkiem dla początkujących programistów. Należy dobrze poznać środowisko, żeby móc wybrać dobrze współpracujące ze sobą komponenty i dopiero wtedy można zacząć tworzyć aplikację. Pod tym względem zupełnym przeciwieństwem jest *Ruby on Rails*. Programista nie musi znać całego środowiska oraz dużej ilości bibliotek, żeby stworzyć aplikację. Jednakże w momencie, kiedy zajdzie taka potrzeba, możliwa jest podmiana niektórych bibliotek bądź też rezygnacja z części w celu zmniejszenia rozmiarów projektu.

4.2.4 Podsumowanie

Wrażenia po korzystaniu z obu frameworków są zgoła odmienne. I oczywiście znacząco zależą od wcześniejszych doświadczeń i przyzwyczajeń w kwestii tworzenia aplikacji internetowych. Osoba, która dopiero zaczyna swoją przygodę z tą dziedziną informatyki może mieć pewne problemy aby zacząć pracę z *Express'em*. Jego modułowość oraz lekkość są w stanie wykorzystać i docenić osoby, które posiadają pewne doświadczenie w kwestii aplikacji webowych oraz wiedzą dokładnie jakich komponentów będą musieli użyć w celu stworzenia aplikacji. Frameworkiem dużo bardziej przyjaznym dla początkujących oraz przy tworzeniu standardowych aplikacji jest *Ruby on Rails*. Programista może, ale oczywiście nie musi, skorzystać z dostarczonych przez twórców rozwiązań, które w większości przypadków są wystarczające.

Zdecydowaną zaletą *Express'a* jest popularność języka programowania *JavaScript*. Zdecydowanie zmniejsza to barierę wejścia w środowisko tego frameworka.

4.3 Phoenix vs Express

Ostatnim porównaniem jest konfrontacja *Phoenix'a* z *Express'em*. Z jednej strony mamy bardzo młody framework typu full-stack, z drugiej niewielki, ale sprawdzony framework, który stawia na minimalizm i modularność. Zestawienie to jest o tyle ciekawe, iż

Phoenix miał wykorzystywać najlepsze elementy innych frameworków, lecz ciężko doszukać się tam rozwiązań znanych z *Express'a*.

4.3.1 Filozofia działania

Podobnie, jak w przypadku zestawienia *Ruby on Rails* z *Express'em*, filozofia powstania znacząco różni się w przypadku *Phoenix'a* i *Express'a*. Pierwszy jest młodym frameworkiem, który wzoruje się na najlepszych rozwiązaniach, oczywiście subiektywnie wybranych przez twórców, z innych frameworków webowych w celu zapewnienia maksymalnego zadowolenia programisty oraz osiągnięcia wysokiej wydajności. Drugi natomiast preferuje dostarczenie jedynie niezbędnych mechanizmów, stawiając na podejście *Configuration over Convention*.

Wyższość jednego podejścia nad drugim jest oczywiście kwestią dyskusyjną. Jednakże porównując liczbę frameworków podobnych do *Phoenix'a*, a więc typu full-stack oraz podobnych do *Express'a*, czyli minimalistycznych, nie trudno zauważyć jak bardzo popularne jest pierwsze podejście.

4.3.2 Architektura aplikacji, elementy składowe

Nietrudno zauważyć, że *Phoenix* ma niewiele wspólnego z *Express'em* w kwestii architektury aplikacji. Sam fakt, iż *Phoenix* jest frameworkiem typu full-stack, definiuje podstawową różnicę. Ponownie, problemem jest tutaj przede wszystkim brak adaptera do baz danych oraz mechanizmu ORM w wypadku *Express'a*. Jest to na tyle często wykorzystywana funkcjonalność, że framework do tworzenia aplikacji działających po stronie serwera powinien mieć wbudowane takie mechanizmy. *Phoenix* narzuca programiście model *MVC*, gdzie *Express* daje dowolność w kwestii użytego wzorca architektonicznego. Jest to z pewnością zaletą *Express'a*, ponieważ model *MVC* jest najbardziej uniwersalny, jednak są sytuacje gdzie inne podejścia dużo lepiej się sprawdzają.

4.3.3 Wygoda użytkowania, szybkość powstawania aplikacji

Phoenix jest frameworkiem bardziej uniwersalnym. Można przy jego pomocy bardzo szybko stworzyć aplikację internetową, która korzysta z bazy danych i innych prostych mechanizmów, np. posiada autentykację użytkowników. Wbudowane mechanizmy pozwalają na realizację takiej funkcjonalności w bardzo szybkim czasie i bez dużych nakładów pracy.

Problemem w przypadku *Phoenixa* może być paradygmat funkcyjny języka *Elixir*. Aby poprawnie pisać aplikacje z wykorzystaniem tego języka należy poznać programowanie funkcyjne, które jest dużo mniej popularne niż programowanie obiektowe.

Express, przynajmniej na początkowym etapie tworzenia aplikacji wymaga od programisty znajomości środowiska oraz bibliotek, które w innych frameworkach uznawane są jako podstawowe. Do momentu, aż programista posiada wyrobione zdanie na temat dostępnych bibliotek, korzystanie z *Express'a* nie jest tak szybkie, jak z *Phoenix'a*.

4.3.4 Podsumowanie

Nie da się ukryć, że porównywane frameworki stoją poniekąd po dwóch stronach barykady. Jest to konfrontacja kompletnego frameworka typu full-stack z frameworkiem, który powstał jako zaprzeczenie tej idei. Oba rozwiązania mają oczywiście swoich fanów oraz przeciwników. Nie da się jednoznacznie wyłonić lepszego z tych dwóch rozwiązań, ponieważ ich przydatność w dużym stopniu zależy od rodzaju aplikacji, jaki mamy wykonać oraz znajomości danego środowiska.

Biorąc pod uwagę podejście obu frameworków do tworzenia aplikacji, *Phoenix* stanowczo jest wygodniejszym narzędziem. Framework ten stworzony jest do szybkiego prototypowania i jak najbardziej się sprawdza w takim zastosowaniu. Przeszkodą dla wielu programistów może być natomiast funkcyjny język programowania *Elixir*, na którym oparty jest *Phoenix*.

Express natomiast sprawdza się jako narzędzie do stworzenia aplikacji internetowej, która nie jest dużych rozmiarów. Podstawowe elementy tego frameworku, jak router zapytań i system widoków, idealnie się nadają do tego typu zastosowań. Problemy zaczynają się dopiero, kiedy aplikacja musi mieć bezpośrednie połączenie z bazą danych oraz wymagane są dodatkowe, bardziej zaawansowane mechanizmy.

Rozdział 5

Projekt komputerowego środowiska eksperymentalnego

5.1 Plan prowadzenia eksperymentów

W celu dokonania analizy porównawczej oraz wydajnościowej wybranych frameworków, stworzono plan przeprowadzania eksperymentów oraz odpowiednie środowisko eksperymentalne. Dzięki temu testy mogły być wykonane rzetelnie, miarodajnie oraz powtarzalnie. Aby przeprowadzone testy były porównywalne, aplikacje stworzone przy pomocy każdego z frameworków powinny spełniać te same założenia.

Zaplanowany eksperyment składa się z kilku faz:

1. stworzenie środowiska eksperymentalnego,
2. implementacja aplikacji w każdym z frameworków,
3. przeprowadzenie testów każdego z frameworków.

5.2 Wykorzystane narzędzia

5.2.1 System kontroli wersji

Aby ułatwić pracę nad kilkoma aplikacjami jednocześnie, zdecydowano się wykorzystać jeden z systemów kontroli wersji, jakim jest *Git*. Rozwiązanie to zapewnia dostęp do historii zmian oraz możliwość pobrania kodu z repozytorium na każdy komputer bądź

serwer, co jest sporym ułatwieniem w kwestii testowania rozwiązań na zdalnej maszynie. Jako hosting został wykorzystany serwis *GitHub*.

5.2.2 Docker

W celu zapewnienia izolacji środowiska developerskiego oraz testowego, zdecydowano się skorzystać z mechanizmu wirtualizacji. Dzięki temu aplikacje można w bardzo prosty sposób przenosić na inne maszyny mając pewność, że zawsze będą uruchamiane dokładnie w ten sam sposób. Jedyną różnicą są dostępne zasoby mocy obliczeniowej oraz pamięci, które zależą oczywiście od parametrów komputera bądź też serwera. Dokładniejszy opis tego rozwiązania przedstawiony jest w sekcji 2.2.3.

Listing 5.1: Przykładowy plik Dockerfile.

```
FROM ruby:2.4.1

ENV APP_ROOT /app
ENV BUNDLE_PATH /bundle

RUN apt-get update -qq && \
    apt-get install -y build-essential libpq-dev nodejs cmake

WORKDIR $APP_ROOT

ADD . $APP_ROOT
```

W celu połączenia kilku kontenerów (kontener aplikacji, bazy danych itd) w jedną sieć zostało wykorzystane narzędzie *docker-compose*. Przykładowa konfiguracja znajduje się na listingu 5.2.

Listing 5.2: Przykładowy plik docker-compose.yml.

```
version: '2'

services:
  db:
    image: postgres:9.6.1 # reuse postgres container
    volumes:
      - /var/lib/postgresql/data
```

```
redis:
  image: redis:3.2.6-alpine

web:
  build:
    dockerfile: Dockerfile
    context: .
  ports:
    - 3000:3000
  volumes:
    - .:/app
  links:
    - db
    - redis
  depends_on:
    - db
    - redis
  command: bundle exec rails s -b 0.0.0.0
  environment:
    - REDIS_URL=redis://redis:6379
    - DATABASE_URL=postgres://postgres@db:5432
  volumes_from:
    - bundle

sidekiq:
  build:
    dockerfile: Dockerfile
    context: .
  volumes:
    - .:/app
  links:
    - db
    - redis
  depends_on:
    - db
    - redis
  command: bundle exec sidekiq
  environment:
    - REDIS_URL=redis://redis:6379
```

```
      - DATABASE_URL=postgres://postgres@db:5432
    volumes_from:
      - bundle

  bundle:
    image: busybox
    command: echo "I'm a little data container, short and stout..."
    volumes:
      - /bundle
```

5.2.3 Locust

Wydajność aplikacji internetowych zwykle bada się pod kątem ilości użytkowników, którzy jednocześnie mogą korzystać z aplikacji. Jednym z rozwiązań, które umożliwia przeprowadzenie badań obciążeniowych aplikacji, jest narzędzie *Locust*. Jest to prosta aplikacja, która składa się z szeregu skryptów w *Python'ie* oraz interfejsu graficznego dostępnego przez przeglądarkę internetową.

Locust tworzy zadaną ilość wątków, które symulują operacje przeprowadzane przez użytkowników. Na podstawie zebranych odpowiedzi od serwera oraz pomiaru czasów, może określić wartość obsłużonych zapytań na sekundę (ang. RPS - requests per second).

Przy pomocy skryptów definiujemy które endpointy aplikacji *Locust* ma przetestować oraz jakie dane powinien wysłać. Dzięki temu można zasymulować rzeczywiste zachowanie użytkownika. Przykładowy skrypt konfiguracyjny znajduje się na listingu 5.3.

Listing 5.3: Przykładowy plik konfiguracyjny narzędzia Locust.

```
from locust import HttpLocust, TaskSet

def login(l):
    l.client.post("/users/sign_in", json={"user":{"email":"marcin.
        mantke@gmail.com", "password":"password"}})

def index(l):
    l.client.get("/")

def temperature_sensor_data(l):
    l.client.get("/temperature_sensor_data")
```

```
def create_temp_data(l):
    l.client.post("/temperature_sensor_data", json={"
        temperature_sensor_datum": {"sensor_id": "1", "utc_timestamp": "
        1496005043", "value": "2"}})

class UserBehavior(TaskSet):
    tasks = {index: 0, temperature_sensor_data: 5, create_temp_data: 1}

    def on_start(self):
        login(self)

class WebsiteUser(HttpLocust):
    task_set = UserBehavior
    min_wait = 5000
    max_wait = 9000
```

Rozdział 6

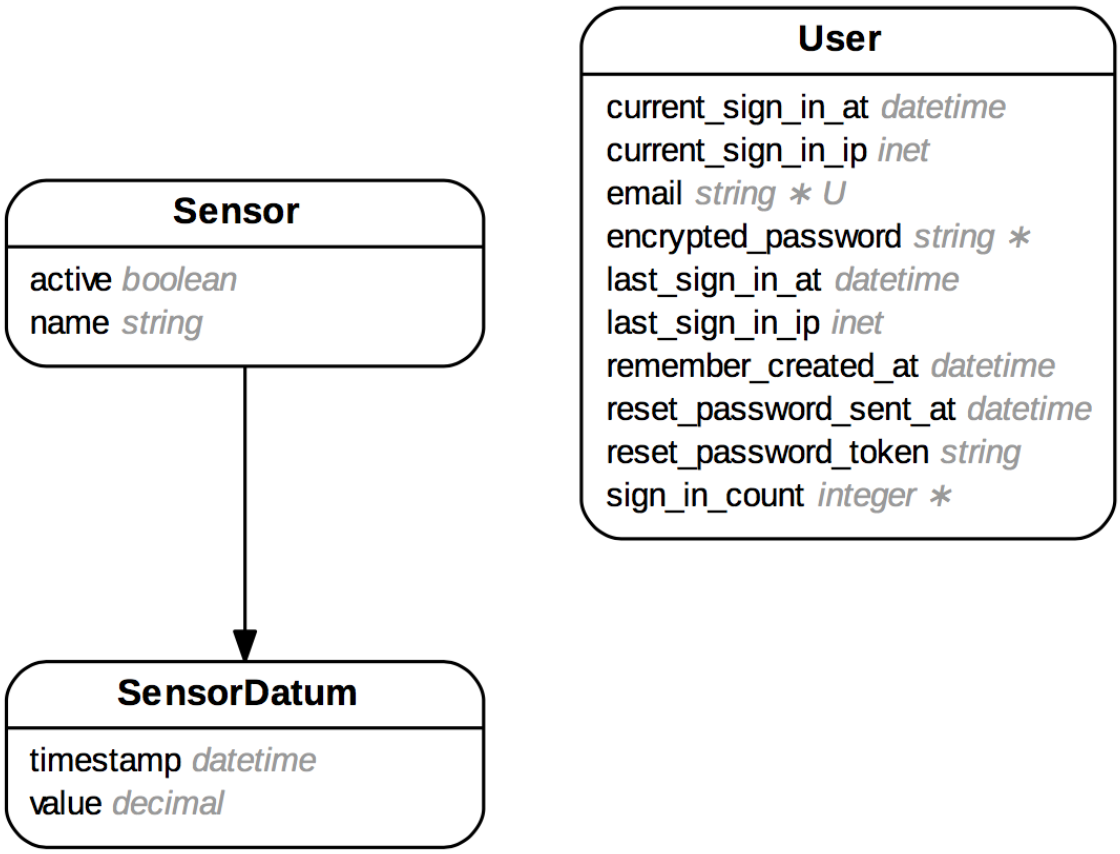
Implementacja

Aplikacje testowe miały za zadanie sprawdzać dostępność najczęściej wykorzystywanych mechanizmów używanych do budowy aplikacji internetowych. Aby rozwiązywały one rzeczywisty problem, miały one pełnić rolę aplikacji do zarządzania inteligentnym budynkiem. Tematyka ta nawiązuje do mojej pracy inżynierskiej, której tematem był „System zarządzania inteligentnym domem z wykorzystaniem Raspberry Pi oraz technologii internetowych".

W każdej z aplikacji zostały zaimplementowane następujące funkcjonalności:

- rejestracja i logowanie użytkowników - system autentykacji,
- dodawanie i usuwanie sensorów,
- dodawanie i usuwanie danych zbieranych przez sensory,
- przetwarzanie danych w tle na przykładzie przetwarzania plików CSV z danymi użytkowników.

Każda z aplikacji obsługuje bazę danych o strukturze, która przedstawiona jest na rysunku 6.1.



Rysunek 6.1: Diagram ERD bazy danych.

Źródło: opracowanie własne.

6.1 Ruby on Rails

Framework *Ruby on Rails* domyślnie korzysta z architektury *MVC*. Aby nie łamać zasady *Convention over Configuration*, zdecydowano się stworzyć aplikację zgodną z ideą frameworka, czyli w tej właśnie architekturze.

6.1.1 Modele

Aplikacja posiada 3 modele: *User*, *TemperatureSensor* oraz *TemperatureSensorDatum*. Dodatkowo, korzystając z mechanizmu *STI* (ang. *Single Table Inheritation*), istnieją dwa modele abstrakcyjne *Sensor* oraz *SensorDatum*, które są bazą dla różnych typów sensorów oraz danych zbierane przez owe sensory. Na listingach 6.1 oraz 6.2 przedstawiony jest przykład implementacji *STI*, czyli abstrakcyjny model sensora oraz dziedziczący po nim model *TemperatureSensor*.

Listing 6.1: Abstrakcyjny model sensora w Ruby on Rails.

```
class Sensor < ApplicationRecord
end
```

Listing 6.2: Model sensora temperatury w Ruby on Rails.

```
class TemperatureSensor < Sensor
  has_many :temperature_sensor_data
end
```

Łatwo zauważyć, że abstrakcyjny model jest praktycznie pusty. Jedyną ważną informacją jest dziedziczenie po klasie *ApplicationRecord*, po której musi dziedziczyć każdy model. Natomiast w klasie *TemperatureSensor* zdefiniowana jest relacja jeden do wielu z danymi sensora.

6.1.2 Kontrolery

Ruby on Rails posiada mechanizm, który pozwala na wygenerowanie szablonu kontrolera. Zawarte są w nim implementacje podstawowych operacji typu *CRUD*, czyli **C**reate - **R**ead - **U**pdate - **D**elete. Jako, że z założenia nie ma możliwości edycji przychodzących danych, pominięta została sekcja *Update*. Kontroler odpowiedzialny za dane z sensorów temperatury przedstawiony jest na listingu 6.3.

Listing 6.3: TemperatureSensorDataController w Ruby on Rails.

```
class TemperatureSensorDataController < ApplicationController
  before_action :set_temperature_sensor_datum, only: :destroy

  # GET /temperature_sensor_data
  def index
    @temperature_sensor_data = TemperatureSensorDatum.all
  end

  # GET /temperature_sensor_data/new
  def new
    @temperature_sensor_datum = TemperatureSensorDatum.new
  end

  # POST /temperature_sensor_data
  def create
```

```
    @temperature_sensor_datum = TemperatureSensorDatum.new(
        temperature_sensor_datum_params)
    respond_to do |format|
      if @temperature_sensor_datum.save
        format.html { redirect_to temperature_sensor_data_url, notice: '
            Temperature sensor datum was successfully created.' }
      else
        format.html { render :new }
        format.json { render json: @temperature_sensor_datum.errors, status
            : :unprocessable_entity }
      end
    end
  end

  # DELETE /temperature_sensor_data/1
  def destroy
    @temperature_sensor_datum.destroy
    respond_to do |format|
      format.html { redirect_to temperature_sensor_data_url, notice: '
          Temperature sensor datum was successfully destroyed.' }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_temperature_sensor_datum
      @temperature_sensor_datum = TemperatureSensorDatum.find(params[:id])
    end

    # Never trust parameters from the scary internet, only allow the white
        list through.
    def temperature_sensor_datum_params
      params.require(:temperature_sensor_datum).permit(:value, :timestamp,
          :sensor_id)
    end
end
```

6.1.3 Widoki

Każda z akcji w kontrolerze powinna mieć swój odpowiednik w warstwie widoków. Framework *Ruby on Rails* dostarcza mechanizm *partial'i*, który pozwala podział widoków na kilka plików w celu umożliwienia korzystania z jednego fragmentu w kilku miejscach. Przykład takiego rozwiązania można pokazać jako podstronę dodawania nowej próbki danych z sensora (listing 6.4), gdzie sam formularz jest reużywalnym fragmentem - *partial'em* (listing 6.5).

Listing 6.4: Widok, na którym można dodać nowe próbki dla sensora temperatury.

```
<h1>New Temperature Sensor Datum</h1>

<%= render 'form', temperature_sensor_datum: @temperature_sensor_datum %>

<%= link_to 'Back', temperature_sensor_data_path %>
```

Listing 6.5: Fragment, na którym jest formularz.

```
<%= form_for(temperature_sensor_datum) do |f| %>
  <% if temperature_sensor_datum.errors.any? %>
    <div id="error_explanation"\>
      <h2><%= pluralize(temperature_sensor_datum.errors.count, "error") %>
          prohibited this temperature_sensor_datum from being saved:</h2>

      <ul>
      <% temperature_sensor_datum.errors.full_messages.each do |message| %>
        <li><%= message %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field"\>
    <%= f.label :value %>
    <%= f.number_field :value, { step: 0.1 } %>
  </div\>

  <div class="field"\>
    <%= f.label :timestamp %>
    <%= f.datetime_local_field :timestamp %>
```

```
  </div>

  <div class="field"\>
    <%= f.label :sensor_id %>
    <%= f.collection_select(:sensor_id, TemperatureSensor.where(active:
        true), :id, :name) %>
  </div>

  <div class="actions"\>
    <%= f.submit %>
  </div>
<% end %>
```

6.1.4 Przetwarzanie w tle - ActiveJob

Framework *Ruby on Rails* posiada wbudowany mechanizm, który jest abstrakcją umożliwiającą przetwarzanie danych w tle - *ActiveJob*. Biblioteka ta posiada szereg adapterów do najbardziej popularnych mechanizmów zajmujących się owym przetwarzaniem. Dzięki temu możliwa jest ewentualna zmiana samego narzędzia, przy zachowaniu kodu. W aplikacji wykorzystane zostało narzędzie *Sidekiq*.

Sam proces przetwarzania składa się z trzech części - widoku z formularzem, przy pomocy którego użytkowniok może wysłać plik csv na serwer, akcji w kontrolerze, która odbiera plik i przekazuje go do ostatniego etapu, czyli przetwarzania w tle. Na listingu 6.1.4 przedstawiony został ostatni etap tego procesu.

```
require "csv"
class UserImportJob < ApplicationJob
  queue_as :default

  def perform(file)
    CSV.foreach(file.path) do |row|
      User.create!(row.to_h)
    end
  end
end
```

6.2 Phoenix

Phoenix, podobnie jak *Ruby on Rails*, jest frameworkiem korzystającym z architektury *Model-View-Controller*.

6.2.1 Modele

Jako, że *Phoenix* nie posiada wbudowanego mechanizmu *STI*, każdy model korzysta z osobnej tabeli w bazie danych. Oznacza to brak abstrakcyjnych modeli i ilość tabel w bazie danych równą ilości używanych modeli. Framework ten w modelu, poza relacjami z innymi modelami, przechowuje schemat danych w tabeli, z której korzysta dany model.

Listing 6.6: Model sensora temperatury we frameworku Phoenix.

```
defmodule App.TemperatureSensor do
  use App.Web, :model

  schema "temperature_sensors" do
    field :name, :string
    field :active, :boolean, default: false

    timestamps()
  end

  def changeset(struct, params \\ %{}) do
    struct
    |> cast(params, [:name, :active])
    |> validate_required([:name, :active])
  end
end
```

6.2.2 Kontrolery

Phoenix posiada mechanizm generowania *scaffoldów*, które bardzo przyspieszają tworzenie aplikacji i generują sporą część kodu kontrolera, dzięki czemu można tworzyć proste aplikacje z minimalną znajomością języka programowania *Elixir*. *Scaffold* domyślnie generuje akcje kontrolera w konwencji *CRUD*, lecz aby aplikacja była zgodna z wcześniejszymi założeniami, usunięta została część odpowiedzialna za edycję danych. Kontroler po tych

zmianach przedstawiony jest na listingu 6.7.

Listing 6.7: Kontroler danych z sensorów temperatur we frameworku Phoenix

```
defmodule App.TemperatureSensorDatumController do
  use App.Web, :controller
  require IEx

  alias App.TemperatureSensorDatum
  alias App.TemperatureSensor

  def index(conn, _params) do
    temperature_sensor_data = Repo.all(TemperatureSensorDatum)
    render(conn, "index.html", temperature_sensor_data:
        temperature_sensor_data)
  end

  def new(conn, _params) do
    changeset = TemperatureSensorDatum.changeset(%TemperatureSensorDatum{})
    query = from(s in TemperatureSensor, select: {s.name, s.id})
    sensors = Repo.all(query)
    render(conn, "new.html", changeset: changeset, sensors: sensors)
  end

  def create(conn, %{"temperature_sensor_datum" =>
      temperature_sensor_datum_params}) do
    temperature_sensor_datum_params =
      if temperature_sensor_datum_params["utc_timestamp"] do
        {:ok, timestamp} = DateTime.from_unix!(
            temperature_sensor_datum_params["utc_timestamp"])
        Map.put(temperature_sensor_datum_params, "utc_timestamp", timestamp
            )
      else
        temperature_sensor_datum_params
      end
    changeset = TemperatureSensorDatum.changeset(%TemperatureSensorDatum{},
        temperature_sensor_datum_params)

    case Repo.insert(changeset) do
      {:ok, _temperature_sensor_datum} ->
        conn
```

```
        |> put_flash(:info, "Temperature sensor datum created successfully
           .")
        |> redirect(to: temperature_sensor_datum_path(conn, :index))
      {:error, changeset} ->
        query = from(s in TemperatureSensor, select: {s.name, s.id})
        sensors = Repo.all(query)
        render(conn, "new.html", changeset: changeset, sensors: sensors)
    end
  end

  def show(conn, %{"id" => id}) do
    temperature_sensor_datum = Repo.get!(TemperatureSensorDatum, id)
    render(conn, "show.html", temperature_sensor_datum:
       temperature_sensor_datum)
  end

  def delete(conn, %{"id" => id}) do
    temperature_sensor_datum = Repo.get!(TemperatureSensorDatum, id)

    # Here we use delete! (with a bang) because we expect
    # it to always work (and if it does not, it will raise).
    Repo.delete!(temperature_sensor_datum)

    conn
    |> put_flash(:info, "Temperature sensor datum deleted successfully.")
    |> redirect(to: temperature_sensor_datum_path(conn, :index))
  end
end
```

6.2.3 Widoki

W kwestii warstwy prezentacji widoczna jest inspiracja frameworkiem *Ruby on Rails*. *Phoenix* również posiada mechanizm fragmentów i działa on na takiej samej zasadzie, jak w oryginale.

Listing 6.8: Widok, na którym można dodać dane dla sensora w Phoenix'ie.

```
<h2>New temperature sensor datum</h2>

<%= render "form.html", changeset: @changeset,
                        action: temperature_sensor_datum_path(@conn, :
                            create),
                        sensors: @sensors %>

<%= link "Back", to: temperature_sensor_datum_path(@conn, :index) %>
```

Listing 6.9: Fragment, na którym jest formularz we frameworku Phoenix.

```
<%= form_for @changeset, @action, fn f -> %>
  <%= if @changeset.action do %>
    <div class="alert alert-danger"\>
      <p>Oops, something went wrong! Please check the errors below.</p>
    </div>
  <% end %>

  <div class="form-group"\>
    <%= label f, :value, class: "control-label" %>
    <%= number_input f, :value, step: "any", class: "form-control" %>
    <%= error_tag f, :value %>
  </div>

  <div class="form-group"\>
    <%= label f, :timestamp, class: "control-label" %>
    <%= datetime_select f,  :timestamp, class: "form-control" %>
    <%= error_tag f, :timestamp %>
  </div>

  <div class="form-group"\>
    <%= label f, :sensor_id, class: "control-label" %>
    <%= select(f, :sensor_id, @sensors) %>
  </div>

  <div class="form-group"\>
    <%= submit "Submit", class: "btn btn-primary" %>
  </div>
<% end %>
```

6.2.4 Przetwarzanie danych w tle

Phoenix nie posiada wbudowanej biblioteki, która nakłada abstrakcję i unifikuje dostęp do mechanizmów odpowiedzialnych za przetwarzanie w tle. Nie oznacza to jednak, że realizacja przetwarzania danych w tle jest niemożliwa. Z pomocą przychodzi biblioteka *Exq*. Sugeruje ona rozszerzenie istniejącej struktury o *Workery*. Zaimplementowany *worker* został przedstawiony na listingu 6.10.

Listing 6.10: Worker umożliwiający import użytkowników z pliku csv we frameworku Phoenix.

```
defmodule App.ImportUsersWorker do

  def perform(file) do
    File.stream!(file.path)
      |> CSV.decode(separator: ?;) |>
        Enum.each(fn row ->
          res = %{
            email: Enum.at(row, 0),
            password: Enum.at(row, 1),
            password_confirmation: Enum.at(row, 1)
          }
          User.changeset(%User{}, res)
          |> Repo.insert
        end)
  end
end
```

Aby dodać zadanie do kolejki, z poziomu kontrolera należy wywołać metodę przedstawioną na listingu 6.11

Listing 6.11: Dodanie zadania do kolejki przetwarzania w tle we frameworku Phoenix.

```
def import(conn, params) do
  {:ok, jid} = Exq.enqueue(Exq, "default", "ImportUsersWorker", [params["
      import"]["csv"]])
  text conn, "Task scheduled"
end
```

6.3 Express

Express nie posiada domyślnie zdefiniowanej architektury, a więc również struktury projektu. Decyzja odnośnie architektury aplikacji leży w całości po stronie programisty. Aby zachować podobieństwo do pozostałych aplikacji, zdecydowano się na zastosowanie modelu *Model-View-Controller*.

6.3.1 Modele

Korzystanie z modeli możliwe jest dzięki zastosowaniu biblioteki *Sequelize.js*. W modelach zawarte są informacje na temat pól danego modelu i ich typów, a więc jego schemat. Dodatkowo, model zawiera informacje na temat relacji z innymi modelami.

Listing 6.12: Model sensora temperatury w Express'ie.

```
var Sequelize = require('sequelize')

var TemperatureSensor = connection.define('temperature_sensors',
                                          TemperatureSensorMeta.attributes,
                                          TemperatureSensorMeta.options)

TemperatureSensor.hasMany(TemperatureSensorDatum);

var attributes = {
  name: {
    type: Sequelize.STRING,
  },
  active: {
    type: Sequelize.BOOLEAN,
  }
}

var options = {
  freezeTableName: true
}

module.exports.attributes = attributes
module.exports.options = options
```

6.3.2 Kontrolery

W przeciwieństwie do *Ruby on Rails* i *Phoenix'a*, *Express* nie posiada mechanizmu generowania *scaffold'ów*. W efekcie wszystkie metody należy stworzyć samodzielnie, co ma wpływ na szybkość powstawania aplikacji, w której większośc akcji jest standardowych, czyli spełniających założenia *CRUD*. Na listingu 6.13 przedstawiony jest kontroller we frameworku *Express*.

Listing 6.13: Kontroler danych z sensora temperatury we frameworku Express.

```
var Model = require('../model/models.js')

module.exports.index = function(req, res) {
  Model.TemperatureSensorDatum.findAll().then(data => {
    res.send(data)
  })
}

module.exports.create = function(req, res) {
  var value = req.body.temperature_sensor_datum.value
  var sensorId = req.body.temperature_sensor_datum.sensor_id

  var newTemperatureSensorDatum = {
    value: parseFloat(value),
    temperatureSensorId: sensorId
  }

  Model.TemperatureSensorDatum.create(newTemperatureSensorDatum).then(
      function() {
    res.redirect('/')
  }).catch(function(error) {
    res.send('error')
    req.flash('error', "Something went wrong...")
  })
}
```

6.3.3 Widoki

W implementacji warstwy widoków został wykorzystany silnik *Handlebards*. Jest to popularne rozwiązanie, spotykane w innych frameworkach webowych zorientowanych na warstwę prezentacji. Przykładowym frameworkiem, który domyślnie korzysta z tego mechanizmu jest *Ember.js*. System ten oparty jest o mechanizm *layout'ów*, w których osadzane są konkretne podstrony. Dzięki temu możliwe jest zachowanie spójnego wyglądu na każdej z podstron. Na listingach 6.14 oraz 6.15 przedstawiony jest layout strony oraz zawartość podstrony do rejestracji użytkownika.

Listing 6.14: Layout używany w aplikacji napisanej przy pomocy Express'a.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" \>
  <title>Example App</title>
  <!-- Bootstrap -->
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap
      /3.3.5/css/bootstrap.min.css" \>
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap
      /3.3.5/css/bootstrap-theme.min.css" \>

  <link rel="stylesheet" href="/styles/app.css" \>

  {{{_sections.head}}}
</head>
<body>
    {{{body}}}

    <!-- Bootstrap -->
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/
        bootstrap.min.js" \></script>
    <!-- IE10 viewport hack for Surface/desktop Windows 8 bug -->
    <script src="../../assets/js/ie10-viewport-bug-workaround.js" \></script
        >
</body>
</html>
```

Listing 6.15: Podstrona rejestracji użytkownika w Express'ie.

```
{{#section 'head'}}
    <link rel="stylesheet" href="/styles/auth.css"\>
{{/section}}

<div class="container"\>
  <form method="POST" action="/users/sign_up" class="form-signin"\>
    <h2 class="form-signin-heading"\>Create an account</h2>

    {{#if errorMessage}}
      <div class="alert alert-danger"\>
        {{errorMessage}}
      </div>
    {{/if}}

    <label for="inputEmail" class="sr-only"\>Email</label>
    <input type="text" id="inputEmail" name="email" class="form-control"
        placeholder="Email" required autofocus>
    <label for="inputPassword" class="sr-only"\>Password</label>
    <input type="password" id="inputPassword" name="password" class="form-
        control" placeholder="Password" required>
    <label for="inputPassword" class="sr-only"\>Repeat Password</label>
    <input type="password" id="inputPassword2" name="password2" class="form
        -control" placeholder="Repeat Password" required>
    <button class="btn btn-lg btn-primary btn-block" type="submit"\>Sign up
        </button>
  </form>
</div>
```

Rozdział 7

Analiza wydajnościowa

7.1 Metody mierzenia wydajności aplikacji internetowych

Głównym problemem aplikacji internetowych jest sytuacja, gdy w jednym momencie próbuje z niej skorzystać duża ilość użytkowników. Błędy aplikacji objawiają się wtedy poprzez długi czas przetwarzania, błąd przetwarzania bądź też odrzucenie zapytania HTTP. Oczywiście takie zachowanie jest bardzo niepożądane, dlatego też nowoczesne frameworki starają się minimalizować możliwość wystąpienia takich sytuacji.

Z tego też powodu aplikacjie poddawane są testom obciążeniowym. Tego typu testy opierają się na symulacji dużej liczby użytkowników próbujących skorzystać z aplikacji i analizie odpowiedzi z aplikacji. Na podstawie testów można uzyskać dane m.in. na temat:

- procentowej ilości błędów,
- wartości minimalnej, maksymalnej, średniej i medianie czasu odpowiedzi,
- rozmiaru odpowiedzi z serwera,
- średniej ilości przetworzonych zapytań w jednostce czasu.

7.2 Wyniki badań

Badania zostały przeprowadzone na serwerze *VPS* firmy *DigitalOcean*. Posiada ona następujące parametry:

- procesor: dwurdzeniowy Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz,
- pamięć RAM: 2GB,
- dysk: 40GB SSD,
- system operacyjny: Ubuntu 16.04.2 LTS (Xenial Xerus)

Zgodnie z zaprojektowanym środowiskiem eksperymentalnym, testy wydajnościowe zostały przeprowadzone z wykorzystaniem narzędzia *Locust*. Wyniki badań są uśrednionymi wartościami z 10 przebiegów testowych.

7.2.1 Czas odpowiedzi z serwera

W kategorii czasu odpowiedzi z serwera mierzone były następujące wartości:

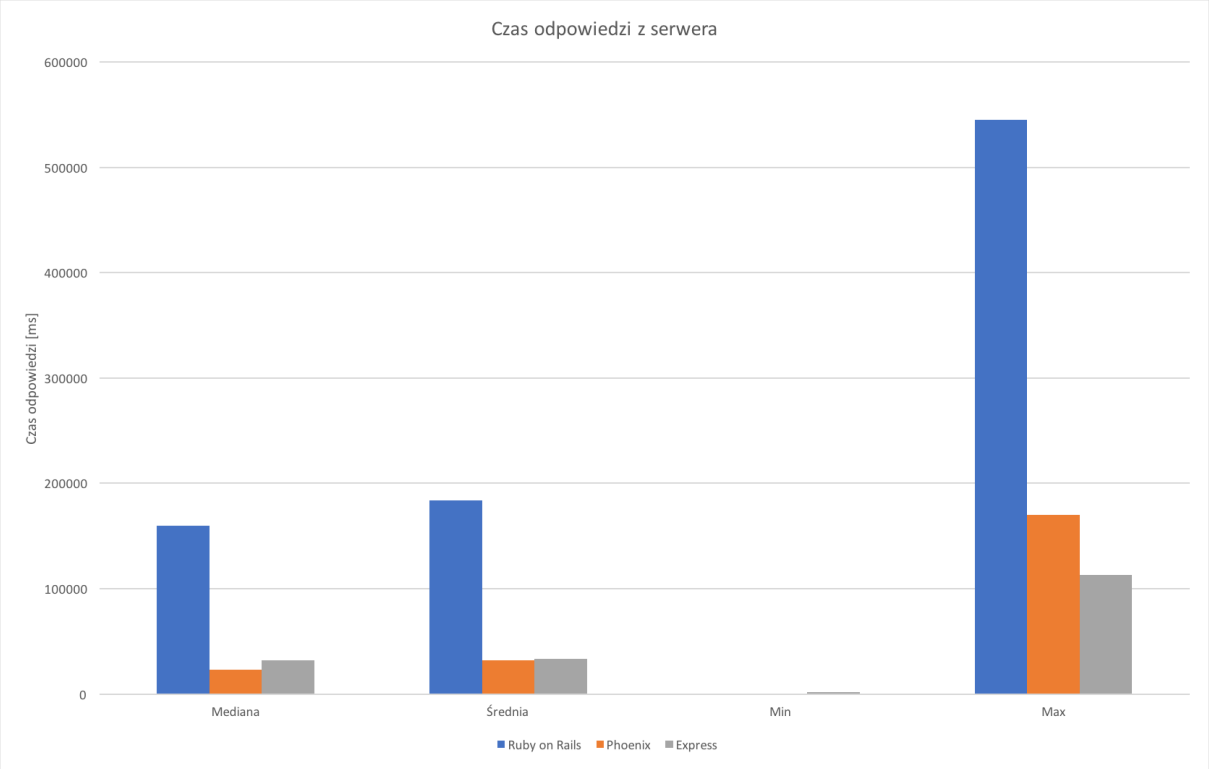
- mediana czasu odpowiedzi,
- średnia czas odpowiedzi,
- minimalny czas odpowiedzi,
- maksymalny czas odpowiedzi.

Wyniki dla poszczególnych frameworków przedstawione są w tabeli 7.1.

Tabela. 7.1: Wyniki badań czasu odpowiedzi.
Źródło: opracowanie własne.

	Mediana [ms]	**Średnia [ms]**	**Min [ms]**	**Max [ms]**
Ruby on Rails	160000	184060	114	544870
Phoenix	23000	31873	623	169825
Express	32000	33193	1514	112874

Już podczas analizy wyników w formie tabelarycznej można zauważyć dużo większe wartości przy frameworku *Ruby on Rails*. Różnice te są dużo bardziej widoczne na zestawieniu pokazanym w formie wykresu słupkowego przedstawionego na rysunku 7.1.



Rysunek 7.1: Porównanie czasów odpowiedzi z serwera.
Źródło: opracowanie własne.

Zgodnie z przewidywaniami, *Ruby on Rails* jest najwolniejszym frameworkiem, jeśli chodzi o czas odpowiedzi z serwera. Średni czas odpowiedzi jest ok. 5,5 razy większy niż w przypadku pozostałych frameworków. Spowodowane jest to faktem, iż język *Ruby* nie należy do najszybszych języków programowania. Jego celem miało być dostarczenie przyjemności z programowania, co niestety odbiło się negatywnie na wydajności.

Najszybszym frameworkiem okazał się, zgodnie z oczekiwaniami, *Phoenix*. Twórcom udało się zrealizować założenie dotyczące wydajności tego frameworka. Wpływ ma na to oczywiście wspomniany wcześniej język *Elixir*. Co ciekawe, wydajność ta została osiągnięta z zachowaniem przyjemności z tworzenia aplikacji, co potwierdza duży potencjał tego frameworka.

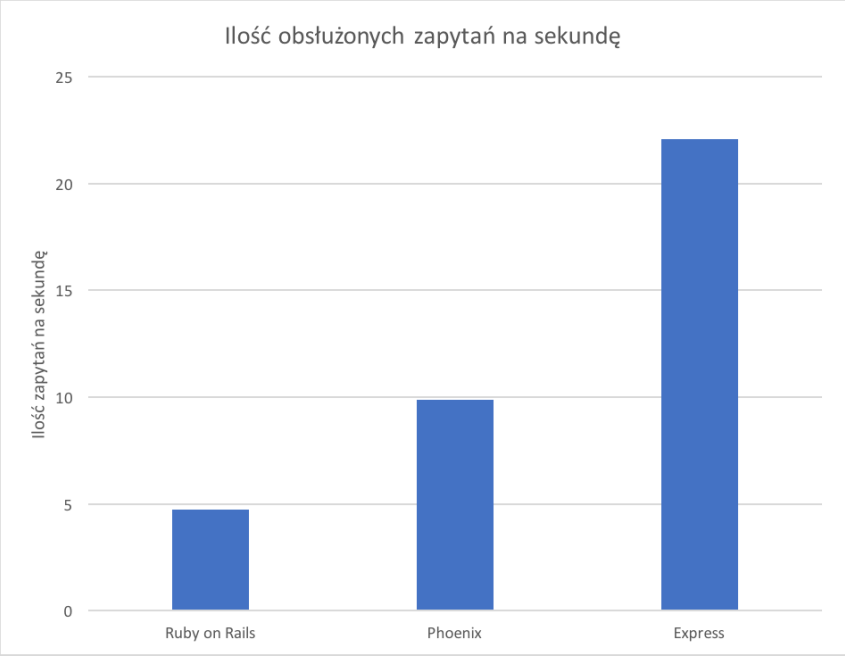
Zaskoczeniem jest wydajność frameworka *Express*. W zestawieniu jest on minimalnie wolniejszy od *Phoenix'a*, co można określić jako bardzo dobry wynik. Wpływ na to ma minimalizm tego narzędzia. *Express* z racji na swoją „ubogość" domyślnych bibliotek jest bardzo małym frameworkiem. W połączeniu z szybką platformą *Node.js*, która odpowiedzialna jest za serwowanie aplikacji, otrzymane zostały bardzo dobre wyniki czasu odpowiedzi.

7.2.2 Ilość obsłużonych zapytań na sekundę

Poza czasem odpowiedzi bardzo ważnym parametrem jest ilość obsłużonych zapytań na sekundę. Analiza tego parametru pozwala określić ilu użytkowników może jednocześnie korzystać z systemu bez problemów z wydajnością. Efektem owej analizy może być decyzja o zwiększeniu mocy obliczeniowej maszyny, która obsługuje aplikację bądź też zastosowanie *load balancing'u*. Wyniki przeprowadzonych badań zostały przedstawione w tabeli 7.2 oraz na wykresie 7.3.

Tabela. 7.2: Wyniki badań ilości zapytań na sekundę.

	RPS
Ruby on Rails	4,74
Phoenix	9,86
Express	22,09



Rysunek 7.2: Porównanie ilości obsłużonych zapytań na sekundę.
Źródło: opracowanie własne.

Wyniki tych badań są nieco zaskakujące. Najwięcej zapytań, z ponad dwukrotnie lepszym wynikiem niż drugi w zestawieniu *Phoenix*, obsłużył framework *Express*. Przyczyn takiego wyniku ponownie można doszukiwać się w platformie *Node.js*, na której oparty

jest ten framework. Jest ona bardzo stabilna i stanowi podstawę dla wielu frameworków, w tym frontendowych, co doprowadziło do optymalizacji wydajności.

Drugi pod względem ilości przetworzonych zapytań na sekundę jest framework *Phoenix*. Przed przeprowadzeniem testów, framework ten był faworytem w osiągnięciu najlepszego wyniku ze względu na język *Elixir*, który pochodzi od *Erlang'a*, a więc jest językiem posiadającym mechanizm *lekkich wątków* (mechanizm podobny do *zielonych wątków*). Testy jednak jednoznacznie pokazały, iż ilość obsłużonych zapytań w czasie sekundy nie jest wyjątkowo duża.

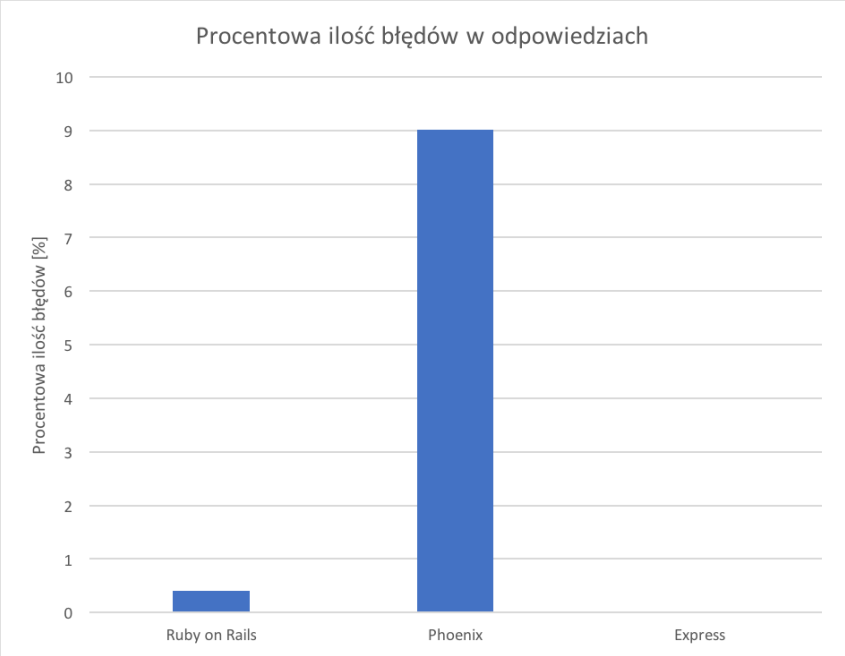
Frameworkiem, który zdołał obsłużyć najmniej zapytań w ciągu sekundy, został *Ruby on Rails*. Także w tym przypadku taki wynik był zgodny z przewidywaniami. *Ruby on Rails* jest 2 razy wolniejszy od *Phoenix'a* i aż 4 razy wolniejszy od *Express'a*. Pokazuje to, że framework ten, zgodnie z jego założeniami, powinien być stosowany głównie w celu prototypowania aplikacji.

7.2.3 Poprawność odpowiedzi z serwera

Poza szybkością działania aplikacji, bardzo ważnym elementem aplikacji internetowych jest ich niezawodność. Nawet jeśli framework jest wydajny, to traci on na wartości, jeśli nie jest niezawodny. W tym celu przeprowadzone zostały badania niezawodności. Ich wynikiem jest procentowa ilość błędów w odniesieniu do ilości zapytań. Wyniki w formie tabelarycznej oraz wykresu są przedstawione poniżej.

Tabela. 7.3: Wyniki ilości błędów zapytań w odniesieniu do ilości zapytań.

	Ilość błędów [%]
Ruby on Rails	0,41
Phoenix	9,01
Express	0,00



Rysunek 7.3: Zestawienie ilości błędnych odpowiedzi z serwerów.
Źródło: opracowanie własne.

Wyniki otrzymane na podstawie badań odpowiadają przewidywanym wynikom. *Ruby on Rails* osiągnął ilość błędów na poziomie $0,41\%$. Jest to wynik bardzo dobry, biorąc pod uwagę obciążenie, któremu została poddana aplikacja. I stanowczo jest to wynik mieszczący się w dopuszczalnych granicach błędów, które nie wpływają negatywnie na odczucia użytkownika.

Niewielką niespodzianką jest wynik frameworka *Phoenix*. 9% błednych odpowiedzi jest dużą wartością. Wynik ten można tłumaczyć jedynie młodym wiekiem tego rozwiązania, ponieważ *Elixir* z założenia stawia na niezawodność. Stanowczo jest to element, który powinien zostać dopracowany w kolejnych wersjach frameworka.

Wręcz idealny wynik osiągnął *Express*. W trakcie badań jego ilość błędów utrzymywała się na poziomie 0%, co oznacza pełną niezawodność. Przyczyną takiego wyniku jest niewątpliwie niewielki rozmiar frameworka. Oczywistym jest, że im mniej elementów może wygenerować błąd, tym większa niezawodność systemu.

7.3 Wnioski z badań

Na podstawie badań można wysnuć kilka wniosków. Po pierwsze - założenia frameworka nie zawsze pokrywają się z rzeczywistością. Przykładowo, *Phoenix* reklamowany

jest jako framework stawiający na szybkość działania, skalowalność i niezawodność. W przypadku pierwszych dwóch własności cel został osiągnięty (chociaż np. w porównaniu do *Express'a* nie jest to wynik dużo lepszy). Problemem jest niestety niezawodność. Nawet najszybszy framework nie będzie wykorzystywany w momencie, kiedy nie jest niezawodny.

Po drugie, minimalizm, szczególnie pod względem wydajności, jest korzystny. Oczywiście taką aplikację trudniej się pisze oraz jest ryzyko, że mimo wszystko rozrośnie się do dużych rozmiarów. Mimo wszystko jeśli ktoś poszukuje wydajności, powinien skorzystać z *Express'a* lub podobnych do niego frameworków.

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.