

**POLITECHNIKA WROCŁAWSKA**  
**WYDZIAŁ ELEKTRONIKI**

---

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

**PRACA DYPLOMOWA  
MAGISTERSKA**

Analiza porównawcza frameworków  
internetowych w języku Ruby w zastosowaniach  
GISowych

Comparative analysis of Ruby's web frameworks  
for Geographic Information Systems

**AUTOR:**  
inż. Mikołaj Grygiel

**PROWADZĄCY PRACĘ:**  
dr inż. Roman Ptak, W4/K9

**OCENA PRACY:**



# Spis treści

<b>Spis rysunków</b>	<b>5</b>
<b>Spis fragmentów kodu</b>	<b>5</b>
<b>1 Wprowadzenie</b>	<b>7</b>
1.1 Cel pracy . . . . .	7
1.2 Zakres i koncepcja pracy . . . . .	7
<b>2 Podstawy teoretyczne</b>	<b>9</b>
2.1 Charakterystyka Systemów Informacji Geograficznej . . . . .	9
2.2 Charakterystyka języka Ruby . . . . .	10
<b>3 Istniejące systemy GIS w języku Ruby</b>	<b>11</b>
3.1 OpenStreetMap . . . . .	11
3.2 MangoMap . . . . .	11
<b>4 Frameworki internetowe w języku Ruby</b>	<b>13</b>
4.1 Ruby on Rails . . . . .	14
4.2 Roda . . . . .	15
4.3 Hanami . . . . .	17
<b>5 Narzędzia dostępne do przetwarzania danych geograficznych w języku Ruby</b>	<b>19</b>
5.1 Przechowywanie danych . . . . .	19
5.1.1 PostgreSQL . . . . .	19
5.1.2 PostGIS . . . . .	20
5.1.3 MySQL Spatial . . . . .	20
5.1.4 SpatiaLite . . . . .	20
5.2 Obiektowe przetwarzanie danych . . . . .	20
5.2.1 RGeo . . . . .	20
5.2.2 GeoRuby . . . . .	21
5.3 Prezentowanie danych . . . . .	21
5.3.1 Leaflet . . . . .	21
5.3.2 GoogleMaps JavaScript API . . . . .	21
<b>6 Aplikacja zaimplementowana na potrzeby badań</b>	<b>23</b>
6.1 Opis aplikacji . . . . .	23
6.2 Implementacja . . . . .	25
6.3 Prezentacja aplikacji . . . . .	25

<b>7 Badania</b>	<b>27</b>
7.1 Plan badań . . . . .	27
7.1.1 Środowisko badawcze . . . . .	27
7.1.2 Narzędzia wykorzystane podczas badań . . . . .	28
7.2 Porównanie funkcjonalności wybranych frameworków . . . . .	28
7.2.1 Przechowywanie danych . . . . .	28
7.2.2 Obiektowe przetwarzanie danych . . . . .	30
7.2.3 Prezentowanie danych . . . . .	30
7.2.4 Podsumowanie . . . . .	31
7.3 Porównanie struktur wykonanych projektów . . . . .	32
7.4 Porównanie wydajności zaimplementowanych aplikacji . . . . .	38
7.4.1 Interfejs użytkownika . . . . .	38
7.4.2 Zapytania HTTP . . . . .	42
7.4.3 Komunikacja z bazą danych . . . . .	49
<b>8 Podsumowanie i wnioski</b>	<b>53</b>
8.1 Podsumowanie wykonanej pracy . . . . .	53
8.2 Analiza wyników badań . . . . .	53
8.3 Realizacja celu projektu . . . . .	53
<b>Literatura</b>	<b>55</b>
<b>A Opis zawartości płyty CD</b>	<b>57</b>

# Spis rysunków

2.1	Histora popularności języka Ruby według rankingu Tiobe . . . . .	10
4.1	Architektura aplikacji w Ruby on Rails, źródło: <a href="http://blog.ifuturz.com/ruby-on-rails/ruby-on-rails-mvc-learn-with-fun.html">http://blog.ifuturz.com/ruby-on-rails/ruby-on-rails-mvc-learn-with-fun.html</a> . . . . .	14
4.2	Podstawowa struktura projektu Ruby on Rails . . . . .	15
4.3	Architektura aplikacji w frameworku Roda . . . . .	16
4.4	Struktura projektu Roda . . . . .	16
4.5	Schemat czystej architektury [19] . . . . .	17
4.6	Podstawowa struktura projektu Hanami . . . . .	18
6.1	Diagram przypadków użycia aplikacji . . . . .	24
6.2	Schemat ERD bazy danych . . . . .	25
6.3	Widok gotowej aplikacji . . . . .	26
7.1	Struktura projektu Ruby on Rails . . . . .	32
7.2	Struktura projektu Roda . . . . .	33
7.3	Struktura projektu Hanami - część I . . . . .	34
7.4	Struktura projektu Hanami - część II . . . . .	35
7.5	Liczba plików w projekcie w zależności od użytego frameworku . . . . .	36
7.6	Liczba folderów w projekcie w zależności od użytego frameworku . . . . .	36
7.7	Liczba linii kodu w projekcie w zależności od użytego frameworku . . . . .	37
7.8	Czas wykonania operacji CRUD dla punktu . . . . .	38
7.9	Czas ładowania widoku w zależności od liczby punktów . . . . .	39
7.10	Czas wykonania operacji CRUD dla linii . . . . .	39
7.11	Czas ładowania widoku w zależności od liczby linii . . . . .	40
7.12	Czas wykonania operacji CRUD dla wielokątów . . . . .	40
7.13	Czas ładowania widoku w zależności od liczby wielokątów . . . . .	41
7.14	Czas wykonania zapytania HTTP dla punktu . . . . .	42
7.15	Czas ładowania danych w zależności od liczby punktów . . . . .	43
7.16	Czas ładowania danych w zależności od liczby użytkowników . . . . .	43
7.17	liczba obsłużonych zapytań na sekunde w zależności od liczby użytkowników	44
7.18	Czas wykonania operacji CRUD dla linii . . . . .	44
7.19	Czas ładowania danych w zależności od liczby linii . . . . .	45
7.20	Czas ładowania danych w zależności od liczby użytkowników . . . . .	45
7.21	Liczba obsłużonych zapytań na sekunde w zależności od liczby użytkowników	46
7.22	Czas wykonania operacji CRUD dla wielokątów . . . . .	47
7.23	Czas ładowania danych w zależności od liczby wielokątów . . . . .	47
7.24	Czas ładowania danych w zależności od liczby użytkowników . . . . .	48
7.25	Liczba obsłużonych zapytań na sekunde w zależności od liczby użytkowników	48
7.26	Czas wykonania operacji CRUD dla punktu . . . . .	49

---

7.27 Czas wczytywania danych z bazy w zależności od liczby punktów . . . . .	50
7.28 Czas wykonania operacji CRUD dla linii . . . . .	50
7.29 Czas wczytywania danych z bazy w zależności od liczby linii . . . . .	51
7.30 Czas wykonania operacji CRUD dla wielokątów . . . . .	51
7.31 Czas wczytywania danych z bazy w zależności od liczby wielokątów . . . . .	52

# Spis fragmentów kodu

4.1	Proste drzewo trasowań . . . . .	16
7.1	Obsługa typów geometrycznych z biblioteki PostGIS przez Hanami . . . . .	30
7.2	Tworzenie obiektu RGeo z zapisu binarnego danych przestrzennych . . . . .	30
7.3	Prezenter dla modelu Point . . . . .	31



# Rozdział 1

## Wprowadzenie

### 1.1 Cel pracy

Język Ruby zajmuje 12. miejsce w rankingu popularności języków programowania *Tiobe*<sup>1</sup>. Dużą popularnością wśród frameworków internetowych cieszy się Ruby on Rails, w rankingu *Hotframeworks*<sup>2</sup> zajmuje 3 miejsce wśród wszystkich frameworków. Ruby on Rails jest bez wątpienia najpopularniejszym frameworkm w języku Ruby, kolejne dwa frameworki w języku Ruby to Sinatra i Hanami, zajmują w wcześniej przytoczonym rankingu odpowiednio miejsca 25. i 73. Jednak w języku Ruby istnieje kilkanaście frameworków przeznaczonych do budowania aplikacji internetowych.

Celem niniejszej pracy jest poznanie wybranych frameworków w języku Ruby, ich porównanie w konkretnym zastosowaniu jakim są systemy informacji geograficznej oraz odpowiedź na pytanie jaki framework najlepiej wybrać do tworzenia systemu GIS.

### 1.2 Zakres i koncepcja pracy

„Framework” można zdefiniować jako szkielet służący do budowania aplikacji, czyli zbiór gotowych rozwiązań powtarzających się problemów i wzór do budowania nowych funkcjonalności. [22]

W niniejszej pracy zostaną omówione wybrane frameworki w języku Ruby w świetle ich użyteczności przy budowie systemu informacji geograficznej. Frameworki zostaną porównane na podstawie informacji zawartych w dostępnej dokumentacji narzędzia oraz zaimplementowanej przykładowej aplikacji, za pomocą każdego z wybranych narzędzi, spełniającej wymagania systemu GIS.

---

<sup>1</sup>Dane z marca 2017 r. dostępne na stronie <https://www.tiobe.com/tiobe-index/>

<sup>2</sup>Ranking <https://hotframeworks.com> bierze pod uwagę liczbę repozytoriów kodu na platformie Github i ilość tematów na forum Stackoverflow dotyczących danego frameworku. Dane z dnia 26.03.2017 r.



# Rozdział 2

## Podstawy teoretyczne

### 2.1 Charakterystyka Systemów Informacji Geograficznej

System Informacji Geograficznej skrótnie nazywany GIS (ang. *Geographic Information System*) można zdefiniować na wiele sposobów. Michael Schmandt w swoim opracowaniu [21] podaje następujące definicje:

**Definicja 1.** GIS to system komputerowy składający się z sprzętu i oprogramowania oraz ludzie, którzy wspomagają zbieranie, zarządzanie, analizowanie i wyświetlanie danych przestrzennych. Stosując tą definicję możemy podzielić system GIS na 4 moduły:

- Moduł wprowadzania danych - zawiera narzędzia pozwalające na wprowadzanie i przechowywanie danych przestrzennych.
- Moduł zarządzania danymi - ta część umożliwia edytowanie oraz przeglądanie zgromadzonych zbiorów danych.
- Moduł analizowania danych - podsystem, który odpowiada za analizowanie danych geograficznych i wyciąganie z nich informacji.
- Moduł prezentowania danych - pozwala na tworzenie map, modeli i statystyk ilustrujących zgromadzone dane.

**Definicja 2.** System informacji geograficznej to system komputerowy, który pozwala na przechowywanie danych powiązanych ze sobą geograficznie.

**Definicja 3.** GIS to narzędzie do wyszukiwania wzorców geograficznych (przestrzennych) w zbiorach danych.

Pierwsza definicja jest najbardziej szeroka i zawiera w sobie dwie następne - definicja druga to dwa pierwsze moduły z **Definicja 1.**, a definicja trzecia to moduł analizowania danych.

W podobny sposób do definicji nr 1 GIS jest zdefiniowany w *Principles of Geographic Information Systems* [18] jako zbiór narzędzi pozwalających operować na danych reprezentujących zjawiska geograficzne. Zbiór ten dzieli się na 4 grupy ze względu na funkcje:

- zbieranie i przygotowywanie danych,

- zarządzanie i przechowywanie danych,
- analiza danych,
- prezentowanie danych.

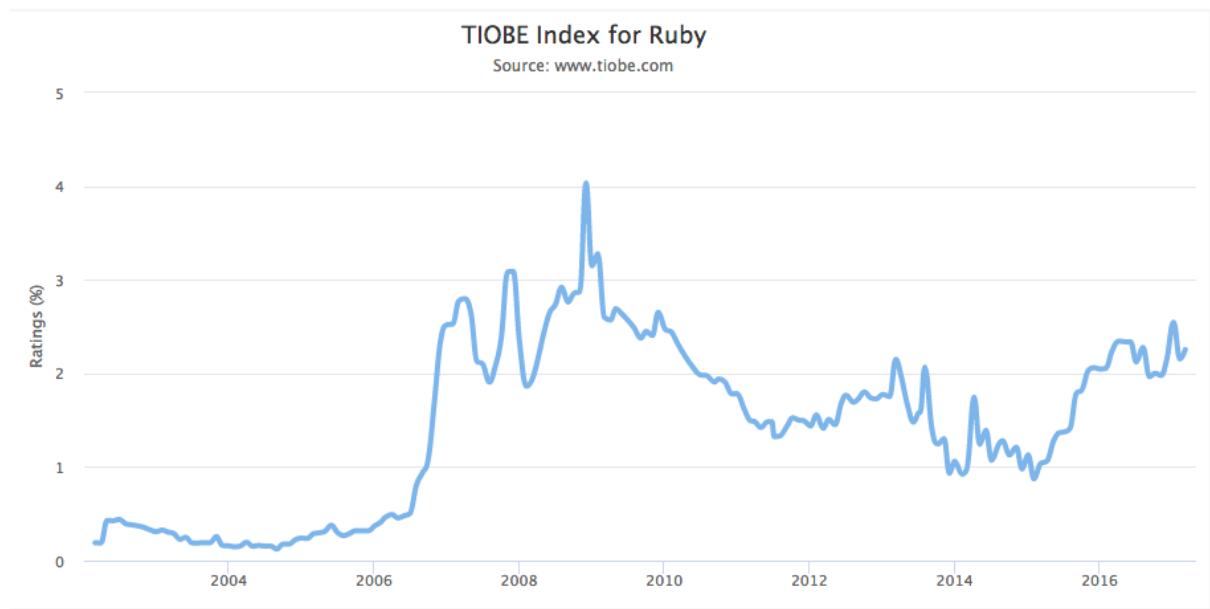
W poniższej pracy przyjmuje się pierwszą definicję Systemu Informacji Geograficznego - GIS to system informatyczny służący do wprowadzania, przechowywania, zarządzania, analizowania i prezentacji danych przestrzennych.

## 2.2 Charakterystyka języka Ruby

Język Ruby został wydany w 1995 roku. Twórca Rubiego, Yukihiro „Matz” Matsumoto, inspirował się takimi językami programowani jak Perl, Smalltalk, Eiffel, Ada i Lisp by stworzyć jego zdaniem język, który zbalansuje programowanie funkcjonalne z programowaniem imperatywnym. [8] Składnia Rubiego ma przypominać język naturalny, autor języka opisuje go jako: *Ruby jest prosty z wyglądu, ale bardzo skomplikowany w środku, tak jak ciało ludzkie.*<sup>1</sup>

Ruby jest językiem ścisłe obiektowym, wszystko postrzegane jest jako obiekt. Każda funkcja jest metodą, ponieważ musi być przyłączona do jakiegoś obiektu. Ruby posiada celowo tylko jednokrotne dziedziczenie, ale pozwala na dołączanie wielu modułów, które są zbiorami metod do klasy. Ruby jest bardzo elastycznym językiem, pozwala na usunięcie lub przeddefiniowanie dowolnej swojej części. Mimo silnie obiektowej natury, dostępne są również elementy programowania funkcyjnego takie jak funkcje anonimowe lub domknięcia.

Szerszą popularność Ruby zyskał w 2006 r., 11 lat po publikacji. Swoją popularność zawdzięcza głównie frameworkowi Ruby on Rails. w rankingu popularności języków programowania Tiobe znajduje się aktualnie na 12 miejscu<sup>2</sup>.



Rysunek 2.1 Historia popularności języka Ruby według rankingu Tiobe

<sup>1</sup> Wypowiedź w liście ruby-talk 12.05.2000 r., źródło:

<http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/2773>

<sup>2</sup> Dane z dnia 08.04.2017 r. <https://www.tiobe.com/tiobe-index/>

# Rozdział 3

## Istniejące systemy GIS w języku Ruby

### 3.1 OpenStreetMap

OpenStreetMap jest internetowym systemem informacji geograficznej z otwartym kodem źródłowym. System zbudowany z wykorzystaniem bazy danych PostgreSQL, frameworku Ruby on Rails oraz biblioteki JavaScriptowej Leaflet służącej do tworzenia interaktywnych widoków z mapami. Dostęp do danych jest otwarty, dane mogą być edytowane przez dowolnego użytkownika, dlatego mogą być niezgodne z rzeczywistością. OpenStreetMap definiuje 4 typy obiektów przestrzennych [12]:

- Węzeł (ang. *node*) - pojedynczy punkt przestrzenny reprezentowany przez długość i szerokość geograficzną.
- Linia (ang. *way*) - jest to uporządkowany zbiór punktów, które mogą reprezentować funkcje liniowe(wektory) lub obszary.
- Relacja (ang. *relation*) - składa się z uporządkowanej listy węzłów, linii i innych relacji.
- Tag (ang. *tag*) - to jednostka informacji dołączona do obiektu jednego z powyżej opisanych typów. Tag składa się z klucza oraz wartości.

OpenStreetMap można wykorzystać przez utworzenie komponentu HTML z wybraną mapą, gotowego do zamieszczenia na dowolnej stronie internetowej lub przez pobranie danych z wybranej mapy. Skompresowane aktualne dane dla całej planety z pojedynczego dnia zajmują prawie 40 GB. Można pobierać również dane historyczne.

### 3.2 MangoMap

MangoMap jest komercyjnym narzędziem do tworzenia map dostępnych przez internet z własnych danych przestrzennych. Ceny za korzystanie z serwisu wynoszą 49-399\$ miesięcznie w zależności od liczby map i udostępnianego miejsca na serwerze do przechowywania danych. System zbudowany jest w oparciu o framework Ruby on Rails. Mapy tworzy się przy użyciu interfejsu graficznego. Stworzone mapy mogą być udostępnione na serwerze MangoMap przez unikalny link lub zamieszczone na zewnętrznej stronie WWW przez komponent HTML. [10]



## Rozdział 4

# Frameworkki internetowe w języku Ruby

W języku Ruby istnieje kilkanaście wspieranych frameworków internetowych. Wybór wykorzystanych frameworków w niniejszej pracy dokonano w następujący sposób:

1. Podzielono frameworki według daty opublikowania pierwszej wersji na 3 grupy:
  - (a) Opublikowane w latach 2004 - 2011 - frameworki o ugruntowanej pozycji.
  - (b) Opublikowane w latach 2012 - 2015 - stosunkowo nowe frameworki.
  - (c) Opublikowane w latach 2016 - 2017 - najnowsze frameworki.
2. Z każdej grupy wybrano framework z największą liczbą pobrań.

Ta metoda ma na celu wyłonienie najpopularniejszych frameworków, które powstały w różnych etapach języka Ruby, jednocześnie każdy z nich współpracuje z najnowszą wersją języka. W ten sposób wybrano *Ruby on Rails*, *Roda* i *Hanami*.

Tablica 4.1 Frameworki internetowe w języku Ruby<sup>1</sup>

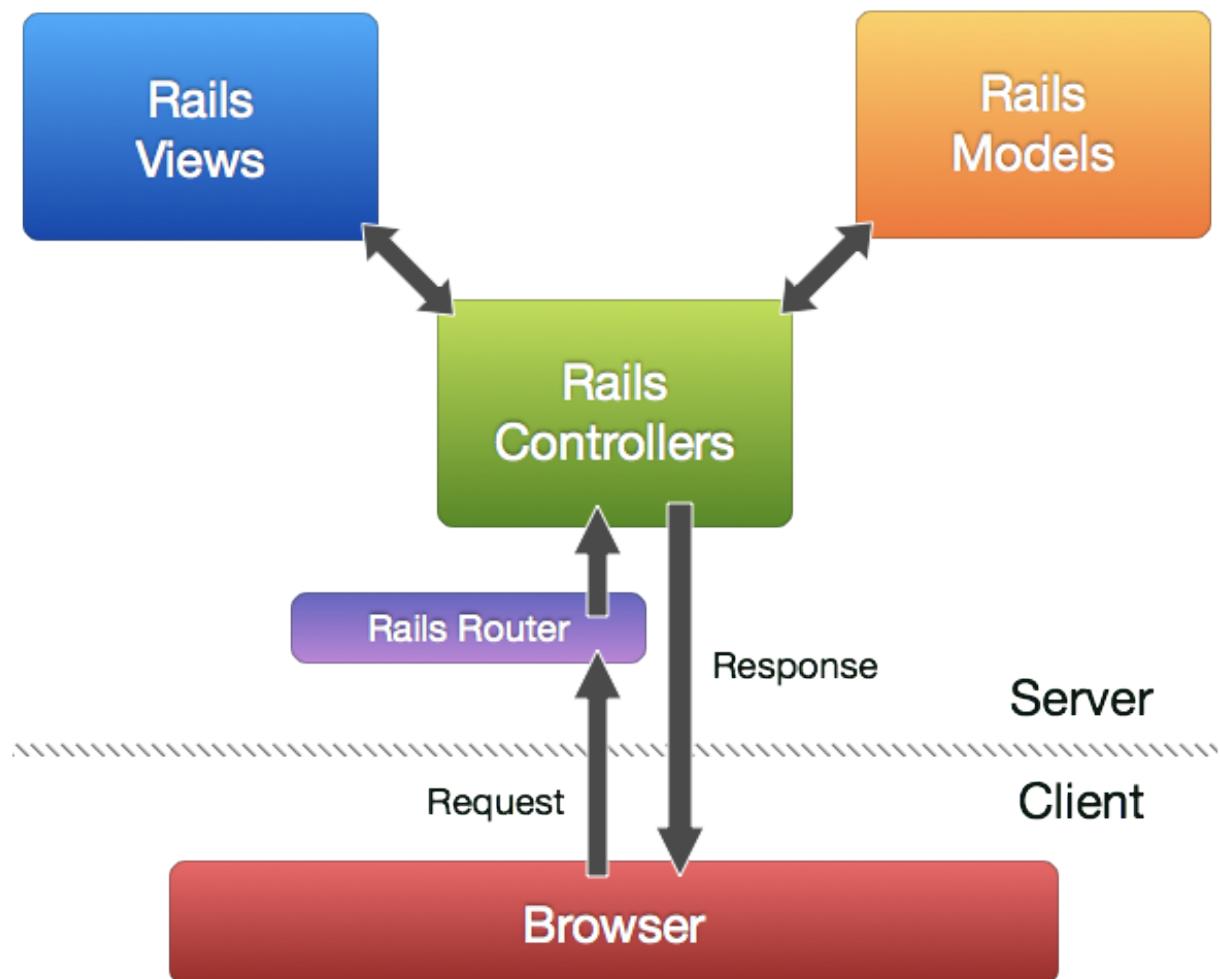
Nazwa	Data opublikowania pierwszej wersji	Data opublikowania najnowszej wersji	Liczba pobrań
Ruby on Rails	25.10.2004 r.	20.03.2017 r.	91 898 706
Hobo	29.04.2007 r.	07.05.2016 r.	211 042
Sinatra	04.10.2007 r.	19.03.2017 r.	45 207 501
Padrino	16.11.2009 r.	23.03.2017 r.	556 481
Cuba	25.04.2010 r.	01.07.2016 r.	124 371
Strelka	24.08.2011 r.	19.01.2017 r.	57 007
Pakyow	20.09.2011 r.	15.07.2016 r.	18 222
Scorched	03.03.2013 r.	12.10.2016 r.	26 929
Trailblazer	26.07.2013 r.	23.01.2017 r.	96 217
Roda	20.07.2014 r.	15.03.2017 r.	105 103
Vanilla	09.05.2015 r.	05.07.2016 r.	80 125
Hanami	20.01.2016 r.	06.04.2017 r.	28 885
Dry-web	21.04.2016 r.	02.02.2017 r.	7 190

<sup>1</sup>Zestawienia przygotowano na podstawie danych z <https://rubygems.org/> oraz <https://www.ruby-toolbox.com>. Pominięto frameworki, których ostatnia wersja ukazała się ponad rok temu. Aktualne na dzień 08.04.2017 r.

## 4.1 Ruby on Rails

Pierwsza stabilna wersja (1.0.0) frameworku Ruby on Rails ukazała się pod koniec 2005 roku, aktualna wersja (5.0.2) została opublikowana 02.03.2017 r. Aplikacja zbudowana z wykorzystaniem RoR jest oparta o wzorzec projektowy Model-Widok-Kontroler [20] (ang. *Model-View-Controller*). Aplikacja oparta na tym wzorcu jest podzielona na 3 części:

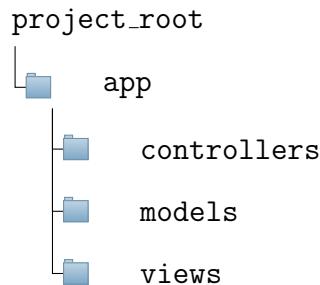
- Modele (ang. *model*) - reprezentują logikę biznesową. W tej warstwie znajdują się wszelkie obiekty, które służą do wykonywania operacji związanych z implementacją funkcjonalności aplikacji.
- Widoki (ang. *view*) - służą do prezentowania danych.
- Kontrolery (ang. *controller*) - obsługują zapytania użytkownika. Nie zawierają w sobie żadnej logiki biznesowej.



Rysunek 4.1 Architektura aplikacji w Ruby on Rails, źródło:  
<http://blog.ifuturz.com/ruby-on-rails/ruby-on-rails-mvc-learn-with-fun.html>

Dwie główne zasady frameworku [15]:

- Nie powtarzaj się (ang. *Don't Repeat Yourself*) - każda informacja powinna mieć pojedynczą, jednoznaczną i autorytatywną reprezentację w kodzie źródłowym systemu. Ułatwia to utrzymywanie kodu.
- Konwencja ponad konfiguracje (ang. *Convention Over Configuration*) - RoR posiada ustalone zasady postępowania w różnych przypadkach. Aplikacja domyślnie zachowuje się według tych zasad, nie wymagając dodatkowej konfiguracji. Pozwala to zredukować ilość kodu źródłowego.



Rysunek 4.2 Podstawowa struktura projektu Ruby on Rails

## 4.2 Roda

Twórcy frameworku Roda jest przy tworzeniu narzędzia podstawili sobie 4 cele [16]:

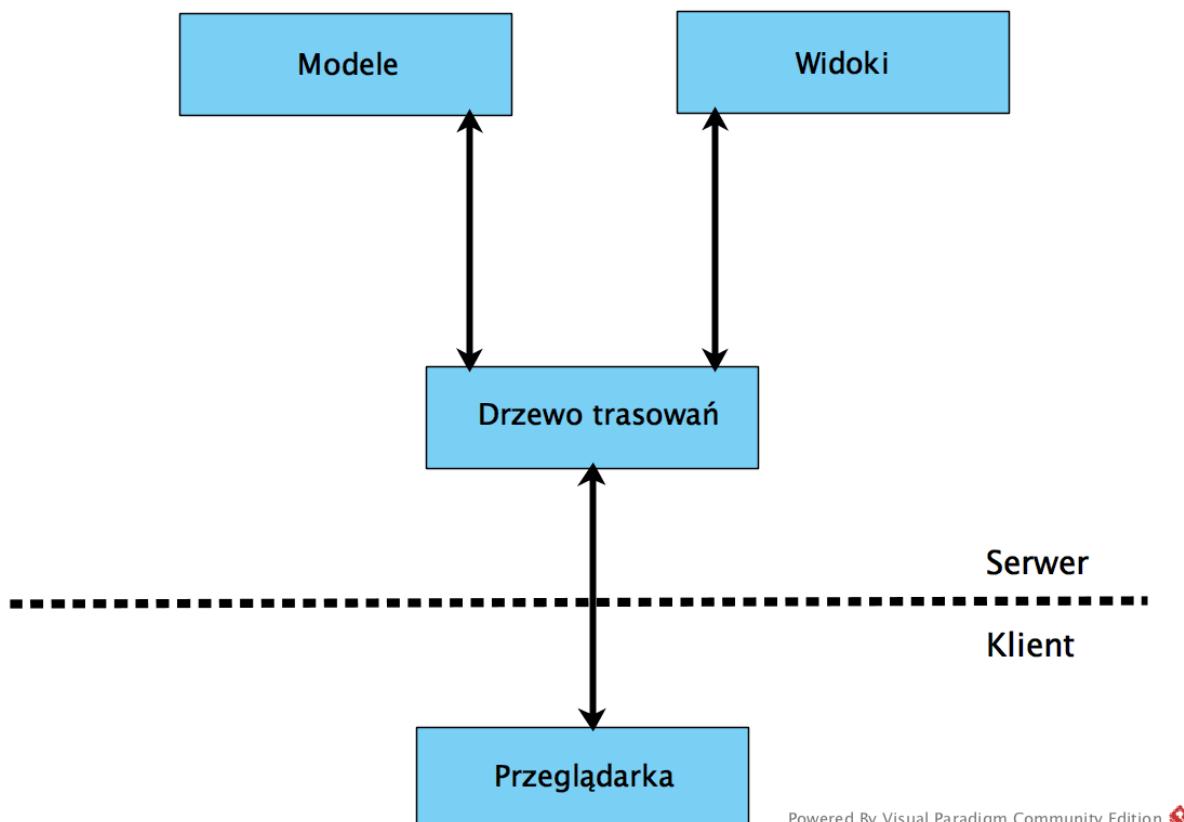
- Prostota (ang. *simplicity*) - framework ma być prosty zarówno wewnętrz (w implementacji), jak i na zewnątrz (dla użytkowników).
- Niezawodność (ang. *reliability*) - Roda wspiera i promuje projektowanie aplikacji z niemutowanym stanem. Roda ogranicza zmienne, stałe i metody przypisane do instancji obiektów aby uniknąć konfliktów z kodem zaimplementowanym przez użytkownika.
- Rozszerzalność (ang. *extensibility*) - framework zbudowany jest całkowicie w oparciu o wtyczki, aby ułatwić dodawanie funkcjonalności do frameworka. Każda część Rody może być zastąpiona własną implementacją przez użytkownika.
- Wydajność (ang. *performance*) - Roda posiada mały koszt obsługi zapytań, drzewo trasowania i inteligentną obsługę pamięci podręcznej co sprawia, że Roda jest szybsza od innych popularnych frameworków języka Ruby.

Roda opiera się o drzewo trasowania, punkty dostępu aplikacji zdefiniowane są w strukturze drzewa. Przykład drzewa trasowań znajduje się we fragmencie kodu 4.1.

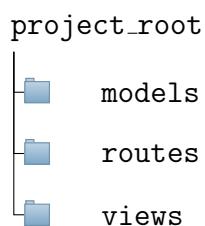
Fragment kodu 4.1 Proste drzewo trasowań

```
r.on "a" do          # /a gałąź
  r.on "b" do          # /a/b gałąź
    r.is "c" do          # /a/b/c zapytanie
      r.get do end      # GET /a/b/c zapytanie
      r.post do end     # POST /a/b/c zapytanie
    end
    r.get "d" do end   # GET /a/b/d zapytanie
    r.post "e" do end  # POST /a/b/e zapytanie
  end
end
```

W przeciwieństwie do Ruby on Rails, Roda nie posiada warstwy kontrolerów i osobnego modułu obsługującego trasowanie, w Rodzie przy definicji danego punktu końcowego znajduje się od razu kod obsługujący otrzymane zapytanie.



Rysunek 4.3 Architektura aplikacji w frameworku Roda

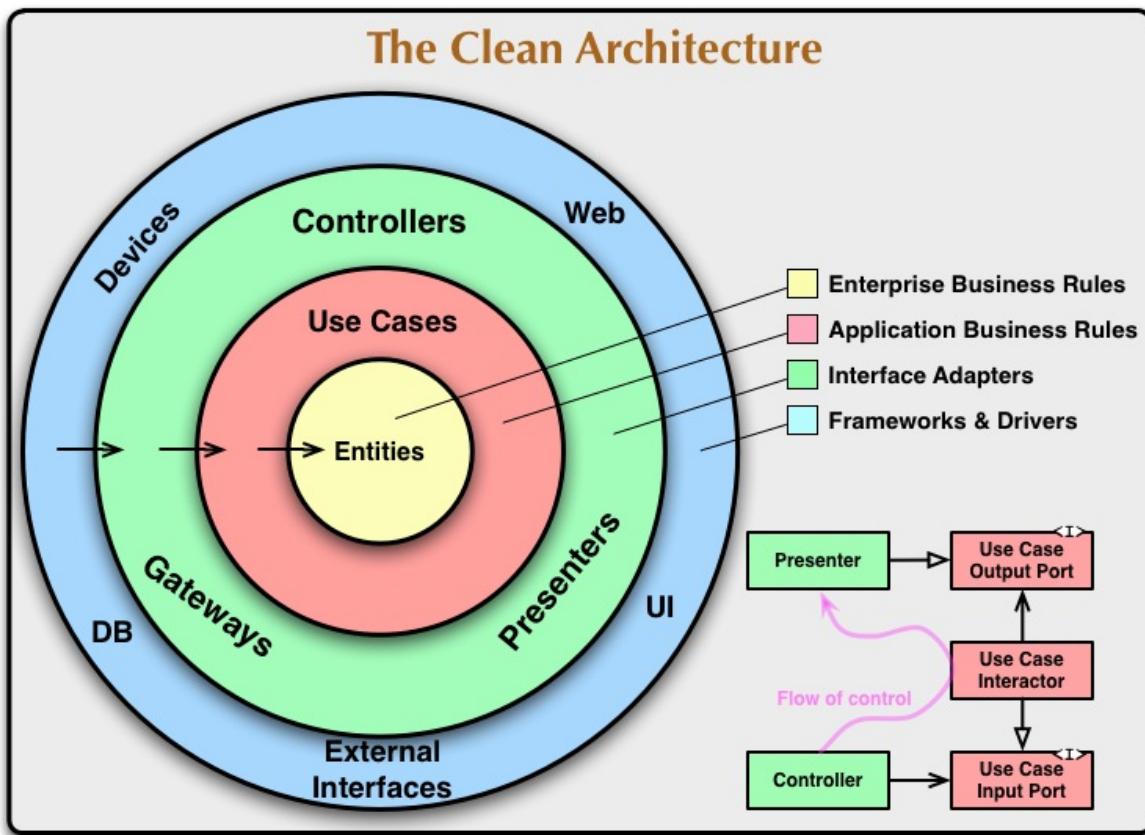


Rysunek 4.4 Struktura projektu Roda

## 4.3 Hanami

Hanami jest lekkim frameworkiem internetowym opartym o architekturę MVC, zbudowanym z wielu mikro-bibliotek. W przeciwieństwie do Ruby on Rails, Hanami nie stawia konwencji ponad konfiguracje, wszystkie informacje powinny być zawarte w napisanym przez programistę kodzie, którego zrozumienie nie wymaga znajomości konwencji frameworka. Architektura projektu jest głównie inspirowana przez dwa podejścia:

- Czysta architektura (ang. *clean architecture*) - schemat tej architektury składa się z kolejnych zawierających się w sobie kół. Każde wewnętrzne koło nie ma żadnych zależności w zewnętrznym kole, czyli w obiektach należących do wewnętrznego koła, nie ma odwołań do obiektów zewnętrznych kół. [19] To podstawowa zasada tego podejścia.



Rysunek 4.5 Schemat czystej architektury [19]

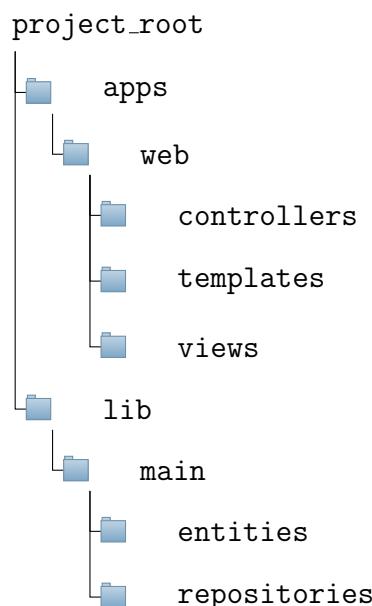
Na schemacie 4.5 można wyróżnić 4 byty:

- Encje (ang. *entities*) - zawierają ogólne reguły biznesowe systemu. Mogą być reprezentowane przez obiekty z metodami, struktury danych lub pojedyncze funkcje.
- Przypadki użycia (ang. *use cases*) - ta warstwa skupia w sobie wszystkie możliwe przypadki użycia systemu przez użytkownika. Znajduje się tu cała logika biznesowa zarządzania encjami. Zmiana w tej warstwie nie powinna wpływać na encje, interfejs użytkownika lub bazę danych.

- Adaptery interfejsów (ang. *interface adapters*) - na tym poziomie dane są konwertowane z formatu najbardziej dogodnego dla przypadków użycia i encji do formatu przyjmowanego przez zewnętrzne narzędzia takie jak baza danych lub przeglądarka internetowa.
  - Frameworki oraz narzędzia (ang. *frameworks and drivers*) - w tej warstwie znajdują się zewnętrzne narzędzia takie jak inne frameworki, baza danych lub przeglądarka internetowa.
- Najpierw monolit (ang. *monolith first*) - według tej zasady budowę systemuaczyna się od monolitycznej aplikacji. Jednak budowa aplikacji od samego początku powinna być jak najbardziej modularna aby można było ją później rozbić na wiele mniejszych aplikacji.

Zgodnie z zasadą *czystej architektury* w utworzonym projekcie systemu z użyciem Hanami oddzielona jest warstwa logiki biznesowej znajdująca się w folderze *lib* projektu od mechanizmu komunikacji z innymi serwisami zawartego w folderze *apps*.

Hanami posiada kontener aplikacji, który pozwala nam utworzyć wiele aplikacji w ramach jednego projektu, które korzystają z tego samego zbioru encji i przypadków użycia, a następnie uruchomić je w jednym procesie Rubiego.



Rysunek 4.6 Podstawowa struktura projektu Hanami

# Rozdział 5

## Narzędzia dostępne do przetwarzania danych geograficznych w języku Ruby

### 5.1 Przechowywanie danych

Omawiane frameworki domyślnie wykorzystują relacyjną bazę danych do przechowywania informacji. Najpopularniejsze bazy danych to PostgreSQL, MySQL, SQLite.

#### 5.1.1 PostgreSQL

PostgreSQL posiada kilka wbudowanych typów przestrzennych [13]:

- Point - reprezentuje punkt na mapie, to podstawowy typ, który służy do budowania pozostałych typów przestrzennych. Dane zapisane są jako para współrzędnych w postaci tekstuowej  $(x, y)$ , gdzie  $x$  i  $y$  to liczby zmienno-przecinkowe.
- Line - prosta w znaczeniu geometrycznym, mogą być reprezentowane przez równanie liniowe  $Ax + By + C = 0$ , wtedy w bazie danych zapisane są współczynniki  $A$ ,  $B$ ,  $C$ . Innym sposobem reprezentacji tego typu są dwa punkty, przez które przechodzi prosta  $\left[ (x_1, y_1), (x_2, y_2) \right]$ .
- Line Segment - w znaczeniu geometrycznym to odcinek, reprezentowany przez dwa punkty, początek i koniec odcinka  $\left[ (x_1, y_1), (x_2, y_2) \right]$ .
- Box - to czworokąt zapisane jako dwa przeciwwległe wierzchołki  $\left( (x_1, y_1), (x_2, y_2) \right)$ .
- Path - lista połączonych ze sobą punktów, ścieżka. Ścieżka może być otwarta, jeśli pierwszy i ostatni punkt nie są ze sobą połączone, lub zamknięta, jeśli pierwszy i ostatni punkt są ze sobą połączone. Nawiązy kwadratowe reprezentują otwartą ścieżkę  $\left[ (x_1, y_1), \dots, (x_n, y_n) \right]$ , zamknięta ścieżka jest przedstawiona za pomocą okrągłych nawiasów  $\left( (x_1, y_1), \dots, (x_n, y_n) \right)$ .
- Polygon - to wielokąt zapisany za pomocą listy punktów, reprezentujących kolejne wierzchołki wielokąta  $\left( (x_1, y_1), \dots, (x_n, y_n) \right)$ .
- Circle - okrąg zapisany przy pomocy środka i promienia  $\langle (x, y), r \rangle$ .

PostgreSQL dostarcza niewiele funkcji do wyszukiwania danych. Sprawdzenie zależności między dwoma punktami takich jak np. przecięcie dwóch obiektów lub zawieranie się jednego obiektu w drugim nie jest dostępne dla typów *Path* i *Polygon*.

### **5.1.2 PostGIS**

PostGIS jest biblioteką rozszerzającą możliwości PostgreSQL w zakresie przetwarzania danych przestrzennych. Biblioteka jest wspierana przez fundacje *Open Source Geospatial*. [14] Dane są zapisane w formie binarnej jako współrzędne geograficzne lub geometryczne w zależności od wyboru. PostGIS zapewnia wsparcie typów zdefiniowanych przez *Open Geospatial Consortium*:

- POINT - pojedynczy punkt na mapie.
- LINESTRING - zbiór połączonych ze sobą punktów reprezentujący ścieżkę.
- POLYGON - zbiór połączonych ze sobą punktów reprezentujący wierzchołki wielokąta.

*Open Geospatial Consortium* definiuje również kolekcje obiektów powyższych typów:

- MULTIPOINT - kolekcja punktów.
- MULTILINESTRING - kolekcja linii.
- MULTIPOLYGON - kolekcja wielokątów.
- GEOMETRYCOLLECTION - kolekcja dowolnych obiektów geometrycznych.

Dla wszystkich typów dostępny jest taki sam zestaw funkcji, które np. sprawdzają zależności między dwoma obiektami, co jest pomocne przy wyszukiwaniu obiektów.

### **5.1.3 MySQL Spatial**

MySQL Spatial jest rozszerzeniem dla bazy MySQL, które dostarcza wsparcie dla danych przestrzennych. Rozszerzenie dostarcza typy danych zdefiniowane przez grupę *Open Geospatial Consortium*. [11] Zbiór funkcjonalności pokrywa się z funkcjonalnościami rozszerzenia Postgis.

### **5.1.4 SpatiaLite**

SpatiaLite jest rozszerzeniem dla bazy danych SQLite, które zawiera wsparcie dla typów przestrzennych zdefiniowanych przez grupę *Open Geospatial Consortium*[17]: *Point*, *Line*, *Multiline*, *Polygon*, *Multipolygon*. Cała baza danych zapisana jest w pojedynczym pliku. Biblioteka udostępniona jest na zasadzie otwartego źródła.

## **5.2 Obiektywne przetwarzanie danych**

### **5.2.1 RGeo**

RGeo wspiera typy danych zdefiniowane przez *Open Geospatial Consortium*, takie jak punkt, linia i wielokąt. Dane są reprezentowane w formie obiektowej. Biblioteka pozwala

na podstawowe operacje analizowania danych przestrzennych, takie jak szukanie punktów przecięcia i obliczenia pola powierzchni. Biblioteka operuje na reprezentacji geograficznej i geometrycznej danych i pozwala konwertować dane pomiędzy dostępymi reprezentacjami. RGeo udostępnia udostępnia jedynie interfejs w języku Ruby dla programistów, a do obsługi danych geograficznych korzysta z bibliotek GEOS i proj.4, te biblioteki są napisane w języku C++. [5]

### 5.2.2 GeoRuby

GeoRuby jest biblioteką całkowicie napisaną za pomocą języka Ruby do przetwarzania danych przestrzennych. Biblioteka ściśle podąża za modelem danych definiowanych przez *Open Geospatial Consortium*, dlatego dobrze łączy się z takimi bazami danych jak Postgis, MySQL Spatial, SpatiaLite i innymi, które również wspierają typy opracowane przez OGC. [2]

## 5.3 Prezentowanie danych

### 5.3.1 Leaflet

Leaflet jest biblioteką napisaną w języku JavaScript, która pozwala dodać interaktywną mapę do widoku aplikacji internetowej. Biblioteka nie dostarcza graficznej reprezentacji mapy, ale zapewnia obsługę mapy z zewnętrznego serwisu np. OpenStreetMap. Do mapy można dodawać interaktywne obiekty. Biblioteka dostępna jest na zasadzie otwartego źródła. [4]

### 5.3.2 GoogleMaps JavaScript API

GoogleMaps JavaScript API pozwala dołączyć mapę z serwisu Google Maps. Mapa jest interaktywna i pozwala wyświetlać obiekty dodane przez programistę. W darmowej wersji GoogleMaps API pozwala wczytać mapę 25 000 razy w ciągu dnia. [3]



# Rozdział 6

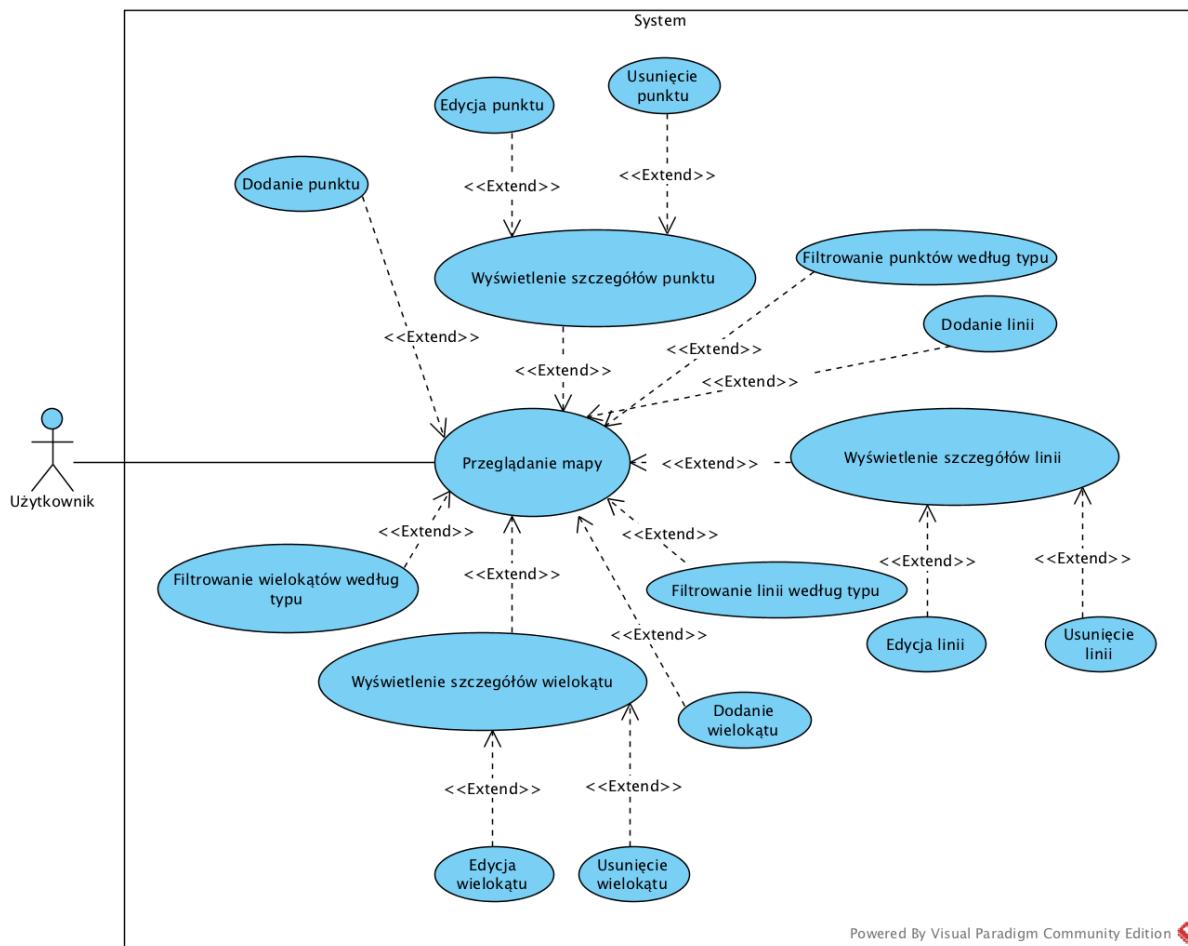
## Aplikacja zaimplementowana na potrzeby badań

### 6.1 Opis aplikacji

W celu wykonania analizy porównawczej w każdym z wybranych frameworków została zaimplementowana aplikacja internetowa operująca na danych geograficznych posiadająca takie same funkcjonalności. Aplikacje zostały zrealizowane zgodnie z architekturą i konwencją wykorzystywaną w danym frameworku zawartych w dokumentacji narzędzia. W projektach wykorzystano jedynie biblioteki niezbędne do zrealizowania wymaganych funkcjonalności. Nie używano bibliotek modyfikujących architekturę aplikacji lub funkcjonalności frameworka. Aplikacja ma za zadanie wykonywać 4 podstawowe operacje na danych geograficznych:

- tworzenie,
- odczytywanie,
- aktualizowanie,
- usuwanie.

Wszystkie funkcjonalności wykonanej aplikacji zostały zaprezentowane na rysunku 6.1.



Rysunek 6.1 Diagram przypadków użycia aplikacji

Aby przeprowadzić badania w realistycznych warunkach aplikacja korzysta z gotowych zbiorów danych:

- Punkty - pochodzące ze zbioru „Państwowy rejestr nazw geograficznych - miejscowości”, zbiór zawiera 256796 obiektów, dostępny jest pod adresem <http://www.codgik.gov.pl/index.php/darmowe-dane/prng.html>.
- Linie - dane uzyskano z serwisu <http://download.geofabrik.de/europe/poland.html> pozwalającego pobrać drogi z terenu Polski z bazy danych OpenStreetMap. Zbiór obejmuje 2519660 obiektów.
- Wielokąty - obiekty pobrano z Państwowego Rejestru Granic dostępnego pod adresem <http://gis-support.pl/baza-wiedzy/dane-do-pobrania/>, zbiór liczy 59978 obiektów.

Każda wersja aplikacji posiada taką samą bazę danych, której schemat znajduje się na rysunku 6.2. Jako bazę danych wykorzystano PostgreSQL z rozszerzeniem PostGIS. Dane geograficzne w warstwie aplikacji przetwarzano używając biblioteki RGeo.

points	
id	SERIAL
coordinates	geometry
name	CHARACTER VARYING
object_type	CHARACTER VARYING
object_class	CHARACTER VARYING
voivodeship	CHARACTER VARYING
county	CHARACTER VARYING
commune	CHARACTER VARYING
terc	CHARACTER VARYING
created_at	TIMESTAMP(6) WITHOUT TIME ZONE
updated_at	TIMESTAMP(6) WITHOUT TIME ZONE

polygons	
id	SERIAL
name	CHARACTER VARYING
coordinates	geometry
terc	CHARACTER VARYING
unit_type	INTEGER
created_at	TIMESTAMP(6) WITHOUT TIME ZONE
updated_at	TIMESTAMP(6) WITHOUT TIME ZONE

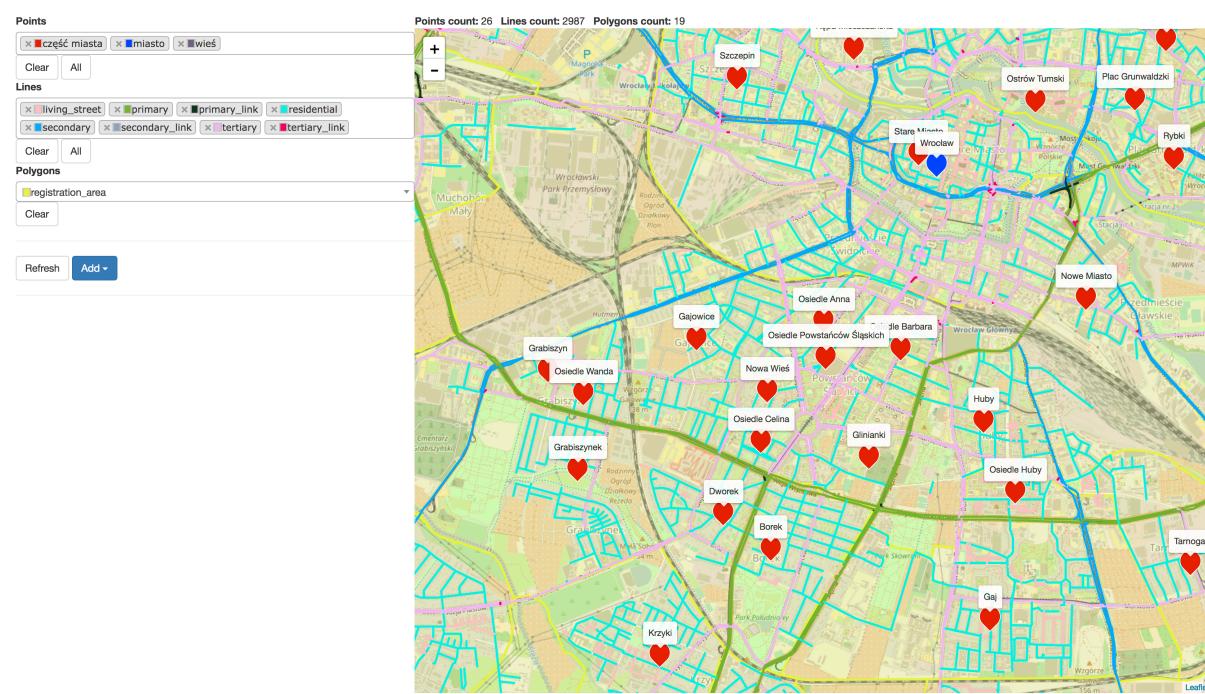
lines	
id	SERIAL
name	CHARACTER VARYING
coordinates	geometry
road_type	CHARACTER VARYING
created_at	TIMESTAMP(6) WITHOUT TIME ZONE
updated_at	TIMESTAMP(6) WITHOUT TIME ZONE

Rysunek 6.2 Schemat ERD bazy danych

## 6.2 Implementacja

## 6.3 Prezentacja aplikacji

Do prezentacji danych na mapie wykorzystano bibliotekę Leflet, ponieważ jest darmowa i ma otwarte źródła. Mapa pochodzi z serwisu OpenStreetMap. Widok gotowej aplikacji znajduje się na rysunku 6.3.



Rysunek 6.3 Widok gotowej aplikacji

# Rozdział 7

## Badania

### 7.1 Plan badań

Na podstawie dostępnej dokumentacji oraz zaimplementowanych aplikacji w każdym z badanych narzędzi, frameworki zostaną porównane w następujących aspektach:

- dostępne funkcjonalności,
- struktura utworzonego projektu,
- wydajność zaimplementowanej aplikacji.

#### 7.1.1 Środowisko badawcze

Do przeprowadzenia badań użyto dwóch komputerów:

##### 1. Komputer - serwer

- MacBook Pro (Retina, 13 cali, model: Early 2015)
- Procesor: Intel Core i5 2,7 GHz 2 rdzenie
- Pamięć RAM: 8GB DDR3 1867 MHz
- Dysk: 120GB APPLE SSD SM0128G

##### 2. Komputer - tester

- MacBook Mini 6.1
- Procesor: Intel Core i5 2,5 GHz 2 rdzenie
- Pamięć RAM: 8GB DDR3 1600 MHz
- Dysk: 240GB GOODRAM SSD CX100

Na *komputerze - serwer* uruchomiona była aktualnie testowana aplikacja. *Komputer - tester* wykonywał zaimplementowane badania. Podczas badań komputery były podłączone ze sobą lokalnie przy pomocy sieci Wi-Fi 802.11n i nie miały dostępu do zewnętrznej sieci. Na komputerach uruchomione były tylko niezbędne aplikacje do przeprowadzenia badań.

### 7.1.2 Narzędzia wykorzystane podczas badań

Do zmierzenia wydajności zaimplementowanych aplikacji zastosowano następujące narzędzia:

- Capybara - biblioteka w języku Ruby pozwalająca symulować interakcje prawdziwego użytkownika z wybraną aplikacją internetową. Capybara dostarcza szereg metod pozwalających zapisać jako scenariusz kolejne kroki postępowania użytkownika m.in. kliknięcie wybranego elementu lub wprowadzenie danych do formularza. Do wspierania JavaScriptu, Capybara używa Selenium. Selenium jest narzędziem, które umożliwia automatyczne wykonanie testów akceptacyjnych aplikacji internetowej w wybranej przeglądarce. W przeprowadzonych badaniach testy uruchomione zostały w przeglądarce Firefox. [1]
- JMeter - aplikacja napisana w języku Java dostępna na licencji otwartego źródła. JMeter służy do wykonywania testów wydajności aplikacji internetowych. W niniejszych badaniach JMeter wykorzystywał protokół HTTP do wysyłania określonych zapytań do aplikacji. W badaniach pomocna również okazała się funkcja symulowania wielu użytkowników - JMeter uruchamia wybrany scenariusz na wielu wątkach aby sprawdzić jak zachowa się aplikacja przy wielu równoległych zapytaniach. [9]
- Ruby Benchmark - biblioteka pozwalająca na zmierzenie czasu wykonywania podanego blogu kodu. Ruby Benchmark został wykorzystany przy zmierzeniu czasu wykonywania scenariuszy utworzonych przy pomocy Capybary oraz czasu wykonywania operacji na bazie danych. [6]

## 7.2 Porównanie funkcjonalności wybranych frameworków

### 7.2.1 Przechowywanie danych

Ruby on Rails posiada największe wsparcie w przechowywaniu danych geograficznych z rozpatrywanych narzędzi. Biblioteka ActiveRecord domyślnie wykorzystywana w Ruby on Rails do mapowania obiektowo-relacyjnego posiada adaptery dla następujących baz danych: PostgreSQL z dodatkiem PostGIS, MysqISpatial, SpatiaLite. Za pomocą adapterów, w migracjach można używać typów kolumn z przestrzennych baz danych. Przy komunikacji typy przestrzenne z bazy danych mapowane są na typy zaimplementowane w bibliotece RGeo. Konfiguracja nie wymaga wiele pracy, wystarczy dodać odpowiednią bibliotekę do projektu i w konfiguracji bazy danych ustawić odpowiedni adapter. Od tego momentu można w migracjach przy definiowaniu kolumn używać wszystkich typów przestrzennych dostarczanych przez wybraną bazę danych. Dane przestrzenne podczas wykonywanych akcji między aplikacją, a bazą danych są automatycznie mapowane.

Tworzenie bazy danych przestrzennej za pomocą Rody nie stanowi problemu. Framework pozwala używać kodu SQL w migracjach dlatego można włączyć dowolne rozszerzenie wybranej bazy danych i używać dodatkowych typów. Roda nie zapewnia specjalnego wsparcia przy mapowaniu danych przestrzennych na obiekty, dane są zwracane w formie tekstowej binarnej wartości pola.

Hanami w migracjach pozwala ozywać kodu napisanego w języku SQL do np. definiowania typów lub włączania dodatków w bazie danych, dlatego bez problemów można stworzyć i wersjonować bazę danych z typami przestrzennymi. Jednak framework, podobnie

jak Ruby on Rails, posiada mechanizm automatycznego mapowania typów bazy danych do typów obiektów, który nie pozwala na definiowanie własnych typów lub własnych reguł mapowania. To ograniczenie powoduje otrzymanie wyjątku przy próbie uruchomienia aplikacji zawierającej nieznany typ. Framework wspiera jedynie typy przestrzenne zdefiniowane w bazie danych PostgreSQL, które są zapisywane postaci tekstowej zawierającej współrzędne. Możliwe jest zapisanie wszystkich 3 podstawowych typów przestrzennych: punktu, linii, wielokąta. PostgreSQL dostarcza podstawowe operacje wyszukiwania danych przestrzennych tylko dla punktów, dlatego zastosowanie PostgreSQL bez dodatku PostGIS mocno ogranicza możliwości budowanej aplikacji. Dodanie obslugi typów geometrycznych dostarczonych przez dodatek PostGIS nie wymaga dużej ilości kodu, wymagana jest dobra znajomość języka Ruby i mechanizmu przeddefiniowania istniejących już klas tzw. „monkey patching”. Najłatwiejsze rozwiązanie pozwalające używać typów geometrycznych z biblioteki PostGIS znajduje się na listingu 7.1. Przy takim rozwiązaniu wartości z kolumny geometrycznej mapowane są na typ String z wartością binarną.

Fragment kodu 7.1 Obsługa typów geometrycznych z biblioteki PostGIS przez Hanami

```
require 'rom/sql/extensions/postgres/inferencer'
require 'rom/sql/extensions/postgres/types'

ROM::SQL::Schema::PostgresInferencer.class_eval do
  alias map_db_type_original map_db_type

  def map_db_type(db_type)
    # najpierw następuje próba użycia oryginalnej metody mapowania typów
    # jeśli zakończy się ona nie powodzeniem, to następuje sprawdzenie
    # czy typ kolumny z bazy danych rozpoczyna się od słowa 'geometry',
    # wtedy dane są rzutowane do typu ROM::Types::String
    map_db_type_original(db_type) || (db_type.start_with?('geometry') ? ROM
      ::Types::String : nil)
  end
end
```

## 7.2.2 Obiektowe przetwarzanie danych

Do obiektowego przetwarzania danych dostępne są dwie biblioteki RGeo i GeoRuby. Ruby on Rails, dzięki wspomnianym wcześniej adapterom, dużo wygodniej jest korzystać z RGeo dzięki automatycznemu mapowaniu danych relacyjnych na obiekty. Frameworki Roda oraz Hanami w żaden sposób nie rozszerzają możliwości bibliotek RGeo i GeoRuby. W Hanami, zaimplementowana metoda mapująca wartości przestrzenne na typ obiektowy zwraca wartość binarną, Roda domyślne zwraca dane przestrzenne w takiej formie. Wartość binarną dalej można zamienić na typ z biblioteki RGeo, co zaprezentowano na listingu 7.2

Fragment kodu 7.2 Tworzenie obiektu RGeo z zapisu binarnego danych przestrzennych

```
module CoordinatesHelper
  FACTORY = RGeo::Geographic.spherical_factory(:srid => 4326)

  # utworzenie obiektu RGeo z binarnego zapisu danych przestrzennych
  def coordinates_object
    RGeo::WKRep::WKBParser.new(FACTORY, support_ewkb: true,
      default_srid: 4326).parse(coordinates)
  end

  # rzutowanie obiektu przestrzennego do postaci "well known text"
  def coordinates_text
    coordinates_object.as_text
  end
end
```

## 7.2.3 Prezentowanie danych

Framework Ruby on Rails posiada biblioteki do integracji obu narzędzi do prezentacji danych geograficznych wspomnianych w rozdziale 5.3:

1. Leaflet-rails - dołącza do projektu Ruby on Rails bibliotekę Leaflet napisaną w języku JavaScript wraz ze stylami CSS oraz dodaje metody do widoków pozwalające na dołączenie mapy i podstawową konfigurację bez potrzeby używania JavaScriptu.

2. Google Maps for Rails - biblioteka opakowuje JavaScriptową bibliotekę GoogleMaps JavaScript API, ułatwiając jej dodanie do projektu.

Dzięki takiej integracji zgodnie z sposobem dołączania bibliotek do projektu w języku Ruby [8], wspomniane biblioteki są przechowywane centralnie w systemie zamiast w projekcie. Roda oraz Hanami w przeciwieństwie do Ruby on Rails nie posiadają bibliotek wspomagających dołączenie do projektu biblioteki Leaflet lub Google Maps JavaScript API. Żeby skorzystać jednej z wymienionych bibliotek należy dołączyć jej kod do projektu.

Przy odczycie danych w formacie JSON, Ruby on Rails oraz Roda automatycznie serializują model do postaci JSON. W projekcie wykorzystującym Hanami, niezbędne jest dodanie zewnętrznej biblioteki realizującej serializowanie danych np. *Roar* i zdefiniowanie odpowiedniego prezentera. Przykładowy prezenter został zaprezentowany na listingu 7.3

Fragment kodu 7.3 Prezenter dla modelu Point

```
require 'roar/decorator'
require 'roar/json'

module Web::Representers
  class Point < Roar::Decorator
    include Roar::JSON

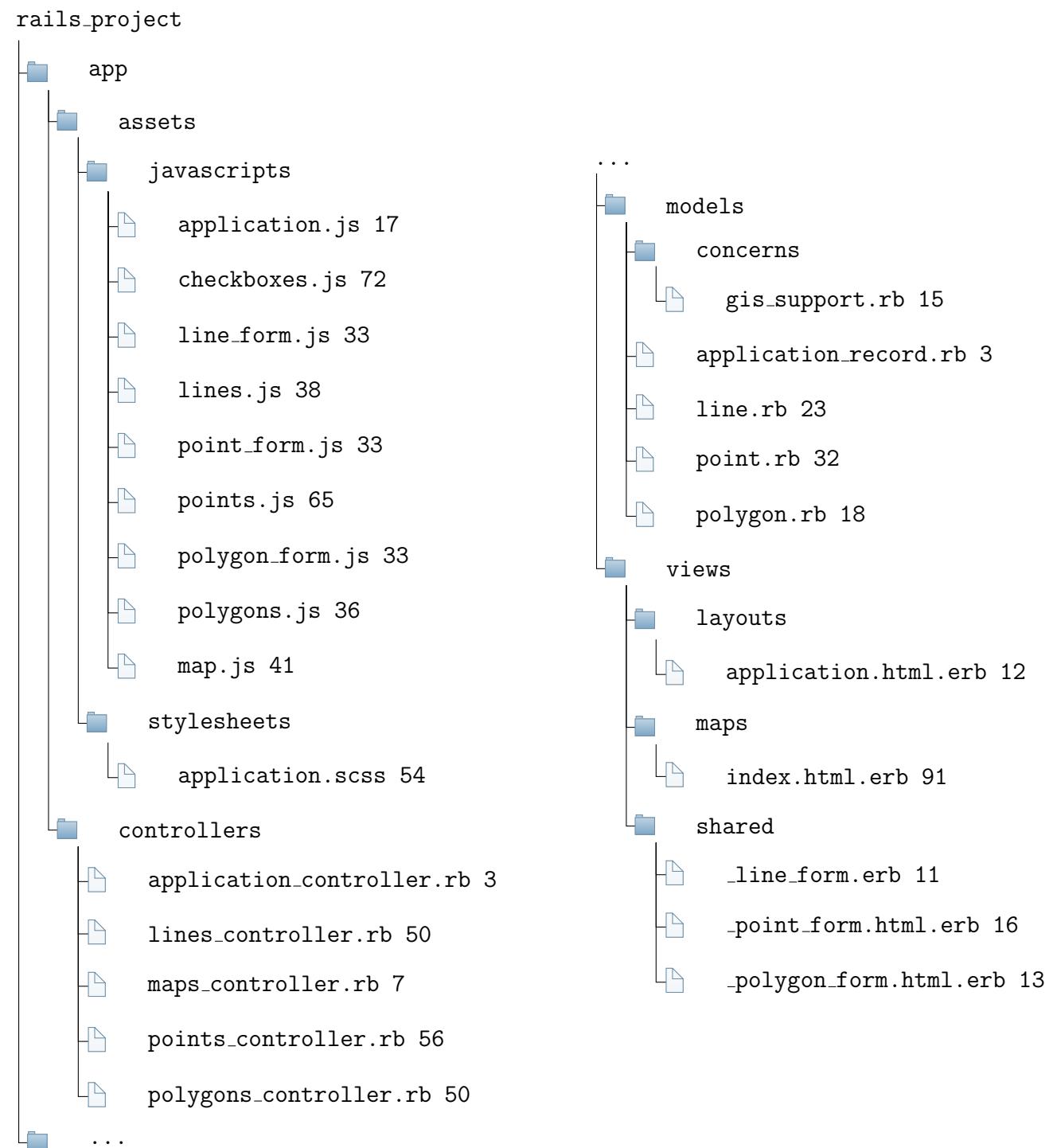
    property :id
    property :name
    property :object_type
    property :object_class
    property :voivodeship
    property :county
    property :commune
    property :terc
    property :coordinates_text
    property :color
  end
end
```

#### 7.2.4 Podsumowanie

Ruby on Rails posiada zdecydowanie najwięcej gotowych narzędzi do pracy z danymi geograficznymi z rozpatrywanych frameworków. Tworzenie aplikacji dzięki temu trwa mniej czasu. Roda mimo mniejszego wsparcia dla danych przestrzennych w przeciwieństwie do Hanami pozwala na ich używanie bez rozszerzania funkcjonalności frameworka. Hanami pozwala zbudować prosty system GIS wykorzystujący podstawowe typy przestrzenne dostępne w bazie danych PostgreSQL, ale implementacja systemu informacji geograficznej wykorzystującego inną bazę danych za pomocą frameworku Hanami wymaga od programisty znacznie więcej pracy i większych umiejętności programistycznych aby rozszerzyć możliwości frameworka.

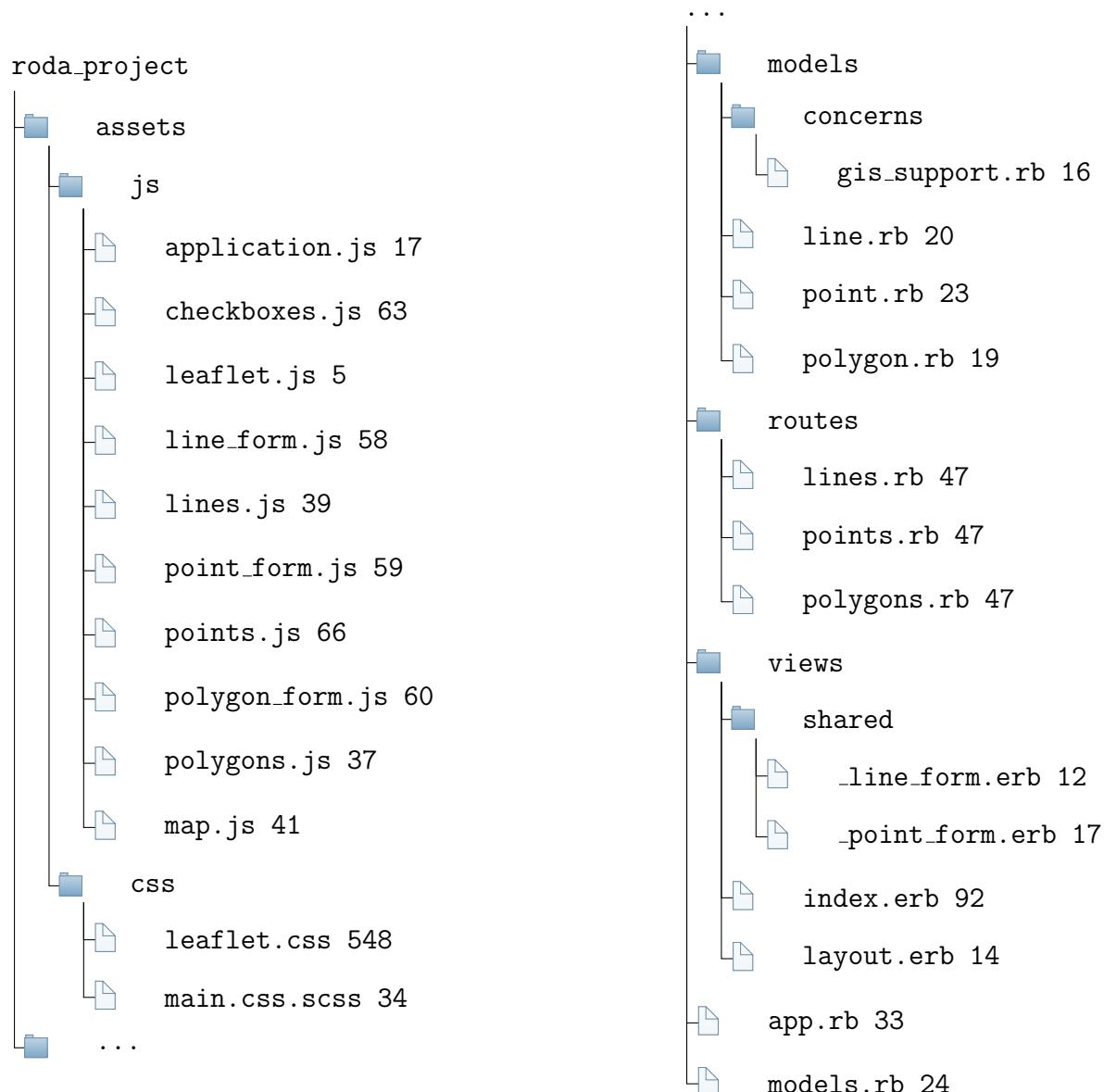
### 7.3 Porównianie struktur wykonanych projektów

Struktura zaimplementowanego projektu przy pomocy Ruby on Rails bezpośrednio odzwierciedla architekturę frameworku. W projekcie mamy jasny podział na Model-Widok-Kontroler.



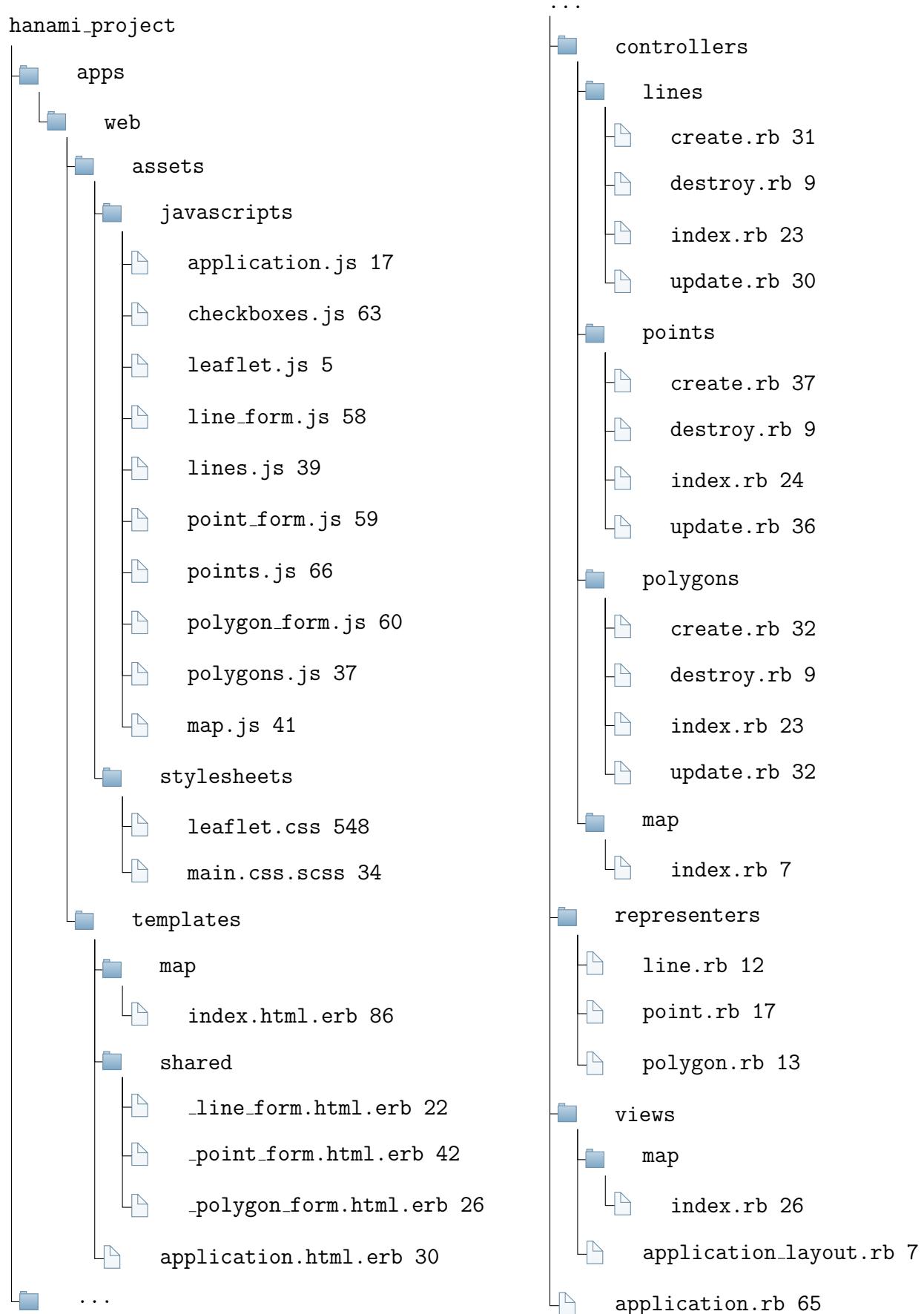
Rysunek 7.1 Struktura projektu Ruby on Rails

Filozofia prostoty w frameworku Roda ma swoje odzwierciedlenie w strukturze projektu, która nie posiada za wiele rozgałęzień.

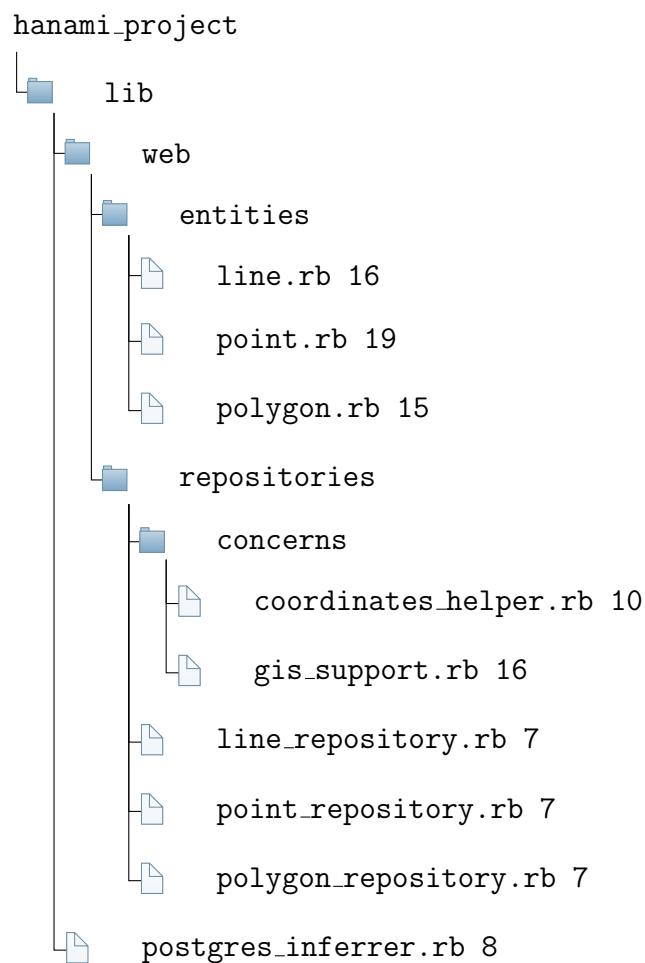


Rysunek 7.2 Struktura projektu Roda

Dodatkowe warstwy abstrakcji w architekturze projektu Hanami przekładają się na bardziej rozbudowaną strukturę projektu.

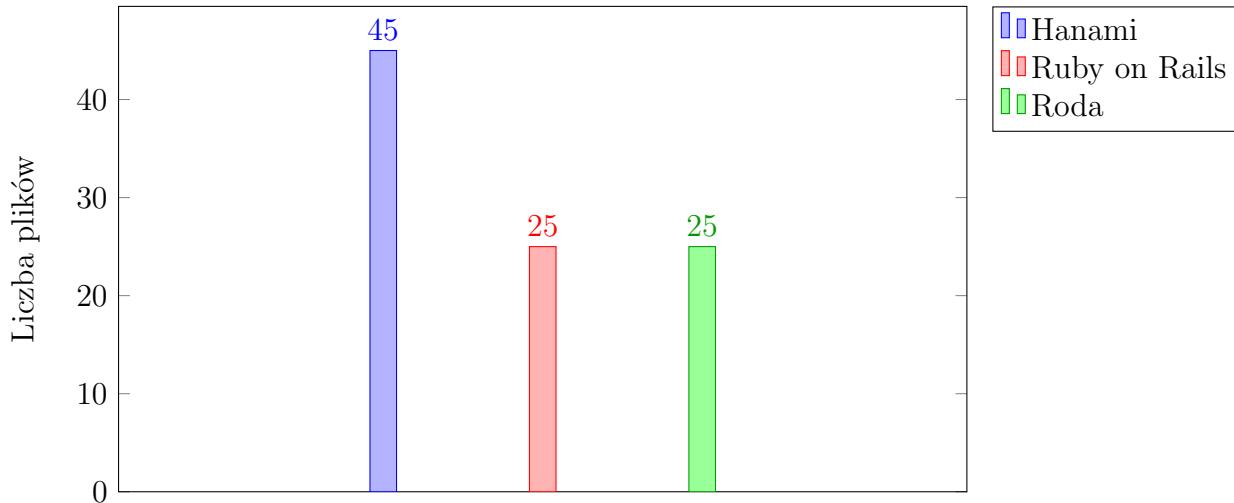


Rysunek 7.3 Struktura projektu Hanami - część I

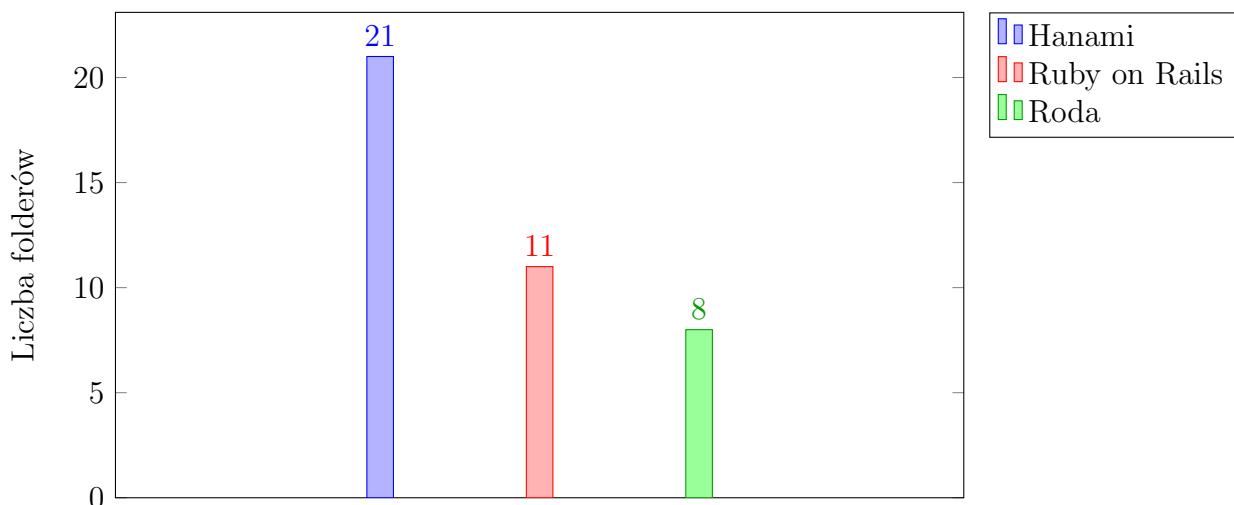


Rysunek 7.4 Struktura projektu Hanami - część II

Na rysunkach 7.5 i 7.6 można zauważyc, że projekt stworzony przy pomocy Hanami zawiera znacznie więcej plików i folderów od dwóch pozostałych projektów. Dodatkowe warstwy abstrakcji w architekturze frameworka przekładają się na bardziej skomplikowaną strukturę projektu. Projekty zrealizowane przy pomocy Ruby on Rails i Roda zawierają dokładnie tyle samo plików, ale Roda ma nieco mniej folderów. Na rysunku 7.2 można zauważyc, że struktura projektu Roda jest bardziej płaska niż struktura Ruby on Rails zaprezentowana na rysunku 7.1.

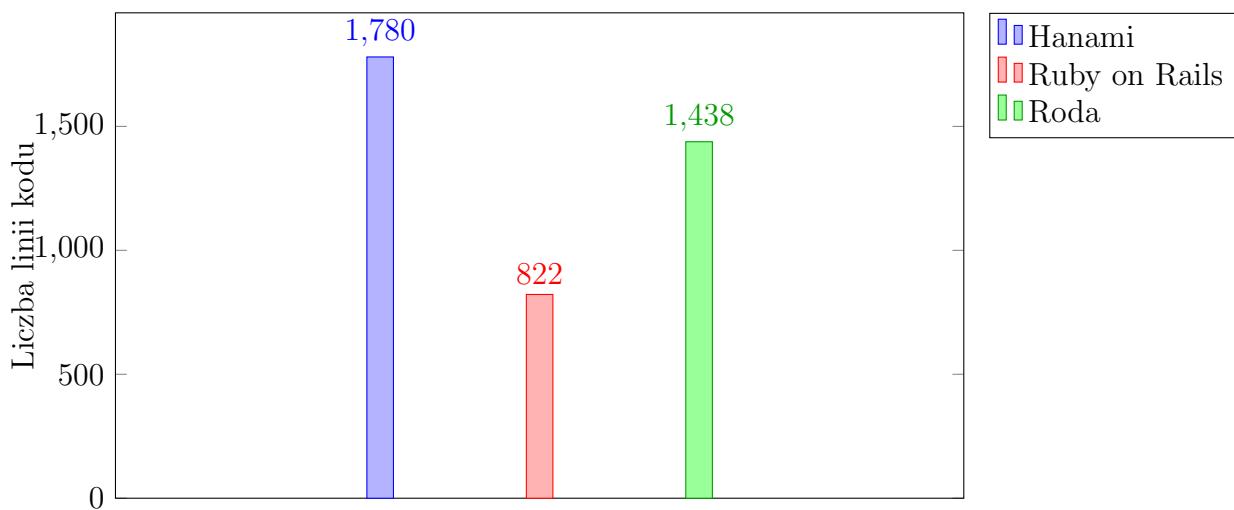


Rysunek 7.5 Liczba plików w projekcie w zależności od użytego frameworku



Rysunek 7.6 Liczba folderów w projekcie w zależności od użytego frameworku

Na poniższym wykresie można zauważyc, że projekt Ruby on Rails zawiera najmniej linii kodu. Framework posiada wiele gotowych, wbudowanych funkcjonalności co ułatwia pracę programistę. Projekt Hanami zawiera najwięcej linii kodu co spowodowane jest skomplikowaną architekturą aplikacji, każda kolejna warstwa abstrakcji narzuca na programistę obowiązek napisania dodatkowego kodu realizującego komunikacje między warstwami.



Rysunek 7.7 Liczba linii kodu w projekcie w zależności od użytego frameworku

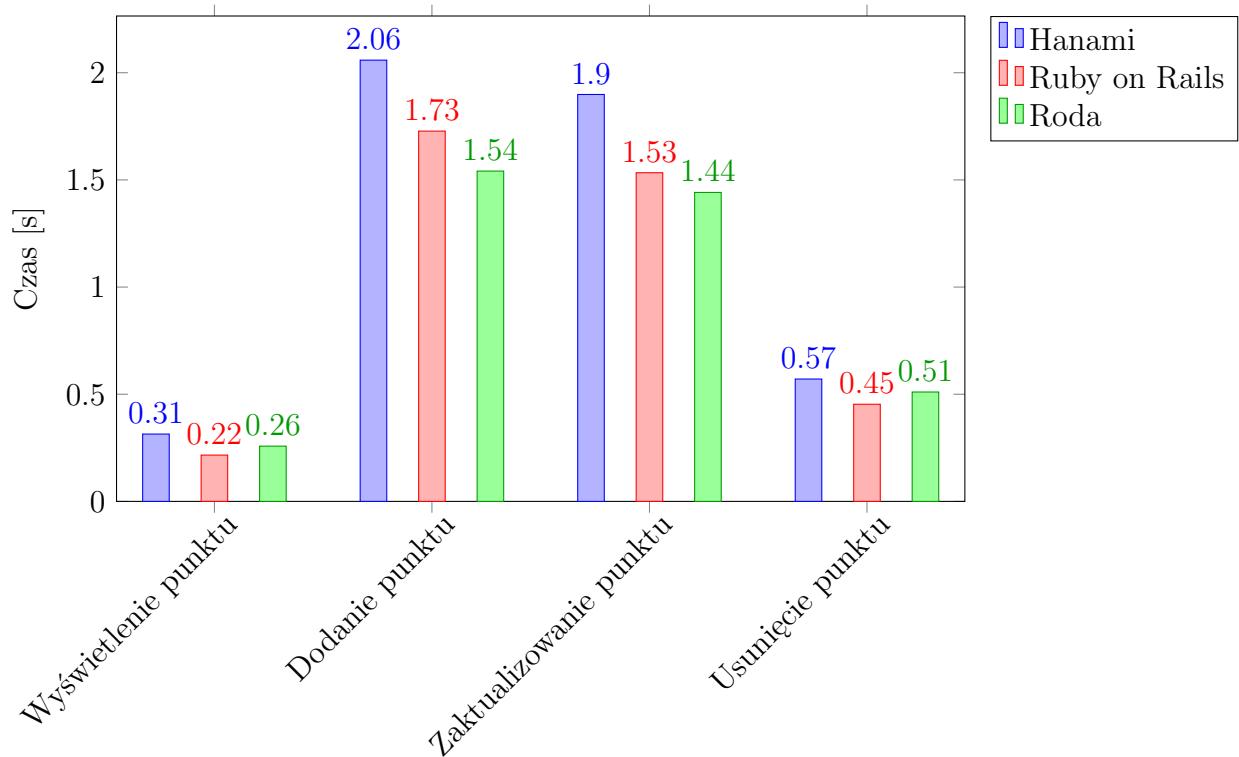
## 7.4 Porównanie wydajności zaimplementowanych aplikacji

Wszystkie wyniki badań wydajnościowych zostały uśrednione po 30 przebiegach.

### 7.4.1 Interfejs użytkownika

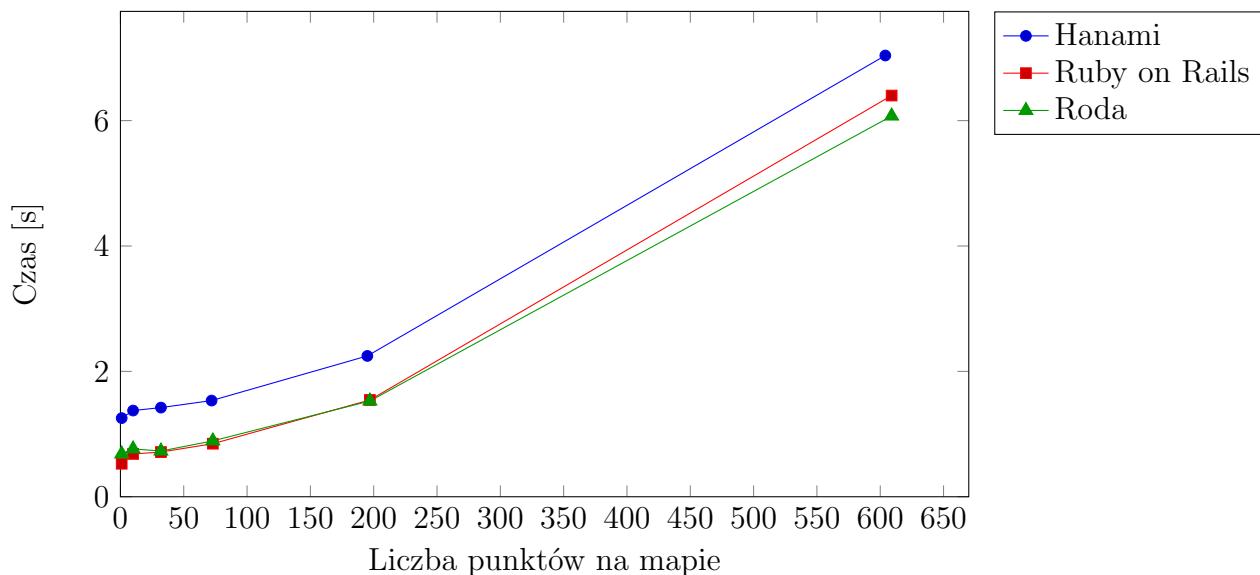
Badania na poziomie interfejsu użytkownika wykonano za pomocą Capybary, Selenium i Ruby Benchmark. Każdy scenariusz obejmował kroki, które użytkownik musi wykonać w aplikacji aby wykonać daną akcję. Badanie danego scenariusza, kończono w chwili pojawienia się efektów wykonanej akcji w interfejsie graficznym.

#### Punkty



Rysunek 7.8 Czas wykonania operacji CRUD dla punktu

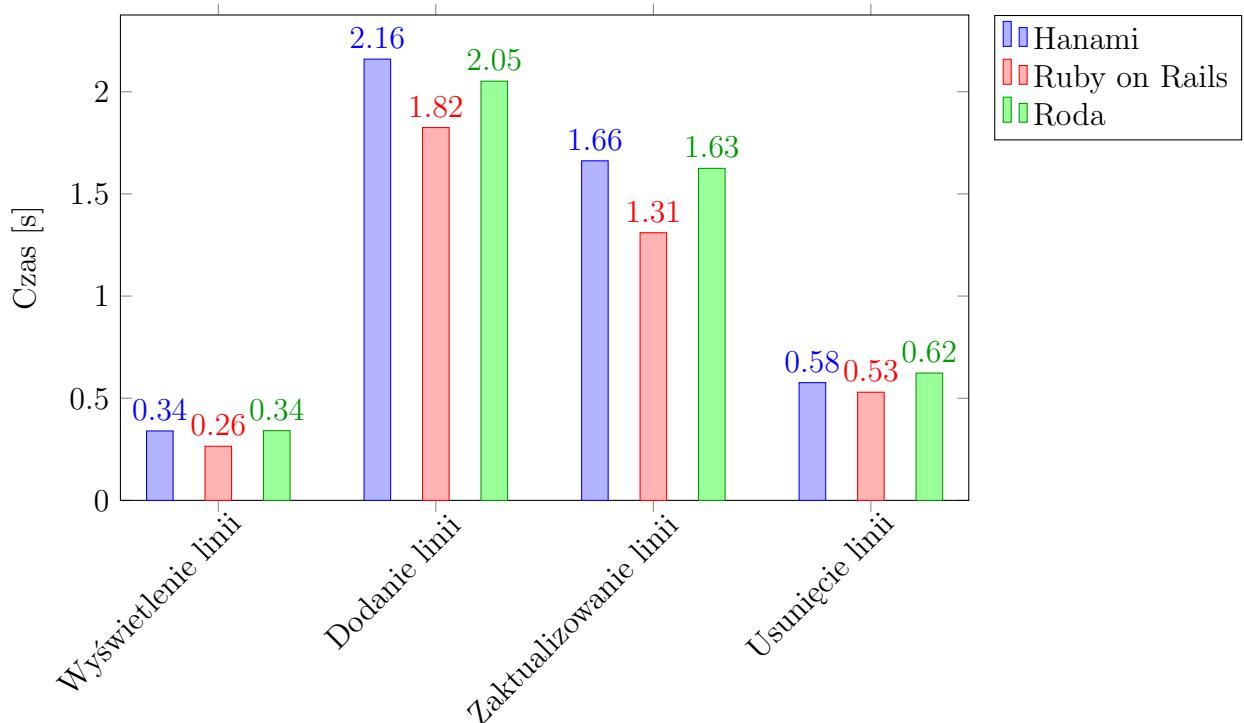
Framework Hanami okazał się najwolniejszy przy pojedynczych operacjach na obiektach typu punkt. Roda jest bardziej wydajna w akcjach dodawania i aktualizowania punktów, natomiast Ruby on Rails ma przewagę przy wyświetlaniu i usuwaniu punktu.



Rysunek 7.9 Czas ładowania widoku w zależności od liczby punktów

Na powyższym wykresie można zauważyc, że przewaga Rody i Ruby on Rails nad Hanami jest stała i nie zależy od liczby punktów.

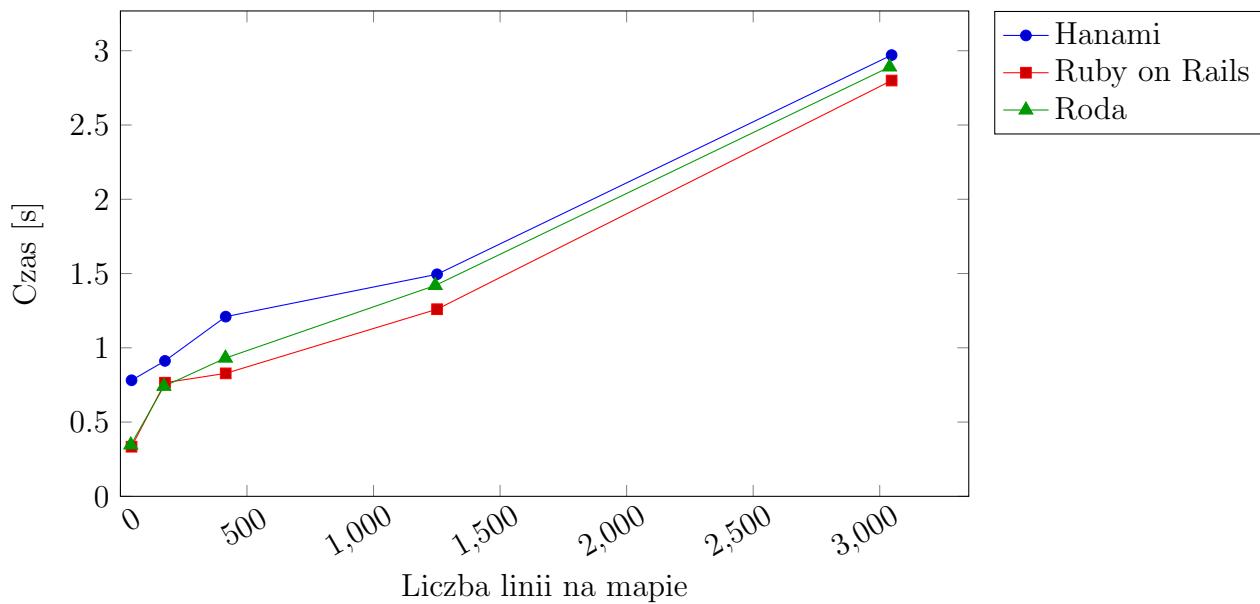
### Linie



Rysunek 7.10 Czas wykonania operacji CRUD dla linii

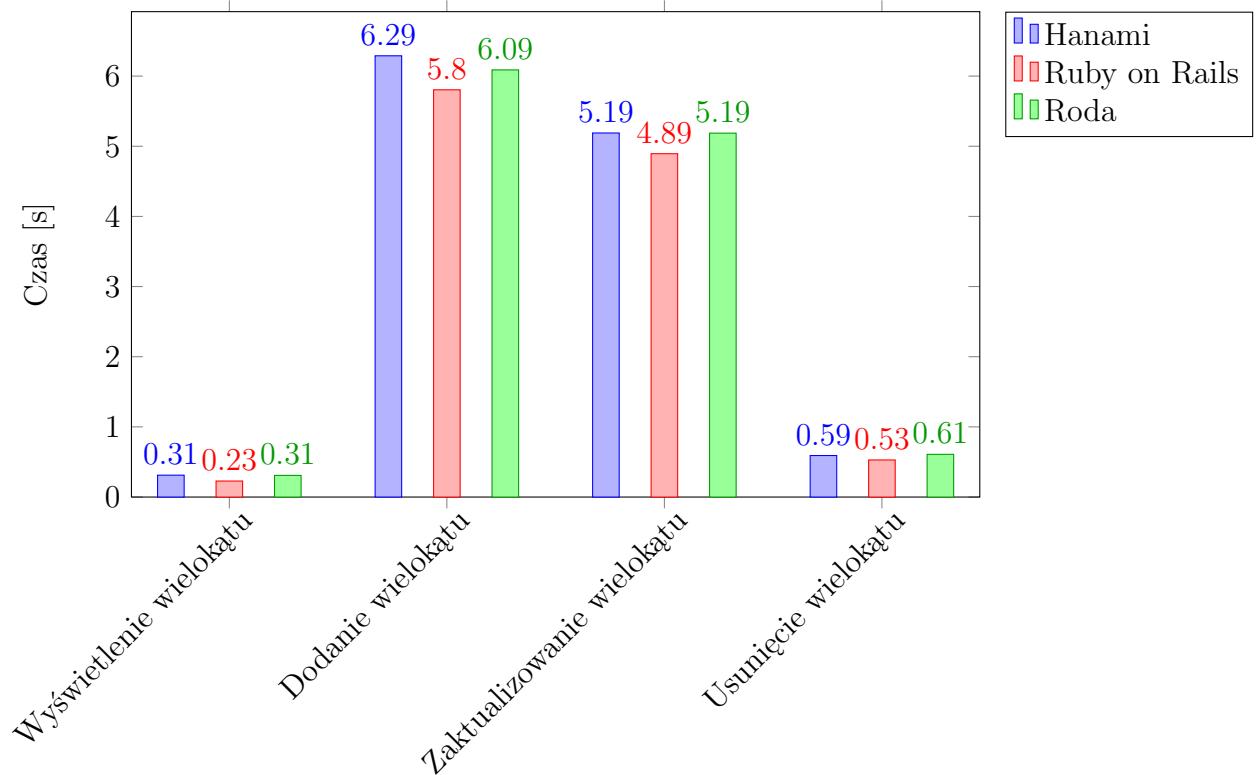
W operacjach na liniach można zauważyc, że Roda nie radzi sobie już tak dobrze jak z obiektami punktowymi. Najlepiej wypada Ruby on Rails, dwa pozostałe frameworki uzyskały porównywalne wyniki.

## Wielokąty



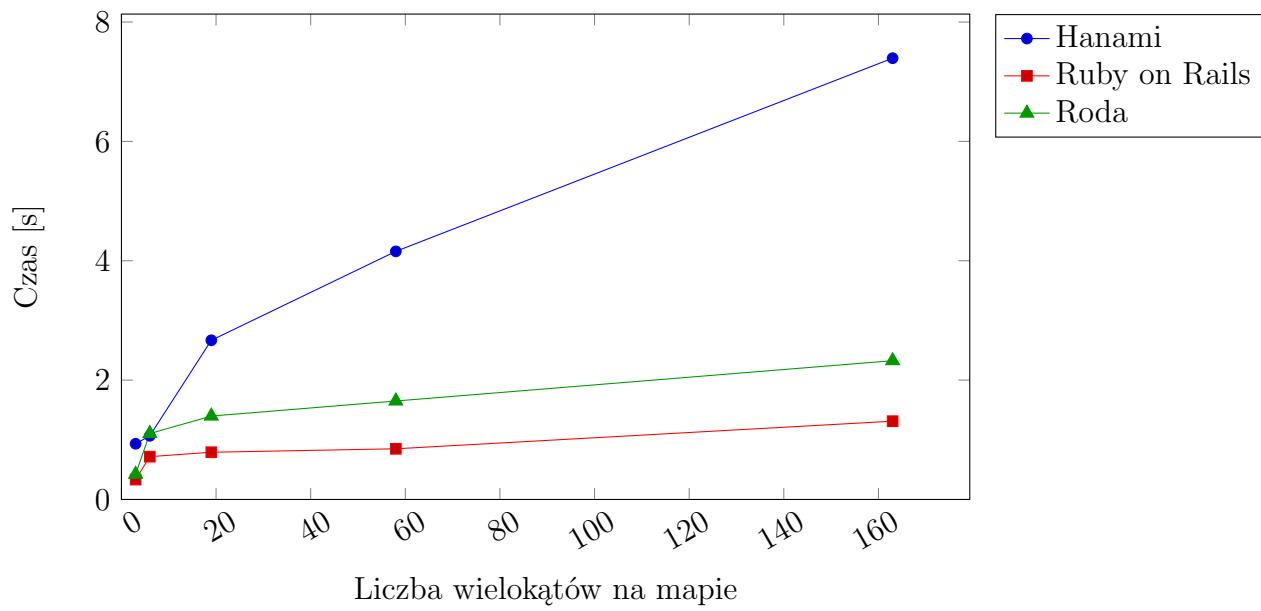
Rysunek 7.11 Czas ładowania widoku w zależności od liczby linii

Przy ładowaniu widoku zawierającego mniej niż 200 linii Roda i Ruby on Rails dają podobne rezultaty. Przy większej liczbie linii Ruby on Rails zyskuje przewagę. Hanami wypada najgorzej, ale różnica między nim, a dwoma pozostałymi frameworkami spada wraz ze wzrostem liczby linii.



Rysunek 7.12 Czas wykonania operacji CRUD dla wielokątów

Podczas operacji na wielokątach Roda i Hanami wypadły porównywalnie, obydwa frameworki dały gorsze rezultaty niż Ruby on Rails.



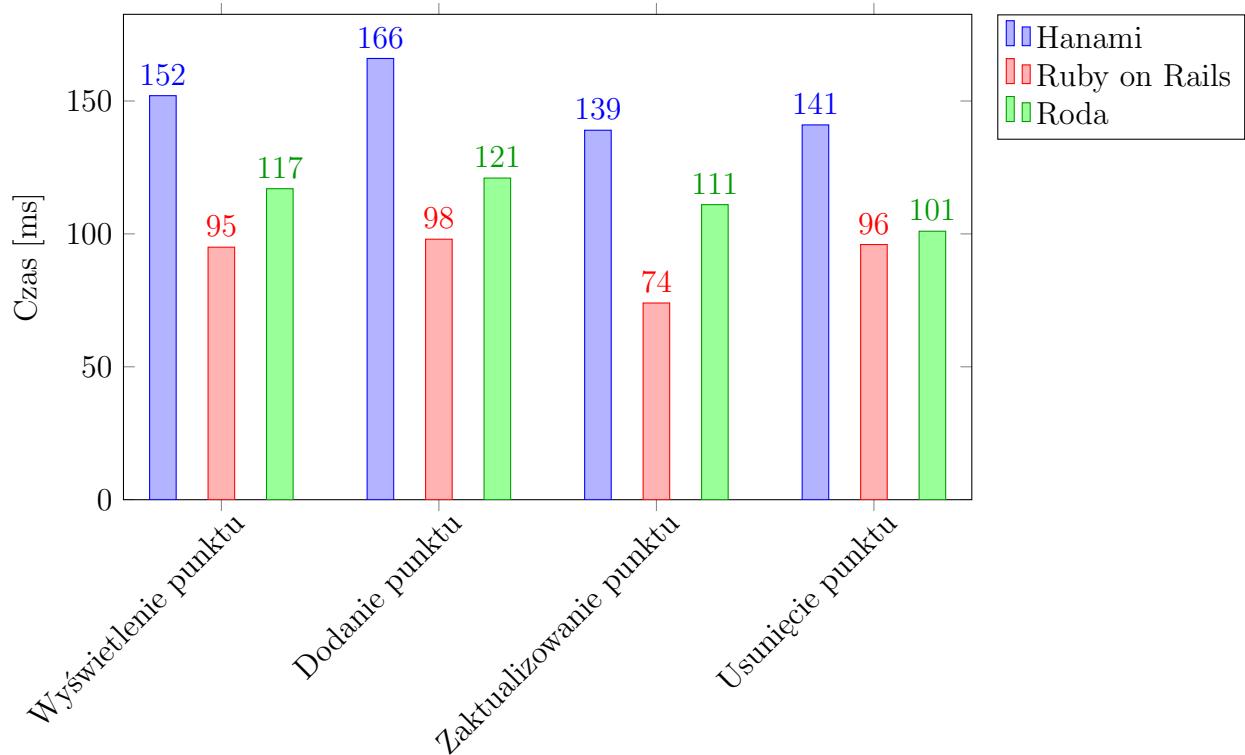
Rysunek 7.13 Czas ładowania widoku w zależności od liczby wielokątów

Podczas ładowania większej liczby wielokątów, najszybciej zwiększał się czas wykonania akcji dla frameworku Hanami. Drugie miejsce zajęła Roda, najlepsze wyniki zarejestrowano z użyciem Ruby on Rails.

#### 7.4.2 Zapytania HTTP

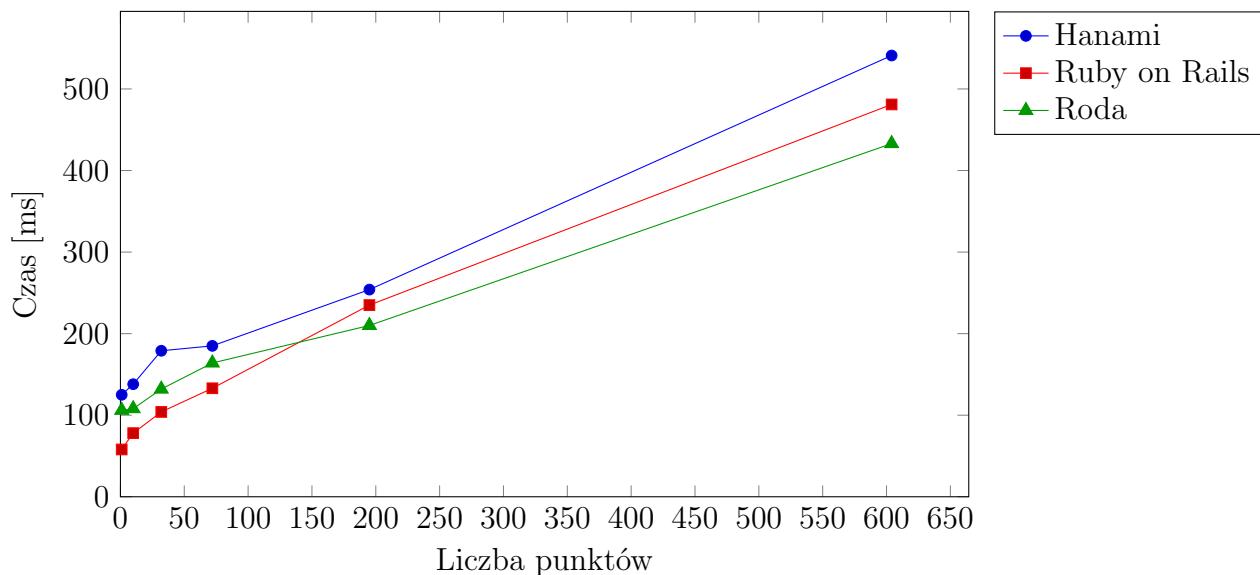
Wszystkie badania w tym rozdziale wykonano przy pomocy aplikacji JMeter.

##### Punkty



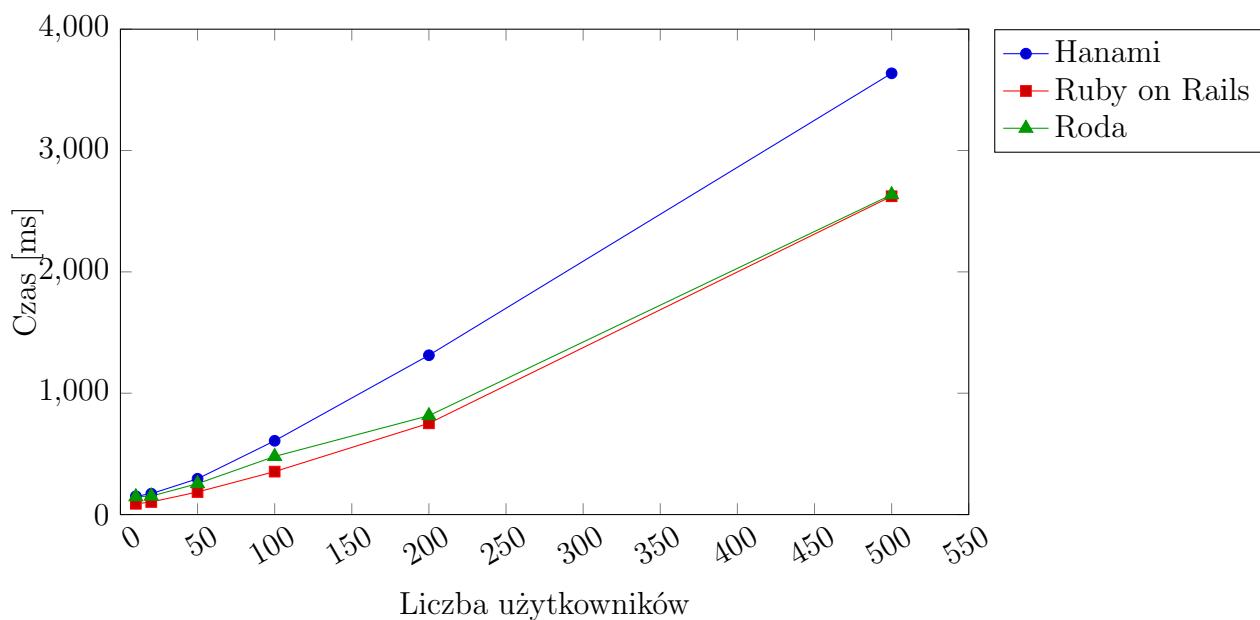
Rysunek 7.14 Czas wykonania zapytania HTTP dla punktu

Czas pojedynczych operacji HTTP dla obiektów punktowych jednoznacznie pokazuje, że najbardziej wydajna jest aplikacja napisana z użyciem Ruby on Rails, następnie Roda, a najmniej wydajny okazał się framework Hanami.



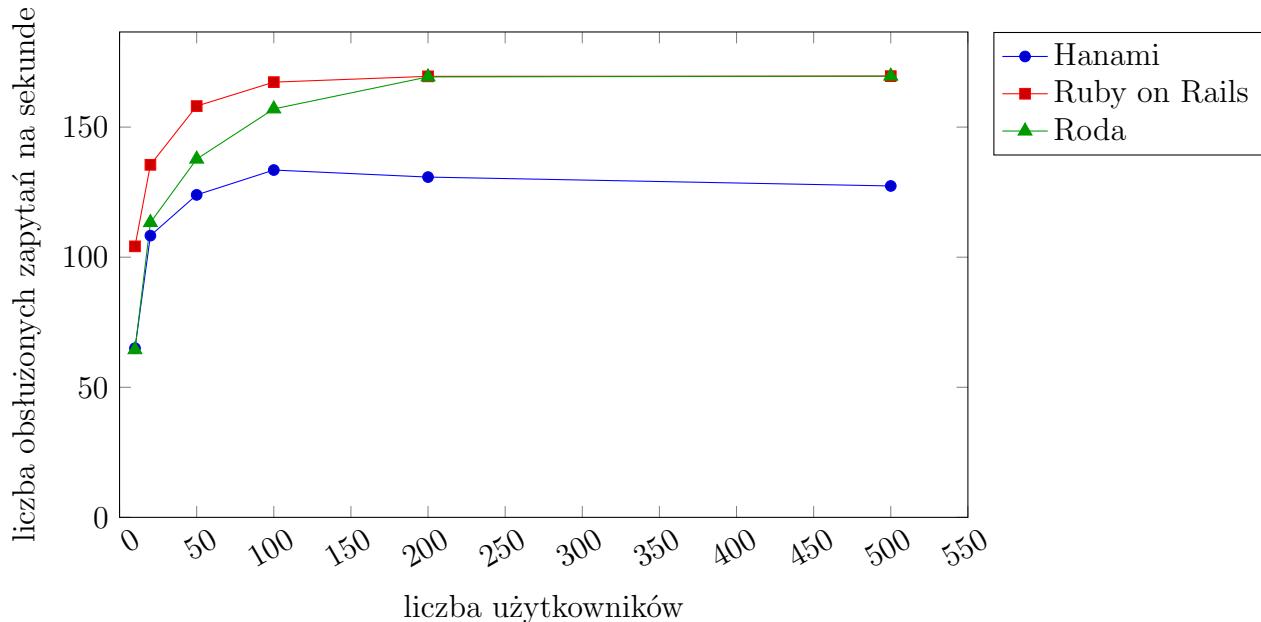
Rysunek 7.15 Czas ładowania danych w zależności od liczby punktów

Hanami bez względu na liczbę punktów wypada gorzej od dwóch pozostałych framework'ów. Jeśli punktów jest mniej niż 40, Ruby on Rails wczytuje dane nieco szybciej niż Roda. Powyżej 70 punktów, lepsze wyniki uzyskano dla frameworku Roda.



Rysunek 7.16 Czas ładowania danych w zależności od liczby użytkowników

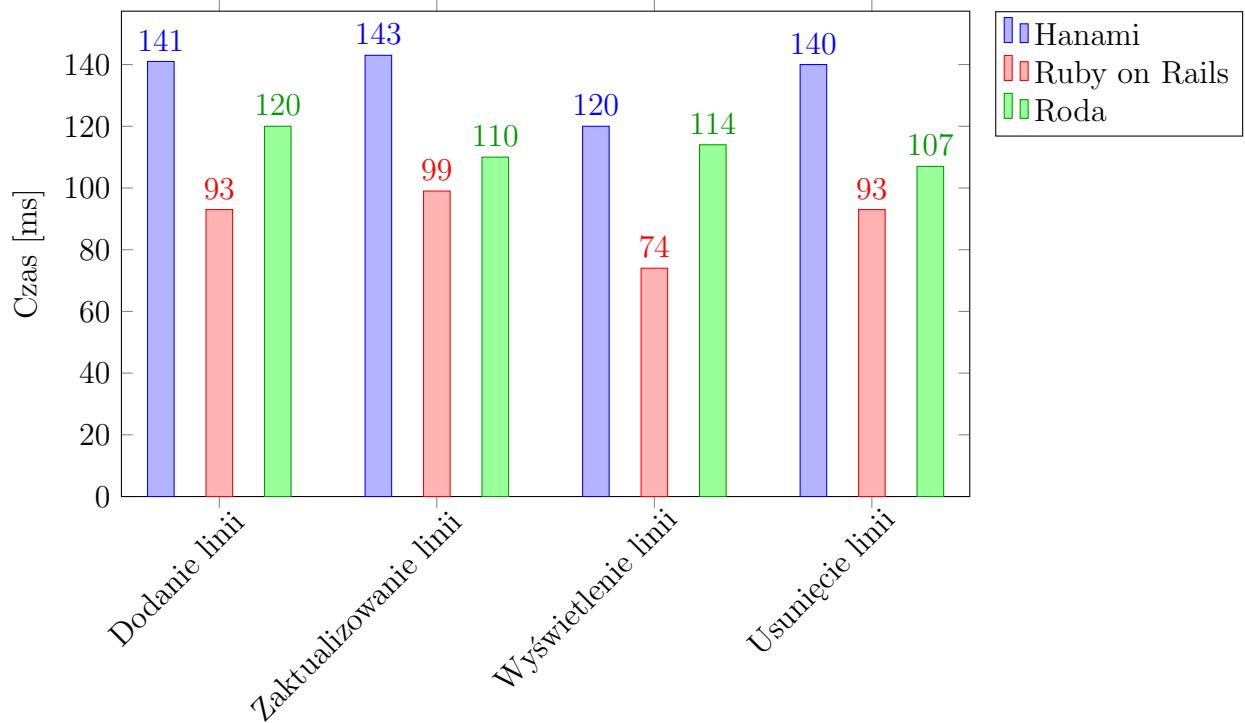
Przy zwiększającej się liczbie użytkowników aplikacja wykorzystująca framework Hanami szybciej zwiększa swój czas odpowiedzi. Pozostałe dwie aplikacje zachowują się podobnie.



Rysunek 7.17 liczba obsłużonych zapytań na sekundę w zależności od liczby użytkowników

Początkowo Ruby on Rails obsługuje najwięcej użytkowników. Powyżej 200 użytkowników, Roda wyrównuje wynik framework'u Ruby on Rails - 169 użytkowników na sekundę.

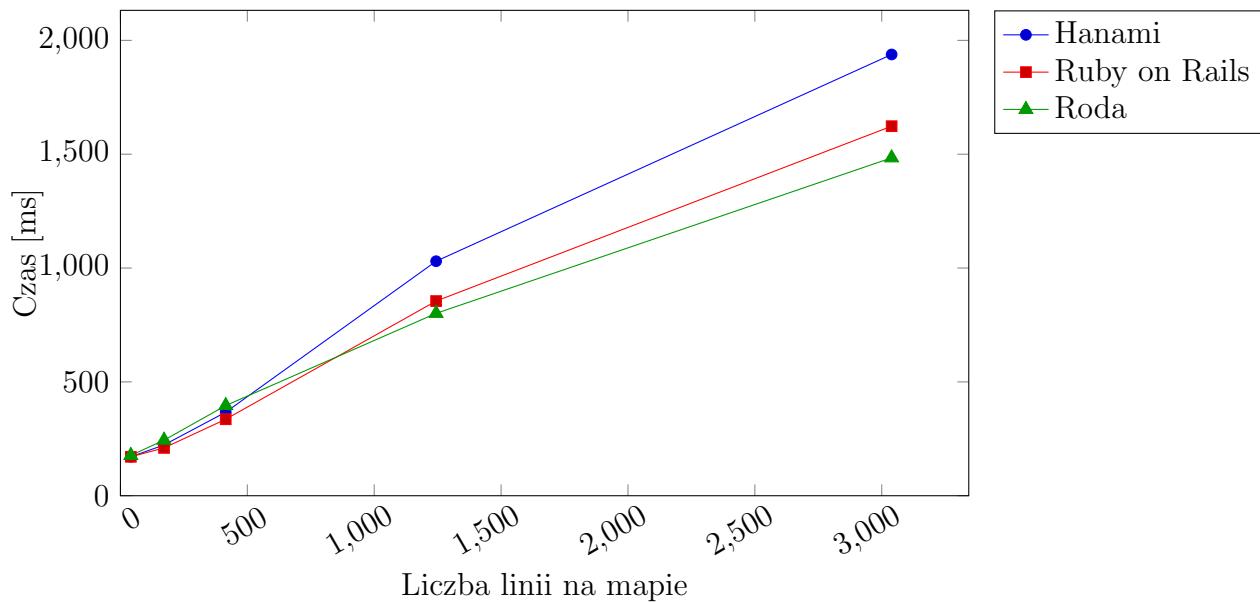
### Linie



Rysunek 7.18 Czas wykonania operacji CRUD dla linii

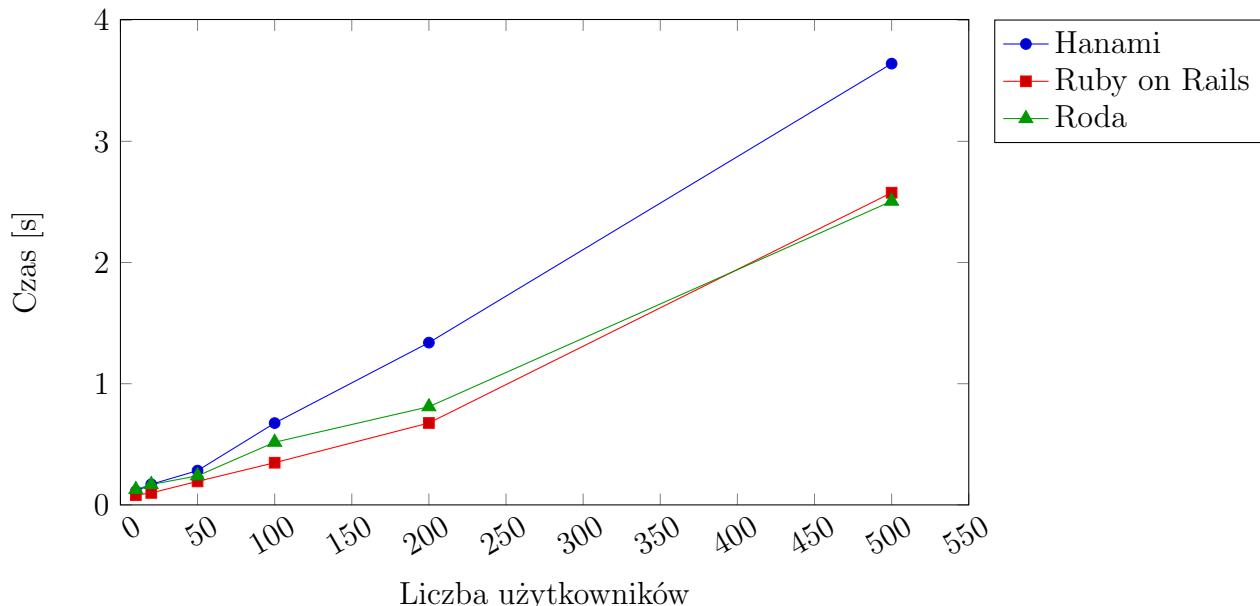
Tak samo jak dla punktów, najkrótszy czas operacji na liniach otrzymano w badaniach aplikacji opartej na frameworku Ruby on Rails. Następne miejsce zajął framework Roda.

Najdłuższy czas odpowiedzi otrzymano wykorzystując Hanami.



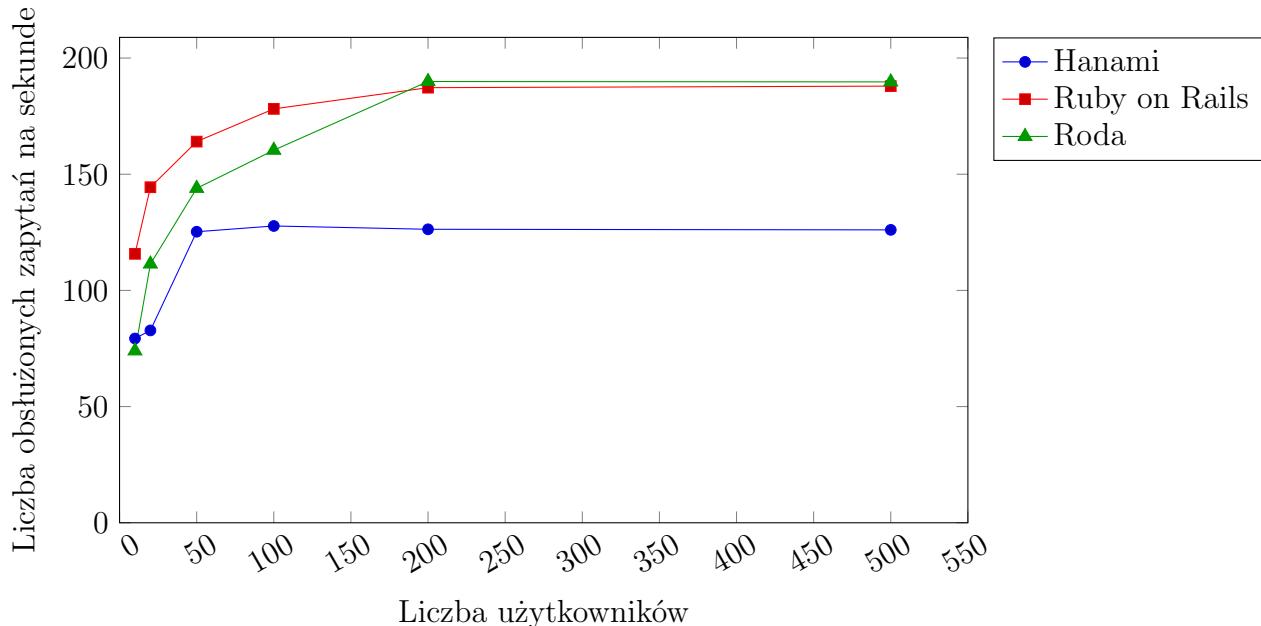
Rysunek 7.19 Czas ładowania danych w zależności od liczby linii

Poniżej 500 linii na mapie wszystkie 3 aplikacje odpowiadały w podobnym czasie. Przy większej liczbie linii, czas odpowiedzi Hanami rósł szybciej niż pozostałych dwóch frameworków. Najkrótszy czas odpowiedzi dla danych liczących ponad 1000 linii zanotowano dla aplikacji wykorzystującej framework Roda.



Rysunek 7.20 Czas ładowania danych w zależności od liczby użytkowników

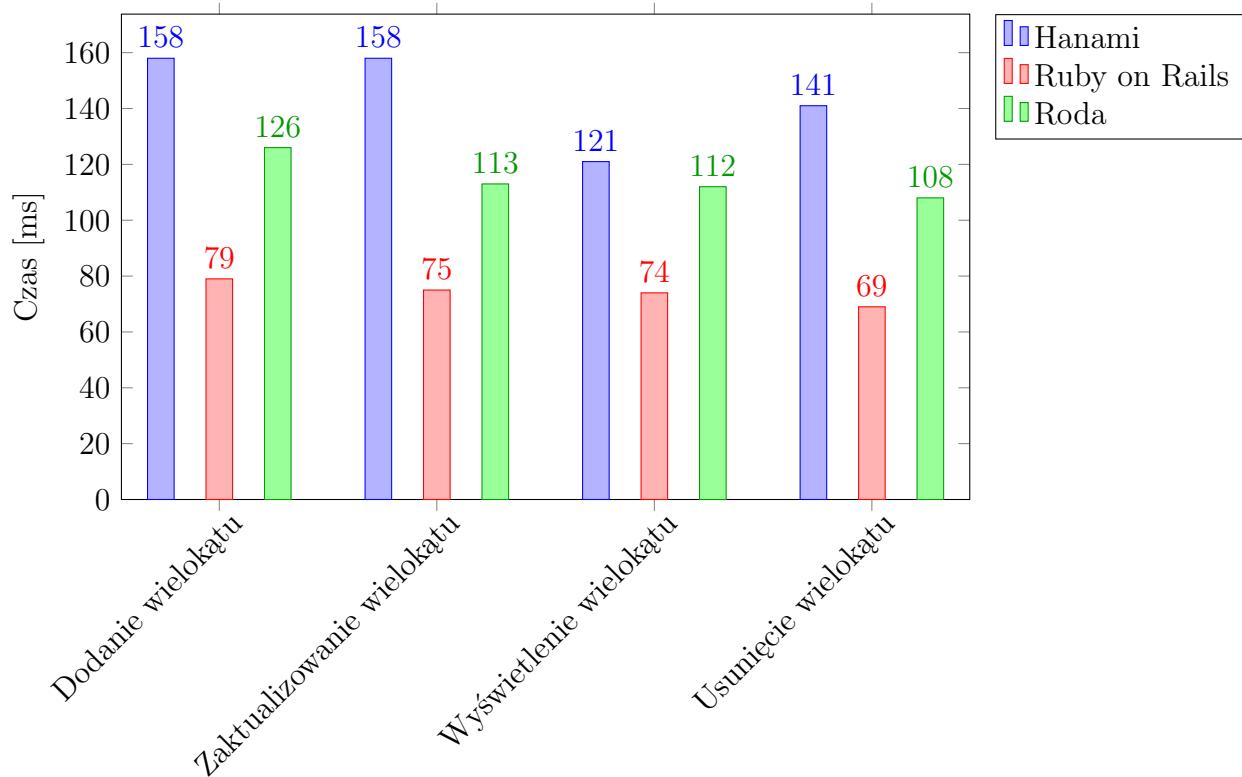
Wraz ze wzrostem liczby użytkowników czas odpowiedzi aplikacji zbudowanej w oparciu o framework Hanami, rósł szybciej niż pozostałych dwóch aplikacji.



Rysunek 7.21 Liczba obsłużonych zapytań na sekundę w zależności od liczby użytkowników

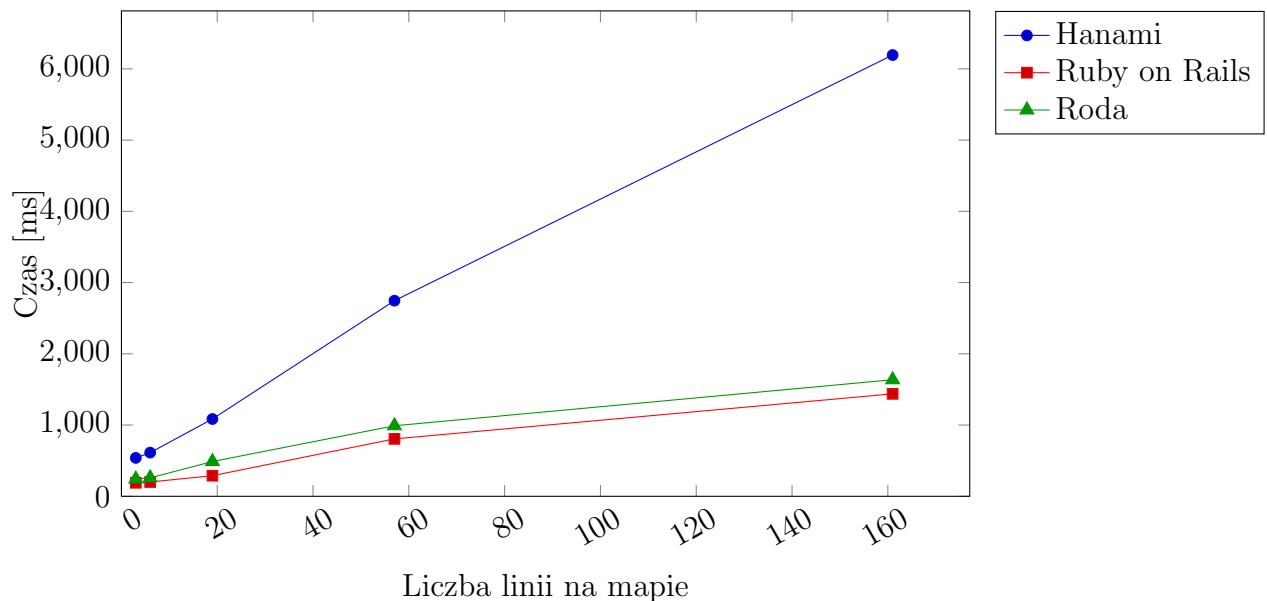
Powyżej 200 użytkowników jednocześnie korzystających z aplikacji, Roda i Ruby on Rails w ciągu sekundy obsługiwały około 190 użytkowników na sekundę. Przy mniejszej liczbie użytkowników Ruby on Rails obsługiwał więcej użytkowników w ciągu sekundy od Roda. Hanami w ciągu całego badania prezentował się znacznie gorzej od dwóch pozostałych framework'ów i maksymalnie w ciągu sekundy odpowiedział na zapytanie 115 użytkowników.

## Wielokąty



Rysunek 7.22 Czas wykonania operacji CRUD dla wielokątów

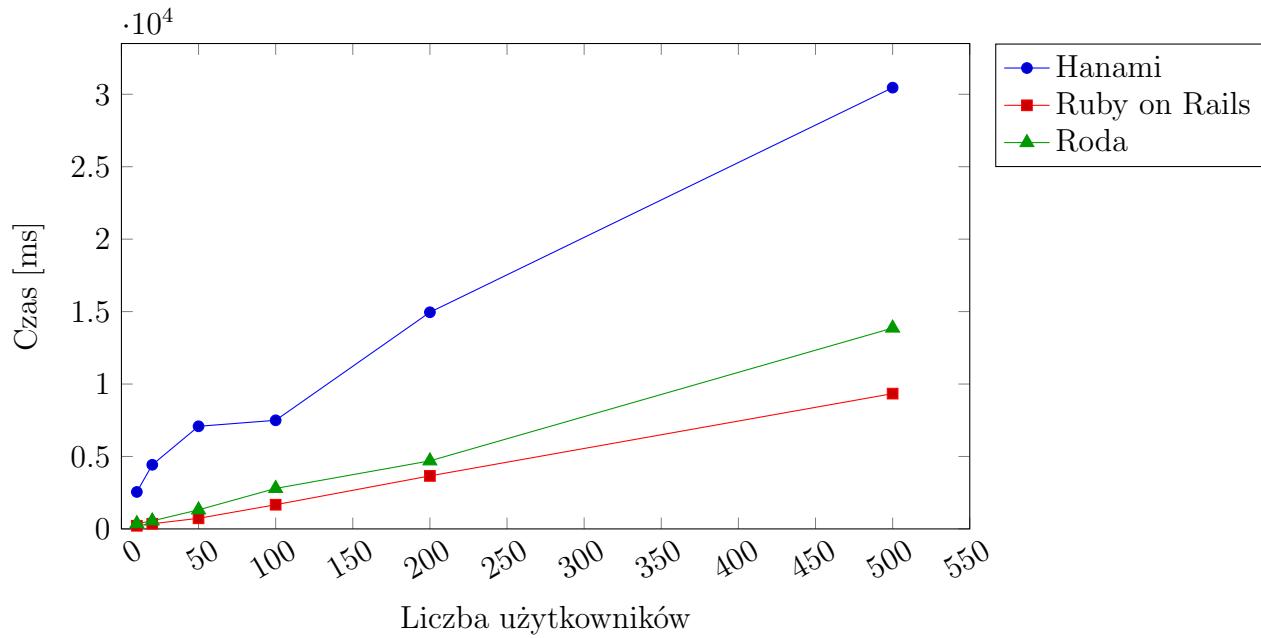
W podstawowych operacjach na wielokątach najlepsze czasy uzyskał framework Ruby on Rails, które były średnio dwukrotnie lepsze od wyników najwolniejszego frameworku Hanami.



Rysunek 7.23 Czas ładowania danych w zależności od liczby wielokątów

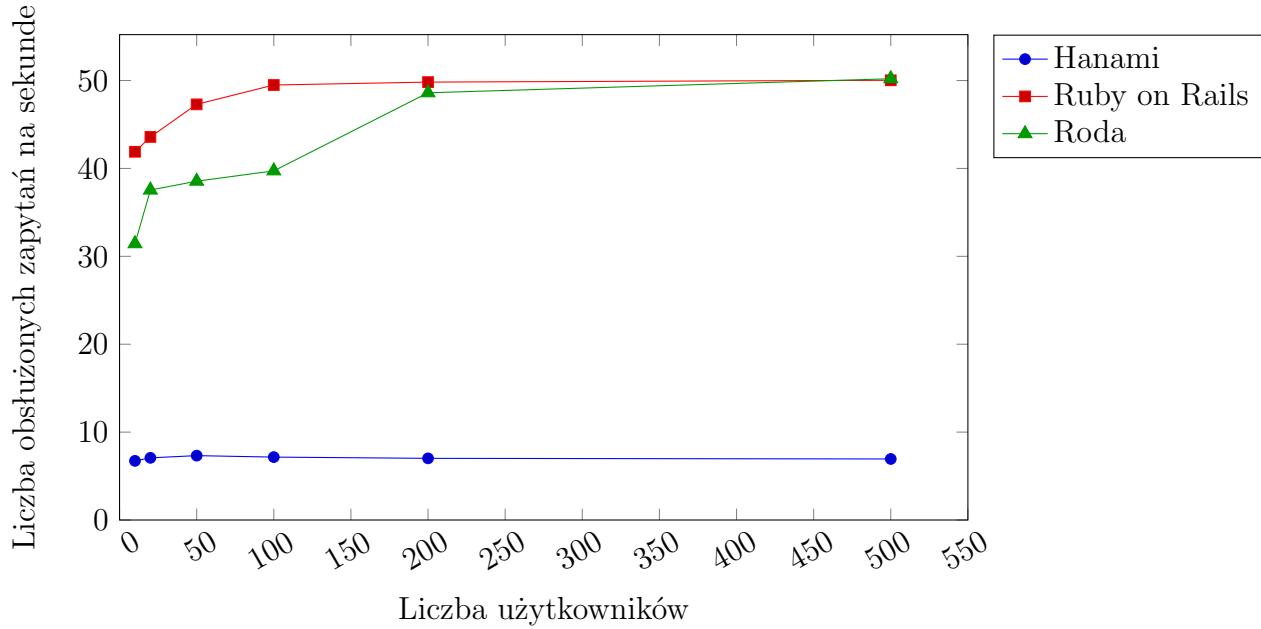
Czas odpowiedzi aplikacji zbudowanej w oparciu o Hanami znacznie szybciej rósł w po-

równaniu do dwóch pozostałych aplikacji. Ruby on Rails i Roda uzyskały zbliżone czasy z małą, stałą przewagą na korzyść Ruby on Rails.



Rysunek 7.24 Czas ładowania danych w zależności od liczby użytkowników

Dla danych typu wielokąt, Hanami znacznie wolniej obsługiwał zapytania przy wzrastającej liczbie użytkowników w porównaniu do pozostałych frameworków. Najlepiej z rosnącą liczbą użytkownik radził sobie framework Ruby on Rails.



Rysunek 7.25 Liczba obsłużonych zapytań na sekunde w zależności od liczby użytkowników

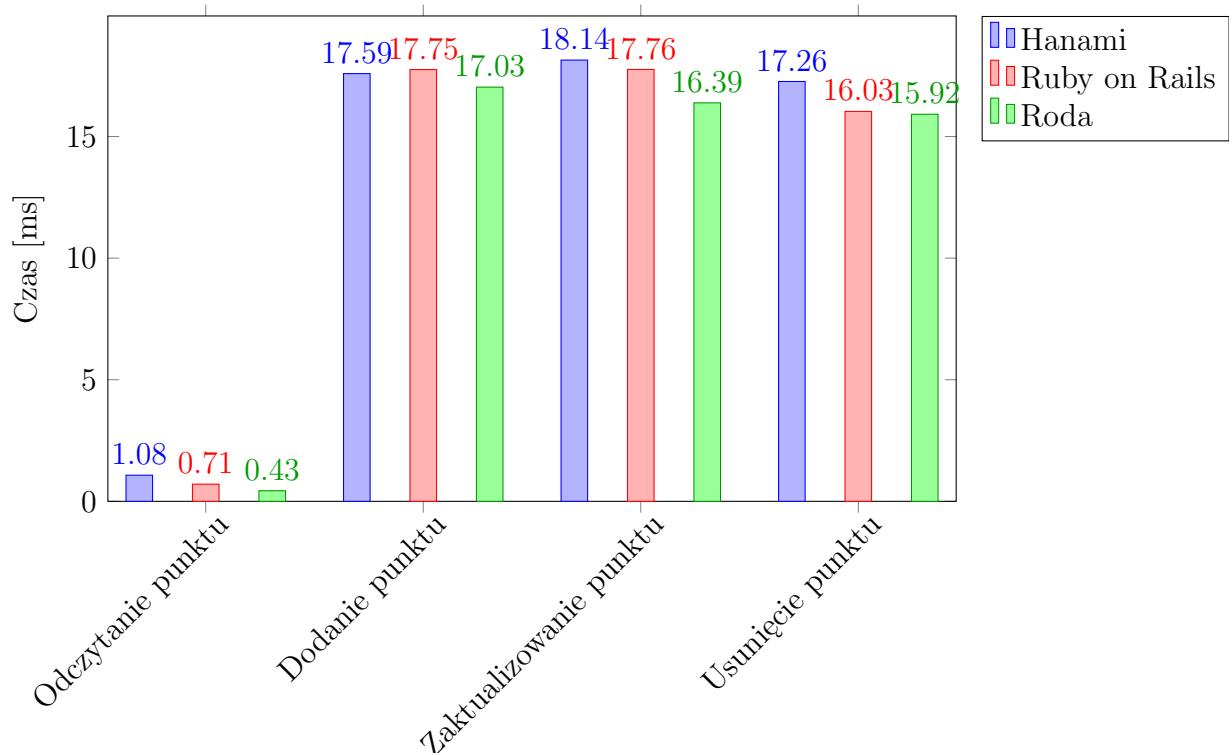
Hanami zdołał maksymalnie obsłużyć 7 użytkowników w ciągu sekundy i ta liczba nie zmieniała się przy rosnącej liczbie użytkowników. Ruby on Rails, podobnie jak dla

innych typów danych przy mniejszej liczbie użytkowników, mniej niż 200, obsługiwał więcej użytkowników niż Roda, jednak od 200 użytkowników jednocześnie korzystających z aplikacji, oba frameworki obsługiwały prawie 50 użytkowników w ciągu jednej sekundy.

### 7.4.3 Komunikacja z bazą danych

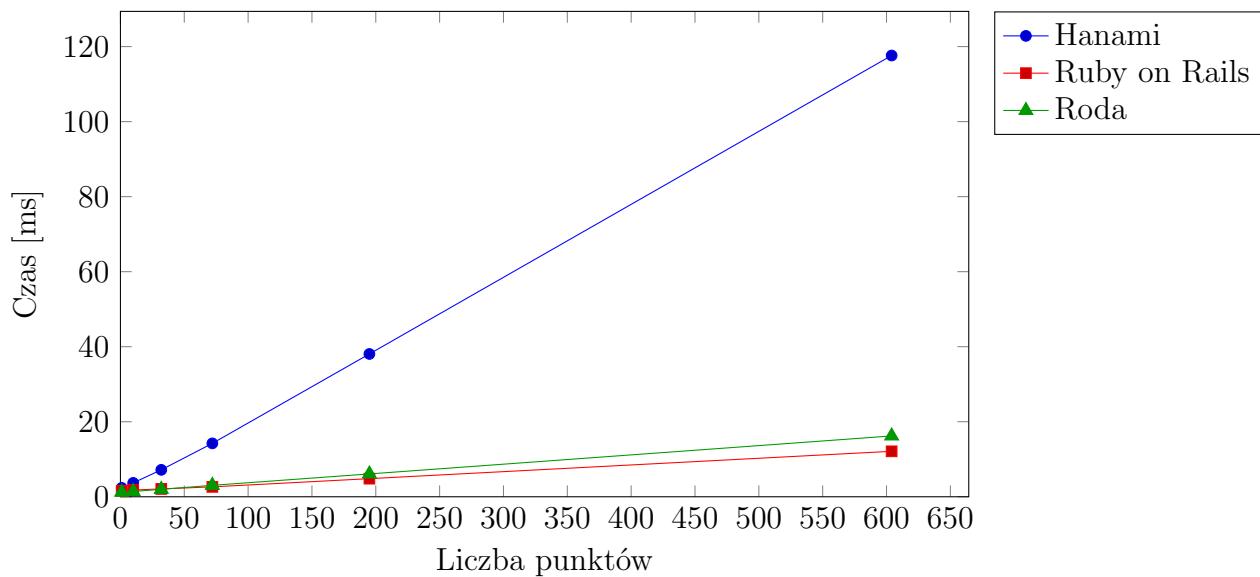
Badania wydajności komunikacji z bazą danych zrealizowano w warstwie modelu każdej z aplikacji. Czas wykonywania metod dla wybranych modeli zmierzono przy pomocy biblioteki Ruby Benchmark.

#### Punkty



Rysunek 7.26 Czas wykonania operacji CRUD dla punktu

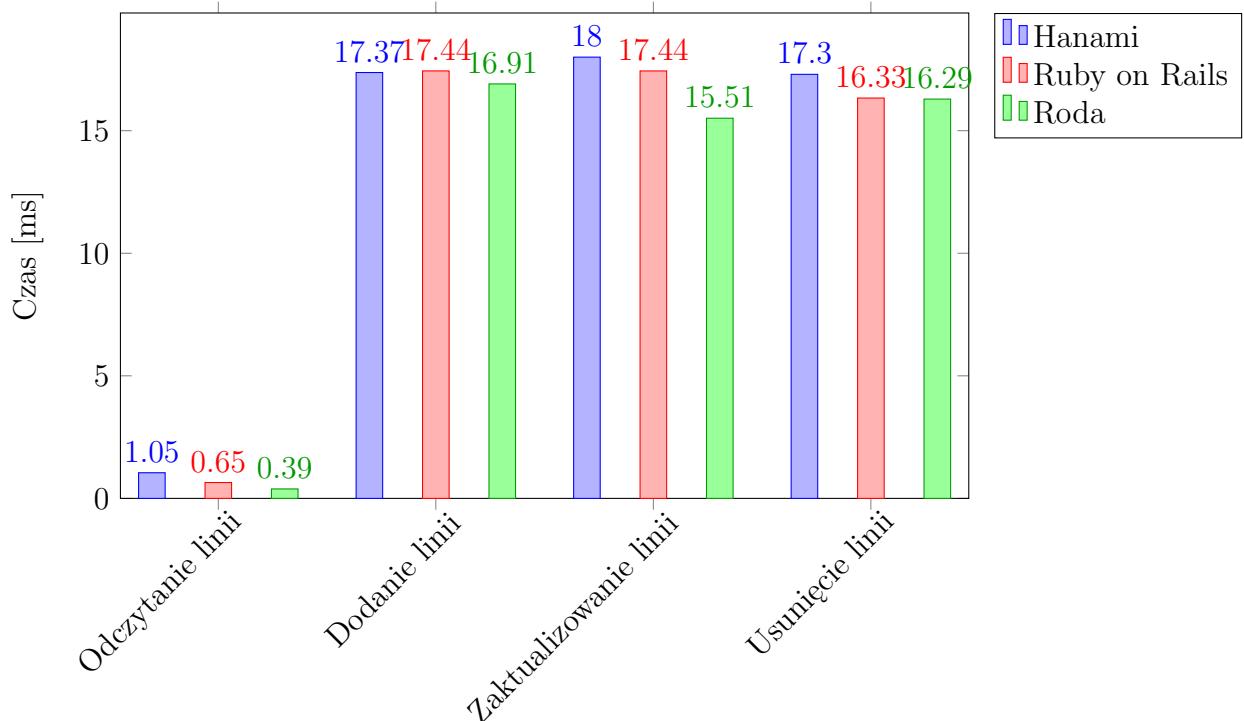
W pojedynczych operacjach w warstwie modelu czasy operacji na punktach były do siebie zbliżone dla 3 badanych frameworków. Najszybciej zadane operacje wykonał framework Roda.



Rysunek 7.27 Czas wczytywania danych z bazy w zależności od liczby punktów

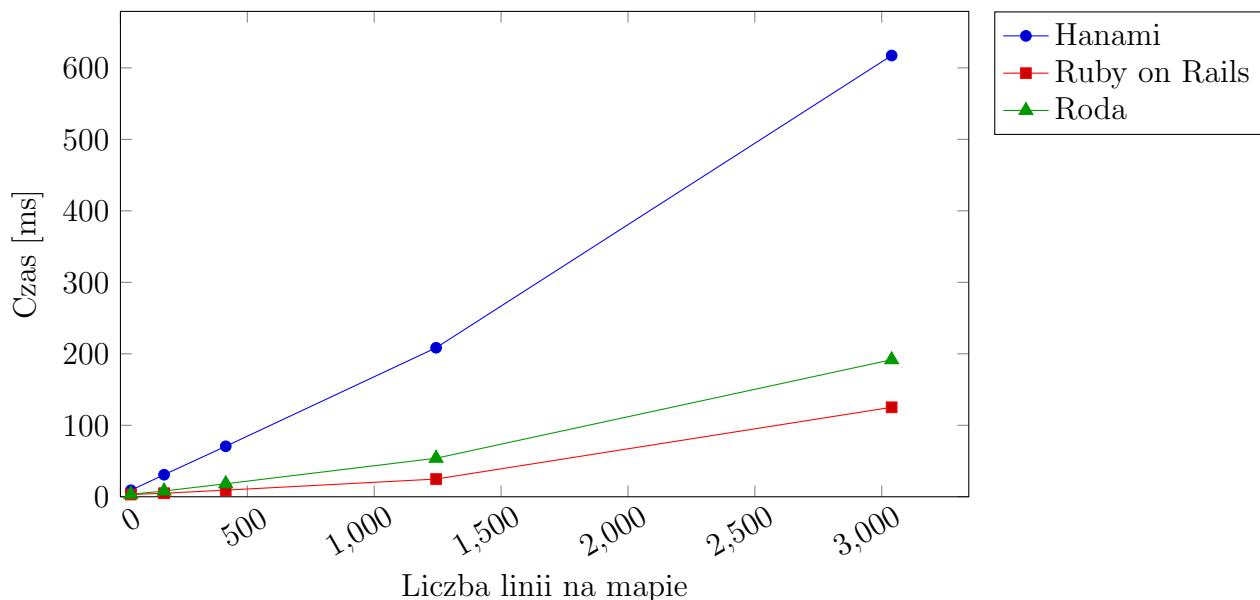
Przy rosnącej liczbie wczytywanych punktów Roda i Ruby on Rails uzyskały podobne wyniki, które były znacznie lepsze od Hanami.

### Linie



Rysunek 7.28 Czas wykonania operacji CRUD dla linii

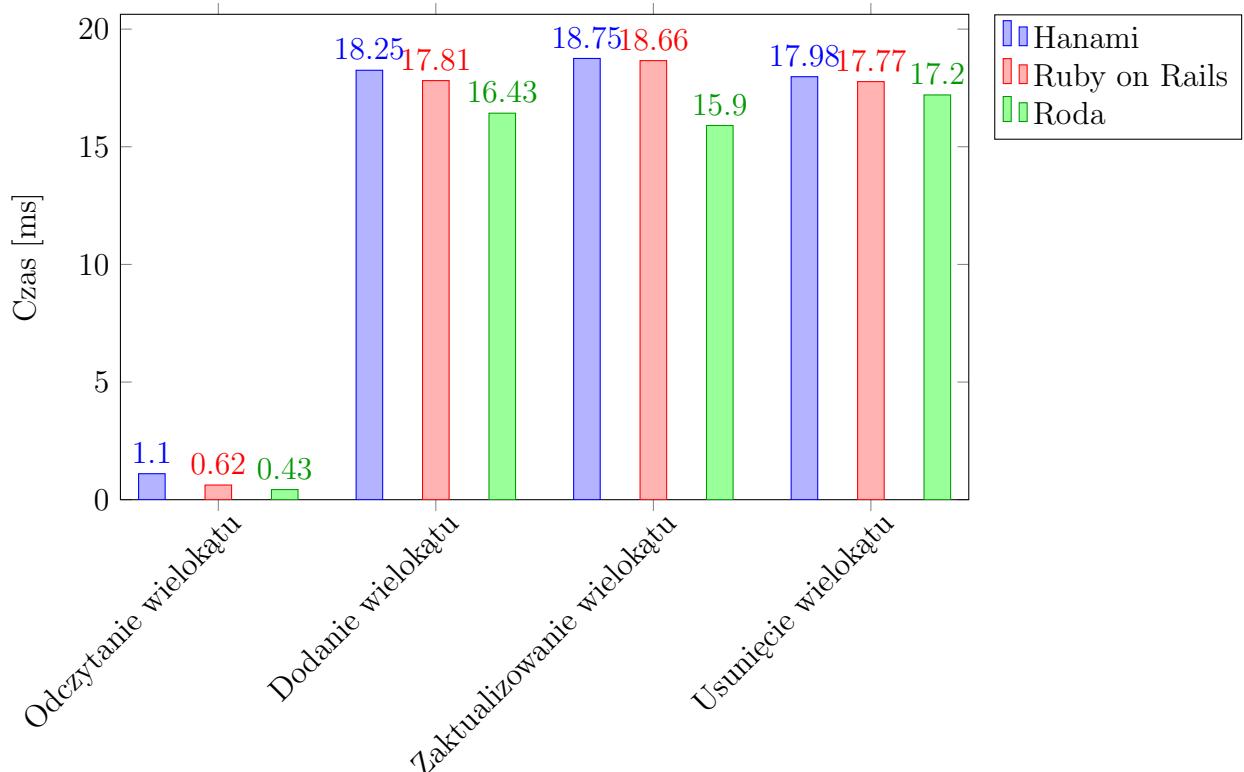
Wyniki dla danych liniowych są analogiczne jak dla punktów, Roda uzyskała najlepsze rezultaty, a najgorsze wyniki uzyskano dla Hanami.



Rysunek 7.29 Czas wczytywania danych z bazy w zależności od liczby linii

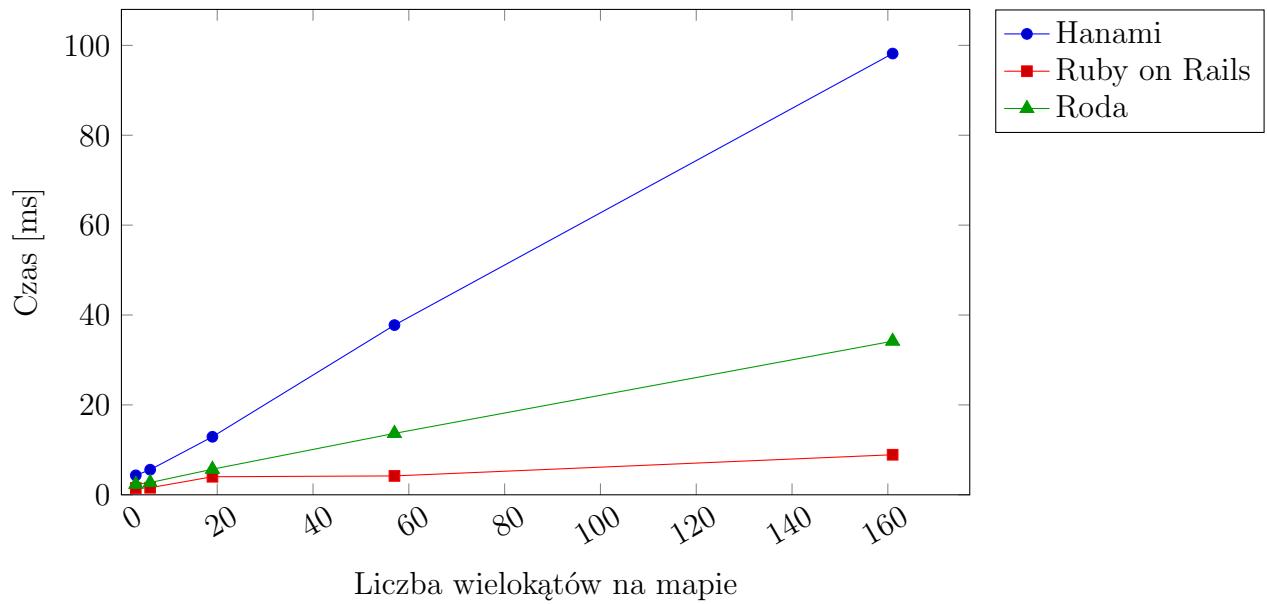
Przy danych liniowych nadal czas odpowiedzi Hanami rósł znacznie szybciej od dwóch pozostałych framework'ów. Można zauważyć, że różnica między Hanami, a Ruby on Rails jest bardziej zauważalna niż dla punktów, z korzyścią na stronę Ruby on Rails.

## Wielokąty



Rysunek 7.30 Czas wykonania operacji CRUD dla wielokątów

Badania dla wielokątów dały analogiczne wyniki jak dla dwóch poprzednich typów danych - Roda wykonywała operacje szybciej niż pozostałe frameworki. Najdłuższy czas otrzymano dla frameworku Hanami.



Rysunek 7.31 Czas wczytywania danych z bazy w zależności od liczby wielokątów

Podobnie jak dla pozostałych typów danych geograficznych, dla aplikacji wykorzystującej framework Hanami odnotowano znacznie gorsze wyniki dla rosnącej liczby wczytywanych wielokątów niż dla pozostałych dwóch aplikacji. Jednak dla wielokątów różnica między frameworkiem Roda, a Ruby on Rails jest bardziej zauważalna.

# Rozdział 8

## Podsumowanie i wnioski

### 8.1 Podsumowanie wykonanej pracy

### 8.2 Analiza wyników badań

Badania wykazały, że przy wyborze frameworku internetowego języka Ruby do budowy systemu informacji geograficznej najgorszym wyborem jest framework Hanami. Jego jedynym atutem może być rozbudowana architektura, która ułatwia organizację kodu źródłowego w rozbudowanych projektach. Jednak w prostszych projektach, wiele warstw abstrakcji wymaga od programistów napisania więcej kodu źródłowego niż w innych frameworkach do zaimplementowania takich samych funkcjonalności. Hanami jest znacznie mniej wydajny w każdej z badanych płaszczyzn - interfejsu użytkownika, zapytań HTTP, komunikacji z bazą danych. Dodatkowo aby użyć innej przestrzennej bazy danych niż PostgreSQL trzeba zaimplementować wsparcie dla nowych typów danych, co nie jest konieczne przy korzystaniu z innych frameworków.

Pozostałe dwa frameworki Ruby on Rails i Roda podobnie wypadły w badaniach wydajnościowych. Roda w operacjach CRUD w warstwie modelu była szybsza dla wszystkich typów danych od frameworku Ruby on Rails, jednak przy wczytywaniu bardziej skomplikowanych typów - linii i wielokątów - dla Ruby on Rails zarejestrowano niższe czasy przy dużej liczbie danych. Na poziomie zapytań HTTP oraz interfejsu użytkownika, aplikacja wykorzystująca Ruby on Rails była bardziej wydajna zarówno dla operacji na pojedynczych obiektach geograficznych, jak i dla wczytywania większej liczby obiektów. Ruby on Rails posiada więcej bibliotek służących do pracy na danych geograficznych co ma przełożenie na mniejszą ilość napisanego kodu dzięki gotowym funkcjom. Struktura projektu Ruby on Rails i projektu Roda jest bardzo podobna, projekty zawierały tyle samo plików.

### 8.3 Realizacja celu projektu

Niniejsza praca udowodniła, że za pomocą frameworków internetowych języka Ruby z powodzeniem można stworzyć system informacji geograficznej. Najlepszym wyborem do budowy systemu GIS okazał się najbardziej popularny framework Ruby on Rails. Duża popularność przekłada się na duże wsparcie, wiele dostępnych bibliotek i narzędzi do pracy z danymi przestrzennymi. Ruby on Rails wymagał najmniej kodu źródłowego do zaimplementowania zaprojektowanego systemu, który wykazywał się największą wydajnością przy przetwarzaniu danych geograficznych.



# Literatura

- [1] *Dokumentacja biblioteki Capybara*, dostępna pod adresem:  
<https://github.com/teamcapybara/capybara>, aktualne na dzień 18.06.2017r.
- [2] *Dokumentacja biblioteki GeoRuby*, dostępna pod adresem:  
<https://github.com/nofxx/georuby>, aktualne na dzień 15.06.2017r.
- [3] *Dokumentacja biblioteki Google Maps JavaScript API*, dostępna pod adresem:  
<https://developers.google.com/maps/documentation/javascript/>, aktualne na dzień 15.06.2017r.
- [4] *Dokumentacja biblioteki Leaflet*, dostępna pod adresem:  
<http://leafletjs.com/reference-1.0.3.html>, aktualne na dzień 15.06.2017r.
- [5] *Dokumentacja biblioteki Rgeo*, dostępna pod adresem:  
<https://github.com/rgeo/rgeo>, aktualne na dzień 15.06.2017r.
- [6] *Dokumentacja biblioteki Ruby Benchmark*, dostępna pod adresem:  
<https://ruby-doc.org/stdlib-1.9.3/libdoc/benchmark/rdoc/Benchmark.html>, aktualne na dzień 18.06.2017r.
- [7] *Dokumentacja Hanami*, dostępna pod adresem:  
<http://hanamirb.org/guides/>, aktualne na dzień 08.03.2017r.
- [8] *Dokumentacja języka Ruby*, dostępna pod adresem:  
<https://www.ruby-lang.org/pl/documentation/>, aktualne na dzień 08.03.2017r.
- [9] *Dokumentacja JMeter*, dostępna pod adresem:  
<http://jmeter.apache.org/index.html>, aktualne na dzień 18.06.2017r.
- [10] *Dokumentacja MangoMap*, dostępna pod adresem:  
<http://help.mangomap.com/>, aktualne na dzień 22.04.2017r.
- [11] *Dokumentacja MySQL*, dostępna pod adresem:  
<https://dev.mysql.com/doc/refman/5.7/en/>, aktualne na dzień 15.06.2017r.
- [12] *Dokumentacja OpenStreetMap*, dostępna pod adresem:  
<http://wiki.openstreetmap.org/>, aktualne na dzień 08.03.2017r.
- [13] *Dokumentacja PostgreSQL*, dostępna pod adresem:  
<https://www.postgresql.org/docs/9.6/static/index.html>, aktualne na dzień 09.06.2017r.
- [14] *Dokumentacja PostGIS*, dostępna pod adresem:  
<http://postgis.net/documentation/>, aktualne na dzień 08.03.2017r.

- [15] *Dokumentacja Ruby on Rails*, dostępna pod adresem:  
<http://guides.rubyonrails.org/>, aktualne na dzień 08.03.2017r.
- [16] *Dokumentacja Roda*, dostępna pod adresem:  
<http://roda.jeremyevans.net/documentation.html>, aktualne na dzień 01.06.2017r.
- [17] *Dokumentacja SpatiaLite*, dostępna pod adresem:  
<https://www.gaia-gis.it/fossil/libspatialite/index>, aktualne na dzień 09.06.2017r.
- [18] Huisman Otto, By (de) Rolf A., *Principles of Geographic Information Systems*, ITC, 2009
- [19] Martin Robert, *The Clean Architecture*, dostępna pod adresem:  
<https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>, aktualne na dzień 20.04.2017r.
- [20] Ruby Sam, Thomas Dave, Hansson Heinemeier David, *Agile Web Development with Rails 5*, Pragmatic Programmers, 2016
- [21] Schmandt Michael, *GIS Commons: An Introductory Textbook on Geographic Information Systems*, dostępne pod adresem:  
<http://giscommons.org/>, aktualne na dzień 07.04.2017r.
- [22] Smyrdek Przemysław, *Czym jest framework i po co go używać*, dostępne pod adresem:  
<http://poznajprogramowanie.pl/czym-jest-framework-i-po-co-go-uzywac/>, aktualne na dzień 20.04.2017r.

# Dodatek A

## Opis zawartości płyty CD

W głównym katalogu załączonej do pracy płyty CD znajdują się:

- mgr\_rails - kod źródłowy projektu Ruby on Rails
- mgr\_roda - kod źródłowy projektu Roda
- mgr\_hanami - kod źródłowy projektu Hanami
- jmeter\_benchmarks - kod źródłowy testów wydajnościowych napisanych za pomocą aplikacji JMeter
- capybara\_benchmarks - kod źródłowy testów wydajnościowych napisanych za pomocą biblioteki Capybara
- praca\_dyplomowa.pdf - niniejszy dokument
- streszczenie.pdf - streszczenie niniejszego dokumentu