Rozproszona aplikacja umożliwiająca przeglądanie fraktali

Autor:
Tymon Tobolski (181037)
Jacek Wieczorek (181043)
Mateusz Lenik (181142)

Prowadzący:
dr inż. Marek Woda

Wydział Elektroniki
III rok
Pn 12.15 - 14.00

Spis treści

1 Opis projektu

Celem projektu jest stworzenie aplikacji internetowej pozwalającej użytkownikowi na przeglądanie fraktali renderowanych na klastrze. Aplikacja pozwalałaby na przybliżanie i oddalanie widocznej części fraktala, zmianę kolorów oraz zapisywanie ostatniej przeglądanej pozycji przez danego użytkownika.

2 Uzasadnienie biznesowe

Projekt może zostać wykorzystany jako proof of concept dla aplikacji przetwarzających duże ilości danych i prezentację ich na mapie. Jest to proste do rozbudowy demo technologiczne, na podstawie którego można stworzyć bardziej skomplikowane aplikacje.

3 Technologie

Aplikacja będzie oparta o technologie wykorzystujące JVM, dzięki czemu możliwe będzie uruchomienie jej na kilku platformach bez konieczności modyfikacji. Dodatkowo ułatwia to korzystanie z różnych języków programowania podczas implementacji aplikacji. Za interfejs webowy będzie odpowiadała aplikacja w Ruby on Rails, która będzie się komunikowała z rozproszonym backendem zaimplementowanym w Scali.

- Platforma: JVM
- Język implementacji: Ruby, Scala, CoffeeScript
- Baza danych: SQLite lub MySQL
- Framework: Ruby on Rails

4 Funkcjonalności systemu

- Obsługa użytkowników
  - Utworzenie konta,
  - Autentykacja za pomocą OAuth,
  - Pobieranie awatarów,
  - Zapisanie ustawień i ostatniej przeglądanej pozycji.
- Wyświetlanie mapy
  - Nawigacja po mapie za pomocą myszy,
  - Wybór renderowanego fraktala,
  - Wybór kolorów dla renderowanego fraktala,
  - Wybór metody podziału obrazu do renderowania.
- Renderowanie
  - Podział obrazu na mniejsze fragmenty do renderowania na backendzie,
  - Przesyłanie żądań z aplikacji do backendu,

5 Implementacja

Praca nad projektem została podzielona na dwie części, aplikację internetową oraz rozproszone API pozwalające na renderowanie części fraktala.
W obecnej wersji system pozwala na wybór renderowanego fraktala spośród dwóch algorytmów, Mandelbrota i zbioru Julii. Oba obrazy można uzyskać zarówno w wersji monochromatycznej, jak i kolorowej. Dla każdego z fraktali można również zdefiniować ilość iteracji algorytmu oraz wymiary płytek.

6 Aplikacja internetowa

Aplikacja internetowa została utworzona przy użyciu frameworka Ruby on Rails. Do wyświetlania fraktala wykorzystana została biblioteka Leaflet. Aplikacja pobiera poszczególne części fraktala łącząc się z rozproszonym API. Obecnie zaimplementowane jest logowanie za pomocą Twitter OAuth. Dla zalogowanego użytkownika zapisywana jest ostatnie przeglądane współrzędne oraz wybrane parametry renderowanego fraktala.

Aplikacja internetowa została zaimplementowana w architekturze MVC z wykorzystaniem architektury REST.

6.1 Kontrolery

6.1.1 PagesController

Kontroler odpowiedzialny za wyświetlanie stron statycznych.

6.1.2 SessionsController

Kontroler odpowiedzialny za tworzenie i kończenie sesji użytkownika

6.1.3 UsersController

Kontroler odpowiedzialny za aktualizację ustawień fraktala dla zalogowanego użytkownika

6.2 Widoki

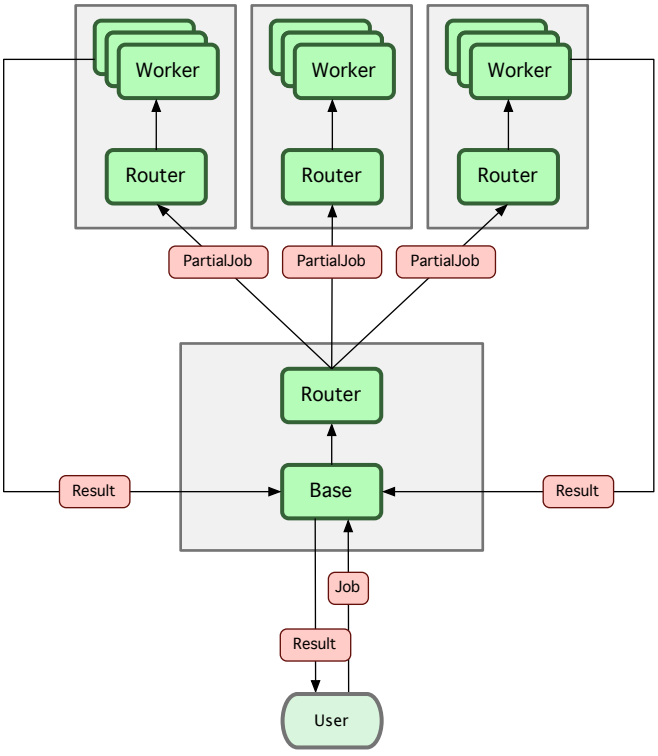
Wszelkei widoki zostały napisane przy uzyciu języka znaczników *Haml*. W celu generowania formularzy użyty został gem *SimpleForm*, a cały layout oparty o bibliotekę styli css - *Bootstrap*. Skrypty wykorzystujące *jQuery*, napisane zostały przy pomocy języka *CoffeScript*, kompilowalnego do kodu javascript.

6.3 Model

W aplikacji został wykorzystany model *User* przechowujący podstawowe dane o zarejestrowanych użytkownikach, a także ich ustawienia fraktali.

7 Rozproszone API

Biblioteka Leaflet pobiera z API obrazki o zdefiniowanych przez użytkownika rozmiarach w pikselach korzystając z URL w formacie `http://{serwer}/img/{kind}/{size}/{quality}/{x}/{y}/{z}` wypełniając pola `kind`, `size`, `quality`, `x`, `y`, `z` odpowiednimi wartościami. Następnie, zgodnie z rysunkiem 1, zapytanie jest przydzielane do odpowiednich węzłów, renderowane, a rezultat zwracany jest do klienta.



Rysunek 1: Load balancing.

7.1 REST API

API zostało zaimplementowane w języku Scala wykorzystując framework *Play 2.0*.

Zadaniem API jest przyjęcie request'u od aplikacji internetowej, sparsowanie danych, rozdzielenie zadań z wykorzysatniem load balancingu serwera nginx, a następnie zwrócenie wygenerowanych obrazów.

```
1
2  object Web extends Application with Controller {
3    implicit val timeout = Timeout(30 seconds)
4
5    def route = {
6      case GET(Path(Seg("img" :: size :: kind :: quality :: xs ::
           ys :: zs :: Nil))) => Action {
7        (for {
8          n <- parseInt(size)
9          maxIter <- parseInt(quality)
10         x <- parseInt(xs)
11         y <- parseInt(ys)
12         z <- parseInt(zs)
13       } yield {
14         println("Starting %d/%d/%d" format (x,y,z))
15
16         Async {
17           (Hardcore.base ? Render(kind, n, x, y, z, maxIter)).
               asPromise.map {
18             case FractalData(data) =>
19               println("Got fractal data %d/%d/%d" format (x,y,z)
                   )
20               val output = render(data)
21               val stream = new ByteArrayInputStream(output.
                   toByteArray)
22               println("Rendered %d/%d/%d" format (x,y,z))
23               SimpleResult(
24                 header = ResponseHeader(OK, Map(
25                   CONTENT_LENGTH -> stream.available.toString,
26                   CONTENT_TYPE -> MimeTypes.types("png")
27                 )),
```

```
28                    Enumerator.fromStream(stream)
29                  )
30              }
31            }
32          }) getOrElse NotFound
33      }
34    }
35
36    def render(data: Array[Array[Int]]) = {
37      println("rendering " + data.length)
38      val img = new BufferedImage(data.length, data.length,
            BufferedImage.TYPE_INT_RGB)
39      for {
40        i <- 0 until data.length
41        j <- 0 until data.length
42      } img.setRGB(i, j, data(i)(j))
43      val output = new ByteArrayOutputStream
44      ImageIO.write(img, "png", output)
45      output
46    }
47
48    def parseInt(s: String) = try { Some(s.toInt) } catch { case _
          => None }
49
50  }
```

7.2 Worker

Bazową klasą dla systemu generującego fraktale jest trait *Fractal*, rozszerzany przez klasy *Mandelbrot* i *Julia*, odpowiadające bezpośrednio za kształt generowanych fraktali. W celu wygenerowania monochromatycznych fraktali, zaimplementowane zostały dwa obiekty *MandelbrotGrayscale* i *JuliaGrayscale*, rozszerzające swoje pierwowzory.

```
1  trait Fractal {
2    def pixel(cx: Double, cy: Double, maxIter: Int): Color
3    def size: Double
4    def xOffset: Double
```

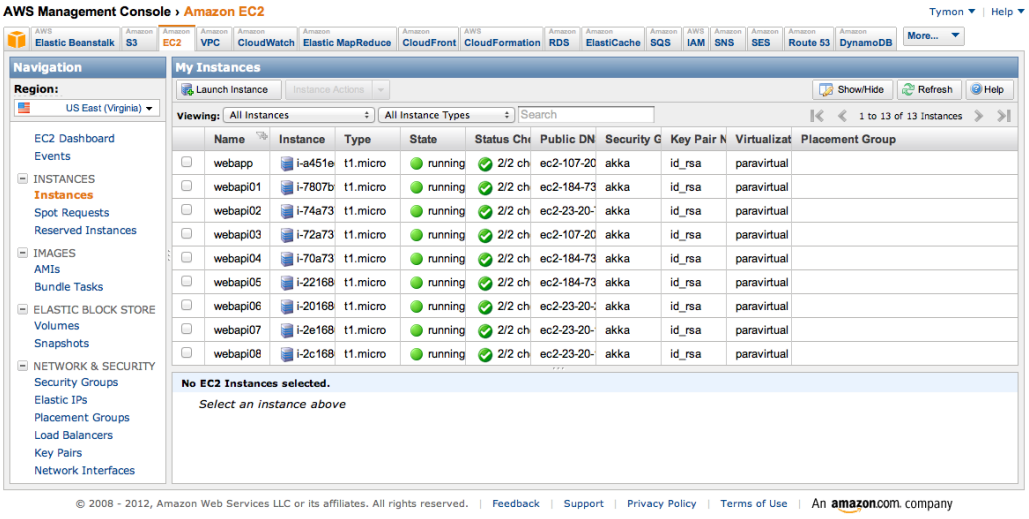
```
5   def yOffset: Double
6
7   def row(n: Int, x: Int, y: Int, z: Int, row: Int, maxIter: Int
        ): Array[Int] = {
8     (0 until n).map { y0 => calculate(n, x * n + row, y * n + y0
          , z, maxIter).getRGB }.toArray
9   }
10
11  def calculate(n: Int, x: Int, y: Int, z: Int, maxIter: Int) =
        {
12    val m = math.pow(2, z) * n
13
14    val cx = (x/m) * size - xOffset
15    val cy = (y/m) * size - yOffset
16
17    pixel(cx, cy, maxIter)
18  }
19 }
20
21 object Fractal {
22   val fractals = Map(
23     "mandelbrot-color" -> Mandelbrot,
24     "mandelbrot-gray" -> MandelbrotGrayscale,
25     "julia-color" -> Julia,
26     "julia-gray" -> JuliaGrayscale
27   )
28   def apply(kind: String) = fractals(kind)
29 }
30
31
32 object Mandelbrot extends Mandelbrot
33 class Mandelbrot extends Fractal {
34   def size = 3.5
35   def xOffset = 2.5
36   def yOffset = 1.75
37
38   def pixel(cx: Double, cy: Double, maxIter: Int) = {
39     var (i, zx, zy) = (0, 0.0, 0.0)
40
```

```
41      while (zx * zx + zy * zy < 4 && i < maxIter) {
42        val temp = zx * zx - zy * zy + cx
43        zy = 2 * zx * zy + cy;
44        zx = temp;
45        i = i + 1;
46      }
47
48      if (i == maxIter) Color.BLACK
49      else Color.getHSBColor(i / 20.0F, 1F, 1F)
50    }
51  }
52
53  object Julia extends Julia
54  class Julia extends Fractal {
55    def size = 4.0
56    def xOffset = 2.0
57    def yOffset = 2.0
58
59    def pixel(cx: Double, cy: Double, maxIter: Int) = {
60      val (a,b) = (-0.8, 0.156)
61      var (i, zx, zy) = (0, cx, cy)
62
63      while (zx * zx + zy * zy < 4 && i < maxIter) {
64        val temp = zx * zx - zy * zy + a
65        zy = 2 * zx * zy + b
66        zx = temp
67        i = i + 1
68      }
69
70      if (i == maxIter) Color.BLACK
71      else Color.getHSBColor(i / 20.0F, 1F, 1F)
72    }
73  }
74
75  object MandelbrotGrayscale extends Mandelbrot with ColorUtils {
76    override def pixel(cx: Double, cy: Double, maxIter: Int) =
          grayscale(super.pixel(cx, cy, maxIter))
77  }
78
```

```
79 object JuliaGrayscale extends Julia with ColorUtils {
80   override def pixel(cx: Double, cy: Double, maxIter: Int) =
       grayscale(super.pixel(cx, cy, maxIter))
81 }
```

8 Wdrożenie

Aplikacja została wdrożona na klastrze obliczeniowym składającym się z dziewięciu maszyn wirtualnych. Ze względu na łatwą konfigurację, jako host maszyn wirtualnych została wybrana platforma Amazon EC2.



Rysunek 2: Klaster obliczeniowy.

9 Testy

W pierwszej wersji systemu, generowanie fraktali polegało na rozdzielaniu pomiędzy workery obliczanie wierszy lub kolumn obrazka. Był to wysoce niewydajny system, który mimo zwiększania ilość workerów, nie poprawiał szybkości działania. Problemem był proces generowania ostatecznego obrazu, który zajmował najwięcej czasu.

Rysunek 3 pokazuje przykładowe działanie systemu. Licząc od góry, piąta stacja odpowiedzialna za renderowanie obrazków, wykorzystuje swoje zasoby procesora i pamięci w maksymalnym stopniu, czego nie można powiedzieć o reszcie workerów, odpowiedzialnych za obliczenia.

W obecnym systemie, proces rozdzielania zadań opiera się o standardowy system lefleta, czyli podziału obrazu na kwadratowe płytki, a każda stacja robocza zwraca wygenerowany fragment. Jest to znacznie bardziej wydajny system pozwalający na dobranie większej liczby parametrów.

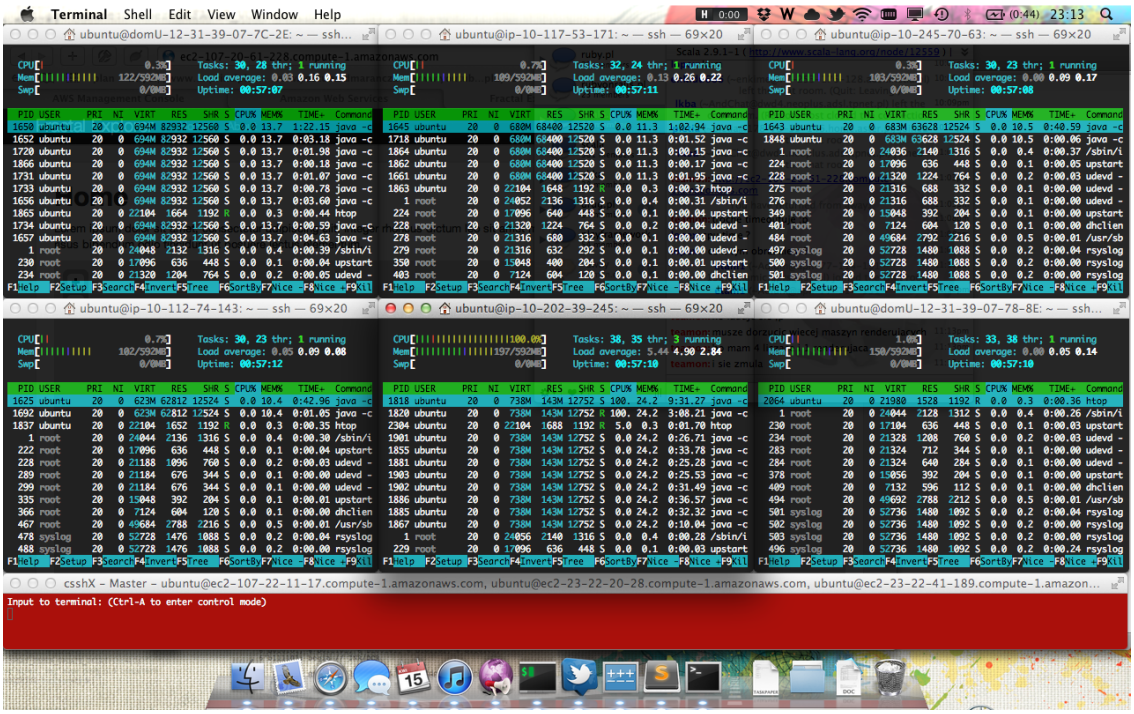
Bezpośredni wpływ na szybkość generowania obrazów ma ilość iteracji oraz wielkość płytki. Dla większych rozmiarów płytek następuje mniej operacji renderowania obrazów png, co może dać zauważalny efekt przy generowaniu fraktala.

Rysunek 4 przedstawia wykres zależności czasu generowania jednej płytki od jej wielkości, dla róznej ilości iteracji algorytmu. Jak widać mimo, iż czas generowania płytki o rozmiarze 32x32 jest najkrótszy, jest to najmniej ekonomiczna forma generowania fraktala. By stworzyć obrazek o wielkoci 256x256, musielibyśmy wygenerować 64 płytki, co dla pojedynczego workera zwiększa czas dziesięciokrotnie. To zastosowanie miałoby sens, tylko przy bardzo dużej liczbie workerów.

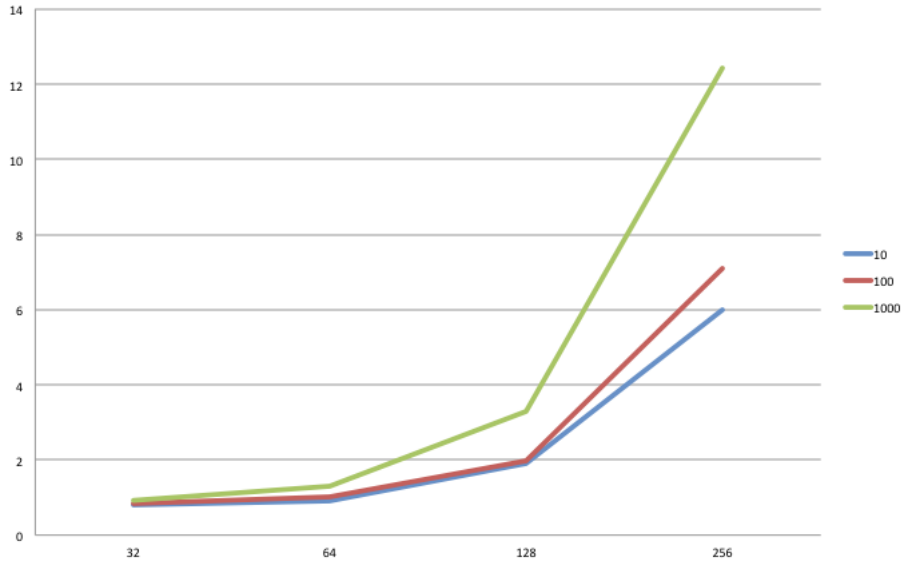
Preprowadzenie idealnych testów sprawdzających wydajność naszego systemu dla różnych parametrów jest zadaniem niemożliwym, z powodu braku odpowiedniego środowiska testowego. Nasza aplikacja została wdrożona na

serwerach amazon, co powoduje, że poważny wpływ na wyniki ma aktualna prędkość łącza, cache przeglądarki, aplikacji oraz serwera.

Zwiększenie liczby workerów, powoduje liniowe zmneijszanie się czasu wykonywania zadania. Nie możemy jednak określić w miarę dokładnych wartości, ponieważ wpływ łącza internetowego i jakość połączenia w danej chwili, ma bardzo duże znaczenie na otrzymane wyniki.



Rysunek 3: Pierwszy system



Rysunek 4: Wykres czasu generowania płytki w zależności od jej rozmiaru w ms