



POLITECHNIKA KRAKOWSKA im. T. Kościuszki
Wydział Mechaniczny
Instytut Informatyki



Kierunek studiów: Informatyka
Specjalność : Informatyka przemysłowa

STUDIA STACJONARNE

PRACA DYPLOMOWA

INŻYNIERSKA

Oskar Kapusta

**MOBILNY SYSTEM POMIARU CZASU
W ZAWODACH NARCIARSKICH**

**MOBILE TIMING SYSTEM FOR
SKIING COMPETITIONS**

Promotor:
prof. dr hab. inż. Leszek Wojnar

Kraków, rok akad. 2012/2013

Autor pracy: Oskar Kapusta

Nr pracy:

OŚWIADCZENIE O SAMODZIELNYM WYKONANIU PRACY DYPLOMOWEJ

Oświadczam, że przedkładana przeze mnie praca dyplomowa magisterska/inżynierska* została napisana przeze mnie samodzielnie. Jednocześnie oświadczam, że ww. praca:

- 1) nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym, a także nie zawiera danych i informacji, które uzyskałem w sposób niedozwolony,
- 2) nie była wcześniej podstawą żadnej innej procedury związanej z nadawaniem tytułów zawodowych, stopni lub tytułów naukowych.

Jednocześnie przyjmuję do wiadomości, że w przypadku stwierdzenia popełnienia przeze mnie czynu polegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzej pracy, lub ustalenia naukowego, właściwy organ stwierdzi nieważność postępowania w sprawie nadania mi tytułu zawodowego (art. 193 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym, Dz.U. z 2012 r. poz. 572).

.....
data i podpis

* niepotrzebne skreślić

Uzgodniona ocena pracy:

.....
podpis promotora

.....
podpis recenzenta

.....
podpis dyrektora instytutu
ds. dydaktyki

Spis treści

1	Cel i zakres pracy	4
2	Wstęp	4
3	Realizacja tematu	6
3.1	Architektura	6
3.2	Implementacja	10
3.2.1	Aplikacja	10
3.2.2	Worker	10
3.3	Obudowa ochronna	10
4	Wnioski	10

1 Cel i zakres pracy

Przedmiotem niniejszej pracy jest budowa mobilnego systemu pomiaru czasu dla zawodów narciarskich. System składa się z dwóch bramek: startowej oraz końcowej, które wykorzystują wiązkę laserową w celu uchwycenia dokładnego momentu przejechania zawodnika przez bramkę. Niniejsza praca będzie się składać z trzech głównych części. Pierwsza z nich poświęcona zostanie architekturze systemu i wykorzystanych technologiach, a w szczególności komunikacji pomiędzy komponentami oraz podstawą teoretycznym działania. Druga część tej pracy zawiera opis implementacji, zaś trzecia część jest poświęcona umieszczeniu systemu w obudowie ochronnej.

2 Wstęp

Obecnie na rynku istnieją podobne systemy do tego, którego budowę ta praca przedstawia, jednak często kosztują tysiące złotych. Praca ta jest próbą stworzenia rozwiązania spełniającego podobne zadanie do ww. systemów korzystając z ogólnie dostępnych podzespołów za nie wielkie pieniądze.

Jako szkielet systemu zostało wybrane Raspberry Pi - platforma komputerowa stworzona przez Raspberry Foundation. W momencie premiery (29 lutego 2012) model B użyty w tej pracy miał cenę początkową US\$ 35. Raspberry Pi oparte jest o chip BCM2835 zawierający procesor ARMv6. Urządzenie działa pod kontrolą dystrybucji systemu Linux Raspbian będącą portem Debiana Wheezy koniecznym z powodu niekompatybilności (oficjalne wydanie Debiana Wheezy na platformę armhf jest dostępne jedynie z procesorami ARMv7 lub nowszymi).

Obie aplikacje (startowa i końcowa) zostały napisane przy użyciu języka Ruby 2.1.0 oraz dla aplikacji startowej stworzony został interfejs web umożliwiający wprowadzanie zawodników oraz podgląd wyników, jak również import oraz eksport. Napisany został przy użyciu CoffeeScript oraz biblioteki JavaScript Backbone.js. CoffeeScript jest językiem inspirowanym elegancką składnią Ruby i Pythona, który kompiluje się do JavaScriptu. Backbone.js natomiast zapewnia strukturę aplikacji.

W momencie pisania kodu aplikacji autor nie zdawał sobie jeszcze sprawy z pewnych dobrych praktyk pisania kodu JavaScript, zatem w kodzie znajdują się błędy, których należy unikać. Z tego powodu ta praca będzie się też starać je pokazać, na czym one polegają i jakie trudności w późniejszym utrzymaniu kodu podowują tak aby czytelnik był ich świadom i sam ich nie popełniał.

W pracy zostały użyte różne gem-y - programy i biblioteki menedżera pakietów RubyGems, których lista wraz z licencjami zostanie przedstawiona na końcu tej pracy.

Kompletny kod źródłowy pracy można znaleźć pod adresami:

<https://github.com/okapusta/skirace>

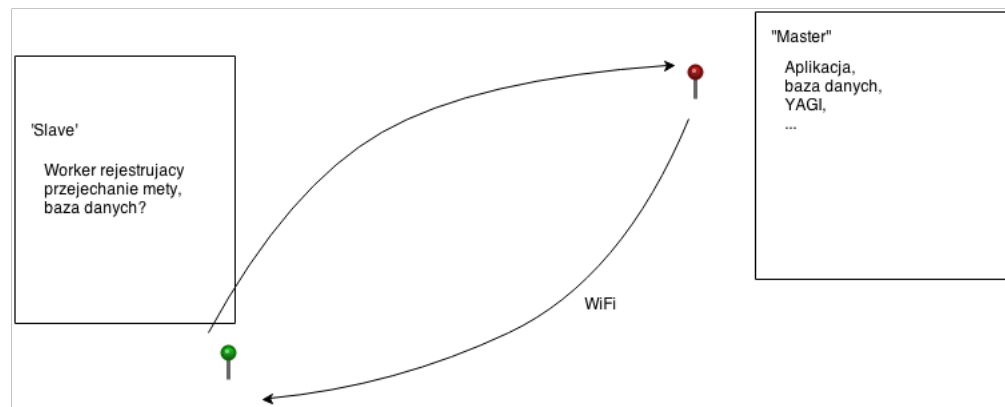
<https://github.com/okapusta/skirace-worker>

3 Realizacja tematu

3.1 Architektura

System, którego ta praca dotyczy został zbudowany w myśl modelu master/slave gdzie mastertem jest aplikacja początkowa. To tam znajduje się interface, baza danych oraz serwer Memcached. W momencie uruchomienia aplikacji, oprócz startu serwera serwującego aplikację WEB tworzony jest nowy wątek zawierający event-loop, który rejestruje przecięcie wiązki lasera, a kiedy to się stanie ustawia godzinę tego zdarzenia w cache.

Zadaniem slavea - workera - jest jedynie pobranie czasu startu z Memcached, obliczenie czasu końcowego oraz wypisanie go na ekranie LCD linii końcowej w momencie finishu.



Komunikacja pomiędzy dwiema bramkami odbywa się poprzez WiFi.

Memcached

Memcached jest to rozproszony system buforowania pamięci podręcznej oryginalnie zaprojektowany na potrzeby serwisu LiveJournal. Pozwala on na przechowywanie obiektów w pamięci RAM przy pomocy kluczy. W aplikacji wykorzystany został w celu przechowania czasów startu i finishu. Serwer Memcached jest uruchomiony tylko na jednej (startowej) bramce, worker natomiast łączy się z cache przy pomocy klienta i może pobierać lub pisać dane do pamięci drugiego urządzenia.

Dependency Injection

W niniejszej pracy został wykorzystany wzorzec projektowy Dependency Injection (DI) polegający na usuwaniu bezpośrednich zależności klas na rzecz wstrzykiwania ich w czasie konstruowania obiektu. Do osiągnięcia tego celu

i uproszczenia DI użyty został gem Dependor, udostępniający prosty DSL (eng. Domain Specific Language) przeznaczony do tego celu.

Poniżej zamieszczone są listingi przedstawiające normalne wstrzykiwanie zależności w Ruby oraz z wykorzystaniem gem-u Dependor.

(a) Ruby

```
1 class A
2   attr_reader :obj
3
4   def initialize(obj)
5     @obj = obj
6   end
7
8   def do_b
9     obj.do_sth
10  end
11 end
12
13 a = A.new(B.new)
14 a.do_b
```

(b) Dependor

```
class A
  takes :b

  def do_b
    b.do_sth
  end
end

a = A.new
a.do_b
```

Chociaż Dependency Injection przy użyciu gemu Dependor wydaje się wygodne w aplikacjach ten gem należy raczej taktować jako proof of concept.

Sprockets

Do kompilacji CoffeeScript oraz szablonów .haml (Haml Coffee Assets) został użyty gem Sprockets zawierający preprocessory dla języków takich jak CoffeeScript czy SCSS. Sprockets w środowisku developerskim pozwala na kompilację assetów (JavaScriptów i CSSów) 'w locie', natomiast w środowisku produkcyjnym assety są prekompilowane. Sprockets pozwala również na minifikację zasobów to jest zastąpienie nazw funkcji pojedynczymi zaktami w celu zmniejszenia rozmiaru kodu, który musi zostać pobrany przez przeglądarkę.

Sprockets działa w kontekście Rack - minimalnego interfejsu dla aplikacji Ruby do komunikacji z popularnymi serwerami WWW. Zasoby serwowane przez sprockets są montowane w pliku config.ru, będącym plikiem konfiguracyjnym dla interfejsu Rack poprzez który aplikacja jest uruchamiana. To tutaj tworzony jest wątek rejestrujący przejechanie linii startu oraz tutaj montowana jest ścieżka serwera WWW '/' tak aby pokazywała na aplikację Sinatra.

Samo sprockets jest montowane w następujący sposób:

```
1 map Skirace::Application.assets_prefix do
2   run Skirace::Application.sprockets
3 end
```

Listing 1: config.ru

Wykorzystane tutaj zmienne klasowe (`assets_prefix`, `sprockets`) aplikacji są definiowane w pliku `application.rb` zawierającym klasę z konfiguracją.

```
1 set :assets_prefix, '/assets'
2 set :assets_path, File.join(public_folder, assets_prefix)
3
4 set :sprockets, Sprockets::Environment.new(root)
```

Listing 2: Ustawienie zmiennych Sprockets

```
1 sprockets.append_path File.join(root, 'app', 'assets', 'javascripts')
2 sprockets.append_path File.join(root, 'app', 'assets', 'stylesheets')
3 sprockets.append_path File.dirname(HamlCoffeeAssets.helpers_path)
```

Listing 3: Przeszukiwane foldery

Powyższe listingi umieszczają skompilowane pliki z wyznaczonych ścieżek w folderze `public/assets` pod nazwami `application.js` oraz `application.css`. Pliki te zawierają jedynie assety załączone poleceniem `require` w pliku `app/assets/application.js`.


```

1  //= require jquery-1.10.2.min
2  //= require jquery.ui.widget
3  //= require jquery.fileupload
4  //= require jquery.cookie
5  //= require bootstrap.min
6  //= require underscore-min
7  //= require backbone-min
8  //= require hamlcoffee
9  //= require app
10 //= require_tree ./lib
11 //= require_tree ./models
12 //= require_tree ./collections
13 //= require_tree ./templates
14 //= require_tree ./services
15 //= require_tree ./views
16 //= require_tree ./routers
17 Skirace.init();

```

Listing 4: app/assets/javascripts/application.js

HAML

W projekcie został użyty HAML (HTML Abstraction Markup Language), który sprawia że kod jest przyjemniejszy do pisania i czytania. W HAML używa się jedynie tagów otwierających a o tym jak osadzone są elementy decyduje indentacja. HAML posiada też skróty na sekcje div o podanym id lub klasie.

(c) HAML

```

1  .klasa
2
3  #id
4    %p
5      Paragraf

```

(d) HTML

```

<div class="klasa"></div>

<div id="id">
  <p>Paragraf</p>
</div>

```

3.2 Implementacja

3.2.1 Aplikacja

3.2.2 Worker

3.3 Obudowa ochronna

4 Wnioski