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

Przedmiotem niniejszej pracy jest budowa mobilnego systemu pomiaru czasu dla zawodów narciarskich. System składa się z dwóch bramek: startowej oraz końcowej, które wykorzystują wiązke laserową w celu uchwicenia 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 stowrzenia 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 Fundation. W momencie premiery (29 luty 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 braku kompatybilności (oficjalne wydanie Debiana Wheezy na platformę armhf działa jedynie z procesorami ARMv7 lub nowszymi).

Obie aplikacjie (startowa i końcowa) zostały napistane przy użyciu języka Ruby 2.1.0 oraz dla aplikacji startowej stworzony został interface web umożliwiający wprowadzanie zawodników oraz podgląd wynikow, jak równiez import oraz eksport. Napisany został on 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 pratyk 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 gemy - programy i biblioteki managera paczek RubyGems, których lista w raz z licencjami zostanie przedstawiona na końcu tej pracy.

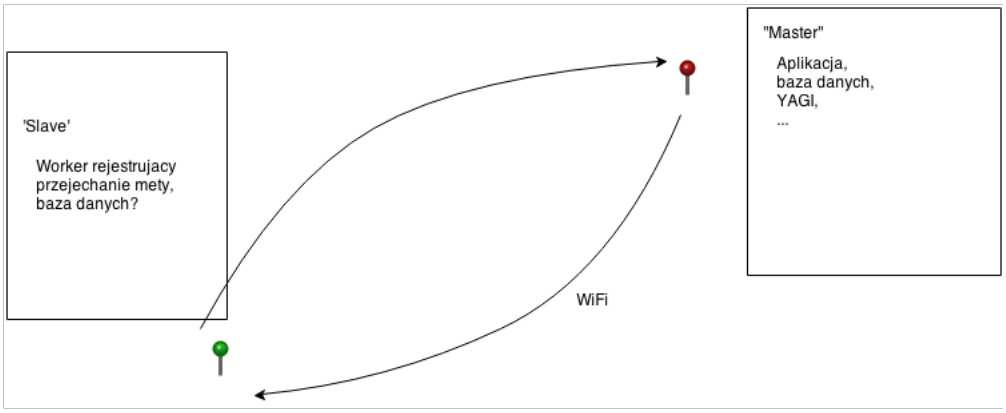
Kompletny kod źródłowy pracy można znaleść pod adresami:
https://github.com/okapusta/skirace
https://github.com/okapusta/skirace-worker

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 sie 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.



Komunikacjia pomiędzy dwiema bramkami odbywa się poprzez WiFi.

3.2 Wybrane technologie

Memcached

Memcached jest to rozproszony system buforowania pamięci podręczniej oryginalnie zaprojektowany na potrzeby serwisu LiveJournal. Pozwala on na przechowywanie obiektów w pamięci RAM przy pomocy kluczy (key-value store). 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 ninejszej pracy został wykorzystany wzorzec projektowy Dependency Injection (DI) polegający na usuwaniu bespośrednich 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 normlne wstrzykiwanie zależności w Ruby oraz z wykorzystaniem gemu Dependor.

(a) Ruby

```ruby
class A
  attr_reader :obj

  def initialize(obj)
   @obj = obj
  end

  def do_b
    obj.do_sth
  end
end

a = A.new(B.new)
a.do_b
```

(b) Ruby + Dependor

```ruby
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 ten gem należy raczej taktować jako proof of concept.

Sprockets

Do kompilacji CoffeeScript oraz szablonów .hamlc (Haml Coffee Assets) został użyty gem Sprockets zawierający preprocessory dla języków takich jak CoffeeScript czy SCSS. Sprockets w środowisku developerskim pozwala na kompilacje assetów (JavaScriptów i CSSów) 'w locie', natomiast w środowisku produkcyjnym assety są prekompilowane. Sprockets pozwala również na minifikacje zasobów to jest zastąpienie nazw funkcji czy zmiennych pojedyńczymi znakami w celu zmiejszenia 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:

```
map Skirace::Application.assets_prefix do
  run Skirace::Application.sprockets
end
```

Listing 1: config.ru

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

```
set :assets_prefix, '/assets'
set :assets_path, File.join(public_folder, assets_prefix)

set :sprockets, Sprockets::Environment.new(root)
```

Listing 2: Ustawienie zmiennych Sprockets

```
sprockets.append_path File.join(root, 'app', 'assets', 'javascripts')
sprockets.append_path File.join(root, 'app', 'assets', 'stylesheets')
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 plikach app/assets/javascripts/application.js i app/assets/stylesheets/application.css.

```
//= require jquery-1.10.2.min
//= require jquery.ui.widget
//= require jquery.fileupload
//= require jquery.cookie
//= require bootstrap.min
//= require underscore-min
//= require backbone-min
//= require hamlcoffee
//= require app
//= require_tree ./lib
//= require_tree ./models
//= require_tree ./collections
//= require_tree ./templates
//= require_tree ./services
//= require_tree ./views
//= require_tree ./routers
Skirace.init();
```

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

```
/*
 *= require bootstrap.min
 *= require bootstrap-responsive.min
 *= require app.css
 *= require_tree .
 */
```

Listing 5: app/assets/stylesheets/application.css

Twitter Bootstrap

Twitter Bootstrap jest frameworkiem front-end dostarczającym style CSS oraz kod JavaScript, który za zadanie ma przyśpieszenie front-endu aplikacji. Bootstrap dostarcza zestaw klas HTML, które posiadają określone style. W tej pracy wykorzystany został Twitter Bootstrap w wersji drugiej. Aktualna wersja Bootstrapa to 3.1.1.

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 elelemny

decyduje indentacja. HAML posiada też skróty na sekcje div o podanym id lub klasie.

(c) HAML

```
.klasa

#id
  %p
    Paragraf
```

(d) HTML

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

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

Backbone.js

Backbone.js jest lekką biblioteką JavaScript nadającą strukturę aplikacją JavaScript. Typowe użycie tej biblioteki jest to zazwyczaj trio pomiędzy samym backbonem a jQuery i underscore.js.

W tej pracy został też użyty gem haml_coffee_assets pozwalający na pisanie szablonów Backbone w języku HAML z osadzonym CoffeeScriptem (podobnie jak w eRuby).

Biblioteka Backbone.js stara się odtworzyć to co jest po stronie serwera w modelu MVC na stronę klienta (przeglądarki) i JavaScriptu. Model odzwierciedla zasób na serwerze i jest odpowiednikiem modelu po stronie serwera. Kolekcja jest grupą modeli pobieraną z serwera, na kolekcji zdefiniowane są zdarzenia (eng. events), które ponownie renderują widok np. kiedy element zostanie dodany do kolekcji. W Backbone Router odpowiedzialny jest za tłumaczenie adresów URL na widoki oraz za obsługę historii przeglądarki. Widok jest odpowiednikiem kontrolera znanego z Rails. Odpowiada na zdarzenia w oknie przeglądarki takie jak kliknięcie myszką i podejmuje odpowiednie akcje. Szablony natomiast są, w przypadku tej pracy, kodem HAML używanym przez widoki do rendowania treści aplikacji.