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 akademicki 2012/2013

Autor pracy: **Oskar Kapusta**

Nr pracy: **…….**

OŚWIADCZENIE O SAMODZIELNYM WYKONANIU PRACY DYPLOMOWEJ

Oświadczam, że przedkładana przeze mnie praca dyplomowa 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

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 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 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 również 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 pracy zostały użyte różne *gemy* - programy i biblioteki managera paczek *RubyGems*, których lista w raz z licencjami zostanie przedstawiona na końcu tej pracy.

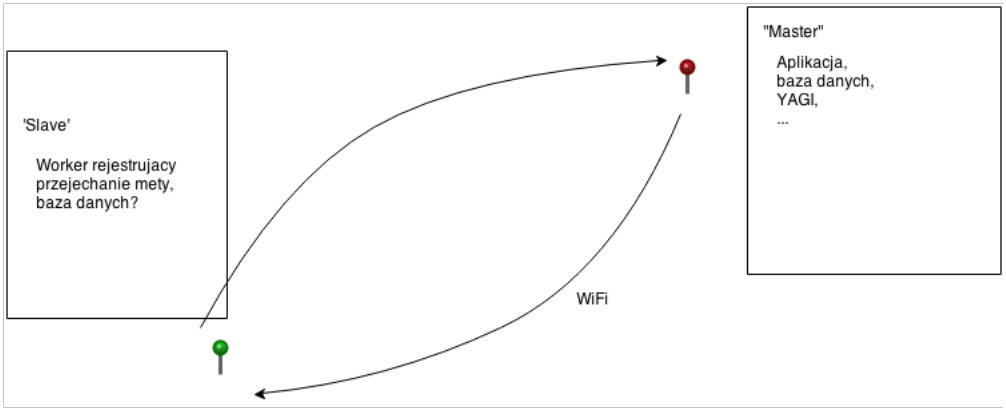
Kompletny kod źródłowy pracy można znaleść pod adresami:
`https://github.com/okapusta/skirace`
`https://github.com/okapusta/skirace-worker`

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.



Rysunek 1: Architektura systemu

Komunikacjia pomiędzy dwiema bramkami odbywa się poprzez WiFi.

3.2 Wybrane technologie

3.2.1 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.

3.2.2 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 zestaw metod i modułów 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
```

3.2.3 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:

```ruby
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 aplikacji Sinatra.

```
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/applica

```
//= 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

3.2.4 Twitter Bootstrap

Twitter Bootstrap jest frameworkiem front-end dostarczającym style CSS oraz kod JavaScript, który za zadanie ma przyśpieszenie budowę 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.

3.2.5 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>
```

3.2.6 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.

3.3 Komunikacja

Oba urządzenia komunikują się ze sobą przy pomocy WiFi w trybie pracy Ad Hoc. Jako karta sieciowa została wybrana karta USB TP-Link TL-WN722N ponieważ posiada antenę o zysku 4dBi, którą można odkręcić oraz zamontować mocniejszą antenę. Poniższy listing przedstawia konfiguracje urządzenia do pracy w trybie Ad Hoc.

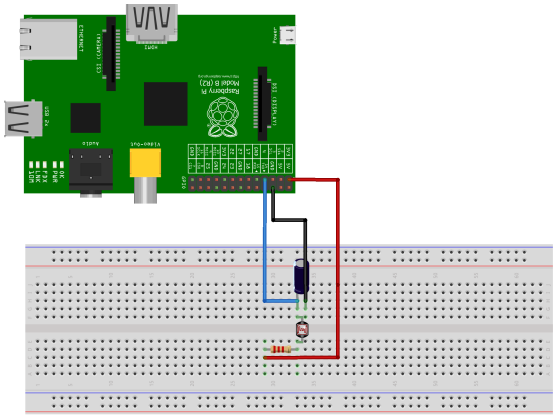
```
1 auto wlan0
2 iface wlan0 inet static
3   address 192.168.1.1
4   netmask 255.255.255.0
5   wireless-channel 1
6   wireless-essid SKIRACE
7   wireless-mode ad-hoc
```

Listing 6: /etc/network/interfaces

Plik */etc/network/interfaces* drugiego urządzenia został skonfigurowany analogicznie. Różni sie on jedynie adresem IP ustawionym na *192.168.1.2*. Urządzenia ustawione w trybie pracy Ad Hoc wykrywają się nawzajem i tworzą sieć o SSID SKIRACE.

3.4 Podstawy działania

Ten rozdział ma na celu przedstawienie budowy oraz podstaw teoretycznych działania czujnika rejestrującego przecięcie wiązki lasera. Składa sie on za fotorezystora, rezystora oraz kondensatora. Poniższy rysunek przedstawia jego budowę.



Rysunek 2: Sensor

W układzie o którym mowa w tym rozdziale resyztor $2.2k\Omega$ działa jako zabezpieczenie przed zbyt dużym napięciem skierowanym na port GPIO. Podłączony jest do fotorezystora, którego rezystancja jest niska kiedy świeci na niego wiązka lasera. Kondensator $1\mu F$ jest ładowany a kiedy przekroczy wartość graniczną wynoszącą około 2V pin GPIO rejestruje wartość HIGH. Działanie to jest wykorzystane w kodzie aplikacji gdzie mierzony jest czas potrzebny na naładowanie kondensatora. Kiedy czas jest krótki znaczy to że na fotorezystor pada wiązka lasera, kiedy się zwiększy znaczy to że laser został przecięty tj. zawodnik przejechał przez start lub metę.

3.5 Implementacja

Ninejszy rozdział zostanie poświęcony implementacji. Składa się on z dwóch podrozdziałów: Aplikacja, który opisuję budowę aplikacji startowej oraz rozdziału Worker opisującego worker rejestrujący przejechanie mety.

3.5.1 Aplikacja

Tak jak była o tym mowa we wstępie do ninejszej pracy aplikację startową można podzielić na dwa osobne komponenty. Jeden stanowi aplikacja napisana we frameworku Sinatra, która udostępnia API dla aplikacji front-end do komunikacji z bazą danych. Ona także serwuje skompilowane zasoby oraz dostarcza połączenia z Memcached. W momencie uruchomienia aplikacji jest też tworzony nowy wątek, którego zadaniem jest zarejestrowanie startu oraz ustawienie czasu tego zdarzenia w Memcached co przedstawia listing 7. Aplikacja front-end natomiast dostarcza interfejs dostępny przez przeglądarke internetową służący to interakcji z back-endem.

```
require './application'

t = Thread.new(Injector.new(OpenStruct.new({}))) do |injector|
  while true
    injector.capacitor.discharge(injector.options.capacitor.pin)
    sleep 0.01; reading = 0;

    while injector.gpio.read(injector.options.capacitor.pin) == LOW
     reading += 1
    end

    if reading > injector.options.activation_threshold
      injector.caching_service.set('start_time', Time.now)
    end
  end
end
t.abort_on_exception = true
```

Listing 7: Utworzenie wątku w pliku config.ru

Wykorzystana tutaj klasa Injector, ma za zadanie budowanie obiektów używających dependency injection w raz z wszelkimi zależnościami przy pomocy gemu Dependor.

W swoim konstuktorze oczekuje obiektów Sinatra dlatego tutaj inicjalizowana jest pustym objektem OpenStruct. OpenStruct odpowiada na dowolne metody oraz zwraca *nil* jeśli nazwa metody wywołanej na rzecz tego obiektu nie zawiera się w haszu, który został użyty do inicjalizacji. Obiekt klasy Injector jest przekazywany w bloku, gdzie później dostarcza obiektów takich jak *capacitor*, którego klasa jest zdefiniowana w pliku *app/services/components/capacitor.rb*.

W klasie Injector są definiowane moduły które mają być przeszukane w poszukiwaniu klas, które maja być wstrzykiwane co przedstawia listing 8. Nazwy przyjmowane w argumentach metody *takes* oraz nazwy metod wywołanych na obiekcie Injector odpowiadają nazwą plików wstrszykiwanych klas. Warunkiem tego jest odpowiednie nazewnictwo klas i plików zgodnie z konwencjami przyjętymi w programowaniu Ruby.

W Ruby przyjęło się nazywanie nazw klas i modułów z dużej litery CamelCase a pliki zawierające te klasy powinny mieć taką samą nazwę jak klasa tylko zapisaną w snake_case z rozszerzeniem *.rb.

```
1  class Injector
2    include Dependor::AutoInject
3    include Dependor::Sinatra::Objects
4
5    look_in_modules ::Repositories,
6    ::Connections,
7    ::RaspberryPi,
8    ::Presenters,
9    ::Uploaders,
10   ::Parsers,
11   ::Components
12
13   def initialize(objects = nil)
14     sinatra_objects(objects)
15
16     io.wiringPiSetup
17   end
18 end
```

Listing 8: Klasa injector odpowiedzialna za tworzenie obiektów z wykorzystaniem DI

Powyższy listing przedstawia jedynie fragment klasy Injector ponieważ jest zbyt długa aby ją całą zamieścić.

Pozostałe metody klasy zawierają jedynie obiekty lub nazwy stałych, które są wstrzy-

kiwane. W konstruktorze klasy Injector jest też inicjalizowane GPIO (General Purpose Input Output).

Obsługę GPIO zapewnia WiringPi-Ruby—wrapper Ruby popularnej biblioteki C WiringPi. W klasie Injector defniuje go metoda pokazana na listingu 9

```
def io
  Wiringpi
end
```

Listing 9: WiringPi

Metoda *io* jest wstrzykiwana do klasy RaspberryPi::Gpio, która jest odpowiedzialna za ustawienie odpowiedniego kierunku pinu (INPUT—OUTPUT) oraz napisanie lub pobranie z niego.

```
class RaspberryPi::Gpio
  takes :io

  def read(pin)
    mode(pin, INPUT)
    io.digitalRead(pin)
  end

  def write(pin, value)
    mode(pin, OUTPUT)
    io.digitalWrite(pin, value)
  end

  private

    def mode(pin, mode)
      io.pinMode(pin, mode)
    end
end
```

Listing 10: app/services/rasperry_pi/gpio.rb

Ta sama klasa 10 użyta jest w workerze. Jest to jedna z zalet obiektowości a w szczególności dependency injection ponieważ klasa nie ma żadnych bezpośrenich zależności i może być łatwo przeniesiona do innej aplikacji. Klasa *RaspberryPi::Gpio* jest w tej aplikacji użyta jedynie w klasie *Components::Capacitor* 11.

```ruby
class Components::Capacitor
  takes :gpio

  def discharge(pin)
    gpio.write(pin, LOW)
  end
end
```

Listing 11: app/services/components/capacitor.rb

Zadaniem tej klasy jest napisanie wartości LOW na pin w celu rozładowania ładunku na kondensatorze.

Dzieje się to na początku pętli *while* w wątku z listingu 7. Następnie w tej samej pętli *śpimy* przez 1ms oraz odczyt ustawiany jest na zero. W kolejnej pętli *while* pin GPIO jest ustawiany na wejście, pobierana jest z niego wartość a kiedy kondensator przekroczy wartorść graniczną jest zwracany jest odczyt inkrementowany w tej pętli.

Jeśli jego wartość przekroczy zadany czas potrzebny na naładowanie kondensatora (tutaj próg aktywacji) w Memcached ustawiany jest dokładny czas tego zdarzenia pod kluczem *start_time*. Odpowiedzialna jest za to klasa CachingService przedstawiona na listingu 12.

```ruby
class CachingService
  takes :memcache_connection

  def get(key)
    memcache_connection.client.get(key)
  end

  def set(key, value)
    if memcache_connection.client.set(key, value)
      return value
    end
  end

  def fetch(key, &block)
    result = get(key)
    return result if result

    set(key, yield)
  end
end
```

Listing 12: app/services/caching_service.rb

Klasa w konstruktorze jako argument przyjume obiekt klasy MemcacheConnection pokazanej na listingu 14. Metody tej klasy *get* oraz *set* mają oczywiste działanie zatem zostaną pominięte. Warto jednak zwtócić uwagę na metodę fetch która próbuje pobrać wartość z cache a kiedy nic nie znajduje się pod przekazanym kluczem wywołuje metodę *set* ustawiającą klucz wartością zwracaną przez przekazany blok. Przykładowe użycie tej metody przedstawiono poniżej.

```ruby
caching_service.fetch('query') do
  User.find(id: @id)
end

```

Listing 13: Przykład wykorzystania metody fetch.

Klasa Connections::MemcaheConnection przyjmuje tutaj w metodzie takes symbol *:dalli_client*, który jest nazwą nazwą metody zawartej w klasie Injector zwracającą klasę *Dalli::Client*.

```
class Connections::MemcacheConnection
  takes :options, :dalli_client


  def client
    @client ||= dalli_client.new(options.memcache_server,
      options.memcache_client)
  end
end
```

Listing 14: app/services/connections/memcache.rb

3.5.1.1 API

API dla aplikacji Backbone (interfejsu) zostało napisane we frameworku Sinatra. Framework ten został zaprojektowany do pisania niewielkich, lekkich aplikacji web i został wybrany ponieważ autor uznał że Ruby on Rails (najpopularnieszy framework Ruby) jest zbyt rozbudowany i uruchomienie aplikacji RoR na RaspberryPi mogło by zużyć zbyt wiele zasobów urządzenia. Sama aplikacja Sinatra uruchamiana jest na porcie 9292. Z tego powodu znajuję się za proxy — serwerem Nginx, który przekierowuje zapytania przychodzące na port 80 na aplikację.

Typowo aplikacjia Sinatra składa się z jednej klasy. W celu wprowadzenia ładu i składu autor zdecydował się ją rozbić na miejsze pliki. Jako pierwszy załączany jest plik *application.rb*, który min. zawiera deklarację modułów, konfigurację Sprockets oraz konfigurację Warden, która zostanie omówiona w rozdziale 3.5.1.2. Na końcu załączane są pozostałe piliki zawierające poszczególne end-pointy, które monkey patchują klasę. Znajdują się w folderze *app/routes* i zostaną przedstawione poniżej przy pomocy zapytań curl i odpowiedzi. W celu autentykacji najpierw jest POSTowany login. Potem możemy autentykować się przy pomocy cookie.

```
curl -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -X POST http://localhost:9292/login \
  -d '{"username":"admin", "password":"password"}' \
  -c cookie
```

W przypadku udanego logowania serwer odpowiada JSONem

```
{
  "user": {
    "authenticated":true,
    "auth_token":"4a59a1427841af26"
    }
}
```

Contestants

W pliku *app/routes/contestants.rb* (listing **??**) zawarte są 2 endpointy:
GET */contestants/:id* zwracający zawodnika o podanym id oraz,
POST */contestants* służący do tworzenia nowych zawodników.

```
curl -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -X GET http://localhost:9292/contestants/1 \
  -b cookie
```

Listing 15: GET */contestants/:id*

```
[
  {
    "contest_id":1,
    "first_name":"Kamil",
    "last_name":"Wójcik",
    "end_time":null
  }
]
```

Listing 16: Odpowiedź zwracana przez GET */contestants/1*

Przy tworzeniu zawodnika jest zwracany jedynie kod odpowiedzi HTTP. Jeśli dane są poprawne i zawodnik zostanie zapisany do bazy zwracany jest kod 200 (OK) w przeciwnym wypdaku zwracany jest kod błędu 422 (Unprocessable Entity).

```
curl -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -X POST http://localhost:9292/contestants \
  -d '{"contestant":
        {"first_name":"oskar", "last_name":"kapusta"},
       "contest":
         {"id":"1"}}'
  -b cookie
```

Listing 17: POST */contestants*

Do metody get '/contestants/:id' przekazywane są w bloku 2 zmienne: contestant_presenter oraz cotestant_repository. contestant_presenter jest obiektem klasy Presenters::ContestantPresenter, której zadaniem jest prezentacja kolekcji zawodników w formatach JSON, CSV, XML oraz jako Hasz.

Użyta tutaj metoda *as_json* mapuje przekazaną kolekcję na tablicę haszy oraz wywołuję na zwracanej wartości metodę *to_json* co przedstawia listing 18.

```
def as_json(collection)
  collection.map do |contestant|
    {
      contest_id: contestant.contest_id,
      first_name: contestant.first_name,
      last_name: contestant.last_name,
      end_time: contestant.end_time
    }
  end.to_json
end
```

Listing 18: *app/services/presenters/contestant_presenter.rb*

Przekazywana także w bloku metody *post* zmienna contestant_repository jest obiektem klasy *Repositories::ContestantRepository* mającej za zadanię budowę, zapis oraz pobieranie z bazy zawodników 19. W konstruktorze przyjmuję model zawodnika *app/models/contestant.rb*.

W bloku metody post przekazywane są też zmienne hash i json_parser metody klasy *Injector* zwracającej stałą JSON, która jest potem użyta do sparsowania ciała rządania HTTP. hash jest obiektem klasy Hash modułu *Skirace*. Klasa ta dziedziczy po Haszu i ma na celu dodanie pomocniczych metod znanych z Rails takich jak użyta tutaj *with_indifferent_access* pozwalająca na odwołanie się do klucza haszu przy pomocy stringa lub symbolu.

Takie podejście zostało wybrane ponieważ autor uważa że monkey patchowanie klas należących do *Ruby core* jest czymś czego nie powinno się robić.

```ruby
class Repositories::ContestantRepository
  takes :db_contestant

  def build(params)
    contestant = db_contestant.new(params[:contestant])
    contestant.contest_id = params[:contest][:id]
    contestant
  end

  def save(contestant)
    contestant.save
  end

  def all
    db_contestant.all
  end

  def get(id)
    db_contestant.where(id: id)
  end
end
```

Listing 19: *app/services/repositories/contestant_repository.rb*

Contests

W pliku *app/routes/contests.rb* znajdują się następujące endpointy:
GET */contests*
Zwraca wszystkie zawody z bazy. Zmienna contest_presenter przekazana w bloku jest

```
get '/contests' do |contest_presenter, contest_repository|
  contest_presenter.as_json(contest_repository.all)
end
```

Listing 20: *app/routes/contests.rb*

```
curl -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -X GET http://localhost:9292/contests \
  -b cookie
```

Listing 21: GET */contests*

```
[
  {
    "id":1,
    "name":"Zakopane 2014"
  },
  {
    "id":2,
    "name":"Sczawnica"
  }
]
```

Listing 22: JSON response

obiektem klasy *Presenters::ContestPresenter* zawieającej jedynie metodę *as_json* działającą analogicznie do metody o tej samej nazwie klasy *Presenters::ContestantPresenter* przedstawionej wcześniej.

contest_repository jest tutaj obiektem klasy *Repositories::ContestRepository*. Metoda all najpierw sprawdza czy w bazie znajdują się zawody metodą *any?* Jeśli w bazie danych istniją zawody są one zwracane w przeciwnym wypadku wywoływana jest prywatna metoda *create_default* 23.

```
class Repositories::ContestRepository
  takes :db_contest

  def all
    db_contest.any? ? db_contest.all : create_default
  end

  private

     def create_default
       [ OpenStruct.new(db_contest.create(name: 'Default').values) ]
     end
end
```

Listing 23: *app/services/repositories/contest_repository.rb*

GET */contests/public*

Jeśli publicznie udustępnianie zawodów jest włączone zwrócony zostanie JSON zawiezający id i nazwę zawodów oraz listę zawodników 26 w przeciwnym wypadku zwrócony zostanie JSON z odpowiedzią *no-public-contests* oraz statusem HTTP 404 (Not Found) 27.

```
get '/contests/public' do |public_contest_presenter, contest_repository|
  begin
    public_contest_presenter.as_json(contest_repository.get_public)
  rescue
    {response: 'no-public-contests', status: 404}.to_json
  end
end
```

Listing 24: *app/routes/contests.rb*

```
curl -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -X GET http://localhost:9292/contests/public
```

Listing 25: GET */contests/public*

```
{
  "id":1,"name":"Zakopane 2014",
  "contestants": [
    {
      "contest_id":1,
      "first_name":"Kamil",
      "last_name":"Wójcik",
      "end_time":null
    }, ...
  ]
}
```

Listing 26: Publiczne udostępnianie zadodów włączone

```
{"response":"no-public-contests","status":404}
```

Listing 27: Publiczne udostępnianie zadodów wyłączone

GET */contests/:id/contestants*

Zwraca listę zawodników dla zawodów o przekazanym id.

```
get '/contests/:id/contestants' do |contestant_presenter, contest_repository|
  env['warden'].authenticate!

  contestants = contest_repository.get(params[:id]).contestants
  contestant_presenter.as_json(contestants)
end
```

Listing 28: *app/routes/contests.rb*

POST */contests*

Tworzy nowe zawody.

```
curl -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -X GET http://localhost:9292/contests/1/contestants \
  -b cookie
```

Listing 29: GET */contests/1/contestants*

```
[
  {
    "contest_id":1,
    "first_name":"Kamil",
    "last_name":"Wójcik",
    "end_time":null
  }, ...
]
```

Listing 30: Odpowiedź JSON

3.5.1.2 Warden

3.5.1.3 Front-end

Inerface apllikacji, tak jak była to tym mowa we wstępie, został napisany przy pomocy Backbone.js. Aplikacja frontend jest inicjalizowana ostatnią linią listingu 4. Obiekty plikacji oraz funkcja inicjalizująca ją są zawarte w pliku *app.coffee* przedstawionym na listingu 31.

Listing 31: app/assets/javascripts/app.coffee

3.5.2 Worker

3.6 Obudowa ochronna

4 Wnioski

5 Licencje Gemów

5.1 Wykaz gemów

- **execjs**
  Autorzy: Sam Stephenson, Josh Peek
  Licencja: MIT

- **haml** MIT
  Autorzy: Hampton Catlin, Nathan Weizenbaum
  Licencja: MIT

- **haml_coffee_assets**
  Autor: Michael Kessler
  Licencja: MIT

- **uglifier**
  Autor: Ville Lautanala
  Licencja: MIT

- **sinatra**
  Autorzy: Blake Mizerany, Konstantin Haase
  Licencja: MIT

- **sequel**
  Autorzy: Sharon Rosner, Jeremy Evans
  Licencja: MIT

- **sprockets**
  Autorzy: Sam Stephenson, Josua Peek
  Licencja: MIT

- **sprockets-helpers**
  Autor: Perer Browne
  Licencja: MIT

- **dependor-sinatra**
  Autor: Adam Pohorecki
  Licencja: MIT

- **therubyracer**
  Autor: Charles Lowell
  Licencja: MIT

- **wiringpi** GNU LGPLv3

- **thin**
  Autor: Marc-Andre Cournoyer
  Licencja: Ruby

- **sqlite3**
  Autorzy: Jamis Bluck, Luis Lavena, Aron Patterson
  Licencja: BSD-3

- **sass**
  Autorzy: Nathan Weizenbaum, Chris Eppstein, Hampton Catlin
  Licancja: MIT

- **binding_of_caller**
  Autor: John Mair
  Licencja: MIT

- **pry**
  Autorzy: John Mair, Conrad Irwin, Ryan Fitzgerald
  Licencja: MIT

- **warden**
  Autor: Daniel Neighman
  Licencja: MIT

- **bcrypt-ruby**
  Autor: Coda Hale
  Licencja: MIT

- **memcache-client**
  Autorzy: Eric Hodel, Robert Cottler, Mike Perham
  Licencja: BSD-3

- **dalli**
  Autor: Mike Perham
  Licencja: MIT

- **nokogiri**
  Autorzy: Aaron Patterson, Mike Dalessio, Yoko Harada, Tim Eliott, Akinori MUSHA
  Licencja: MIT