Module 4, Summative Assignment: Web Services

This assignment will evaluate your ability to implement a RMM Level Two (2) REST-based Web Service consisting of:

- URIs
- URI, Query, and Post Data Parameters
- HTTP Methods
- Error Status Responses
- Representations with JSON and XML
- Resource Implementations with JSON
- Headers and Caching
- and an HTTP Client

The functional goal of the assignment is to:

- Implement a REST-based Web Service to have access to `Race` and `Entrant` information by re-using the previous assignment's data tier and HTML page levels
- Implement `Race` result reporting through an HTTP Web Service interface using JSON
- Implement cache re-validation headers to limit the amount of server-side work required to keep client copies of race results up to date.

This assignment requires you to base your solution from the "triresults" application you constructed in the previous module. The instructions of this assignment are focused at the HTTP level. However, you are free to also visualize your changes to the database using the web pages constructed in the previous assignment.

The length of the assignment is partially due to the many examples shown throughout each section. The core of this assignment is within the Web Service tier. When relevant, you will be given solution **hints** to remind you of how things work in the data tier.

Functional Requirements

1. Implement web service access to get information for `racer` and `race` resources.
2. Implement web service access to perform basic CRUD for `race`s.
3. Implement error handling required to provide proper error feedback to web service clients.
4. Implement web service access to obtain XML and JSON representations for `race`s.
5. Implement web service access for race result reporting updates and status.
6. Implement web caching to more efficiently keep web service clients in sync with results.

Getting Started

1. Start your MongoDB server using `mongod`.
2. Create a copy of your `triresults` solution from the previous module to begin focusing on the web service requirements for this module.

   ```
   $ cp -r triresults (new_location)
   $ cd (new location)
   ```

3. Remove your rspec tests from the previous assignment.

   ```
   $ rm -rf spec/*
   ```

4. Optionally change your database. If you cloned the application with the original `config/mongoid.yml` file, the two applications will share the same database. If you decide to separate the two, edit the following lines.

   ```
   development:
     clients:
       default:
   ```

```
      database: (new_name)_development
test:
  clients:
    default:
      database: (new_name)_test
```

5. Create an `app/services` directory for non-model and non-controller code.

   ```
   $ mkdir app/services
   ```

   Add this directory to the `eager_load_paths` using the `config/application.rb` file. The extra step with the paths should not be necessary as Rails should automatically detect and load classes in any directory below `app`. However, we have seen issues with that functionality not working once Mongoid has been added. By adding the following line, you can guarantee the new directory path will be added.

   ```
   # config/application.rb
   Mongoid.load!('./config/mongoid.yml')
   config.eager_load_paths += %W( #{config.root}/app/services )
   ```

6. For testing purposes, edit the following settings in the `config/environments/test.rb` file from their default settings. Without these changes, the tests will detect runtime errors and fail before getting error pages and responses from the HTTP request. For example, a 404/NotFound status response may be the correct response for the test and without the changes below – any 4xx or 5xx response will get blindly treated as a failure, regardless what status result the test is attempting to verify.

   ```
   # config/environments/test.rb
   #config.consider_all_requests_local       = true
   config.consider_all_requests_local       = false

   #config.action_dispatch.show_exceptions = false
   config.action_dispatch.show_exceptions = true
   ```

7. Use the `rails console` during your development to invoke your solutions for the data tier. Remember to call `reload!` after making changes to your source code.

   ```
   $ rails c
   > ...
   > reload!
   > ...
   ```

8. Download and extract the starter set of boostrap files for this assignment.

   ```
   student-start/
   |-- Gemfile
   |-- .rspec (an important hidden file)
   `-- spec
       |-- ..._spec.rb
       |-- ..._spec.rb
       `-- data
             |-- races.json
             |-- racers.json
             `-- results.json
   ```

   - Overwrite your existing Gemfile with the Gemfile from the bootstrap fileset. They should be nearly identical, but this is done to make sure the gems and versions you use in your solution can be processed by the automated grader when you submit. Any submission should be tested with this version of the file.

     NOTE the Gemfile includes a section added for testing.

     ```
     group :test do
         gem 'rspec-rails', '~> 3.0'
         gem 'mongoid-rspec', '3.0.0'
         gem 'capybara'
     end
     ```

as well as a new definition for the following items:

  - `tzinfo-data` gem conditionally included on Windows platforms
  - `mongoid` gem used to implement your data tier
  - `httparty` gem will be used to implement client access to your web services
  - `responders` gem to be optionally used to build responses

```
# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
gem 'mongoid', '~> 5.0.0'
gem 'httparty'
gem 'responders', '~> 2.0'
```

- Add the `spec/*.rb` files provided with the bootstrap fileset to a corresponding `spec/` directory within your `triresults` application. These files contain tests that will help determine whether you have completed the assignment. Be sure to also copy the hidden `.rspec` file in the root directory.
- Add the provided `spec/data/*.json` data files to your `spec/data` directory.

9. Run the `bundle` command to make sure all gems are available.

```
$ bundle
```

10. Ingest the sample data. This data will assist you as you build and test your solution. The data is time-sensitive, so repeat as necessary to continue to have upcoming races in the future.

```
$ rake assignment:setup_data

importing data...
races=144, racers=1000, results=2880
updating database: triresults_development
updating race dates to current by (N) years
updating birth years to current by (N) years
updating creation and update times to (today)
```

Note that the data ingested into the development database is not used for grading. The tests will generate new data in the test database. You can refresh your development database by repeating the above commands if you delete or insert data you wish to remove.

11. Run the rspec test(s) to receive feedback. `rspec` must be run from the root directory of your application. There are several test files provided for this assignment. Many of those files are designed to test your code at specific points as you proceed through the technical requirements of this assignment. Initially, majority of tests will (obviously) fail until you complete the requirements necessary for them to pass.

```
$ rspec
...
(N) examples, (N) failures, (N) pending
```

To focus test feedback on a specific step of the requirements, add the specific file (path included) with the tests along with "-e rq##" to the rspec command line to only evaluate a specific requirement. Pad all step numbers to two digits.

```
$ rspec spec/uri_spec.rb -e rq02
...
(N) examples, (N) failures, (N) pending
```

12. Implement your solution to the technical requirements and use the rspec tests to help verify your completed solution.

13. Submit your Rails app solution for grading.

Technical Requirements

If you do this assignment as it is intended, you will create all controller and view classes from scratch manually, without the aid of any `rails generate` template commands or wholesale copy/paste of files. You should end up with an implementation where everything added was for a need and, ideally, because you knew why it was needed and what it performs. The API interface is being developed separate from the HTML/UI interface so that you can focus more on serving web service clients.

URIs

In this section you will implement a new URI in triresults and a new set of controller and view classes to focus on web service API implementation.

1. Create a set of custom URIs within Rails to serve as resources for posting and accessing race results using a web service interface. The actual names for the URL parameters (e.g., `:id`, `:racer_id`) are not important to satisfying this requirement since they will only be visible to your controller code. The resource URIs shall be:

   - `/api/races` - to represent the collection of races
   - `/api/races/:id` - to represent a specific race
   - `/api/races/:race_id/results` - to represent all results for the specific race
   - `/api/races/:race_id/results/:id` - to represent a specific results for the specific race
   - `/api/racers` - to represent the collection of racers
   - `/api/racers/:id` - to represent a specific racer
   - `/api/racers/:racer_id/entries` - to represent a the collection of race entries for a specific racer
   - `/api/racers/:racer_id/entries/:id` - to represent a specific race entry for a specific racer
   - defined for at least GET access at this time. Do not yet worry about the implementation to back it.

   Hint: The above step is achieved by configuring your `config/routes.rb` to have both – new global resources for `Racer`s and `Race`s and a set of nested resources to represent the dual role of `Entrant`.

   You can test your solution by using `rake routes`. There is no rspec test for the URIs until a follow-on step.

2. Without using `rails g` command (suggestion/recommendation - not graded), create (a set of) controllers and empty action methods to be the target of the URI action requests. The purpose of these empty methods is to just provide a complete path that maps the URI through to a method that you will implement later. By default, a resource with the following URI...

   `/api/races`

   ...will be mapped to the following target for GET requests.

   `api/races#index`

   A target with that value needs to be matched with a RacesController class within the `api` namespace or module. A namespace can be implemented by placing your new controller class(es) in a sub-directory that matches the name of the controller in the action of the route.

   ```
   # solution/app/controllers/api/races_controller.rb
   module Api
     class RacesController < ApplicationController
     end
   end
   ```

   Note that you have full control to tweak your controller trees how you see fit as long as the externally visible URI remains as specified. To complete this step, you must have an action method defined in the controller(s) to be the routed target for each `GET` URI defined in the previous step (**Hint**: `index` and `show` would be the default method names for these).

   Hint: Check your `log/test.log` during the rspec test or the server console output during development – to see if there are any syntax errors being reported for your controller class(es) that prevent the step from passing.

   ```
   $ rspec spec/uri_spec.rb -e rq02
   ```

3. Provide implementations for the method stubs you created in the previous step to act as the target of GET requests to the URIs implemented above. The primary task here is to have an action in place that can receive the call and be able to understand the URI parameters passed. These controller actions must

   - return a `200/OK` response and render a hard-coded plain text string that matches the URI used to call it.
   - continue to perform the above functionality when more complete implementations are added in follow-on sections for GETs and the `Accept` format has not been specified

   Hint: Wrap your initial implementations in a condition statement that will only fire when content-type is not specified. Place your real implementation in an else-block as this assignment implementation progresses. That way older tests won't fail when implementations are completed.

```
def show
  if !request.accept || request.accept == "*/*"
    render plain: "/api/racers/#{params[:racer_id]}/entries/#{params[:id]}"
  else
    #real implementation ...
  end
end
```

   Hint: Although you know the URI template used to invoke each of your actions, the intent is that you locate the URI parameters from the `params` hash available to your code within the controller. Try not to cheat by looking for the complete URI called from the request object made accessible to the controller.

   You can demonstrate your URI access using `rails console` and `HTTParty`. In a separate terminal, start your `rails server`. The following shows the controller returning a plain text response:

```
> HTTParty.get("http://localhost:3000/api/races").response.body
 => "/api/races"
> HTTParty.get("http://localhost:3000/api/races/123").response.body
 => "/api/races/123"
> HTTParty.get("http://localhost:3000/api/races/123/results").response.body
 => "/api/races/123/results"
> HTTParty.get("http://localhost:3000/api/races/123/results/456").response.body
 => "/api/races/123/results/456"
> HTTParty.get("http://localhost:3000/api/racers").response.body
 => "/api/racers"
> HTTParty.get("http://localhost:3000/api/racers/abc").response.body
 => "/api/racers/abc"
> HTTParty.get("http://localhost:3000/api/racers/abc/entries").response.body
 => "/api/racers/abc/entries"
> HTTParty.get("http://localhost:3000/api/racers/abc/entries/def").response.body
 => "/api/racers/abc/entries/def"
```

   The above calls all returned 200/OK with a

```
> response=HTTParty.get("http://localhost:3000/api/racers/abc/entries/def")
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.content_type
 => "text/plain"
```

   If you call a URI not implemented, you should get a 404 status response.

```
> response=HTTParty.get("http://localhost:3000/api/racers/abc/foobar/def")
> response.response
 => #<Net::HTTPNotFound 404 Not Found  readbody=true>
```

   Hint: If you fail the test because your application correctly returned a 404/NotFound for the bogus URI, then make sure you have updated your `config/environments/test.rb` file as described in the `Getting Started` section above.

```
ActionController::RoutingError:
       No route matches [GET] "/api/races/abc/foo/def"
```

```
$ rspec spec/uri_spec.rb -e rq03
```

4. Implement a controller method to act as the target of POST requests to the `/api/races` URI implemented above. The primary task here is to have an action in place that can receive the call. This method must:

   - render a `200/OK` response. The content of the response will not be checked. **Hint**: The following will render an empty response with a `200/OK` status response.

   ```
   render plain: :nothing, status: :ok
   ```

   - continue to perform the above functionality when more complete implementations are added in the follow-on sections for POST and the `Accept` format has not been specified. **Hint**: The following will perform the requirements of this step when the `Accept` header is not supplied or has not been set to a specific MIME type.

   ```
   if !request.accept || request.accept == "*/*"
     render plain: :nothing, status: :ok
   else
     #real implementation
   end
   ```

   Hint: Note that you must configure forgery protection in the controller class for the API to use when implementing POST HTTP method actions from web service clients.

   ```
   protect_from_forgery with: :null_session
   ```

   You can demonstrate access to your new URI action using the rails console.

   ```
   > response=HTTParty.post("http://localhost:3000/api/races")
   > response.response
    => #<Net::HTTPOK 200 OK  readbody=true>
   ```

   ```
   $ rspec spec/uri_spec.rb -e rq04
   ```

5. Implement a utility/service class in `app/services` called `TriResultsWS` that can act as our primary client interface to the API. This class must:

   - include the `HTTParty` mixin
   - declare a `base_uri` to reference your server (e.g., `http://localhost:3000`)

   Hint: The following represents an implementation of this requirement. The class implementation also includes a declaration to output debug which will include the details of the HTTP message exchange.

   ```
   # app/services/tri_results_ws.rb
   class TriResultsWS
     include HTTParty
     debug_output $stdout
     base_uri "http://localhost:3000"
   end
   ```

   You can demonstrate your new web service client class using the rails console after a `reload!`. If you define `debug_output` for the class you can see the full HTTP exchange between the client and server. Feel free to work at the verbosity level that you are comfortable with.

   ```
   > reload!
   > TriResultsWS.get("/api/races")
   opening connection to localhost:3000...
   opened
   <- "GET /api/races HTTP/1.1\r\n
   Accept-Encoding: gzip;q=1.0,deflate;q=0.6,identity;q=0.3\r\n
   Accept: */*\r\n
   User-Agent: Ruby\r\n
   Connection: close\r\n
   ```

```
Host: localhost:3000\r\n\r\n"

-> "HTTP/1.1 200 OK \r\n"
-> "X-Frame-Options: SAMEORIGIN\r\n"
-> "X-Xss-Protection: 1; mode=block\r\n"
-> "X-Content-Type-Options: nosniff\r\n"
-> "Content-Type: text/plain; charset=utf-8\r\n"
-> "Etag: W/\"b5dd0c9690b186a0e7629e3e9b2be60c\"\r\n"
-> "Cache-Control: max-age=0, private, must-revalidate\r\n"
-> "X-Request-Id: f8fa9363-6ac8-44bf-8bd2-78bccf23c1bd\r\n"
-> "X-Runtime: 0.069018\r\n"
-> "Server: WEBrick/1.3.1 (Ruby/2.2.2/2015-04-13)\r\n"
-> "Date: Sun, 17 Jan 2016 02:07:55 GMT\r\n"
-> "Content-Length: 10\r\n"
-> "Connection: close\r\n"
-> "\r\n"
reading 10 bytes...
-> ""
-> "/api/races
```

```
$ rspec spec/uri_spec.rb -e rq05
```

Query Parameters and POST Data

In the previous section we demonstrated how we can cause certain actions to fire and be passed parameters within the URI. In this section we will expand our options and pass in query arguments thru two additional means:

- query parameters
- POST data

1. Update the controller for GET `/api/races` to handle a query string that represents paging parameters. This change must:

   - extract the supplied `offset` and `limit` parameters also from the params object.
   - append them to the returned text string, with values within square braces ([]) as shown in example format below `, offset=[#{}], limit=[#{}]`

   You can demonstrate your handling of query parameters using the rails console. Pass the query parameters as a hash to HTTParty using the `:query` element.

   ```
   > TriResultsWS.get("/api/races",:query=>{:offset=>4,:limit=>10}).response.body
    => "/api/races, offset=[4], limit=[10]"
   ```

   Notice from the HTTParty debug that the query parameters were passed to the server together with the URI

   ```
   <- "GET /api/races?offset=4&limit=10
   ```

   ```
   $ rspec spec/params_spec.rb -e rq01
   ```

2. Update the controller for POST `/api/races` to handle POST data. The payload will contain a `race` with the name property within a hash. Your method must:

   - return a status of 200/OK, Content-Type `text/plain`, and the name of the `race` provided in the POST data. The following shows the structure of the posted payload with the `:name` field the method must return back.

     ```
     {:race=>{:name=>"(race name)"}}
     ```

   - continue to perform the above functionality when more complete implementations are added in the follow-on sections for POST and the `Accept` format has not been specified. **Hint**: The following shows how this requirement can be latched to execute only when the `Accept` header is not specified or does not specify a specific MIME type.

```
    if !request.accept || request.accept == "*/*"
      #return post params
    end
```

You can demonstrate your new method handling using the rails console. In the example, we are passing the properties in the body of the POST using the `:body` property in the call to HTTParty.

```
> response=TriResultsWS.post("/api/races",:body=>{:race=>{:name=>"Meager Mile"}})
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.body
 => "Meager Mile"
```

Notice from the HTTParty debug that, unlike the GET, the POST data is not passed with the URI. It is provided in the body of the POST and encoded (by HTTParty) using `application/x-www-form-urlencoded` encoding.

```
<- "POST /api/races ...
Content-Type: application/x-www-form-urlencoded\r\n
\r\n"
<- "race[name]=Meager%20Mile"
```

Notice that we passed in a Ruby hash with no `Content-Type` specification in the previous example. HTTParty processed the hash into `form encoding` format (as if it where a call from an HTML form) and supplied a `Content-Type` specification stating it was `application/x-www-form-urlencoded` encoding. If we pass HTTParty straight JSON text, it will be sent unchanged but adds a `Content-Type` of `application/x-www-form-urlencoded`. Rails will **NOT** be able to parse the content and our POST method will not get the parameters it needs, resulting in an error response.

```
> response=TriResultsWS.post("/api/races",
                             :body=>{:race=>{:name=>"Meager Mile"}}.to_json)
> response.body
 => ""
```

```
<- "POST /api/races ...
Content-Type: application/x-www-form-urlencoded\r\n
\r\n"
<- "{\"race\":{\"name\":\"Meager Mile\"}}"
```

If we do the opposite and pass a `Content-Type` without the `:body` element being a JSON string format already, we also get a header spec and format mis-match except this time we get an error because the server cannot parse the format we claimed the body was in.

```
> response=TriResultsWS.post("/api/races",
                             :body=>{:race=>{:name=>"Meager Mile"}},
                             :headers=>{"Content-Type"=>"application/json"})
> response.response
 => #<Net::HTTPBadRequest 400 Bad Request  readbody=true>
```

```
<- "POST /api/races ...
Content-Type: application/json\r\n
\r\n"
<- "race[name]=Meager%20Mile"
```

If we provide a JSON string in the `:body` and a matching `application/json` in the `:header` for `Content-Type`, our method can be called using a JSON POST body and works.

```
> response=TriResultsWS.post("/api/races",
                             :body=>{:race=>{:name=>"Meager Mile"}}.to_json,
                             :headers=>{"Content-Type"=>"application/json"})
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.body
 => "Meager Mile"
```

```
<- "POST /api/races ...
Content-Type: application/json\r\n
\r\n"
<- "{\"race\":{\"name\":\"Meager Mile\"}}"
```

```
$ rspec spec/params_spec.rb -e rq02
```

Methods

In this section you are going to demonstrate a more complete vocabulary of HTTP method implementations using the data tier copied over from the previous module. In a previous section, you were required to expose specific URIs, but only required to implement GETs and a single POST. That left you many choices in how to proceed with the assignment. In this section you will need to have POST in place for the `api/races` URI and a GET, PUT, and DELETE in place for the `/api/races/:id` URI. You are not restricted in how you achieve the mapping from URI and HTTP Method to the controller#action as long as you have an action in place that provides the required behavior below.

1. Implement POST `/api/races` to create a new `Race` in the database. Note that some of the requirements listed are automatically done for you by the server. This action must:

   - accept a JSON string in the POST body with name and date

   ```
   "{\"race\":{\"name\":\"First Race\",\"date\":\"2016-01-17\"}}"
   ```

   - add a new `Race` to the database with that name and date
   - return a `201/CREATED` status with a `text/plain` response with the race name. **Hint**: use `status: :created` to supply the non-default `201/CREATED` status.

   ```
   render plain: name, status: :created
   ```

   - not eliminate the stub behavior implemented for these methods in the earlier URI and parameter sections when Accept is not supplied. (**Hint**: Implement this behavior when `request.accept` is not nil and not equal to `*/*`)

   You can test out your new implementation using the rails console. In the example below, we explicitly create a JSON formatted string with the `Race` data and then POST that information using the appropriate `Content-Type` in the header. Note that we must provide an `Accept` header to bypass our earlier stub implementations.

   ```
   > json_string={race:{name:"First Race", date:Date.current.iso8601}}.to_json
    => "{\"race\":{\"name\":\"First Race\",\"date\":\"2016-01-17\"}}"
   > response=TriResultsWS.post("/api/races",
                                :body=>json_string,
                                :headers=>{"Content-Type"=>"application/json",
                                           "Accept"=>"text/plain"})
   ```

   The response should be a `201/CREATED` and we should be able to locate the new `Race` in the database based in the inputs provided. If your response was `200/OK`, check that your controller logic is not falling into the stub implementation implemented earlier.

   ```
   > response.response
    => #<Net::HTTPCreated 201 Created  readbody=true>
   2.2.2 :130 > response.body
    => "First Race"

   > pp Race.find_by(:name=>"First Race", :date=>Date.current).attributes
   {"_id"=>BSON::ObjectId('569bd20ee301d0aa30000000'),
    "next_bib"=>0,
    "n"=>"First Race",
    "date"=>2016-01-17 00:00:00 UTC,
    "updated_at"=>2016-01-17 17:40:30 UTC,
    "created_at"=>2016-01-17 17:40:30 UTC}
   ```

   Note there are several ways you could have tried to implement using the parameters to create the race and many of them likely failed due to a `ActiveModel::ForbiddenAttributesError`.

- **Passed explicit hash arguments**: This would have worked but is verbose and undesirable as the number of arguments increase.

```
race=Race.create(:name=>params[:race][:name], :date=>params[:race][:date])
```

- **Direct mass assignment**: Here you know that `:race` should have two elements with the exact names you would have used to create this `Race` using Mongoid in the rails console. We can use the debugger or log statements to help demonstrate.

```
(byebug) params[:race]
{"name"=>"First Race", "date"=>"2016-01-17"}
race=Race.create(params[:race])
```

This does not work because of security measures Rails has put into place to prevent callers from adding anything they want to the data tier thru the mass assignment technique that is common within Rails. If you look close in a debugger session, you will notice that the parameters are not a hash type.

```
(byebug) params[:race].class
ActionController::Parameters
```

ActiveModel implementations (i.e., Mongoid) will not accept parameters of this type that have not been inspected and white-listed for processing.

- **Hash mass assignment**: We can bypass Rails security measures and blindly turn our `ActionController::Parameters` into a Ruby hash. This will work but leaves you open to every parameter passed in being passed to the model.

```
race=Race.create(params[:race].to_hash)
```

You can fix that slightly using `slice`,

```
race=Race.create(params[:race].to_hash.slice(:name, :date))
```

but Rails has a more standard approach to addressing this using white-listing

- **White-listed mass assignment**: The `ActionController::Parameters` type has a set of methods (`require` and `permit`) that scale the parameters down to a set of white-listed values. A version of this was generated for you by `rails g` scaffold commands for the web pages generated in the previous module.

```
race=Race.create(race_params)
...
private
  def race_params
    params.require(:race).permit(:name, :date)
  end
```

```
$ rspec spec/methods_spec.rb -e rq01
```

2. Implement GET `/api/races/:id` to retrieve an existing `Race` from the database. This action must:

- accept the ID of the `Race` as a URI parameter as designed in the URI design section of this assignment
- find the Race in the database
- return a 200/OK response and a rendering of the `Race` as JSON. **Hint**: You can have the default JSON rendering applied using the following in your method:

```
render json: race
```

- not eliminate the stub behavior implemented for these methods in the earlier URI and parameter sections when Accept is not supplied. (**Hint**: Implement this behavior when `request.accept` is not nil and not equal to `*/*`)

You can test your new method using the rails console. In the following example we locate a specific race in the database and use it in forming a URI passed in the GET to the server.

```
> race=Race.find_by(:name=>"First Race", :date=>Date.current)
> response=TriResultsWS.get("/api/races/#{race.id}",
                            :headers=>{"Accept"=>"application/json"})
```

The action returns a `200/OK` status with a `Content-Type` of `application/json` and a payload. **Note** that this is the default JSON rendering of the object from the database and contains database information we may or may not have wanted to be passed to the caller.

```
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.content_type
 => "application/json"
> pp response.parsed_response
{"_id"=>{"$oid"=>"569bd20ee301d0aa30000000"},
 "created_at"=>"2016-01-17T17:40:30.540Z",
 "date"=>"2016-01-17",
 "loc"=>nil,
 "n"=>"First Race",
 "next_bib"=>2,
 "updated_at"=>"2016-01-17T17:40:30.540Z"}
```

If you are receiving a `text/plain` response from the stub created in the earlier URI section, verify that you have properly updated your logic such that you perform real actions when an `Accept` header is provided.

```
$ rspec spec/methods_spec.rb -e rq02
```

3. Execute HEAD `/api/races/:id` as an example of how you could determine if a race ID existed without having to receive a rendering of the race.

   You can demonstrate the use of HEAD with the rails console and the ID used in the previous step. **Note** that we still need to provide an `Accept` header otherwise we will fall back to the stub implementation implemented during the URI section.

   ```
   > response=TriResultsWS.head("/api/races/#{race.id}",
              :headers=>{"Accept"=>"application/json"})
   > response.response
    => #<Net::HTTPOK 200 OK  readbody=true>
   > response.content_length
    => 0
   ```

   You can also demonstrate when the resource cannot be found using this method.

   ```
   > TriResultsWS.head("/api/races/bogus",
      :headers=>{"Accept"=>"application/json"}).response
    => #<Net::HTTPNotFound 404 Not Found  readbody=true>
   ```

   There is no rspec test for this step.

4. Implement PUT `/api/races/:id` to replace the values of a specific `Race`. Some of the requirements listed below are handled for you by the server. This action must:

   - accept the ID of the `Race` as a URI parameter as designed in the URI design section of this assignment
   - accept a JSON string in the payload data of the HTTP method
   - replace the Race in the database with the supplied values.
   - return a 200/OK response and a rendering of the `Race` as JSON.

   Hint: You can have the default JSON rendering applied using the following in your method:

   ```
   render json: race
   ```

   Hint: Avoid the `ActiveModel::ForbiddenAttributesError` error by re-using the same technique you used during POST to get white-listed values to pass to Mongoid.

   You can demonstrate your new action using the rails console. In the example below, we:

   - locate a race in the database and verify its name.
   - explicitly form a JSON string (from a hash) that has the new values we want the race replaced with.
   - invoke PUT on the URI, providing the JSON string and `Content-Type`
   - receive a 200/OK status code indicating success

```
> race=Race.find_by(:name=>"First Race", :date=>Date.current)
> race.name
 => "First Race"
> json_string={race:{name:"Modified Race", date:Date.current.iso8601}}.to_json
 => "{\"race\":{\"name\":\"Modified Race\",\"date\":\"2016-01-17\"}}"
> response=TriResultsWS.put("/api/races/#{race.id}",
                            :body=>json_string,
                            :headers=>{"Content-Type"=>"application/json"})
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
```

Note that an `Accept` header was not necessary since we did not implement a stub method for PUT commands and the controller implementation is hard-coded to return `application/json`. We can begin verifying success by inspecting the returned payload and note that the name (and date) have be replaced.

```
> response.content_type
 => "application/json"
> pp response.parsed_response
{"_id"=>{"$oid"=>"569bd20ee301d0aa30000000"},
 "created_at"=>"2016-01-17T17:40:30.540Z",
 "date"=>"2016-01-17",
 "loc"=>nil,
 "n"=>"Modified Race",
 "next_bib"=>17,
 "updated_at"=>"2016-01-17T19:47:54.990Z"}
```

We can also verify the state of the race by locating it directly within the database.

```
> Race.find(race.id).name
 => "Modified Race"
    > pp Race.find(race.id).attributes
{"_id"=>BSON::ObjectId('569bd20ee301d0aa30000000'),
 "next_bib"=>17,
 "n"=>"Modified Race",
 "date"=>2016-01-17 00:00:00 UTC,
 "updated_at"=>2016-01-17 19:47:54 UTC,
 "created_at"=>2016-01-17 17:40:30 UTC}
```

One thing to note about the state of the database. You will notice that `next_bib` is greater than zero (0) even though we have not assigned a single `Entrant`. That is because the default JSON marshaller is calling the getter for `next_bib` each time we request the object and this is incrementing the `next_bib` value in the database.

```
$ rspec spec/methods_spec.rb -e rq04
```

5. **Note** that you can also invoke the same action as PUT with a PATCH command and determine which was actually being called by inspecting the `request`. PUT is meant to be a replacement and PATCH is meant to be a partial update. How you implement that is up to you but it can make a difference to an Internet cache when is sees a PUT ("possibly cache - that will be the new state of the resource") versus a PATCH ("don't cache – that is only a partial update to the state of the resource"). The easiest way to think of this is to consider a URI referencing a document and the PUT overwrites the document and the PATCH makes changes to the document.

   To verify this point, lets insert a debug statement in our action to print out the HTTP method called.

   ```
   # PUT,PATCH /api/races/:id
   def update
     Rails.logger.debug("method=#{request.method}")
     race=Race.find(params[:id])
   ```

   When we issue a PUT, we get a method=PUT in the rails server output. Although not always enabled for a URI – an intermediate Internet cache between the callers and this service may believe the state passed in is a full replacement for what will be served by the next GET.

   ```
   Started PUT "/api/races/569c01b2e301d0cbcf000001" for 127.0.0.1 at 2016-01-17 16:15:52 -0500
   ```

```
Processing by Api::RacesController#update as HTML
  Parameters: {"race"=>{"name"=>"Modified Race", "date"=>"2016-01-17"}, "id"=>"569c01b2e301d0cbcf000001
Can't verify CSRF token authenticity
method=PUT
```

When we issue a PATCH, we get a method=PATCH in the rails server output. This will allow us to receive partial updates and apply them accordingly and the intermediate Internet caches will know the incoming state was not a full replacement.

```
Started PATCH "/api/races/569c01b2e301d0cbcf000001" for 127.0.0.1 at 2016-01-17 16:15:59 -0500
Processing by Api::RacesController#update as HTML
  Parameters: {"race"=>{"name"=>"Modified Race", "date"=>"2016-01-17"}, "id"=>"569c01b2e301d0cbcf000001
Can't verify CSRF token authenticity
method=PATCH
```

There are no tests for this step.

6. Implement DELETE `/api/races/:id` to destroy the race associated with that specific `Race` ID. Some of the requirements listed below are automatically handled for you by the server. This action must:

   - accept the ID of the `Race` as a URI parameter as designed in the URI design section of this assignment
   - remove the race with the supplied ID from the database
   - return a 204/NO-CONTENT response without any content in the response **Hint**: You can have the default JSON rendering applied using the following in your method:

```
render :nothing=>true, :status => :no_content
```

You can test your new action using the rails console. In the example below, we

- verify that our race is still in the database
- issue a DELETE on the URI for the race
- check the response is a 2xx-level response, indicating success and 204/NO CONTENT indicating success with no provided response
- verified the race is no long in the database

```
> Race.where(:id=>race.id).first.name
 => "Modified Race"

> response=TriResultsWS.delete("/api/races/#{race.id}")
> response.response
 => #<Net::HTTPNoContent 204 No Content  readbody=true>

> Race.where(:id=>race.id).first
 => nil
```

```
$ rspec spec/methods_spec.rb -e rq06
```

Error Conditions

Up to this point we have been focusing on the "happy path" and satisfied when we get the correct responses for correct inputs. However, you likely noticed that when errors did occur, the web service client receives some unwanted information – especially in the Rails development environment profile. In this section you will be asked to address error conditions in a way that provides the API caller with a clean error response instead of an HTML page dump.

As setup for this section, observe the default response to a PUT request for a race that does not exist. Notice we correctly received a 404/NOT FOUND, but we also received 97,566 bytes of HTML text – which is of no use to a web service client.

```
> response=TriResultsWS.put("/api/races/foobar",
                            :body=>json_string,:headers=>{"Content-Type"=>"application/json"})
> response.response
 => #<Net::HTTPNotFound 404 Not Found  readbody=true>
> response.content_length
```

```
 => 97566
> response.content_type
 => "text/html"
```

1. Add rescue handling such that the exception thrown by Mongoid when a document is not found returns a `text/plain` message with the ID of the document asked for. This response must:

   - return a 404/NOT FOUND status code with a text/plain Content-Type and a text error message with the identity of the race requested located inside the text using the format `race[:id]`. Example:

   ```
   woops: cannot find race[foobar]
   ```

   Hint: You can add a rescue block per method or you can also address the requirement using a global `rescue_from` construct within your controller class. **Note** that you can usually locate the specific exception thrown by looking at the server log.

   ```
   rescue_from Mongoid::Errors::DocumentNotFound do |exception|
       render plain: "woops: cannot find race[#{params[:id]}]", status: :not_found
   end
   ```

   You can demonstrate your handling of "race not found" error using the rails console. In the following example, we

   - attempt to update a race that does not exist using a PUT
   - receive a 404/NOT FOUND as we did before
   - receive text/plain with an error message better suited for a web service client
   - within the error message is the required pattern of `race[:id]` with `:id` being the race we requested

   ```
   > response=TriResultsWS.patch("/api/races/foobar",
                 :body=>json_string,:headers=>{"Content-Type"=>"application/json"})
   > response.response
    => #<Net::HTTPNotFound 404 Not Found  readbody=true>
   > response.content_type
    => "text/plain"
   > response.content_length
    => 31
   > response.body
    => "woops: cannot find race[foobar]"
   ```

   ```
   $ rspec spec/errors_spec.rb
   ```

Representations

This section you will focus on rendering different representation formats from your web service. Up to now we have limited the output to text/plain with a minor amount of JSON. This was hard-coded in some of the hints provided in this assignment. Examples:

```
render plain: name, status: :created
render json: race
```

In this section we are going to keep the renderings simple, but expand it such that the client can control what `Content-Type` they receive based on runtime requests and what is implemented by the web service. You are required to support both `application/json` and `application/xml` `Content-Type`s.

Since the solution will be evaluated from the API client-perspective, you are not constrained in how you implement the rendering decisions. The lecture demonstrated a way to keep the rendering format separate from the controller in the `HelloController`. The `render documentation` will also be of some help. The following builders will also be of interest

- `JSON JBuilder`
- `XML Builder`

The rails scaffold generator does mostly the same thing except it is complicated by the re-directs required by the web page access. The beauty of using the separate API class is that we can separate ourselves from the details of web page navigation and concentrate on completing actions.

1. Respond to a GET `/api/races/:id` request using an XML rendering of the requested race when the request contains an `Accept:application/xml` header. This action must:

   - be invoked using GET `/api/races/:id`
   - locate the requested `Race` by ID
   - return a 404/NOT FOUND status with an XML-formatted error message containing some error text in a root-level element called `msg` when not found. Example:

   ```
   <error>
     <msg>woops: cannot find race[foobar]</msg>
   </error>
   ```

   - return a 200/OK status with an XML-formatted message containing a representation of the requested `Race` with minimum of `name` and `date` elements at the root level below a `race` element. Example:

   ```
   <race>
     <name>Upper Tract Iron</name>
     <date>2014-10-20</date>
   </race>
   ```

   Hint: To get started, you are going to want to remove any pre-determined, hard-coded rendering configuration in your controller and make rendering decisions based on the `Accept` header.

   ```
   #render json: race
   render race
   ```

   The `sayhello` example expressed that thru different views.

   You can demonstrate your ability to provide XML content when the client provides an `Accept` header in the request for a GET using the rails console. In this example we

   - obtain a valid race from the database
   - display the race name
   - request the race from the API, supplying the ID of the race in the URI and the `Accept` header stating we would only take XML.

   ```
   > race=Race.first
   > race.name
    => "Upper Tract Iron"
   > response=TriResultsWS.get("/api/races/#{race.id}",
                               :headers=>{"Accept"=>"application/xml"})
   ```

   We received:

   - a 200/OK status message telling us our request was satisfied
   - a `Content-Type` of `application/xml` telling us the format of the body of the response
   - an XML payload body with the required name and date in root-level elements

   ```
   > response.response
    => #<Net::HTTPOK 200 OK  readbody=true>
   > response.content_type
    => "application/xml"
   > y response.body
   --- |-
     <race>
       <name>Upper Tract Iron</name>
       <date>2014-10-20</date>
     </race>
   ```

   Hint: In an earlier section you added a rescue block that had hard-coded the response content type to `text/plain`. To complete this step, you should conditionally execute the previous implementation in the absence of an `Accept` header or a specific mime type in the header.

   ```
   if !request.accept || request.accept == "*/*"
     render plain: "woops: cannot find race[#{params[:id]}]", status: :not_found
   ```

```
else
  #Accept-specific response here
end
```

Hint: One way you could implement the Accept-specific implementation is to define the return status and message information independent of the returned type. In this specific example, there would be a `app/views/api/error_msg.xml.builder` in place to accept the inputs and render a result.

```
render :status=>:not_found,
       :template=>"api/error_msg",
       :locals=>{ :msg=>"woops: cannot find race[#{params[:id]}]"}
```

If we issue a race ID that does not exist:

- a 404/NOT FOUND status is returned
- Content-Type is set to `application/xml`
- an error message is supplied in XML format

```
> response=TriResultsWS.get("/api/races/foobar",
                            :headers=>{"Accept"=>"application/xml"})
> response.response
 => #<Net::HTTPNotFound 404 Not Found  readbody=true>
> response.content_type
 => "application/xml"
> y response.body
--- |-
  <error>
    <msg>woops: cannot find race[foobar]</msg>
  </error>
```

```
$ rspec spec/representations_spec.rb -e rq01
```

2. Respond to a GET `/api/races/:id` request using an JSON rendering of the requested race when the request contains an `Accept:application/json` header. Functionally this may be the same as what you quickly implemented with `render :json` in earlier steps, but the implementation here must be within a framework that allows the caller to specify the desired content-type rather than hard-coding the value in the action. This action must:

   - be invoked using GET `/api/races/:id`
   - locate the requested `Race` by ID
   - return a 404/NOT FOUND status with an JSON-formatted error message containing some error text in a root-level property called `msg` when not found. Example:

```
{"msg"=>"woops: cannot find race[foobar]"}
```

   - return a 200/OK status with an JSON-formatted message containing a representation of the requested `Race` with minimum of `name` and `date` properties at the root level. Example:

```
{"name"=>"Upper Tract Iron", "date"=>"2014-10-20"}
```

   You can demonstrate your ability to provide JSON content when the client provides an `Accept` header in the request for a GET using the rails console. In this example we

   - obtain a valid race from the database
   - display the race name
   - request the race from the API, supplying the ID of the race in the URI and the `Accept` header stating we would only take JSON.

```
> race=Race.first
> race.name
 => "Upper Tract Iron"
> response=TriResultsWS.get("/api/races/#{race.id}",
                            :headers=>{"Accept"=>"application/json"})
```

   We received:

- a 200/OK status message telling us our request was satisfied
- a `Content-Type` of `application/json` telling us the format of the body of the response
- an JSON payload body with the required name and date in root-level elements

```
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.content_type
 => "application/json"
> response.parsed_response
 => {"name"=>"Upper Tract Iron", "date"=>"2014-10-20"}
```

If we issue a race ID that does not exist:

- a 404/NOT FOUND status is returned
- Content-Type is set to `application/json`
- an error message is supplied in JSON format

```
> response=TriResultsWS.get("/api/races/foobar",
                            :headers=>{"Accept"=>"application/json"})
> response.response
 => #<Net::HTTPNotFound 404 Not Found  readbody=true>
> response.content_type
 => "application/json"
> response.parsed_response
 => {"msg"=>"woops: cannot find race[foobar]"}
```

```
$ rspec spec/representations_spec.rb -e rq02
```

3. Respond to a GET `/api/races/:id` request using a plain text rendering when the formats in the `Accept` header are not supported. You can make this another `rescue_from` case – as you did in the previous section on reporting errors. Your action must:

   - return a `415/UNSUPPORTED MEDIA TYPE` status with a `text/plain` error message stating the mime type requested in square brackets (i.e, content-type[#{request.accept}]) when the caller supplies an `Accept` header with no supported MIME types. Example:

   ```
   woops: we do not support that content-type[application/foobar]
   ```

   - return a `200/OK` when the `Accept` header contains unsupported media types mixed with a supported media type. The response type will be the supported type.

   Hint: This should only require 2-3 lines of code to implement. If you find yourself implementing entire algorithms, stop and review the lecture materials and examples.

   Hint: Note that the rescue syntax allows for the block to receive the exception. If you do nothing with that exception you will likely loose useful information in locating the source of a problem. It is suggested that you log the exception to restore some of the useful default behavior.

   ```
   rescue_from ActionView::MissingTemplate do |exception|
     Rails.logger.debug exception
   ...
   ```

   You can demonstrate your new `Accept` content type error handling using the rails console. In this example we have provided a valid race ID, but an unsupported MIME type. The error response is returned as a `415/UNSUPPORTED MEDIA TYPE` and the message we generated states the format requested.

   ```
   > race=Race.first
   > response=TriResultsWS.get("/api/races/#{race.id}",
                               :headers=>{"Accept"=>"application/foobar"})
   > response.response
    => #<Net::HTTPUnsupportedMediaType 415 Unsupported Media Type  readbody=true>
   > response.content_type
    => "text/plain"
   ```

```
> response.body
 => "woops: we do not support that content-type[application/foobar]"
```

If we combine the unsupported type with a supported type, Rails will use the supported type.

```
> response=TriResultsWS.get("/api/races/#{race.id}",
                            :headers=>{"Accept"=>"application/foobar,application/json"})
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.content_type
 => "application/json"
> response.parsed_response
 => {"name"=>"Upper Tract Iron", "date"=>"2014-10-20"}
```

Note there are still some error cases that are beyond this assignment. Take into consideration the case where a client requests an invalid `Race` and an unsupported MIME type. The `404/NOT FOUND` exception handling can throw an exception itself if it cannot locate a content to render the error message. That may add complications to your error handlers or require errors be reported in a standard format.

```
$ rspec spec/representations_spec.rb -e rq03
```

Resource Implementation (JSON)

In this section we will concentrate on implementing what is needed to get race results from the API. In an earlier section, you designed a `/api/race/:race_id/results` in order to represent the collection of results for a race. Now we want to implement a GET of results for at least JSON. Since there were no internal design requirements of how you implemented your controller(s), you will need to translate some of the hints to your specific situation.

1. Update the controller method that is implementing the action for GET `/api/races/:race_id/results/:id` so that it finds the specified `Entrant`. This action must:

   - obtain the `:race_id` and (`Entrant`) `:id` provided in the URI
   - find the `Entrant` in the database
   - make the `Entrant` available to the view

   Hint: the following snippet will be of use in locating the `Entrant` in the database.

   ```
   @result=Race.find(params[:race_id]).entrants.where(:id=>params[:id]).first
   ```

   What you are now adding should implement the stub created earlier when no `Accept` header was supplied.

   ```
   if !request.accept || request.accept == "*/*"
     render plain: "/api/races/#{params[:race_id]}/results/#{params[:id]}"
   else
     #real implemenation goes here
   end
   ```

   If you used a partial to implement the view for an `Entrant` result, then a partial called `_result.json.jbuilder` could get invoked using the following:

   ```
   render :partial=>"result", :object=>@result
   ```

   At the completion of this step you should either be:

   - manually calling a view similar to how the `HelloController#say method` worked in the class examples.
   - using automatic routing the way the `HelloController#sayhello` worked in the class examples.

   With that in place, you likely are getting an error stating a 404/NOT FOUND on the client-side and a template or partial not found on the server-side.

   There is no test for this step.

2. Implement an `XXX.json.jbuilder` script to act as the view that the above controller is trying to render. We will use a custom view so that we can better control what goes into the data marshalled back to the client. The document returned to the client must have:

- `place` - overall place
- `time` - time formatted (long) overall time
- `last_name` - entrant's last_name
- `first_name` - entrant's first_name
- `bib` - bib number
- `city` - entrant's city
- `state` - entrant's state
- `gender` - entrant's gender (Hint: entrant.racer_gender)
- `gender_place` - entrant's placing within gender
- `group` - entrant's group_name (Hint: entrant.group_name)
- `group_place` - entrant's placing within group
- `swim` - time formatted (long) swim time
- `pace_100` - time formatted (short) swim_pace_100
- `t1` - time formatted (short) t1_secs
- `bike` - time formatted (long) bike_secs
- `mph` - formated bike_mph
- `t2` - time formatted (short) t2_secs
- `run` - time formatted (long) run_secs
- `mmile` - time formatted (short) run_mmile
- `result_url` - url of this entrant's race result
- `racer_url` - if entrant.racer.id is not nil

When formatting time of the fields

- time formatting should be in either HH:MM:SS format ("%k:%M:%S") for longer durations or MM:SS ("%M:%S") for shorter durations.
- floating point numbers like MPH should be to one decimal place
- you implemented helper methods in module 3 for the HTML pages to format the fields. They can be re-used here.

Hint: It is recommened that you implement this view as a partial so that it is re-usable when we go to render the resource collection in a follow-on step.

Hint: You should be able to use the following sources of information as reference

- your `app/views/races/show.html.erb` implementation form the previous assignment. The table constructed at the bottom of that web page will tell you where the data can be obtained from.
- the `JBuilder Reference Docs`
- the following sample partial snippet

```
# .../_result.json.jbuilder
json.place result.overall_place
json.time format_hours result.secs
json.last_name result.last_name
...
json.mmile format_minutes result.run_mmile
json.result_url api_race_result_url(result.race.id, result)
if result.racer.id
  json.racer_url api_racer_url(result.racer.id)
end
```

With the controller and view in place, you can demonstrate your new access to the race results using the rails console. In the example we start by locating a race in the past with results and a sample of one of its entrants. **Note** that we are using the named scope `past` implemented in the previous assignment to easily obtain a query returning races that have been held prior to today.

```
> race=Race.past.first
> race.name
 => "Upper Tract Iron"
> race.id
 => BSON::ObjectId('569a58b8e301d083c3000000')
```

```
> entrant=race.entrants.first
> entrant.first_name
 => "Tiesha"
> entrant.last_name
 => "Beck"
> entrant.id
 => BSON::ObjectId('569a58bfe301d083c30007f2')
```

We now issue the request for the entrant's results using the nested resource `races/:race_id/results/:id` and get back a full JSON document with our requested data.

```
> response=TriResultsWS.get("/api/races/#{race.id}/results/#{entrant.id}",
                            :headers=>{"Accept"=>"application/json"})
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.content_type
 => "application/json"
> response.content_length
 => 481
```

The following contains the information for the winner.

```
> pp response.parsed_response
{"place"=>1,
 "time"=>"10:17:16",
 "last_name"=>"Beck",
 "first_name"=>"Tiesha",
 "bib"=>11,
 "city"=>"Ampthill",
 "state"=>"VA",
 "gender"=>"F",
 "gender_place"=>1,
 "group"=>"30 to 39 (F)",
 "group_place"=>1,
 "swim"=>" 0:57:56",
 "pace_100"=>"01:30",
 "t1"=>"00:55",
 "bike"=>" 6:35:17",
 "mph"=>17.0,
 "t2"=>"00:41",
 "run"=>" 2:42:26",
 "mmile"=>"00:06",
 "result_url"=>
  "http://localhost:3000/api/races/569a58b8e301d083c3000000/results/569a58bfe301d083c30007f2",
 "racer_url"=>"http://localhost:3000/api/racers/569a58bde301d083c3000647"}
```

Next we pick a `Race` that is upcoming to show race results that have not yet occurred. Notice in this rendering we are chosing to show all fields (nil and non-nil). That will make it easier to identify the fields when we make updates.

```
> race=Race.upcoming.first
> entrant=race.entrants.first
> response=TriResultsWS.get("/api/races/#{race.id}/results/#{entrant.id}",
                            :headers=>{"Accept"=>"application/json"})
> pp response.parsed_response
{"place"=>nil,
 "time"=>" 0:00:00",
 "last_name"=>"Ruiz",
 "first_name"=>"Lynwood",
```

```
 "bib"=>1,
 "city"=>"Pentagon",
 "state"=>"DC",
 "gender"=>"M",
 "gender_place"=>nil,
 "group"=>"20 to 29 (M)",
 "group_place"=>nil,
 "swim"=>nil,
 "pace_100"=>nil,
 "t1"=>nil,
 "bike"=>nil,
 "mph"=>nil,
 "t2"=>nil,
 "run"=>nil,
 "mmile"=>nil,
 "result_url"=>
  "http://localhost:3000/api/races/569a58b8e301d083c300000c/results/569a58bfe301d083c30009d2",
 "racer_url"=>"http://localhost:3000/api/racers/569a58bbe301d083c3000489"}
```

```
$ rspec spec/resources_spec.rb -e rq02
```

3. Implement a PATCH `/api/races/:race_id/results/:id` action to accept race result updates. This action must:

   - be callable using `PATCH /api/races/:race_id/results/:id`
   - find the `Entrant` that is associated with the `:race_id` and `:entrant_id`
   - accept a payload that is keyed by `result` and has one or more of the following fields in it:
     - swim
     - t1
     - bike
     - t2
     - run

     These fields contain floating point numbers that are measuring in seconds the time taken to complete that particular leg of the race.
   - return a `200/OK` response when a successful update is made. The rspec test will only look for a `200/OK` and will not be providing an `Accept` header. This means that you may `render :nothing` or hard-code the format of what you render. To place a requirement on the return document would have taken the focus away from our primary point here – which is to implement the update.

   Note how we are using PATCH here to knife into the resource and tweak a few fields. We are not replacing the entire document with what we are posting.

   Hint: The previous assignment should have made the update easier but we are left to implement this ourselves as part of this assignment. The following is a reminder of how we can

   - initialize the results entry with the race event
   - update the time for the result

```
result=params[:result]
if result
  if result[:swim]
    entrant.swim=entrant.race.race.swim
    entrant.swim_secs = result[:swim].to_f
  end
  if result[:t1]
    entrant.t1=entrant.race.race.t1
    entrant.t1_secs = result[:t1].to_f
  end
```

```
  ...
  entrant.save
```

With your new PATCH action in place, you can demonstrate its functionality from the rails console. In the following example, we grab a random sample `Race` form the upcoming races and grab a random `Entrant` from that `Race`. The entrant starts out without any times and with equal `created_at` and `updated_at` timestamps.

```
> race=Race.upcoming.to_a.sample
> entrant=race.entrants.sample
> Entrant.where(id:entrant.id).pluck(:created_at, :updated_at, :secs)
 => [[2016-01-17 00:00:00 UTC, 2016-01-17 00:00:00 UTC, nil]]
```

The action is called five(5) times in total, with a value for each of the legs of the event. We can check the database in between calls to see the `secs` and `updated_at` changing.

```
> response=TriResultsWS.patch("/api/races/#{race.id}/results/#{entrant.id}",
              :body=>{:result=>{:swim=>15.minute}}.to_json,
              :headers=>{"Content-Type"=>"application/json"})
> Entrant.where(id:entrant.id).pluck(:created_at, :updated_at, :secs)
 => [[2016-01-17 00:00:00 UTC, 2016-01-24 09:05:52 UTC, 900.0]]
...
:body=>{:result=>{:t1=>1.minute}}.to_json
:body=>{:result=>{:bike=>40.minutes}}.to_json
:body=>{:result=>{:t2=>40.seconds}}.to_json
:body=>{:result=>{:run=>25.minutes}}.to_json
> Entrant.where(id:entrant.id).pluck(:created_at, :updated_at, :secs)
 => [[2016-01-17 00:00:00 UTC, 2016-01-24 09:08:45 UTC, 4900.0]]
```

Hint: If you are having difficulty passing this test due to 'CSRF token authenticty' types of errors, ensure all your API controllers are implementing the `protect_from_forgery` option, as shown with the `RacesController`.

```
**Hint**: If you are having difficulty passing this test due to a view template
not being found -- realize that this step does not require a document response
and your solution can either `render :nothing` or hard-code a format type as
long as it returns `200/OK` when the update completes successfully.
```

When we retrieve the results for the `Entrant` after posting the event times, we see that all times have been recorded and the overall time has been updated. The sum of the leg times should equal the entrant time.

```
> response=TriResultsWS.get("/api/races/#{race.id}/results/#{entrant.id}",
                            :headers=>{"Accept"=>"application/json"})
> pp response.parsed_response
{"place"=>nil,
 "time"=>" 1:21:40",
...
 "swim"=>" 0:15:00",
 "pace_100"=>"02:00",
 "t1"=>"01:00",
 "bike"=>" 0:40:00",
 "mph"=>18.0,
 "t2"=>"00:40",
 "run"=>" 0:25:00",
 "mmile"=>"00:08",
...
```

```
$ rspec spec/resources_spec.rb -e rq03
```

4. Update the controller method that is implementing the action for GET `/api/races/:race_id/results` so that it finds the specified `Race` and its entrants. This action must:

   - obtain the `:race_id` provided in the URI
   - find the `Race` in the database
   - make the collection entrants for the `Race` available to the view

Hint: The following is a snippet of code that you can likely drop into your existing controller method implementing the action.

```
@race=Race.find(params[:race_id])
@entrants=@race.entrants
```

What you are now adding should implement the stub you have in place from the URI section. you added in the initial URI section of this assignment.

```
  if !request.accept || request.accept == "*/*"
    render plain: "/api/races/#{params[:race_id]}/results"
  else
    #implementation here
  end
```

At the completion of this step you should be either be:

- manually calling a view similar to how the `HelloController#say method` worked in the class examples.
- using automatic routing the way the `HelloController#sayhello` worked in the class examples.

With that in place, you likely are getting an error stating a 404/NOT FOUND on the client-side and a template or partial not found on the server-side.

There is no test for this step.

5. Implement an `XXX.json.jbuilder` script to act as the view the above controller is trying to render. We will use a custom view so that we can better control what goes into the data marshalled back to the client. It is recommened that the individual rows of this collection view be implemented with the partial you recently implemented in a previous step. The document returned to the client must have:

   - an array element for each `Entrant` result in the collection
   - the same `Entrant` result properties defined in the previous step in this section.

   Hint: The following snippet shows how to form the collection loop and invoke the partial. Dependending on what you called your partial and whether you even used a partial – yours may look different.

   - the `ignore_nil!` was added to have nil fields not included in the rendered response
   - the `json.array!` iterates over the collection provided and passed the element into the body using the `element` variable.
   - the `json.partial!` renders the `result` partial, passing in `entrant` to be used as the `result` veriable in that partial.

```
# app/views/api/results/index.json.jbuilder
json.ignore_nil!   #don't marshal nil values
json.array!(@entrants) do |entrant|
  json.partial! "result", :locals=>{ :result=>entrant }
end
```

With your new controller and view in place for race results, you should be able to demonstrate getting race results using the rails console. In this example we start off by locating a race that is in the past. The following query makes use of a named scope `past` implemented in the previous assignment.

```
> race=Race.past.first
> race.name
 => "Upper Tract Iron"
> race.date
 => Mon, 20 Oct 2014
```

We start off by issuing the request for the `Race`. This was implemented in a previous step of this assignment. This shows we are getting back the `Race` we expect.

```
response=TriResultsWS.get("/api/races/#{race.id}",
                          :headers=>{"Accept"=>"application/json"})
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
```

```
> response.content_type
 => "application/json"
> response.parsed_response
 => {"name"=>"Upper Tract Iron", "date"=>"2014-10-20"}
```

We now issue the request for the results using the nested collection resource below the races/:race_id resource and get back a full JSON document with our requested data. You may want to look at this data using a browser to be able see the information better.

```
> response=TriResultsWS.get("/api/races/#{race.id}/results",
                            :headers=>{"Accept"=>"application/json"})
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.content_type
 => "application/json"
> response.content_length
 => 9681
> response.parsed_response.count
 => 20
```

The following is a sample of the response showing the first place winner.

```
> pp response.parsed_response.first
{"place"=>1,
 "time"=>"10:17:16",
 "last_name"=>"Beck",
 "first_name"=>"Tiesha",
 "bib"=>11,
 "city"=>"Ampthill",
 "state"=>"VA",
 "gender"=>"F",
 "gender_place"=>1,
 "group"=>"30 to 39 (F)",
 "group_place"=>1,
 "swim"=>" 0:57:56",
 "pace_100"=>"01:30",
 "t1"=>"00:55",
 "bike"=>" 6:35:17",
 "mph"=>17.0,
 "t2"=>"00:41",
 "run"=>" 2:42:26",
 "mmile"=>"00:06",
 "racer_url"=>"http://localhost:3000/api/racers/569a58bde301d083c3000647.json"}
```

Next we pick a `Race` that is upcoming to show race results that have not yet occurred.

```
> race=Race.upcoming.first
> response=TriResultsWS.get("/api/races/#{race.id}/results",:headers=>{"Accept"=>"application/json"})
> pp response.parsed_response.first
{"time"=>" 0:00:00",
 "last_name"=>"Ruiz",
 "first_name"=>"Lynwood",
 "bib"=>1,
 "city"=>"Pentagon",
 "state"=>"DC",
 "gender"=>"M",
 "group"=>"20 to 29 (M)",
 "racer_url"=>"http://localhost:3000/api/racers/569a58bbe301d083c3000489.json"}
```

```
$ rspec spec/resources_spec.rb -e rq05
```

Headers and Caching

In this section we are going to repeat some of the earlier actions except with some conditional behavior based on caching headers.

1. Update the `results` controller so that when the collection of race results is returned; a `Last-Modified` header is returned with it. This value must:

   - represent the newest `updated_at` property of all `Entrant` documents associated with the Race.

   Hint: `race.entrants.max(:updated_at)` **Hint**: Use `fresh_when` to set the `Last-Modified` value as the newsest `updated_at` timestamp for all the entrants within the race.

   With your `Last-Modified` solution in place, you can demonstrate their presence using the rails console.

   ```
   > response=TriResultsWS.get("/api/races/#{race.id}/results",
                               :headers=>{"Accept"=>"application/json"})
   > response.response
    => #<Net::HTTPOK 200 OK  readbody=true>
   > response.header["Last-Modified"]
    => "Mon, 18 Jan 2016 09:23:07 GMT"
   ```

   ```
   $ rspec spec/caching_spec.rb -e rq01
   ```

2. Update the previous action to only return an update, if the client does not already have the current version of the race results. This method must:

   - locate the `Race` in the database
   - locate the newest `updated_at` value for each of its `Entrants`
   - look for and process the `If-Modified-Since` header
   - compare the timestamp of the `If-Modified-Since` header with the newest entrant timestamp and short circuit the call (returning a 304/Not-Modified) if the client already has the lastest copy of the data.

   Hint: Replace `fresh_when` with a conditional block using `stale?`.

   With your new conditional logic in place, you can demonstrate this using the rails console. In the example below, we seed the scenario with an initial request of the full race results. A `Last-Modified` is returned, which should represent the last time updated for the newest update across all `Entrants` in the Race. We can verify that as well.

   ```
   > response=TriResultsWS.get("/api/races/#{race.id}/results",
                               :headers=>{"Accept"=>"application/json"})
   > response.response
    => #<Net::HTTPOK 200 OK  readbody=true>
   > last_modified=response.header["Last-Modified"]
    => "Mon, 18 Jan 2016 09:23:07 GMT"
   > response.content_length
    => 6420
   ```

   Request a copy of the `Race` results using a last-modified that is up to date with the database's current state.

   ```
   > last_modified
     => "Mon, 18 Jan 2016 09:34:09 GMT"
   > response=TriResultsWS.get("/api/races/#{race.id}/results",
              :headers=>{"Accept"=>"application/json",
                         "If-Modified-Since"=>last_modified})
   > response.response
    => #<Net::HTTPNotModified 304 Not Modified  readbody=true>
   > response.header["Last-Modified"]
    => "Mon, 18 Jan 2016 09:34:09 GMT"
   ```

   ```
   $ rspec spec/caching_spec.rb -e rq02
   ```

3. Issue an update to one of the entrant's results and re-issue the GET from the previous step.

   In the following example we:

- locate an entrant for a race
- extract their URL
- issue a result PATCH to that URL

```
> response=TriResultsWS.get("/api/races/#{race.id}/results",
                            :headers=>{"Accept"=>"application/json"})
> result_url=response.parsed_response.first["result_url"]
 => "http://localhost:3000/api/races/569a58b8e301d083c300008a/results/569a58c4e301d083c3002412"
> response=HTTParty.patch(result_url,:body=>{:result=>{:swim=>15.minute}}.to_json,
          :headers=>{"Content-Type"=>"application/json","Accept"=>"application/json"})
```

Hint: If you are having difficulty with passing this test due to 'CSRF token authenticty' types of errors, ensure all your API controllers are implementing the `protect_from_forgery` option, as shown with the `RacesController`.

When we obtain the newest `updated_at` timestamp in the database for the race we notice that the value is now newer than the last_modified we had been using.

```
> last_modified
 => "Sun, 24 Jan 2016 09:08:45 GMT"
> race.entrants.max(:updated_at)
 => 2016-01-24 09:50:02 UTC
```

When we re-issue the GET passing the older `Last-Modified` timestamp in the `If-Modified-Since` header, we should get a `200/OK` with a full updated. This is because our latest updated caused the status of this resource to be newer than the client previously held.

```
> last_modified
 => "Mon, 18 Jan 2016 09:34:09 GMT"
> response=TriResultsWS.get("/api/races/#{race.id}/results",
           :headers=>{"Accept"=>"application/json",
                      "If-Modified-Since"=>last_modified})
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.content_type
 => "application/json"
> response.header["Last-Modified"]
 => "Sun, 24 Jan 2016 09:50:02 GMT"
```

```
$ rspec spec/caching_spec.rb -e rq03
```

Self Grading/Feedback

Some unit tests have been provided in the bootstrap files and provide examples of tests the grader will be evaluating for when you submit your solution. They must be run from the project root directory.

```
$ rspec (file)
...
(N) examples, 0 failures
```

You can run as many specific tests you wish be adding `-e rq## -e rq##`

```
$ rspec (file) -e rq01 -e rq02
```

Submission

Submit an .zip archive (other archive forms not currently supported) with your solution root directory as the top-level (e.g., your Gemfile and sibling files must be in the root of the archive and *not* in a sub-folder. The grader will replace the spec files with fresh copies and will perform a test with different query terms.

```
|-- app
|   |-- assets
```

```
|   |-- controllers
|   |-- helpers
|   |-- mailers
|   |-- models
|   `-- views
|-- bin
|-- config
|-- config.ru
|-- db
|-- Gemfile
|-- Gemfile.lock
|-- lib
|-- log
|-- public
|-- Rakefile
|-- README.rdoc
|-- test
`-- vendor
```

Last Updated: 2016-05-20