Module 3, Lesson 1: Mongoid Setup and Document CRUD

The overall goal of the assignment is to give you practice in:

- Setting up an application for use with Mongoid
- Mapping and interacting with Mongoid Document class instances
- Creating new and updating existing documents in the database

The functional goal of the assignment is to:

- To configure a Mongoid database connection to build a data tier for Race results.
- Create, update, find, and delete `Racer` instances.

This is an introductory practice exercise that is intended to give you experience with some of the most basic CRUD capabilities within Mongoid. Most of the code is provided to you and all of the development is through the interactive `rails console`. There is nothing to turn in.

Since the development is through the `rails console`, the tests provided will be evaulating the state of the database after your CRUD interactions. As a result you must ensure that you are operating on the 'test' database that the rspec tests are running against. This simply requires the addition of an option when you start up the console:

```
$ rails c -e test
```

The console will indicate that the test environment is loaded after you issue the command.

You can find details of the Mongoid API on the Mongoid Tutorial Page

Functional Requirements

1. Setup the database and get a connection and reference to a document collection.
2. Instantiate a Mongoid object and change its attribute state.
3. Create, find, update and delete documents to/from the database.

Gettings Started

1. Start your MongoDB server using `mongod`
2. Create a new Rails application called `triracers`.

   ```
   $ rails new triracers
   $ cd triracers
   ```

3. Download and extract the starter set of files. The root directory of this starter set will be referred to as the root directory of your solution. When extracted correctly – the `spec` folder and `Gemfile` should be at the same (root) level.

   ```
   --- student-start
       |-- .rspec (important hidden file)
       `-- spec
           |-- test_utils.rb
           |-- lecture1_spec.rb
           |-- lecture2_spec.rb
           |-- lecture3_spec.rb
           |-- rails_helper.rb
           `-- spec_helper.rb
   ```

- spec - this directory contains tests to verify your solution. You should not modify anything in this directory

4. Implement the technical requirements.
5. Run the `rspec` command from the project root directory. The spec files are written per-lecture. The steps taken in one lecture can impact the results of a preceding lecture. However, if you execute all sections correctly, you will be able to execute all rspec tests at the end and pass.

```
$ rspec

(N) examples, (N) failures
...
```

Technical Requirements

Lecture 1: Mongoid Connection

In this section we will focus on integrating Mongoid into an existing application and gaining access to a client connection and collections thru Mongoid.

1. Add the `mongoid` gem to your Gemfile and run `bundle`.

```
gem 'mongoid', '~> 5.0.0'

$ bundle
```

Also add the 'rspec-rails' gem for executing the tests and run `bundle` again

```
gem 'rspec-rails', '~> 3.0'

$ bundle
```

2. Generate a `mongoid.yml` configuration file.

```
$ rails g mongoid:config
      create  config/mongoid.yml
```

3. Inspect the generated defaults in the file and determine if there are any changes required for your development environment. The file is well commented and shows other options that are available but will not be used in class.

```
$ egrep -v '#|^$' config/mongoid.yml
development:
  clients:
    default:
      database: triracers_development
      hosts:
        - localhost:27017
      options:
  options:
test:
  clients:
    default:
      database: triracers_test
      hosts:
        - localhost:27017
      options:
        read:
          mode: :primary
        max_pool_size: 1
```

4. Add the `mongoid.yml` file to the `config/application.rb` file so that tools like `rails console` do not have to manually load the configuration.

```
module Triracers
  class Application < Rails::Application
    ...
    #bootstraps mongoid within applications -- like rails console
    Mongoid.load!('./config/mongoid.yml')
```

5. [Optionally] set the default `orm` template engine for `rails generate` commands within the same `config/application.rb` file. With the `mongoid` gem installed within your application – `mongoid` will be the default `orm` engine. You can use this technique to set the default engine back to `active_record` or specify the `orm` engine to use on the command line with the `--orm (name)` option.

```
#which default ORM are we using with scaffold
#add  --orm none, mongoid, or active_record
#    to rails generate cmd line to be specific
#config.generators {|g| g.orm :active_record}
#config.generators {|g| g.orm :mongoid}
```

6. Generate a skeletal Mongoid `Racer` model class using the `rails generate model` command. No properties are required at this time.

```
$ rails g model Racer
```

7. Use the `rails console` to verify you can obtain a connection to the database and access the collection associated with `Racer`.

```
$ rails c -e test

> Racer.mongo_client
 => #<Mongo::Client:0x43995560 cluster=localhost:27017>
> Racer.collection
 => #<Mongo::Collection:0x43955740 namespace=triracers_test.racers>
> Racer.collection.name
 => "racers"
> Racer.count
 => 0
```

```
$ rspec spec/lecture1_spec.rb
```

Lecture 2: Model Class/Document

This section focuses on mapping a Rails model class to a MongoDB collecion.

1. Re-generate the Mongoid `Racer` model class using the `rails generate model` command.

   - **Hint 1**: `rails d model (ModelName)` will remove an existing model class and fixtures.
   - **Hint 2**: `rails g model (ModelName)` will prompt you before overwriting an existing file.
   - **Hint 3**: adding the `--force` option will overwrite existing files without prompts.

   This class must:

   - be named `Racer`
   - include the `Mongoid::Document` mixin
   - have a `first_name` attribute defined to hold a value of type String
   - have a `last_name` attribute defined to hold a value of type String

- have a `date_of_birth` attribute defined to hold a value of type Date
- have a `gender` attribute defined to hold a value of type String

```
$ rails g model Racer ...
```

2. Map the class properties to documentation properties found within our documents. This is not always necessary or desired, but being able to form a class that is consistent with Rails naming conventions and then custom mapped to the names of the collection and document fields isolates the possible differences from the rest of the application. Your model class mappings must:

   - map the model class to be stored in the `racer1` collection
   - alias the model class `first_name` attribute to the `fn` document property
   - alias the model class `first_name` attribute to the `ln` document property
   - alias the model class `date_of_birth` attribute to the `dob` document property
   - keep the model class `gender` attribute mapped to the `gender` document property

   You can create an empty instance at this point to print a debug message that shows the field to document mapping.

   Hint: use the `reload!` command to see the new implementation of your model class within `rails console`.

```
> reload!
> Racer.new
 => #<Racer _id: 5674dafce301d01ff2000001,
     fn(first_name): nil, ln(last_name): nil, dob(date_of_birth): nil, gender: nil>
```

3. Add support for timestamps to the model class. Your model class must:

   - include the `Mongoid::Timestamps` mixin

   At this point you can create an empty `Racer` instance to print a debug message. This will show the *field-to-document* mapping to include the timestamp information (e.g., `created_at` and `updated_at`). The timestamps are currently `nil` since the instance has not been saved to the database.

```
> Racer.new
 => #<Racer _id: 5674dbf6e301d01ff2000002, created_at: nil, updated_at: nil,
     fn(first_name): nil, ln(last_name): nil, dob(date_of_birth): nil, gender: nil>
```

4. Using the `rails console`, create an instance of your `Racer` class using a bulk assignment with a mixture of class and document mappings. **Do not save it to the database yet**.

```
> r=Racer.new(:fn=>"cat", :ln=>"hat")
 => #<Racer _id: 5674dd1ee301d01ff2000004, created_at: nil, updated_at: nil,
     fn(first_name): "cat", ln(last_name): "hat", dob(date_of_birth): nil, gender: nil>
```

   Notice that the document was populated with the document field names and the class instance reports both the document and mapped (`class_field_names`) for each property.

```
> r.attributes
 => {"_id"=>BSON::ObjectId('5674dd1ee301d01ff2000004'), "fn"=>"cat", "ln"=>"hat"}

> r
 => #<Racer _id: 5674dd1ee301d01ff2000004, created_at: nil, updated_at: nil,
     fn(first_name): "cat", ln(last_name): "hat", dob(date_of_birth): nil, gender: nil>
```

5. Use the various getters to obtain the state of your instance.

```
> r.first_name
 => "cat"
> r.fn
 => "cat"
> r[:fn]
 => "cat"
> r.read_attribute(:fn)
 => "cat"
```

6. Use the various setters to change the state of your instance.

   Change the `first_name` of the instance using the model class setter method.

   ```
   > r.first_name="thing"
    => "thing"
   ```

   Change the `last_name` of the instance using the `hash[:key]` assignment method.

   ```
   > r[:last_name]="one"
    => "one"
   ```

   Set the `dob` document field using the `write_attribute` reflection method.

   ```
   > r.write_attribute(:dob, Date.new(1957,3,12))
    => 1957-03-12 00:00:00 UTC
   ```

7. From the list of methods for the object, inspect the state of your instance by locating a method we believe is called `changed`. Use the `methods` command to get a (huge) list of all methods that can be called. Then, use the array `grep` command to reduce that list to methods that are related to our `changed` search criteria.

   ```
   > r.methods.grep /changed/
    => [:changed_for_autosave?, ... :changed?, :children_changed?, :changed_attributes]
   ```

   Call the `changed?` command to determine if the object needs to be saved.

   ```
   > r.changed?
    => true
   ```

   Verify the state was is not yet written to the database by using the collection.

   ```
   > Racer.collection.find(:_id=>r.id).first
    => nil
   ```

8. Save your instance to the database using the `save` method.

   ```
   > r.save
    => true
   ```

   Verify the object reports that it has been saved.

   ```
   > r.persisted?
    => true
   > r.changed?
    => false
   ```

   Verify the state is in the database.

   ```
   > Racer.collection.find(:_id=>r.id).first
    => {"_id"=>BSON::ObjectId('5674dd1ee301d01ff2000004'),
     "fn"=>"thing", "ln"=>"one", "dob"=>1957-03-12 00:00:00 UTC,
     "updated_at"=>2015-12-19 04:37:31 UTC, "created_at"=>2015-12-19 04:37:31 UTC}
   ```

9. Now that your object is persisted, modify and save its state such that it has the following properties:

   - `first_name` => "Sally"
   - `last_name` => nil
   - `gender` => "F"
   - `date_of_birth` => 1957-01-01

   You can inspect your updated object instance and database state using the Rails console.

```
> r
 => #<Racer _id: 5674dd1ee301d01ff2000004,
   created_at: 2015-12-19 04:37:31 UTC, updated_at: 2015-12-19 04:40:30 UTC,
   fn(first_name): "Sally", ln(last_name): nil,
   dob(date_of_birth): 1957-01-01 00:00:00 UTC, gender: "F">

> r.attributes
 => {"_id"=>BSON::ObjectId('5674dd1ee301d01ff2000004'),
    "fn"=>"Sally", "ln"=>nil, "dob"=>1957-01-01 00:00:00 UTC,
    "updated_at"=>2015-12-19 04:40:30 UTC, "created_at"=>2015-12-19 04:37:31 UTC,
    "gender"=>"F"}
```

```
$ rspec spec/lecture2_spec.rb
```

Lecture 3: Basic CRUD

In the previous section you instantiated and interacted with a transient instance. This section concentrates on creating and deleting document objects.

1. Create a new instance of a `Racer` within the database without going through `new/save`. Use the `create` method to insert a document with the following `Entrant` information, while also manually assigning the `_id` of the object.

   - `_id` => 1
   - `fn` => "cat"
   - `ln` => "inhat"

   Your output from the shell evaluation should be as follows:

```
 => #<Racer _id: 1,
   created_at: 2015-12-19 16:06:27 UTC, updated_at: 2015-12-19 16:06:27 UTC,
   fn(first_name): "cat", ln(last_name): "inhat", dob(date_of_birth): nil, gender: nil>
> Racer.collection.find(:_id=>1).first
 => {"_id"=>1, "fn"=>"cat", "ln"=>"inhat",
   "updated_at"=>2015-12-19 16:06:27 UTC, "created_at"=>2015-12-19 16:06:27 UTC}
```

2. Accidentally (on purpose) attempt to create the same object a second time.

   Hint: use up arrow for command recall

   Your output from the shell command will include the following:

```
Mongo::Error::OperationFailure: E11000 duplicate key error index: ....$_id_
    dup key: { : 1 } (11000)
```

3. Create a transient instance of the object this time (with `new` instead of `create`) with the same `_id` but different field properties. Use the `upsert` method to update any existing object with the assigned `_id`, if found, or create a new instance. Either way, we will end up with a document in the database with the provided identifiying information.

```
> r=Racer.new(:_id=>1, :fn=>"thing", :ln=>"one")
> r.upsert
> r
 => #<Racer _id: 1, created_at: nil, updated_at: nil,
   fn(first_name): "thing", ln(last_name): "one",
   dob(date_of_birth): nil, gender: nil>
> Racer.collection.find(:_id=>1).first
 => {"_id"=>1, "fn"=>"thing", "ln"=>"one"}
```

One annoying thing to notice is that the original `created_at` was lost and the `updated_at` was set to nil instead of the upsert time. You can compensate for the loss of `update_at` using the `touch` command.

```
> r.touch
> r
 => #<Racer _id: 1, created_at: nil, updated_at: 2015-12-19 16:16:50 UTC,
   fn(first_name): "thing", ln(last_name): "one",
   dob(date_of_birth): nil, gender: nil>
> Racer.collection.find(:_id=>1).first
 => {"_id"=>1, "fn"=>"thing", "ln"=>"one", "updated_at"=>2015-12-19 16:16:50 UTC}
```

You can automate the generation of the `update_at` by adding the following callback to the `Racer` model class.

```
class Racer
  ...
  include Mongoid::Timestamps
  ...

  before_upsert do |doc|
    doc.set_updated_at
  end
end
```

The following is a complete sequence with the callback included. We still lose the `created_at` during the upsert but the `updated_at` is auotmatically inserted prior to the upsert call being made to the database.

```
> reload!
> Racer.find(1).delete
> Racer.create(:_id=>1, :fn=>"cat", :ln=>"inhat")
 => #<Racer _id: 1,
   created_at: 2015-12-19 17:47:00 UTC, updated_at: 2015-12-19 17:47:00 UTC,
   fn(first_name): "cat", ln(last_name): "inhat", dob(date_of_birth): nil, gender: nil>
> r=Racer.new(:_id=>1, :fn=>"thing", :ln=>"one")
 => #<Racer _id: 1,
   created_at: nil, updated_at: nil,
   fn(first_name): "thing", ln(last_name): "one", dob(date_of_birth): nil, gender: nil>
> r.upsert
 => true
> r
 => #<Racer _id: 1,
   created_at: nil, updated_at: 2015-12-19 17:48:00 UTC,
   fn(first_name): "thing", ln(last_name): "one", dob(date_of_birth): nil, gender: nil>
> Racer.collection.find(:_id=>1).first
 => {"_id"=>1, "fn"=>"thing", "ln"=>"one", "updated_at"=>2015-12-19 17:48:00 UTC}
```

4. Create a second instance with a unique `_id` and properties. This time you should get a new instance created.

   - `_id` => 2
   - `fn` => “thing”
   - `ln` => “two”

```
> two=Racer.new ...
 => #<Racer _id: 2,
   created_at: nil, updated_at: nil,
   fn(first_name): "thing", ln(last_name): "two", dob(date_of_birth): nil, gender: nil>
> two.upsert
 => true
> two
 => #<Racer _id: 2,
   created_at: nil, updated_at: 2015-12-19 17:54:06 UTC,
   fn(first_name): "thing", ln(last_name): "two", dob(date_of_birth): nil, gender: nil>
> Racer.collection.find(:_id=>2).first
 => {"_id"=>2, "fn"=>"thing", "ln"=>"two", "updated_at"=>2015-12-19 17:54:06 UTC}
```

Note that because the upsert is implemented in the database, the client cannot reliably supply a conditional `created_at` with upsert.

5. Get a reference to `Racer.id=1` and issue a destroy (or delete to bypass callbacks) on that specific Racer.

```
> Racer.where(:id=>1).first
 => #<Racer _id: 1,
   created_at: nil, updated_at: 2015-12-19 17:48:00 UTC,
   fn(first_name): "thing", ln(last_name): "one", dob(date_of_birth): nil, gender: nil>
> Racer.where(:id=>1).delete
 => 1
> Racer.where(:id=>1).first
 => nil
```

6. Create a `Racer` also with the name "Sally" in the database. This first name should match the `Racer` created in the lecture 2 section.

   After you have created your new Racer, you should have two racers with the `first_name` of "Sally" if you have the `Racer` from the previous section. If not – you will have just a single `Racer`.

```
> Racer.where(:first_name=>"Sally").map {
  |r| "#{r.id} #{r.fn}, #{r.ln}, dob=#{r.dob}, created_at=#{r.created_at}"}
 => ["5674dd1ee301d01ff2000004 Sally, , dob=1957-01-01, created_at=2015-12-19 04:37:31 UTC",
     "5675a659e301d01ff2000005 Sally, , dob=, created_at=2015-12-19 18:47:53 UTC"]
```

7. Delete all `Racers` not having the `first_name` of `Sally` using the `where()` and `destroy_all()` methods. We have not specifically covered the `where()` function and syntax yet and where it differs from `collection.find()`, but it plays an important role with `delete/destroy_all()`. Append ".ne" to the property name when testing for inequality. With the `Racer` from the previous section – you should complete this assignment with two `Racers` remaining and they both should have the `first_name` of Sally.

   After you destroy all racers not having the `first_name` as `Sally`, you should end up with `0` matching the following query.

```
> Racer.where(:first_name.ne=>"Sally").count
 => 0
```

```
$ rspec spec/lecture3_spec.rb
```

Self Grading/Feedback

Unit tests have been provided in the bootstrap files that can be used to evaluate your solution. They must be run from the same directory as your solution.

```
$ rspec
........

(N) examples, 0 failures
```

Submission

There is no submission required for this assignment but the skills learned will be part of a follow-on assignment so please complete this to the requirements of the unit test.

Last Updated: 2016-01-30