Module 2, Lesson 2: Schema Design

In this assignment you will get some practice reshaping documents and relationships according to how you wish to represent individual document fields and document relationships.

The overall goal of the assignment is to:

- change the data type for fields in a document
- add missing fields in documents
- normalize fields in a document
- change the structure of documents to support different relationships

The functional goal of the assignment is to:

- update the `Racer` document fields to be a more well-designed schema

This assignment is written as a practice exercise with self grading. You were shown many of the commands used here in a previous lesson and now called to use them as a part of implementing better document schema. You will perform all of it from within the `irb` shell. All queries are provided for you to inspect, modify, and verify the contents of the database. There is no test or anything to turn in but the format should leave you in a state where you are confortable making schema changes in your collections with the hands on experience and examples covered here.

Getting Started

1. Start your MongoDB server
2. Download and extract the starter set of files. The instructions are written assuming you are inside this directory.

   ```
   --- student-start
       |-- racer.rb
       `-- race2_results.json
   ```

   - racer.rb - a helper class to obtain MongoDB server connections and access convenience methods to load and reset the test data
   - race2_results.json - test data for the exercise.
3. Install the following gems. You may already have them installed.

   ```
   $ gem install mongo -v 2.1.2
   ```
4. Helper methods have been provided to get a connection to MongoDB server and collections. You can override these values using environment variables if you are not using the defaults.

   - MONGO_URL="mongodb://localhost:27017"
   - MONGO_DATABASE="test"
   - RACE_COLLECTION="race2"
   - RACER_COLLECTION="racer2"
5. Prepare your `irb` shell environment by executing the following commands.

   ```
   $ irb
   > require "./racer.rb"
   > Racer.reset
    => 1000
   ```

`Racer.reset` takes an optional path to a data file, deletes everything in the collection, and loads in a fresh copy of the data. Nothing is touched if the file does not exist. If successfull, 1000 documents will be in the initial race results collection (`race2`) and no documents will be in the racer collection (`racer2`).

6. You may reset your environment at any time by calling `Racer.reset`. You can also gain access to the specific collections being used by doing the following:

```
> Racer.reset
 => 1000
> races=Racer.races_collection
 => #<Mongo::Collection:0x10465900 namespace=test.race2>
> races.name
 => "race2"
> races.count
 => 1000
> racers=Racer.racers_collection
 => #<Mongo::Collection:0x9918740 namespace=test.racer2>
> racers.name
 => "racer2"
> racers.count
 => 0
```

Since we will be using the races collection the most, the default `Racer.collection` and `Racer.races_collection` both reference the same thing.

```
> Racer.collection.name
 => "race2"
```

Exercise

Field Normalization

Part 1: Consistent Field Types

In this section you are going to locate fields of a particular type using the `$type` operator and update the field to a desired type using `update_one`.

1. Locate a sample of `:number` fields from the documents in the database and notice how some are strings and others are integer type.

```
> Racer.collection.find(name: {:$regex=>"JONES"}).projection({_id:0, number:1}).to_a
 => [{"number"=>7}, {"number"=>"8"}]
```

2. Use the following query to find all `:number` properties of type `String`. We use the enum ordinal `2` for the BSON data type for String. (BSON Types)

```
> Racer.collection.find(number: {:$type=>2}).count
 => 492
```

`16` is the ordinal type for a 32-bit integer.

```
> Racer.collection.find(number: {:$type=>16}).count
 => 508
```

3. Use the `update_one` operator to update only the `:number` field of documents with `:numbers` of String type with the Integer form of the String using `to_i`.

```
> racer=Racer.collection
> racer.find(number: {:$type=>2}).each do |r|
    racer.update_one({:_id=>r[:_id]}, {:$set=>{:number=>r[:number].to_i}})
  end

> Racer.collection.find(number: {:$type=>2}).count
 => 0
> Racer.collection.find(number: {:$type=>16}).count
 => 1000
```

Part 2: Consistent Fields Supplied

In this part we are going to fix some inconsistencies between documents where some are missing the `:gender` field. We are going to make use of the `$nil` comparison value and the `update_many` function to correct these documents.

1. Use the following query to locate all documents that do not have a `:gender` field.

   ```
   > Racer.collection.find(gender:$nil).count
   => 508
   ```

2. Use the following query to add the `:gender=>"F"` to each document missing that field. This assumes we already know that each document missing a `:gender` is female.

   ```
   > result=Racer.collection.find(gender:$nil).update_many(:$set=>{:gender=>"F"})
    => #<Mongo::Operation::Result:21076800 documents=[{"ok"=>1, "nModified"=>508, "n"=>508}]>
   > result.modified_count
    => 508
   > Racer.collection.find(gender:$nil).count
    => 0
   ```

Part 3: Normalized Fields (compound values)

In this part we are going to replace the compound `:name` field with a normalized `:first_name` and `:last_name` field. Luckily we know that in our database, all first and last names are a single word (no spaces) and first is always before last in the existing `:name` field.

1. Use the following query to show the `:name` fields. Notice how the first and last name of each person is combined into a single field. The later query tells us how many documents in the collection still have a `:name` field.

   ```
   > Racer.collection.find.projection({_id:0, name:1}).limit(3).to_a
    => [{"name"=>"ARACELY SMITH"}, {"name"=>"JOEL JOHNSON"}, {"name"=>"MARTIN JOHNSON"}]
   > Racer.collection.find(:name=>{:$exists=>true}).count
    => 1000
   ```

2. Use the following query to locate all documents with a `:name` field, extract the `first_name` and `last_name` from the `:name` field with a regular expression, and update the document. The document will have the two new fields added and the older one removed.

   ```
   > racers=Racer.collection
   > racers.find(:name=>{:$exists=>true}).each do |r|
       matches=/(\w+) (\w+)/.match r[:name]
       first_name=matches[1]
       last_name=matches[2]
       racers.update_one({:_id=>r[:_id]},
                         {:$set=>{:first_name=>first_name, :last_name=>last_name},
                          :$unset=>{:name=>""}})
     end
   ```

3. Verify all `:name` properties were deleted and we now have `:first_name` and `:last_name` fields.

```
> racers.find(:name=>{:$exists=>false},
              :first_name=>{:$exists=>true},
              :last_name=>{:$exists=>true}).count
 => 1000
```

Normalizing Relationships

For the purposes of the next sections, it has been discovered that racers race in many races and we need to model our `Racer` and `Race` as a many-to-one relationship.

Part 1: Creating a Linked Relationship

In this section we are going to create a separate collection for the `Racer` with `name` and a new `_id`. We are going to remove the `name` property from the `Races` collection and insert a new `racer_id` set to the `_id` of the `Racer`.

1. Use the following to create references to the two collections. If they do not yet exist – they will be automatically created.

```
> racers=Racer.racers_collection
> races=Racer.races_collection
```

2. Use the following to reset the collections back to a default state.

```
> Racer.reset
```

3. Use the following queries and algorithm to:

   - locate all `Races` that still have a `name`
   - upsert the `Racer` by attempting to locate them by name and creating a new document if not found
   - create a new `racer_id` property in the `Race` set to the `upserted_id` of the of the `Racer`

```
> races.find(:name=>{:$exists=>true}).count
 => 1000
> racers.find.count
  => 0

> races.find(:name=>{:$exists=>true}).each do |r|
    result=racers.update_one({:name=>r[:name]}, {:name=>r[:name]}, {:upsert=>true})
    id=result.upserted_id
    races.update_one({:_id=>r[:_id]},{:$set=>{:racer_id=>id},:$unset=>{:name=>""}})
  end

> races.find(:name=>{:$exists=>true}).count
 => 0
> racers.find.count
 => 1000
```

4. Use the following query to inspect both collections and perform a client "join" on the two collections. Your `first` document may be different than the example shown below.

```
> racers.find.first
 => {"_id"=>BSON::ObjectId('563ef169e301d0dc61002e51'), "name"=>"FREDERICKA BALDWIN"}
> pp races.find( :racer_id => racers.find.first[:_id] ).first; nil
 => {"_id"=>BSON::ObjectId('563ef169e301d0dc61002e51'),
     "number"=>857,
     "group"=>"20 to 20",
     "time"=>"3594 secs",
     "racer_id"=>BSON::ObjectId('563ef169e301d0dc61002e51')}
```

Part 2: Creating an Embedded Relationship

In this section we are going to create an embedded representation of the relationship with `Races` inside the `Racer`.

1. Use the following query and algorithm to:

   - locate all `Races` with `Racer` names
   - create a new `Racer` and insert the `Race` as the first element within an array within `Racer` called `races`
   - remove the `Race`

```
> Racer.reset
> racers=Racer.racers_collection
> races=Racer.races_collection
> races.find(:name=>{:$exists=>true}).each do |r|
    result=racers.update_one({:name=>r[:name]},
                             {:name=>r[:name],
                              :races=>[
                                 {:_id=>r[:_id],
                                  :number=>r[:number],
                                  :group=>r[:group],
                                  :time=>r[:time]}
                                 ]}, {:upsert=>true})
    races.find(:_id=>r[:_id]).delete_one
  end
  > races.find.count
   => 0
  > racers.find.count
   => 1000
```

2. Use the following query to look at the structure of the document. Notice the `Race` is embedded within an array within the `Racer`.

```
> pp racers.find.first; nil
{"_id"=>BSON::ObjectId('563efb5e0878cba85bcb014a'),
 "name"=>"AMPARO WEBER",
 "races"=>
  [{"_id"=>BSON::ObjectId('563efb35e301d0dc610040e5'),
    "number"=>613,
    "group"=>"14 and under",
    "time"=>"3489 secs"}
  ]}
```

3. Use the following query to unwind the embedded we can re-create a 1:1 relationship between the `Racer` and one of its `Races` by creating copies of the `Racer` for each `Race` in the array.

```
> racers.find.aggregate([ {:$unwind=>"$races"}, {:$project=>{_id:0, name:1, races:1}} ]).first
 => {"name"=>"AMPARO WEBER",
     "races"=>{"_id"=>BSON::ObjectId('563efb35e301d0dc610040e5'),
               "number"=>613,
               "group"=>"14 and under",
               "time"=>"3489 secs"}}
```

Self Grading/Feedback

All feedback is performed through execution of commands and queries provided. You are welcome to go off on many tangents and can always get back to the start point with `Racers.reset`.

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: 2015-12-14