Module 2, Summative Assignment: Advanced MongoDB Ruby Driver Topics

This assignment will evaluate your ability to implement a data tier using some of the more advanced MongoDB Ruby Driver Topics discussed during this module. It is lengthy – but much of the length is the result of providing:

- extensive examples that seek to further explain what is being asked to implement
- demonstrations of how you can interactively look at your in-progress solution and self-evaluate the result

The overall goal of the assignment is to:

- Implement a model class and a set of supporting classes to represent a complex type for a collection.
- Implement standard queries
- Implement aggregation pipeline queries
- Create an index and implement geolocation queries
- Implement GridFS file storage and retrieval
- Implement and navigate a relationship from a model class
- Demonstrate the data tier's ability to serve content to the web
- Demonstrate the data tier's ability to leverage relationships between collections to the web.

The functional goal of the assignment is to:

- Implement a data tier to manage `place`s and `photo`s and associate `photo`s with a nearby `place`
- Implement a web tier to view `place`s and their associated `photo`s.

Functional Requirements

1. Implement a model and supporting classes to encapsulate access to the `places` collection. Import data – primarily thru the `google maps API` – into this collection.
2. Implement standard queries for the `places` collection. This will get you familiar with the schema as well as provide some necessary functions for the overall data tier.
3. Implement advanced queries for the `places` collection using the aggregation framework. This will locate information that is embedded within the nested `places` schema.
4. Implement geolocation queries using a `2dsphere` index for the `places` collection, which will locate a place within a distance threshold.
5. Implement a model class called `Photo` that will encapsulate actions performed on photos (`jpeg` only). This model class will:
    - import photo images from files
    - extract geolocation information from the image using the `exifr` gem. The images are `geotagged jpeg` images and contain geolocation coordinates in the `exif` portion of the image. You will will use the `exifr` gem to extract the `exif` geolocation information from the `jpeg` images.
    - store, update, and retrieve photo information and data from GridFS
6. Add support functions to locate the nearest `places` for a `photo` and associate each `photo` with a `place` within distance tolerances.
7. Populate the data tier using `$rake db:seed` for demonstration with the web tier
8. Display raw `photo` image content from a web URI.
9. Display `place` information and associated `photo` images from a set of web URIs.

Getting Started

1. Create a new Rails application called `places`.

```
$ rails new places
$ cd places
```

2. Add the `mongo` and `mongoid` gems to the Gemfile and run `bundle`. The `mongoid` gem will automatically install the `mongo` gem but since we are still focusing on the MongoDB Ruby Driver, we want to explicitly show that dependency here. More recent versions of the `mongo` and `mongoid` gems will likely be installed as a part of the `bundle` command.

```
gem 'mongo', '~> 2.1.0'
gem 'mongoid', '~> 5.0.0'
```

```
$ bundle
```

3. Configure `mongoid` within the application by generating a configuration file and loading that within `places/config/application.rb`. The defaults generated should be fine.

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

   - If the `load!` statement is not present in your application.rb, add it just before the `end` statement in the class `Application` definition

```
$ grep mongoid config/application.rb
#bootstraps mongoid within applications -- like rails console
Mongoid.load!('./config/mongoid.yml')
```

   - Verify (or update) that `development:clients:default:database` references `places_development`

```
$ egrep -v '\#|$^' config/mongoid.yml
development:
  clients:
    default:
      database: places_development
      hosts:
        - localhost:27017
      options:
  options:
```

4. After starting your MongoDB Database via `mongod`, use the `rails console` during your development to interactively test your data tier solutions. Remember to use `reload!`
after making changes to your source code.

```
$ rails c
> Mongoid::Clients.default
 => #<Mongo::Client:0x39062920 cluster=localhost:27017>
```

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

```
student-start/
|-- Gemfile
|-- db
|   |-- image1.jpg
|   |-- image2.jpg
|   |-- image3.jpg
|   |-- image4.jpg
|   |-- image5.jpg
|   |-- image6.jpg
|   `-- places.json
|-- .rspec (an important hidden file)
`-- spec
    |-- aggregation_spec.rb
    |-- collection_spec.rb
    |-- geo_spec.rb
    |-- images_spec.rb
    |-- photos_spec.rb
    |-- query_spec.rb
```

```
|-- rel_spec.rb
|-- seed_spec.rb
`-- web_spec.rb
```

- 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 'capybara'
  end
  ```

  as well as a new definition for the following items:

  - `tzinfo-data` gem conditionally included on Windows platforms
  - `mongo` gem containing the MongoDB Ruby Driver
  - `mongoid` gem we will use to obtain connections
  - `exifr` gem we will use to extract geo coordinates from `jpeg` images

  ```
  # Windows does not include zoneinfo files, so bundle the tzinfo-data gem
  gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
  gem 'mongo', '~> 2.1.0'
  gem 'mongoid', '~> 5.0.0'
  gem 'exifr'
  ```

- Add the provided image and json data files to your `db/` directory.
- Add the `spec/*.rb` files provided with the bootstrap fileset to a corresponding `spec/` directory within your `places` 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.

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

   ```
   $ bundle
   ```

7. Run the `rake db:migrate` command to resolve potential `db/schema.rb` warnings that checks for its existence.

8. 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/collection_spec.rb -e rq01
   ...
   (N) examples, (N) failures, (N) pending
   ```

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

10. Submit your Rails app solution for grading.

Technical Requirements

Places Collection

In this section you must implement a model class called `Place` and two supporting classes called `Point` and `AddressComponent`. The three (3) classes are used to encapsulate the properties of the data within the `places` collection. `Place` will be the primary class for database interaction. `Point` encapsulates a geolocation coordinate. `AddressComponent` encapsulates many address aliases within a `Place`.

0. Confirm that your rails application is appropriately structured.

   ```
   $ rspec spec/collection_spec.rb -e rq00
   ```

1. Create a model class called `Place`. This class must:

   - provide a class method called `mongo_client` that returns a MongoDB Client from Mongoid referencing the default database from the `config/mongoid.yml` file (Hint: `Mongoid::Clients.default`)
   - provide a class method called `collection` that returns a reference to the `places` collection.

   You can demonstrate your new model class and methods using the `rails console`.

   ```
   > reload!
   > Place.mongo_client
    => #<Mongo::Client:0x39062920 cluster=localhost:27017>
   > Place.collection
    => #<Mongo::Collection:0x15439920 namespace=places_development.places>
   ```

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

2. Implement a class method called `load_all` that will bulk load a JSON document with `places` information into the places collection. This method must

   - accept a parameter of type `IO` with a JSON string of data
   - read the data from that input parameter (Note: this is similar handling an uploaded file within Rails)
   - parse the JSON string into an array of Ruby hash objects representing places (Hint: `JSON.parse`)
   - insert the array of hash objects into the `places collection` (Hint: `insert_many`)

   You can demonstrate your new model class and methods using the Rails console.

   ```
   > f=File.open("./db/places.json"); nil
   > Place.load_all(f)
   > Place.collection.count
    => 39
   > pp Place.collection.find.first; nil
   {"_id"=>BSON::ObjectId('56521833e301d0284000003d'),
    "address_components"=>
     [{"long_name"=>"Wilsden",
       "short_name"=>"Wilsden",
       "types"=>["administrative_area_level_4", "political"]},
      {"long_name"=>"Bradford District",
       "short_name"=>"Bradford District",
       "types"=>["administrative_area_level_3", "political"]},
      {"long_name"=>"West Yorkshire",
       "short_name"=>"West York",
       "types"=>["administrative_area_level_2", "political"]},
      {"long_name"=>"England",
       "short_name"=>"England",
       "types"=>["administrative_area_level_1", "political"]},
      {"long_name"=>"United Kingdom",
       "short_name"=>"GB",
       "types"=>["country", "political"]}],
    "formatted_address"=>"Wilsden, West Yorkshire, UK",
    "geometry"=>
   ```

```
 {"bounds"=>
   {"northeast"=>{"lat"=>53.8399586, "lng"=>-1.8368897},
    "southwest"=>{"lat"=>53.8078964, "lng"=>-1.8899853}},
  "location"=>{"lat"=>53.8256035, "lng"=>-1.8625303},
  "location_type"=>"APPROXIMATE",
  "viewport"=>
   {"northeast"=>{"lat"=>53.8399586, "lng"=>-1.8368897},
    "southwest"=>{"lat"=>53.8078964, "lng"=>-1.8899853}}},
 "place_id"=>"ChIJRRp9xvble0gR7M3MlaQDgok",
 "types"=>["administrative_area_level_4", "political"]}
```

Note that you can always clear the `places` collection and start over using `delete_many`.

```
> Place.collection.delete_many
```

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

3. Implement a custom type called `Point`. This class must have:

   - a read/write (Integer) attribute called `longitude` (Hint: `attr_accessor`)
   - a read/write (Integer) attribute called `latitude` (Hint: `attr_accessor`)
   - a `to_hash` instance method that will produce a `GeoJSON Point` hash (**Hint**: see example below)
   - an `initialize` method that can set the attributes from a hash with keys `lat` and `lng` or `GeoJSON Point` format.

   Example hash:

```
{:type=>"Point", :coordinates=>[ -1.8625303, 53.8256035]} #GeoJSON Point format
{:lat=>53.8256035, :lng=>-1.8625303}

**Note**: You may encounter string or :symbols as keys in your hashes. If you do
not know what to expect, you may want to try using `hash.symbolized_keys()` or
`hash.deep_symbolized_keys()` if you want to easily prepare for both forms.
```

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

4. Implement a custom type called `AddressComponent`. This class must have:

   - a read-only (String) attribute called `long_name`
   - a read-only (String) attribute called `short_name`
   - a read-only (array of Strings) attribute called `types`
   - an `initialize` method that can set the attributes from a hash with keys `long_name`, `short_name`, and `types`.

   Example hash:

```
{:long_name=>"Bradford District",
 :short_name=>"Bradford District",
 :types=>["administrative_area_level_3", "political"]},
```

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

5. Implement read/write attributes for the following properties in `Place`. (Hint: `attr_accessor`)

   - a read/write (String) attribute called `id`
   - a read/write (String) attribute called `formatted_address`
   - a read/write (Point) attribute called `location`
   - a read/write (collection of AddressComponents) attribute called `address_components`
   - an `initialize` method to `Place` that can set the attributes from a hash with keys `_id`, `address_components`, `formatted_address`, and `geometry.geolocation`. (**Hint**: use `.to_s` to convert a `BSON::ObjectId` to a `String` and `BSON::ObjectId.from_string(s)` to convert it back again.)

   Example hash:

```
{"_id":BSON::ObjectId('56521833e301d0284000003d'),
 "address_components":
```

```
  [
    {"long_name":"Wilsden", "short_name":"Wilsden", "types":["administrative_area_level_4", "political"
    {"long_name":"Bradford District", "short_name":"Bradford District", "types":["administrative_area_l
  ],
 "formatted_address":"Wilsden, West Yorkshire, UK",
 "geometry":
    {
        "location":{"lat":"53.8256035, "lng":-1.8625303},
        "geolocation":{"type":"Point", "coordinates":[-1.8625303, 53.8256035]}
    }
 }
```

Note: The example above is shown using string hash keys. The RSpec tests typically call methods with symbols. You may want to try `hash.deep_symbolized_keys()` if you want to easily prepare for both forms.

Note: The `GeoJSON Point` format was added to the test data from the original information obtained from google maps. MongoDB `2dsphere` index and `2dsphere` search functions require this format to function correctly. Anything we do with geolocation in this exercise will use the `geolocation` property.

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

Standard Queries

In this section you must implement a few standard queries for `Places`. This is a short warm-up to get familiar with the schema and utlizes the query topics from the previous module.

1. Implement a class method called `find_by_short_name` that will return a `Mongo::Collection::View` with a query to match documents with a matching `short_name` within `address_components`. This method must:

   - accept a String input parameter
   - find all documents in the `places` collection with a matching `address_components.short_name`
   - return the `Mongo::Collection::View` result

   You can demonstrate your new class method using the Rails console. Notice how the view returned from the `find` command can be extended with sorting and paging commands.

   ```
   > Place.find_by_short_name("GB").first[:formatted_address]
    => "Wilsden, West Yorkshire, UK"
   > Place.find_by_short_name("GB").sort(:formatted_address=>1).skip(4).first[:formatted_address]
    => "Bradford, West Yorkshire BD15, UK"
   ```

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

2. Implement a helper class method called `to_places` that will accept a `Mongo::Collection::View` and return a collection of `Place` instances. This method must:

   - accept an input parameter
   - iterate over contents of that input parameter
   - change each document hash to a Place instance (**Hint**: `Place.new`)
   - return a collection of results containing `Place` objects

   You can demonstrate your new class helper method using the Rails console. Notice that by separating the find from the `Place` collection creation, we can allow sorting and paging be independently applied.

   ```
   > Place.to_places(Place.find_by_short_name("GB")).first.location
    => #<Point:0x00000004dc6f00 @latitude=53.8256035, @longitude=-1.8625303>

   > Place.to_places(Place.find_by_short_name("GB").limit(3)).each {|r| p r.formatted_address }; nil
   "Wilsden, West Yorkshire, UK"
   "8 Badgergate Ave, Wilsden, Bradford, West Yorkshire BD15 0LJ, UK"
   "Wilsden, West Yorkshire, UK"
   ```

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

3. Implement a class method called `find` that will return an instance of `Place` for a supplied `id`. This method must:

   - accept a single String `id` as an argument
   - convert the `id` to `BSON::ObjectId` form (**Hint**: `BSON::ObjectId.from_string(s)`)
   - find the document that matches the `id`
   - return an instance of `Place` initialized with the document if found (Hint: `Place.new`)

   You can demonstrate your new class method using the Rails console.

```
> id=Place.collection.find.first[:_id].to_s
 => "56521833e301d0284000003d"

> Place.find(id).formatted_address
 => "Wilsden, West Yorkshire, UK"
> Place.find(id).location
 => #<Point:0x000000054b2fa8 @latitude=53.8256035, @longitude=-1.8625303>
```

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

4. Implement a class method called `all` that will return an instance of all documents as `Place` instances. This method must:

   - accept two optional arguments: `offset` and `limit` in that order. `offset` must default to no offset and `limit` must default to no limit
   - locate all documents within the `places` collection within paging limits
   - return each document as in instance of a `Place` within a collection

   You can demonstrate your new class method using the Rails console. Notice that the return type is a collection of `Place` instances and that `offset` (default=0) and `limit` (default=unlimited) are the default.

```
> Place.all(4,3).map {|place| place.formatted_address}
 => ["Costa Rica", "Desert Hot Springs, CA 92241, USA",
     "Zieglmeierstra.e 11, 82383 Hohenpei.enberg, Germany"]

> Place.all.count
 => 39
```

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

5. Implement an instance method called `destroy` in the `Place` model class that will delete the document associtiated with its assigned `id`. This method must:

   - accept no arguments
   - delete the document from the `places` collection that has an `_id` associated with the `id` of the instance.

   You can demonstrate your new method using the Rails console. In the example below we grab a random sample `place` and call destroy.

```
> place=Place.all.sample
> place.destroy
 => #<Mongo::Operation::Result:48622960 documents=[{"ok"=>1, "n"=>1}]>
```

   In the following example, we use `all` and `destroy` together to implement a somewhat expensive but convenient and functional way to clear the collection.

```
> Place.all.each {|place| place.destroy }
> Place.all.count
 => 0
```

   Note: Remember you can restore your places collection (with new `_id`s) by using your `load_all` method implemented earlier.

```
> Place.load_all(File.open('./db/places.json'))
```

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

Aggregation Framework Queries

In this section you must implement several queries using the aggregation framework using various `pipeline operators` to derive the proper query result.

1. Create a `Place` class method called `get_address_components` that returns a collection of hash documents with `address_components` and their associated `_id`, `formatted_address` and `location` properties. Your method must:

   - accept optional `sort`, `offset`, and `limit` parameters
   - extract all `address_component` elements within each document contained within the collection (**Hint**: `$unwind`)
   - return only the `_id`, `address_components`, `formatted_address`, and `geometry.geolocation` elements (Hint: `$project`)
   - apply a provided `sort` or no sort if not provided (Hint: `$sort` and `q.pipeline` method)
   - apply a provided `offset` or no offset if not provided (Hint: `$skip` and `q.pipeline` method)
   - apply a provided `limit` or no limit if not provided (Hint: `$limit` and `q.pipeline` method)
   - return the result of the above query (Hint: `collection.find.aggregate(...)`)

   You can demonstrate your new class method using the Rails console. Notice how the output has been cut down to just the `_id`, `address_components`, `formatted_address`, and `geometry.location` elements. Each `address_component` has been flattened out so that `_id`, `formatted_address` and `geometry.geolocation` elements are repeated for each element in the collection. Apply a different search criteria and paging parameters to adjust the output.

   Note: In order to mirror similiar results below, it is suggested to restore your `places` collection by using the `load_all` call mentioned previously. However, since we are sorting on `_ids` that were dynamically assigned during your ingest – your exact results will vary.

```
> pp Place.get_address_components({:_id=>-1}, 48,3).to_a; nil
[{"_id"=>BSON::ObjectId('56ad4adde301d07b9e000018'),
  "address_components"=>
   {"long_name"=>"United Kingdom",
    "short_name"=>"GB",
    "types"=>["country", "political"]},
  "formatted_address"=>"England, UK",
  "geometry"=>
   {"geolocation"=>
     {"type"=>"Point", "coordinates"=>[-1.1743197, 52.3555177]}}},
 {"_id"=>BSON::ObjectId('56ad4adde301d07b9e000017'),
  "address_components"=>
   {"long_name"=>"England",
    "short_name"=>"England",
    "types"=>["administrative_area_level_1", "political"]},
  "formatted_address"=>"West Yorkshire, UK",
  "geometry"=>
   {"geolocation"=>
     {"type"=>"Point", "coordinates"=>[-1.76261, 53.81081760000001]}}},
 {"_id"=>BSON::ObjectId('56ad4adde301d07b9e000017'),
  "address_components"=>
   {"long_name"=>"West Yorkshire",
    "short_name"=>"West York",
    "types"=>["administrative_area_level_2", "political"]},
  "formatted_address"=>"West Yorkshire, UK",
  "geometry"=>
   {"geolocation"=>
     {"type"=>"Point", "coordinates"=>[-1.76261, 53.81081760000001]}}}]
```

   Notice how you should be able to invoke the method and have defaults applied.

```
> Place.get_address_components.count
```

```
 => 141
```

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

2. Create a `Place` class method called `get_country_names` that returns a distinct collection of country names (`long_names`). Your method must:

   - accept no arguments
   - create separate documents for `address_components.long_name` and `address_components.types` (Hint: `$project` and `$unwind`)
   - select only those documents that have a `address_components.types` element equal to `"country"` (Hint: `$match`)
   - form a distinct list based on `address_components.long_name` (Hint: `$group`)
   - return a simple collection of just the country names (`long_name`). You will have to use application code to do this last step. (Hint: `.to_a.map {|h| h[:_id]}`)

   You can demonstrate your new class method using the Rails console. Notice how the output is a distinct list of country `long_names` as stripped down strings in a collection.

```
> Place.get_country_names
 => ["X X", "Yy", "Zz", "A A"]
```

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

3. Create a `Place` class method called `find_ids_by_country_code` that will return the `id` of each document in the `places` collection that has an `address_component.short_name` of type `country` and matches the provided parameter. This method must:

   - accept a single `country_code` parameter
   - locate each `address_component` with a matching `short_name` being tagged with the `country` type (Hint: `$match`)
   - return only the `_id` property from the database (Hint: `$project`)
   - return only a collection of `_id`s converted to Strings (Hint: `.map {|doc| doc[:_id].to_s}`)

   You can demonstrate your new class method using the Rails console. Notice how this method can be used to locate a group of primary keys that can be fed back into find. This is an expensive way to implement a `find` but it may be a necessary implementation when we form links across collections later.

```
> Place.find_by_country_code "GB"
 => ["56521833e301d0284000003d", "565218a9e301d02840000069", ... "565218a9e301d0284000006a"]

> Place.find_ids_by_country_code("GB").slice(0,2).each { |id| puts Place.find(id).formatted_address}
  Wilsden, West Yorkshire, UK
  8 Badgergate Ave, Wilsden, Bradford, West Yorkshire BD15 0LJ, UK
```

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

Geolocation Queries

In this section you must create a geolocation index within the `places` collection and implement a geolocation search that locates `places` within tolerances given a geographic point.

1. Create two `Place` class methods, one called `create_indexes` and the other `remove_indexes`. These will be used to create and remove a `2dsphere` index to your collection for the `geometry.geolocation` property. These methods must exhibit the following behavior:

   - `create_indexes` must make sure the `2dsphere` index is in place for the `geometry.geolocation` property (**Hint**: `Mongo::Index::GEO2DSPHERE`)
   - `remove_indexes` must make sure the `2dsphere` index is removed from the collection (**Hint**: `Place.collection.indexes.map {|r| r[:name] }` displays the names of each index)

   You can demonstrate your new class methods using the Rails console. The second example below shows how you can locate the name of each index. This name must be used when removing the index.

```
> Place.create_indexes
 => #<Mongo::Operation::Result:43671200 documents=[{"createdCollectionAutomatically"=>false,
    "numIndexesBefore"=>1, "numIndexesAfter"=>2, "ok"=>1.0}]>

> Place.collection.indexes.map {|r| r[:name] }
 => ["_id_", "geometry.geolocation_2dsphere"]

> Place.remove_indexes
 => #<Mongo::Operation::Result:43441700 documents=[{"nIndexesWas"=>2, "ok"=>1.0}]>
```

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

2. Create a `Place` class method called `near` that returns places that are closest to provided `Point`. This method must:

   - accept an input parameter of type `Point` (created earlier) and an optional `max_meters` that defaults to no maximum
   - performs a `$near` search using the `2dsphere` index placed on the `geometry.geolocation` property and the `GeoJSON` output of `point.to_hash` (created earlier). (**Hint**: `Query a 2dsphere Index`)
   - limits the maximum distance – if provided – in determining matches (**Hint**: `$maxDistance`)
   - returns the resulting view (i.e., the result of find())

   You can demonstrate your new class methods using the Rails console. You can use one of a number of queries to locate a specific document within the `places` collection and then create a `Place` instance to represent that document.

   Note: You may need to re-invoke `create_indexes` prior to executing this find, given the previous step demonstrated `remove_index`

```
> pa_doc=Place.find_by_short_name("PA").first
> pa_place=Place.new(pa_doc)
```

   `place` encapsulates a `location` object of type `Point` which has a `to_hash` method added to it (we did this earlier) that generates a hash in `GeoJSON Point` format.

```
> pa_point=pa_place.location
 => #<Point:0x000000036aff10 @latitude=39.874572, @longitude=-75.56709699999999>
> pa_point.to_hash
 => {:type=>"Point", :coordinates=>[-75.56709699999999, 39.874572]}
```

   You will need to `Hash` form of the `Point` in your query. You might find it interesting that both a `Point` and `Hash` instance both support the method `to_hash`. So the real requirement of this method for `point` is that it accept an object with a to_hash method that produces a GeoJSON formatted hash. Do not forget to call `to_hash` on whatever you are passed and you will automatically enable the use of `Hash` and `Point` types if desired.

```
> pa_point.to_hash
 => {:type=>"Point", :coordinates=>[-75.56709699999999, 39.874572]}
> pa_point.to_hash.to_hash
  => {:type=>"Point", :coordinates=>[-75.56709699999999, 39.874572]}
```

   `point` can be used to locate other places nearby using an optional maximum distance threshold measured in meters (There are 1609.4 meters in a mile). The `to_places` method can be used to convert the collection of matching documents to a collection of `Place` instances.

```
> pa_near=Place.to_places(Place.near(pa_point, 10*1609.4))
#or based on what we learned above, the following should also work
#when passed a Hash with no additional work on your part
> pa_near=Place.to_places(Place.near(pa_point.to_hash, 10*1609.4))
```

   The collection of near `Places` can be iterated over and properties printed to gain insight into which places are closer than others.

```
> pa_near.each { |place| p place.formatted_address }; nil
"1399 Baltimore Pike, Chadds Ford, PA 19317, USA"
```

```
"Chadds Ford, PA, USA"
"Chadds Ford, PA 19317, USA"
"Delaware County, PA, USA"
```

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

3. Create an instance method (also) called `near` that wraps the class method you just finished. This method must:

   - accept an optional parameter that sets a maximum distance threshold in meters
   - locate all `places` within the specified maximum distance threshold
   - return the collection of matching documents as a collection of `Place` instances using the `to_places` class method added earlier.

   You can demonstrate your new class methods using the Rails console. Once you have an instance of a `Place`, it should be very easy to locate other places near it.

```
> pa_place.near(10*1609.4).each {|r| p r.formatted_address }; nil
"1399 Baltimore Pike, Chadds Ford, PA 19317, USA"
"Chadds Ford, PA, USA"
"Chadds Ford, PA 19317, USA"
"Delaware County, PA, USA"
```

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

Photos

In this section you must implement a model class called `Photo`. The purpose of this model class is to encapsulate all information and content access to a photograph. This model uses `GridFS` – rather than a usual MongoDB collection like `places` since there will be an information aspect and a raw data aspect to this model type. This model class will also be responsible for extracting geolocation coordinates from each `photo` and locating the nearest `place` (within distance tolerances) to where that `photo` was taken. To simplify the inspection of the photo image data, all photos handled by this model class will be assumed to be `jpeg` images. You may use the [`exifr gem`] (https://rubygems.org/gems/exifr/) to extract available geographic coordinates from each `photo`.

1. Create a model class called `Photo`. Since the storage for this class is primarily within GridFS, there is no need for a `collection` method. This class must:

   - provide a class method called `mongo_client` that returns a MongoDB Client from Mongoid referencing the default database from the `config/mongoid.yml` file (**Hint**: `Mongoid::Clients.default`)

   You can demonstrate your new model class and methods using the Rails console.

```
> Photo.mongo_client
 => #<Mongo::Client:0x44048040 cluster=localhost:27017>
```

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

2. Implement the following attributes in the `Photo` class

   - a read/write attribute called `id` that will be of type `String` to hold the String form of the GridFS file `_id` attribute
   - a read/write attribute called `location` that will be of type `Point` to hold the location information of where the photo was taken.
   - a write-only (for now) attribute called `contents` that will be used to import and access the raw data of the photo. This will have varying data types depending on context.

   You can demonstrate your new model attributes and access methods using the Rails console. Notice that we initialized `contents` to be an `IO` (`File`) object that can be read from and is not the data itself. This will become important later.

```
> place=Place.all.first
> f=File.open('./db/image1.jpg','rb')
> photo = Photo.new
> photo.location = place.location
```

```
> photo.contents = f
> pp photo
 #<Photo:0x000000070a2150
 @contents=#<File:./db/image1.jpg>,
 @location=
  #<Point:0x00000006ec97c0 @latitude=33.875467, @longitude=-116.3016158>>
```

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

3. Add an `initialize` method in the `Photo` class that can be used to initialize the instance attributes of `Photo` from the hash returned from queries like `mongo_client.database.fs.find`. This method must

   - initialize `@id` to the string form of `_id` and `@location` to the `Point` form of `metadata.location` if these exist. The document hash is likely coming from query results coming from `mongo_client.database.fs.find`.
   - create a default instance if no hash is present

   You can demonstrate your new method using the Rails console. In the first set of commands the default initialize is being called and then the location is being set to a new `Point` instance.

```
> photo=Photo.new
 => #<Photo:0x000000062119d8>

> photo.location=Point.new(:type=>"Point", :coordinates=>[-116.30161960177952, 33.87546081542969])
 => #<Point:0x00000006193290 @longitude=-116.30161960177952, @latitude=33.87546081542969>
```

   In the second set of commands, the file information for a GridFS file is retrieved using a `find` command and directly used to initialize the instance.

```
> doc=Photo.mongo_client.database.fs.find.first
  {"_id"=>BSON::ObjectId('5652d94de301d0c0ad000001'),
   "chunkSize"=>261120,
   "uploadDate"=>2015-11-23 09:15:57 UTC,
   "contentType"=>"binary/octet-stream",
   "metadata"=>{"location"=>{"type"=>"Point", "coordinates"=>[-116.30161960177952, 33.87546081542969]}},
   "length"=>601685,
   "md5"=>"871666ee99b90e51c69af02f77f021aa"}

> photo=Photo.new doc
 => #<Photo:0x000000060fcd18
   @location=#<Point:0x000000060fc8b8 @longitude=-116.30161960177952, @latitude=33.87546081542969>,
   @id="5652d94de301d0c0ad000001">
```

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

4. Add an instance method to the `Photo` class called `persisted?` to return true if the instance has been created within GridFS. This method must:

   - take no arguments
   - return true if the `photo` instance has been stored to GridFS (**Hint**: `@id.nil?`)

   You can demonstrate your new method using the Rails console as a part of implementing the next requirement (`save`).

5. Add an instance method to the `Photo` class called `save` to store a new instance into GridFS. This method must:

   - check whether the instance is already persisted and do nothing (for now) if already persisted (**Hint**: use your new `persisted?` method to determine if your instance has been persisted)
   - use the `exifr` gem to extract geolocation information from the `jpeg` image.
   - store the content type of `image/jpeg` in the `GridFS contentType` file property.
   - store the `GeoJSON Point` format of the image location in the `GridFS metadata` file property and the object in class' `location` property.
   - store the data contents in `GridFS`
   - store the generated `_id` for the file in the `:id` property of the `Photo` model instance.

Lets take a quick look at the `exfir` gem. The `EXIFR::JPEG initialize` method can read the contents of a file and further provide the geolocation information through the call to `gps`

```
> f = File.open('./db/image1.jpg','rb')
 => #<File:./db/image1.jpg>

> gps=EXIFR::JPEG.new(f).gps
 => #<struct EXIFR::TIFF::GPS latitude=33.87546081542969, longitude=-116.30161960177952, ...
```

The `gps` object can then be inspected for `latitude` and `longitude` properties that can be used to instantiate the `Point` class we have created for this assignment. The `Point` class can produce a location in `GeoJSON Point` format. This can be stored in the `metadata` properties of the file using the `location` property.

```
> location=Point.new(:lng=>gps.longitude, :lat=>gps.latitude)
 => #<Point:0x00000006731210 @latitude=33.87546081542969, @longitude=-116.30161960177952>

> location.to_hash
 => {:type=>"Point", :coordinates=>[-116.30161960177952, 33.87546081542969]}
```

Hint: Both `EXIFR` and `GridFS` will be reading the same file. You must call rewind() on the file in between calls for the proper number of bytes to be stored in `GridFS`.

```
> f.rewind
```

You can demonstrate your new `save` method using the Rails console.

```
> f = File.open('./db/image1.jpg','rb')
 => #<File:./db/image1.jpg>
> photo=Photo.new
 => #<Photo:0x00000005fb4690>
> photo.contents = f
 => #<File:./db/image1.jpg>
> id=photo.save
 => "5652df83e301d0c0ad00000d"
> photo.location
 => #<Point:0x00000005ed32a8 @latitude=33.87546081542969, @longitude=-116.30161960177952>
```

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

6. Add a class method to the `Photo` class called `all`. This method must:

   - accept an optional set of arguments for skipping into and limiting the results of a search
   - default the offset (**Hint**: `skip`) to 0 and the limit to unlimited
   - return a collection of `Photo` instances representing each file returned from the database (**Hint**: `...find.map {|doc| Photo.new(doc) }`)

   You can demonstrate your new method using the Rails console. By supplying no arguments, we are able to access all documents in the collection. When we add the first parameter (`offset`), we skip that number of documents in the collection. When we add the second parameter (`limit`), we constrain the results to a limit of documents. Notice the method returns instances of `Photo`.

```
> Photo.all.count
 => 4
> Photo.all(1).count
 => 3
> Photo.all(1,2).count
 => 2
> pp Photo.all(1,2).first
#<Photo:0x000000067baf38
 @id="5652df09e301d0c0ad000005",
 @location=
  #<Point:0x000000067bac90
```

```
  @latitude=33.87546081542969,
  @longitude=-116.30161960177952>>
```

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

7. Create a class method called `find` that will return an instance of a `Photo` based on the input `id`. This method must:

   - accept a single String parameter for the `id`
   - locate the file associated with the `id` by converting it back to a `BSON::ObjectId` and using in an `:_id` query.
   - set the values of `id` and `location` witin the model class based on the properties returned from the query.
   - return an instance of the `Photo` model class

   Hint: You can use the following example as a guide to how you may locate the file info.

```
> pp Photo.mongo_client.database.fs.find(:_id=>BSON::ObjectId.from_string(id)).first
{"_id"=>BSON::ObjectId('5652df83e301d0c0ad00000d'),
 "chunkSize"=>261120,
 "uploadDate"=>2015-11-23 09:42:27 UTC,
 "contentType"=>"binary/octet-stream",
 "metadata"=>
  {"location"=>
    {"type"=>"Point",
     "coordinates"=>[-116.30161960177952, 33.87546081542969]}},
 "length"=>601685,
 "md5"=>"871666ee99b90e51c69af02f77f021aa"}
```

   You can demonstrate your new `find` method using the Rails console.

```
> photo=Photo.find id
 => #<Photo:0x0000000426a378 @id="5652df83e301d0c0ad00000d",
    @location=#<Point:0x0000000423ab28 @longitude=-116.30161960177952, @latitude=33.87546081542969>>
> photo.location
 => #<Point:0x0000000423ab28 @longitude=-116.30161960177952, @latitude=33.87546081542969>
```

```
$ rspec spec/photos_spec.rb -e rq07
```

8. Create a custom getter for `contents` that will return the data contents of the file. This method must:

   - accept no arguments
   - read the data contents from GridFS for the associated file
   - return the data bytes

   You can demonstrate your new custom getter method for `contents` using the Rails console.

```
> f=File.open('test.jpg','wb')
 => #<File:test.jpg>
> f.write(photo.contents)
 => 624744
```

   After writing the contents of the file accessed from GridFS onto the file system, the size of the two file should be the same and you should be able to see the same image as the original.

```
$ ls -l test.jpg db/image1.jpg
... 624744 Nov 23 03:26 db/image1.jpg
... 624744 Nov 23 05:18 test.jpg
```

   Hint: If your byte count is less than expected, check that you are calling rewind() on the file in between reading it in both `EXIFR` and `GridFS` during the save.

```
$ rspec spec/photos_spec.rb -e rq08
```

9. Add an instance method called `destroy` to the `Photo` class that will delete the file and contents associated with the ID of the object instance. This method must:

   - accept no arguments

- delete the file and its contents from GridFS

You can demonstrate your new method using the Rails console. We will start out by creating a new Photo from a file, saving the contents, and verifying we can locate the `Photo` by calling `find` on the ID.

```
> photo=Photo.new
> photo.contents=File.open('./db/image1.jpg','rb')
 => #<File:./db/image1.jpg>
> photo.save
 => "565515efe301d0c0ad000015"
> Photo.find(photo.id)
 => #<Photo:0x000000046b25c0 @location=#<Point:0x000000046b1df0 ...
```

We then can call destroy and use `find` again to verify the file can no longer be located in GridFS.

```
> photo.destroy
 => #<Mongo::Operation::Result:36999680 documents=[{"ok"=>1, "n"=>1}]>
> Photo.find(photo.id)
 => nil
```

Of course, if you ever want to clean up and start over with a fresh set of `Photos`, you can leverage your `all` and `destroy` method together.

```
> Photo.all.each {|photo| photo.destroy }
```

```
$ rspec spec/photos_spec.rb -e rq09
```

Relationships

In this section you must implement a `many-to-one` relationship from `Photo` to `Place`. A foreign key to the `place` will be inserted into the `photo` information to realize this relationship and navigation must be bi-directional (i.e., `photo.place` and `place.photos`). We will also select which relationships to form based on distance a `place` is from where the `photo` was taken.

1. Create a `Photo` helper instance method called `find_nearest_place_id` that will return the `_id` of the document within the `places` collection. This `place` document must be within a specified distance threshold of where the photo was taken. This `Photo` method must:

   - accept a maximum distance in meters
   - uses the `near` class method in the `Place` model and its location to locate places within a maximum distance of where the photo was taken.
   - limit the result to only the nearest matching place (**Hint**: `limit()`)
   - limit the result to only the `_id` of the matching place document (**Hint**: `projection()`)
   - returns zero or one `BSON::ObjectIds` for the nearby place found

   You can demonstrate your new method using the Rails console. We first use the `all` class method written earlier to locate a sample photo and verify it has a location.

   ```
   > photo=Photo.all.first
   > photo.location
    => #<Point:0x000000065dbbe0 @longitude=-116.30161960177952, @latitude=33.87546081542969>
   ```

   We then use the new method added here to locate the closest place to the photo within one (1) mile (1609.34 meters/mile).

   ```
   > photo.find_nearest_place_id(1*1609.34)
    => BSON::ObjectId('5652b509e301d03daf000075')
   ```

   We can then use the returned `ID` to inspect the place located.

   ```
   > place=Place.find "5652b509e301d03daf000075"
   > place.location
    => #<Point:0x000000065a8d30 @longitude=-116.3016158, @latitude=33.875467>
   ```

```
> place.formatted_address
 => "77713-77735 Dillon Rd, Desert Hot Springs, CA 92241, USA"
```

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

2. Update the logic within the existing `save` instance method to update the file properties (not the file data – just the file properties/metadata) when called on a persisted instance. Previously, the method only handled a new `Photo` instance that was yet persisted. This method must:

   - accept no inputs
   - if the instance is not yet persisted, perform the existing logic to add the file to GridFS
   - if the instance is already persisted (Hint: `persisted?` helper method added earlier) update the file info (Hint: `find(...).update_one(...)`)

   You can demonstrate your new method using the Rails console. In the first set of methods we get a reference to a sample photo and print the current location.

```
> photo=Photo.all.first
> photo.location
 => #<Point:0x00000005e48928 @longitude=-116.3016158, @latitude=33.875467>
```

   In the next block of commands we set the `location` to a new `Point` and call the new `save` behavior on the photo instance. Since the instance has already been persisted, an update to GridFS id done for the file info properties. We verify the update was performed by retrieving a new instance from the database using our `find` method in the model class.

```
> photo.location=Point.new(:type=>"Point", :coordinates=>[-116.0000000,33.000000])
 => #<Point:0x00000005de2ad8 @longitude=-116.0, @latitude=33.0>
> photo.save
 => #<Mongo::Operation::Result:49114080 documents=[{"ok"=>1, "nModified"=>1, "n"=>1}]>
> Photo.find(photo.id).location
 => #<Point:0x00000005d64958 @longitude=-116.0, @latitude=33.0>
```

   This last block of commands repeats the above to put the previous `location` back in place.

```
> photo.location=Point.new(:type=>"Point", :coordinates=>[-116.3016158,33.875467])
 => #<Point:0x00000005d1ba28 @longitude=-116.3016158, @latitude=33.875467>
> photo.save
 => #<Mongo::Operation::Result:48676360 documents=[{"ok"=>1, "nModified"=>1, "n"=>1}]>
> Photo.find(photo.id).location
 => #<Point:0x0000000577ca18 @longitude=-116.3016158, @latitude=33.875467>
```

   Note: You should likely also re-test that the insert logic for `save` still works for instances that have not yet been persisted.

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

3. We will be adding to `Photo` the functionality to support a relationship with `Place`. Add a new `place` attribute in the `Photo` class to be used to realize a `Many-to-One` relationship between `Photo` and `Place`. The `Photo` class must:

   - add support for a `place` instance attribute in the model class. You will be implementing a custom setter/getter for this attribute
   - store this new property within the file metadata (`metadata.place`)
   - update the `initialize` method to cache the contents of `metadata.place` in an instance attribute called `@place`
   - update the `save` method to include the `@place` and `@location` properties under the parent `metadata` property in the file info.
   - add a custom getter for `place` that will find and return a `Place` instance that represents the stored ID (**Hint**: `Place.find`)
   - add a custom setter that will update the `place` ID by accepting a `BSON::ObjectId`, String, or `Place` instance. In all three cases you will want to derive a a `BSON::ObjectId` from what is passed in.

You can demonstrate your new method using the Rails console. We first use the `all` to obtain a sample photo and show that it does not yet have a place.

```
> photo=Photo.all.first
> photo.place
 => nil
```

We then find the `BSON::ObjectId` for the nearest location and assign that to the `photo.place` and inspect the Photo state attributes to find the stored `BSON::ObjectId` in the `place` attribute. We then can get an instance of the `Place` by calling `photo.place`.

```
> place_id=photo.find_nearest_place_id(1*1609.34)
 => BSON::ObjectId('5652b509e301d03daf000075')
> photo.place=place_id
 => BSON::ObjectId('5652b509e301d03daf000075')
> photo
 => #<Photo:0x00000006728368
    @id="5652d94de301d0c0ad000001",
    ...
    @place=BSON::ObjectId('5652b509e301d03daf000075')>
```

We then can save the `place` ID to the database to form the relationship between `Photo` and `Place`. We can verify the information was saved to the database by getting a fresh copy of the `Photo` instance using the `Photo.find` model method we added that locates a photo by `id`.

```
> photo.save
 => #<Mongo::Operation::Result:53897660 documents=[{"ok"=>1, "nModified"=>1, "n"=>1}]>
> Photo.find(photo.id).place.formatted_address
 => "77713-77735 Dillon Rd, Desert Hot Springs, CA 92241, USA"
```

We can delete the relationship by assigning the `place` property to `nil` and saving the change to the database. We can again verify the database state using the `find` method to retrieve a new instance of that `Photo`.

```
> photo.place=nil
 => nil
> photo.save
 => #<Mongo::Operation::Result:53713540 documents=[{"ok"=>1, "nModified"=>1, "n"=>1}]>
> Photo.find(photo.id).place
 => nil
```

The following shows some of the same place assignment functionality implemented using a String and `Place` instance. In these two cases we constructed a `BSON::ObjectId` from what was passed in. This will require your custom `place` setter method to check the type of what is being passed in and form the necessary `BSON::ObjectId` from the information provided by that type. **Hint**:

```
case
when object.is_a?(Place)
  @place=BSON::ObjectId.from_string(object.id)
...
```

```
> photo.place=place
> photo.place
 => BSON::ObjectId('5652b509e301d03daf000075')
> photo.place='5652b509e301d03daf000075'
 => "5652b509e301d03daf000075"
> photo.place.formatted_address
 => "77713-77735 Dillon Rd, Desert Hot Springs, CA 92241, USA"
> photo.place=nil
> photo
 => #<Photo:0x0000000598d758
    @id="5652d94de301d0c0ad000001",
```

```
    ...
    @place=nil>
```

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

4. Add a class method called `find_photos_for_place` that accepts the `BSON::ObjectId` of a `Place` and returns a collection view of photo documents that have the foreign key reference. This method must:

   - accept the ID of a `place` in either `BSON::ObjectId` or `String` ID form (Hint: `BSON::ObjectId.from_string(place_id.`
   - find GridFS file documents with the `BSON::ObjectId` form of that ID in the `metadata.place` property.
   - return the result view

   You can demonstrate your new method using the Rails console. We first clear our database of all photos using `all` and `destroy`, ingest new file contents using `save`, and update the document with the foreign key using `all` and `save`. We could have saved a trip to the database by assigning `place` within the first block but the command was getting a little long to fit on a single line within this document.

```
> Photo.all.each {|photo| photo.destroy }
> 5.times {photo=Photo.new; photo.contents=File.open('./db/image1.jpg','rb'); photo.save}
> place=Place.all.first
> Photo.all.each {|photo| photo.place=place; photo.save}
```

   We can now use our new method to obtain all `photo` documents that have the requested foreign key stored in their document. We placed the query on this side of the relationship so that `Place` did not have to know the details of `Photo`

```
> Photo.find_photos_for_place(place.id).map {|r| r[:_id] }
 => [BSON::ObjectId('56551a82e301d0c0ad000019'),
     BSON::ObjectId('56551a82e301d0c0ad00001d'),
     BSON::ObjectId('56551a83e301d0c0ad000021'),
     BSON::ObjectId('56551a83e301d0c0ad000025'),
     BSON::ObjectId('56551a83e301d0c0ad000029')]
```

   Note that because the `find_photos_for_place` method returns the query view and not a completed result, the caller can apply paging properties to the collection returned.

```
> Photo.find_photos_for_place(place.id).skip(3).limit(3).map {|r| r[:_id] }
 => [BSON::ObjectId('56551a83e301d0c0ad000025'),
     BSON::ObjectId('56551a83e301d0c0ad000029')]
```

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

5. Add a new instance method called `photos` to the `Place` model class. This method will return a collection of `Photos` that have been associated with the place. This method must:

   - accept an optional set of arguments (`offset`, and `limit`) to skip into and limit the result set. The offset should default to `0` and the limit should default to unbounded.

   You can demonstrate your new method using the Rails console.

```
> place=Photo.all.first.place
> place.photos.count
 => 5
```

   Note that because we have implemented paging within the getter, we now page through an unbounded set of photos for a place.

```
> pp place.photos(2,1).first
#<Photo:0x0000000681d4f8
 @id="56551a83e301d0c0ad000021",
 @location=
  #<Point:0x0000000681d390
   @latitude=33.87546081542969,
   @longitude=-116.30161960177952>,
 @place=BSON::ObjectId('5652b509e301d03daf000075')>
```

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

Data Tier Population

In this section you must implement a data initialization/population script in `db/seeds.rb` that will be runnable from the operating system shell using `$ rake db:seed`. In this Ruby script, you must clear the database of existing records, ingest the `Place`s and `Photo`s, and form `one-to-many` linked relationships between `photo`s and `place`s. This should simply be the grand finale of most of the model class capabilities you implemented above in order to populate the data tier for use in the follow-on web tier.
Your `seeds.rb` must:

1. Clear GridFS of all files. You may use the model commands you implemented as a part of this assignment or lower-level GridFS or database commands to implement the removal of all files.
2. Clear the `places` collection of all documents. You may use the model commands you implemented as a part of this assignment or lower-level collection or database commands to implement the removal of all documents from the `places` collection.
3. Make sure the `2dsphere` index has been created for the nested `geometry.geolocation` property within the `places` collection.
4. Populate the `places` collection using the `db/places.json` file from the provided bootstrap files in `student-start`.
5. Populate GridFS with the images also located in the `db/` folder and supplied with the bootstrap files in `student-start`.

   Hint: The following snippet will loop thru the set of images. You must ingest the contents of each of these files as a `Photo`.

   ```
   > Dir.glob("./db/image*.jpg") { |f| p f}
   "./db/image3.jpg"
   ...
   "./db/image2.jpg"
   ```
6. For each `photo` in GridFS, locate the nearest `place` within one (1) mile of each `photo` and associated the `photo` with that `place`. (Hint: make sure to convert miles to meters for the inputs to the search).
7. As a self-test, verify that you have the following `place`s – shown by their formatted address – associated with a `photo` and can locate this association with a reference to the `place`.

   ```
   > pp Place.all.reject {|pl| pl.photos.empty?}.map {|pl| pl.formatted_address}.sort

   ["1399 Baltimore Pike, Chadds Ford, PA 19317, USA",
    "77713-77735 Dillon Rd, Desert Hot Springs, CA 92241, USA",
    "8 Badgergate Ave, Wilsden, Bradford, West Yorkshire BD15 0LJ, UK",
    "Flamingo Beach Road, Playa Flamingo, Costa Rica",
    "Hamanasu Line, Ohatamachi, Mutsu-shi, Aomori-ken 039-4401, Japan",
    "Zieglmeierstra.e 11, 82383 Hohenpei.enberg, Germany"]
   ```

   ```
   $ rake db:seed
   $ rspec spec/seed_spec.rb
   ```

Serve Photo Images

In this section you must build a minimal web tier to serve up your photos thru a raw URI. This is primarily a demonstration and test of what you have accomplished at the data tier. All assembly instructions will be provided here. The success of this section will be based on whether a `jpeg` image is served to the web client when accessing the `/photos/:id/show` URI for a known `id`.

1. Create a controller class for serving up `Photo` contents using `rails g controller`. Add a single action called `show` to the controller. This will be used to serve up the contents of the `photo`.

```
$ rails g controller photos show
      create  app/controllers/photos_controller.rb
       route  get 'photos/show'
  ...
```

In addition to the controller class, Rails will create a URI route to the action using the URI shown below. Additionally, a helper method called `photos_show_path` is created and refers to that URI. However, this is not good enough because the URI must be able to express an `id` of the desired image.

```
$ rake routes
   Prefix Verb URI Pattern            Controller#Action
   photos_show GET  /photos/show(.:format) photos#show
```

2. Update the entry in `config/routes.rb` to include an `:id` parameter to the `show` action. The `:id` is a key after the `photos` resource collection in the URI. Once you add that – Rails will want some additional information specified – to include controller and action (since we have customized this somewhat). We can also restore the helper method by specifying the the `as:` parameter.

```
#get 'photos/show'
get 'photos/:id/show', to: 'photos#show', as: 'photos_show'
```

```
$ rake routes

 Prefix Verb URI Pattern                Controller#Action
 photos_show GET  /photos/:id/show(.:format) photos#show
```

You can verify your URI is correct by navigating to the following URL and seeing the default page displayed.

Note: If not done already, launch your rails server using `rails s`

```
http://localhost:3000/photos/1/show
Photos#show
Find me in app/views/photos/show.html.erb
```

3. Implement the `photos#show` action within the controller class. All files within this assignment are mime type `image/jpeg`.

```
def show
    @photo = Photo.find(params[:id])
    send_data @photo.contents, { type: 'image/jpeg', disposition: 'inline'}
end
```

Locate a sample `id` using the Rails console.

```
> Photo.all.sample.id
 => "56554251e301d0ed8c00003f"
```

Use that `id` in the URI to see a sample image.

```
http://localhost:3000/photos/56554251e301d0ed8c00003f/show
```

```
$ rspec spec/images_spec.rb
```

Show Places and Photo Images

In this section you must build a minimal web tier to serve up your places and associated photos. This is primarily a demonstration and test of what you have accomplished at the data tier. All assembly instructions will be provided here. The success of this section will be based on whether the `places#index` and `places#show` pages have been implemented. You are free to explore how to expand on this view once the assignment has been submitted.

1. Create a complete scaffold with controller and views for `Place` using the `rails g scaffold_controler` command.

```
$ rails g scaffold_controller place formatted_address
      create  app/controllers/places_controller.rb
      invoke  erb
```

```
create    app/views/places
create    app/views/places/index.html.erb
create    app/views/places/edit.html.erb
create    app/views/places/show.html.erb
create    app/views/places/new.html.erb
create    app/views/places/_form.html.erb
```

2. Add the `places#index` as the default URI for the application and register the `places` resource. This will generate the full suite of URIs for the resource. Since we are only going to use `index` and `show` witin the scope of this assignment – limit the actions to only those two.

   ```
   root 'places#index'
   resources :places, only: [:index, :show]
   get 'photos/:id/show', to: 'photos#show', as: 'photos_show'
   ```

   Once you made these updates in `config/routes.rb`, invoke the `rake routes` command

   ```
        Prefix Verb URI Pattern                Controller#Action
          root GET  /                          places#index
        places GET  /places(.:format)          places#index
         place GET  /places/:id(.:format)      places#show
   photos_show GET  /photos/:id/show(.:format) photos#show
   ```

3. Update the model and and generated view classes to be able to view the index page

   Include the `ActiveModel::Model` mixin. This quickly adds several key properties to the model that are expected by the scaffold-generated view.

   ```
   class Place
     include ActiveModel::Model
   ```

   Remove the following lines from the `index` page (`app/views/places/index.html.erb`). We have removed these links.

   ```
   <td><%= link_to 'Edit', edit_place_path(place) %></td>
   <td><%= link_to 'Destroy', place, method: :delete, data: { confirm: 'Are you sure?' } %></td>
   <%= link_to 'New Place', new_place_path %>
   ```

   You should now be able to view the index page of `formatted_addresses`

   ```
   http://localhost:3000/places
   ```

   Add thumbnail-sized preview images on the index page by updating the index page one last time.

   ```
   <% @places.each do |place| %>
   <% photo=place.photos.sample %>
     <tr>
       <td><%= place.formatted_address %></td>
       <% if photo %>
         <td><img height="50px" width="65px" src= <%= photos_show_path("#{photo.id}")%>/></td>
       <% end %>
   ```

4. Update the model and and generated view classes to be able to view the show page

   Add the `persisted?` method that returns true if the model instance has been saved to the database. This will allow it to use the `:id` to navigate from the index page to the show page.

   ```
   class Place
     include ActiveModel::Model

     def persisted?
       !@id.nil?
     end
   ```

The `rails g` command generated a `show` page (`app/views/places/show.html.erb`) that looks like the following. Note that the `Formatted address` text is used by the rspec tests to determine if this page successfully displayed. Please do not remove that text and re-add it if it is missing.

```
<p id="notice"><%= notice %></p>

<p>
  <strong>Formatted address:</strong>
  <%= @place.formatted_address %>
</p>

<%= link_to 'Edit', edit_place_path(@place) %> |
<%= link_to 'Back', places_path %>
```

Remove the following line from the `show` page with links to a page we will not be implementing in this assignment.

```
<%= link_to 'Edit', edit_place_path(@place) %> |
```

Update the place `show` page to display each photo associated with the place.

```
<% @place.photos.each do |photo| %>
  <p>
    <img height="500px" width="650px" src= <%= photos_show_path("#{photo.id}")%>/>
  </p>
<% end %>
```

You should now be able to view the individual page for each `place`

5. The test data, unfortunately has only a single photo for a specific location. Go back into rails console, import and associate the photos multiple times to see multiple images on the show page.

   ```
   $ rspec spec/web_spec.rb
   ```

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: 2016-07-17