Integrating Mongo Ruby API with Rails

This project provides an example of integrating the MongoDB with Rails using the MongoDB Ruby Driver. To implement this, we will create a model class that encapsulates all interaction with the MongoDB collection. The class methods will implement the standard ActiveModel `all()` and `find()` methods in addition to convenience methods to return the `mongo_client` and `collection`. Instances of the class will represent a specific document and its properties. The `save()`, `update()`, and `destroy()` instance methods are added to support further standard ActiveModel behavior. Other properties required by the Rails scaffolding, which expects the model class to be an ActiveModel instance, will also be added.

Create Application and Configure MongoDB Connection

Create a new application

```
$ rails new zips
```

Add Mongoid to Application to supply MongoDB Connection

Add the Mongoid gem to the `Gemfile`

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

Run the bundler

```
$ bundle
```

Generate a mongo database configuration file.

```
$ rails g mongoid:config
(output:)
      create  config/mongoid.yml
```

While the defaults generated from the above command are fine, do make sure that the names defined for `development:clients:default:database` and `test:clients:default:database` agree with where you import the JSON in a later step.

```
$ cat config/mongoid.yml | grep -v \# | grep -v '^$'
development:
  clients:
    default:
      database: zips_development
      hosts:
        - localhost:27017
      options:
  options:
test:
  clients:
    default:
      database: zips_test
      hosts:
        - localhost:27017
      options:
        read:
          mode: primary
        max_pool_size: 1
```

Next, we need to add some configurations to `config/application.rb`. This is used by stand-alone programs like `rails console` to be able to load the Mongoid environment with fewer steps. This also configures which ORM your scaffold commands use by default. Adding the mongoid gem had the impact of making it the default ORM. The lines below show how we can set it back to ether ActiveRecord or Mongoid. I am leaving it as ActiveRecord here so that we do not rock the boat too soon.

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

    # 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}

    # Do not swallow errors in after_commit/after_rollback callbacks.
    config.active_record.raise_in_transactional_callbacks = true
  end
end
```

Start Web Server

```
$ rails s
```

Download zips.json from MongoDB Web Site and Import

Download the zips.json from the MongoDB web site. The following shows an example of downloading it using `curl`. You may use a web browser. Place the downloaded file in `db/zips.json`, relative to the root of your application. The `db` directory should already exist.

```
curl http://media.mongodb.org/zips.json -o db/zips.json
```

Import the `zips.json` into the `zips` collection in the `zips_development` database.

Note 1: The `--db` name must match the name defined in `config/mongoid.yml`

Note 2: If not yet running, start your MongoDB Server using `mongod`

```
$ mongoimport --drop --db zips_development --collection zips --file db/zips.json
2015-09-12T19:44:10.785-0400    connected to: localhost
2015-09-12T19:44:11.554-0400    imported 29353 documents
```

To help familiarize you with the data we just imported, below is a representative document from the `zips.json` data set

```
{
    "_id" : "01007",
    "city" : "BELCHERTOWN",
    "loc" : [ -72.410953, 42.275103 ],
    "pop" : 10579,
    "state" : "MA"
}
```

Create a Zip Class to Access the Database

We will be manually creating an `app/model/zip.rb` class that initially can obtain a connection to the MongoDB server and database. We will be evolving this class in the subsequent sections, so to start, the `zip.rb` model will implement the following:

- a `mongo_client` class method that returns the default database connection
- a `collection` class method that returns a reference to the zips collection.

These methods are consistent with methods implemented when using mongoid ORM.

Add the `mongo_client` and `collection` class methods

```
class Zip
  # convenience method for access to client in console
  def self.mongo_client
   Mongoid::Clients.default
  end

  # convenience method for access to zips collection
  def self.collection
   self.mongo_client['zips']
  end
end
```

In the `rails console`, invoke the `zip` class methods to verify a connection can be made and to ensure we are interacting with the intended database within Mongo. This database should match with what was used as part of the `mongoimport` command and should contain the zip codes imported from `zips.json` into our `zips` collection.

```
$ rails c
Loading development environment (Rails 4.2.3)
> Zip.mongo_client
 => #<Mongo::Client:0x34369420 cluster=localhost:27017>

> Zip.mongo_client[:zips]
 => #<Mongo::Collection:0x22032060 namespace=zips_development.zips>

> Zip.collection
 => #<Mongo::Collection:0x21998940 namespace=zips_development.zips>

> Zip.collection.find.count
DEBUG | zips_development.count | STARTED | {"count"=>"zips", "query"=>{}}
 => 29353
```

Update the Zip Class with CRUD Methods

Now that we have a connection to the database and can access the collection, it is time to get started implementing methods to perform CRUD against our model.

Add an `all()` Class Method to Find All Documents

Lets add a method that will return all documents from the zips collection. We will be starting with a projection that only returns the fields of interest – in our case: `_id`, `city`, `state`, and `pop`. Since we started off earlier examples by eliminating location, I am choosing to show a projection that eliminates that property from our output.

```
def self.all
    collection.find
              .projection({_id:true, city:true, state:true, pop:true})
end
```

Within the `rails console`, verify we can use our implementation of the `all()` method that also omits the `location` property.

```
> Zip.all.count
 => 29353

> Zip.all.first
 => {"_id"=>"99743", "city"=>"HEALY", "pop"=>1058, "state"=>"AK"}
```

In order to expand this query, lets augment this method with the following:

- an optional find-by-prototype
- sorting
- paging parameters

Additionally, we will map the document term `pop` with an internal term `population` so that we don't conflict with a reserved word later. Make sure we keep a stable sort when manipulating the sort hash or our ordering expressed to the DB could be randomized.

Please note that the example provided is a bit more complicated than required because we have added a multi-level sort in addition to the field mappings. The ordering of keys within the sort hash must state stable while performing the mapping from `population` within the application to `pop` within the database..

```
  def self.all(prototype={}, sort={:population=>1}, offset=0, limit=100)
    #map internal :population term to :pop document term
    tmp = {} #hash needs to stay in stable order provided
    sort.each {|k,v|
      k = k.to_sym==:population ? :pop : k.to_sym
      tmp[k] = v  if [:city, :state, :pop].include?(k)
    }
    sort=tmp

    #convert to keys and then eliminate any properties not of interest
    prototype=prototype.symbolize_keys.slice(:city, :state) if !prototype.nil?

    Rails.logger.debug {"getting all zips, prototype=#{prototype}, sort=#{sort}, offset=#{offset}, limit=#{

    result=collection.find(prototype)
          .projection({_id:true, city:true, state:true, pop:true})
          .sort(sort)
          .skip(offset)
    result=result.limit(limit) if !limit.nil?

    return result
  end
```

Verify that the default arguments to our augmented `all()` method returns only 100 documents. Observe the MongoDB API debug statement to notice that the results are sorted by population.

```
> Zip.all.to_a.count
getting all zips, prototype={}, sort={:pop=>1}, offset=0, limit=100
```

```
zips_development.find | {"find"=>"zips", "filter"=>{},
    "projection"=>{"_id"=>true, "city"=>true, "state"=>true, "pop"=>true},
    "skip"=>0, "limit"=>100, "sort"=>{"pop"=>1}}

 => 100
```

Verify that the former default behavior can be obtained by passing in empty values for `prototype`, `sort`, and `limit`.

```
> Zip.all({},{},0,nil).to_a.count
getting all zips, prototype={}, sort={}, offset=0, limit=
zips_development.find | {"find"=>"zips", "filter"=>{},
  "projection"=>{"_id"=>true, "city"=>true, "state"=>true, "pop"=>true}, "skip"=>0, "sort"=>{}}

 => 29353
```

Lets attempt a few additional query combinations for our updated `all()` method

```
> Zip.all({'state':'NY'},{'population':-1},0,1).first

getting all zips, prototype={:state=>"NY"}, sort={:pop=>-1}, offset=0, limit=1
zips_development.find | {"find"=>"zips", "filter"=>{"state"=>"NY"},
    "projection"=>{"_id"=>true, "city"=>true, "state"=>true, "pop"=>true},
    "skip"=>0, "limit"=>1, "sort"=>{"pop"=>-1}}

 => {"_id"=>"11226", "city"=>"BROOKLYN", "pop"=>111396, "state"=>"NY"}

> Zip.all({'state':'NY'},{'population':1},0,1).first

getting all zips, prototype={:state=>"NY"}, sort={:pop=>1}, offset=0, limit=1
zips_development.find {"find"=>"zips", "filter"=>{"state"=>"NY"},
    "projection"=>{"_id"=>true, "city"=>true, "state"=>true, "pop"=>true},
    "skip"=>0, "limit"=>1, "sort"=>{"pop"=>1}}

 => {"_id"=>"13436", "city"=>"RAQUETTE LAKE", "pop"=>0, "state"=>"NY"}
```

Add Instance Support for the Fields and Add the Ability to Initialize from a Hash

Implement the `initialize()` method so that it can accept a hash for both the internal (`:id` and `:populate`) and external (`:_id` and `:pop`) views of our fields. The mapping of `:_id` to `:id` will help integrate with Rails scaffold. The mapping from `:pop` to `:populate` helps us avoid overriding a method introduced later.

Notice the `initialize()` method's `params` hash must account for the `id` coming in from the view as `:id`, while coming in from the database as `:_id`. Both Rails and MongoDB have specific names they want for this key and we must account for this difference.

```
  attr_accessor :id, :city, :state, :population

  def to_s
    "#{@id}: #{@city}, #{@state}, pop=#{@population}"
  end

  # initialize for both Mongo and a Web hash
  def initialize(params={})
    #switch between both internal and external views of id and population
    @id=params[:_id].nil? ? params[:id] : params[:_id]
    @city=params[:city]
```

```
    @state=params[:state]
    @population=params[:pop].nil? ? params[:population] : params[:pop]
  end
```

Test out the initializer by creating an instance of the Zip class from a document returned from MongoDB query.

```
> doc=Zip.all({'state':'NY'},{'population':-1},0,1).first
 => {"_id"=>"11226", "city"=>"BROOKLYN", "pop"=>111396, "state"=>"NY"}
> obj=Zip.new(doc)
 => #<Zip:0x0000000829c130 @id="11226", @city="BROOKLYN", @state="NY", @population=111396>
```

Add a `find()` Class Method to Find/Return a Specific Instance

Enable `find()` to query by the `:_id=>id` passed into the method. If found, return a `Zip` instance initialized from the document hash returned by MongoDB.

```
  def self.find id
    Rails.logger.debug {"getting zip #{id}"}

    doc=collection.find(:_id=>id)
                  .projection({_id:true, city:true, state:true, pop:true})
                  .first
    return doc.nil? ? nil : Zip.new(doc)
  end
```

Lets test this method by obtaining a `Zip` instance for a particular zipcode and report its population.

```
> Zip.find("11226").population
getting zip 11226
zips_development.find | {"find"=>"zips", "filter"=>{"_id"=>"11226"},
    "projection"=>{"_id"=>true, "city"=>true, "state"=>true, "pop"=>true}}

 => 111396
```

Implement a `save()` Instance Method to Insert a New Zip

The `save()` method should preserve the state of the current `zip` instance. Note that although we are manually assigning the `_id` in this case and already know the value without consulting the `result`. However, for dyanamic `_id` assignment cases, we can get the `inserted_id` from the `result`.

```
  def save
    Rails.logger.debug {"saving #{self}"}

    result=collection.insert_one(_id:@id, city:@city, state:@state, pop:@pop)
    @id=result.inserted_id
  end
```

We will create a `Zip` instance and then call `save()` to insert it into the Database

```
> zip=Zip.new({'id':"00001",'city':"Fake City",'state':"WY",'population':3})
 => #<Zip:0x00000006fd90c0 @id="00001", @city="Fake City", @state="WY", @population=3>

> zip.save
saving 00001: Fake City, WY, pop=3
zips_development.insert | STARTED | {"insert"=>"zips", "documents"=>[{"_id"=>"00001",
    "city"=>"Fake City", "state"=>"WY", "pop"=>3}], "writeConcern"=>{"w"=>1}, "ordered"=>true}
 => "00001"
```

Add an `update()` Instance Method to Change the Values in the Database

Create an `update()` method that accepts a hash and performs an update on those values after accounting for any name mappings.

Note that here – again – our method would be much simpler if we did not have to manually map `pop` to `population` internally.

```
  def update(updates)
    Rails.logger.debug {"updating #{self} with #{updates}"}

    # map internal :population term to :pop document term
    updates[:pop]=updates[:population]  if !updates[:population].nil?
    updates.slice!(:city, :state, :pop) if !slice.nil?

    self.class.collection
              .find(_id:@id)
              .update_one(:$set=>updates)
  end
```

To test `update()` we will obtain a zip instance using `find()`, then update its `population` from 3 to 4. Notice that `:$set` was used to change specific field(s), without changing fields not supplied.

```
> zip=Zip.find "00001"
getting zip 00001
 => #<Zip:0x00000005835f68 @id="00001", @city="Fake City", @state="WY", @population=3>

> zip.update({:population=>4})
updating 00001: Fake City, WY, pop=3 with {:population=>4}
zips_development.update | {"update"=>"zips", "updates"=>[{"q"=>{:_id=>"00001"},
    "u"=>{:$set=>{:pop=>4}}, "multi"=>false, "upsert"=>false}], "writeConcern"=>{:w=>1}, "ordered"=>true}
 => #<Mongo::Operation::Result:46208000 documents=[{"ok"=>1, "nModified"=>1, "n"=>1}]>

> zip=Zip.find("00001").population
getting zip 00001
 => 4
```

Add a `destroy()` Instance Method to Remove the Current Zip from the Database

`destroy()` will delete the document from the database that is associated with the instance's `:id`.

```
  def destroy
    Rails.logger.debug {"destroying #{self}"}

    self.class.collection
              .find(_id:@id)
              .delete_one
  end
```

Load an instance with the state of one of the cities and remove that city from the database.

```
> zip=Zip.find "00001"
getting zip 00001
zips_development.find | {"find"=>"zips", "filter"=>{"_id"=>"00001"},
    "projection"=>{"_id"=>true, "city"=>true, "state"=>true, "pop"=>true}}
 => #<Zip:0x000000047cee8 @id="00001", @city="Fake City", @state="WY", @population=4>
```

```
> zip.destroy
destroying 00001: Fake City, WY, pop=4
zips_development.delete | {"delete"=>"zips",
    "deletes"=>[{"q"=>{"_id"=>"00001"}, "limit"=>1}], "writeConcern"=>{"w"=>1}, "ordered"=>true}
 => #<Mongo::Operation::Result:52648160 documents=[{"ok"=>1, "n"=>1}]>

> zip=Zip.find "00001"
getting zip 00001
 => nil
```

Include `ActiveModel::Model` Mixin Behavor

We need to include the `ActiveModel::Model` mixin and override its `persisted?` implementation to simply return the result of whether a primary key has been assigned. JSON marshalling will also expect a `created_at` and `updated_at` by default.

```
class Zip
  include ActiveModel::Model
...
  def persisted?
    !@id.nil?
  end
  def created_at
    nil
  end
  def updated_at
    nil
  end
```

Full Model Class

Our class should now be at a point where it can integrate with Rails as an official Model class – with some help.

```
class Zip
  include ActiveModel::Model

  attr_accessor :id, :city, :state, :population

  def to_s
    "#{@id}: #{@city}, #{@state}, pop=#{@population}"
  end

  # initialize from both a Mongo and Web hash
  def initialize(params={})
    #switch between both internal and external views of id and population
    @id=params[:_id].nil? ? params[:id] : params[:_id]
    @city=params[:city]
    @state=params[:state]
    @population=params[:pop].nil? ? params[:population] : params[:pop]
  end

  # tell Rails whether this instance is persisted
  def persisted?
    !@id.nil?
  end
```

```ruby
def created_at
  nil
end
def updated_at
  nil
end

# convenience method for access to client in console
def self.mongo_client
 Mongoid::Clients.default
end

# convenience method for access to zips collection
def self.collection
 self.mongo_client['zips']
end

# implement a find that returns a collection of document as hashes.
# Use initialize(hash) to express individual documents as a class
# instance.
#   * prototype - query example for value equality
#   * sort - hash expressing multi-term sort order
#   * offset - document to start results
#   * limit - number of documents to include
def self.all(prototype={}, sort={:population=>1}, offset=0, limit=100)
  #map internal :population term to :pop document term
  tmp = {} #hash needs to stay in stable order provided
  sort.each {|k,v|
    k = k.to_sym==:population ? :pop : k.to_sym
    tmp[k] = v  if [:city, :state, :pop].include?(k)
  }
  sort=tmp
  #convert to keys and then eliminate any properties not of interest
  prototype=prototype.symbolize_keys.slice(:city, :state) if !prototype.nil?

  Rails.logger.debug {"getting all zips, prototype=#{prototype}, sort=#{sort}, offset=#{offset}, limit=#{

  result=collection.find(prototype)
        .projection({_id:true, city:true, state:true, pop:true})
        .sort(sort)
        .skip(offset)
  result=result.limit(limit) if !limit.nil?

  return result
end

# locate a specific document. Use initialize(hash) on the result to
# get in class instance form
def self.find id
  Rails.logger.debug {"getting zip #{id}"}

  doc=collection.find(:_id=>id)
                .projection({_id:true, city:true, state:true, pop:true})
                .first
  return doc.nil? ? nil : Zip.new(doc)
end
```

```
  # create a new document using the current instance
  def save
    Rails.logger.debug {"saving #{self}"}

    self.class.collection
              .insert_one(_id:@id, city:@city, state:@state, pop:@population)
  end

  # update the values for this instance
  def update(updates)
    Rails.logger.debug {"updating #{self} with #{updates}"}

    #map internal :population term to :pop document term
    updates[:pop]=updates[:population]  if !updates[:population].nil?
    updates.slice!(:city, :state, :pop) if !updates.nil?

    self.class.collection
              .find(_id:@id)
              .update_one(:$set=>updates)
  end

  # remove the document associated with this instance form the DB
  def destroy
    Rails.logger.debug {"destroying #{self}"}

    self.class.collection
              .find(_id:@id)
              .delete_one
  end
end
```

Create Controller and View

Generate the controller and view using a scaffold command that does not create a model class.

```
$ rails g scaffold_controller Zip id city state population:integer
      create  app/controllers/zips_controller.rb
      invoke  erb
      create    app/views/zips
      create    app/views/zips/index.html.erb
      create    app/views/zips/edit.html.erb
      create    app/views/zips/show.html.erb
      create    app/views/zips/new.html.erb
      create    app/views/zips/_form.html.erb
      invoke  test_unit
      create    test/controllers/zips_controller_test.rb
      invoke  helper
      create    app/helpers/zips_helper.rb
      invoke    test_unit
      invoke  jbuilder
      create    app/views/zips/index.json.jbuilder
      create    app/views/zips/show.json.jbuilder
```

Verify the route to the new controller is in place in `config/routes.rb`

```
Rails.application.routes.draw do
  resources :zips
```

Access the new page and observe an error between what was returned by the Model (a hash with an `:_id` key) and what is required by the view (an instance of a class with the id() method).

```
http://localhost:3000/zips

undefined method ‘id’ for {"_id"=>"99773", "city"=>"SHUNGNAK", "pop"=>0, "state"=>"AK"}:BSON::Document
```

```
    <% @zips.each do |zip| %>
      <tr>
        <td><%= zip.id %></td>
        <td><%= zip.city %></td>
        <td><%= zip.state %></td>
        <td><%= zip.population %></td>
```

Add the following helper method to the `app/helpers/zips_helper.rb` to convert a Mongo document to a Ruby class instance. We left it as a document so the `all()` method did not have to EAGERly access every document in the result set before it was know it would be used.

```
module ZipsHelper
  def toZip(value)
    #change value to a Zip if not already a Zip
    return value.is_a?(Zip) ? value : Zip.new(value)
  end
end
```

Add a call to the helper method in `app/views/zips/index.html.erb`

```
    <% @zips.each do |zip| %>
      <% zip=toZip(zip) %>
```

Not there is a similar issue with the JSON view as well

```
http://localhost:3000/zips.json
key not found: :id

json.array!(@zips) do |zip|
  json.extract! zip, :id, :id, :city, :state, :population
  json.url zip_url(zip, format: :json)
end
```

Fix the JSON view error by calling the helper method in `app/helpers/index.json.builder`.

```
json.array!(@zips) do |zip|
  zip=toZip(zip)
  json.extract! zip, :id, :id, :city, :state, :population
  json.url zip_url(zip, format: :json)
end
```

Test Drive

In our `ZipsController`, prior to a `:show`, `:edit`, `:update`, or `:destroy` action being executed, we will first invoke the `set_zip()` method to retreive the specific `Zip` by `id`. The generated helper comes ready to call `Zip.find` and expect to get an instance back.

The `zip_params()` method restricts mass assignments for `:zip` parameters to the fields of `:id`, `:city`, `:state`, and `:populaton`.

```ruby
class ZipsController < ApplicationController
  before_action :set_zip, only: [:show, :edit, :update, :destroy]

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_zip
      @zip = Zip.find(params[:id])
    end

    # Never trust parameters from the scary internet, only allow the white list through.
    def zip_params
      params.require(:zip).permit(:id, :city, :state, :population)
    end
```

Index

`index()` retrieves all the `Zips`. The generated action method comes ready to call `Zip.all`.

```ruby
http://localhost:3000/zips
http://localhost:3000/zips.json
def index
  @zips = Zip.all
end
```

Show

`show()` retrieves a specific `Zip` based upon its `id`. The generated action method is fully implemented by the generated `set_zip` helper method.

```ruby
#GET /zips/{id}
#GET /zips/{id}.json
  before_action :set_zip, only: [:show, :edit, :update, :destroy]
  def set_zip
    @zip = Zip.find(params[:id])
  end

  def show
  end
```

New + Create

`new()` returns an initial prototype to the form to create a new `Zip`.

`create()` accepts the results and creates a new `Zip` instance in the database.

Both of these generated action methods come ready to call `Zip.new`, which uses the `initialize` method. The generated create action also comes ready to invoke `save` on the `Zip` instance.

```ruby
#POST /zips/new
  def new
    @zip = Zip.new
  end

#POST /zips
  def create
    @zip = Zip.new(zip_params)
```

```
    respond_to do |format|
      if @zip.save
        format.html { redirect_to @zip, notice: 'Zip was successfully created.' }
        format.json { render :show, status: :created, location: @zip }
      else
        format.html { render :new }
        format.json { render json: @zip.errors, status: :unprocessable_entity }
      end
    end
  end
```

Edit + Update

`edit()` retrieves the instance from the database based upon its `id`.

`update()` applies the changes to the retrieved instance provided by `edit()`.

Both generated action methods rely on the `before_action` and the generated `update` action also comes ready to call `update` on the `Zip` instance.

`http://localhost:3000/zips/00002/edit`

```
#GET /zips/{id}
  before_action :set_zip, only: [:show, :edit, :update, :destroy]
  def set_zip
    @zip = Zip.find(params[:id])
  end

  def edit
  end

#PUT /zips/{id}
  def update
    respond_to do |format|
      if @zip.update(zip_params)
        format.html { redirect_to @zip, notice: 'Zip was successfully updated.' }
        format.json { render :show, status: :ok, location: @zip }
      else
        format.html { render :edit }
        format.json { render json: @zip.errors, status: :unprocessable_entity }
      end
    end
  end
```

Destroy

`destroy()` removes a specific `Zip` instance based upon its `id`. The generated action method comes ready to call `destroy` on the `Zip` instance.

```
#DELETE /zips/{id}
  def destroy
    @zip.destroy
    respond_to do |format|
      format.html { redirect_to zips_url, notice: 'Zip was successfully destroyed.' }
      format.json { head :no_content }
    end
  end
```

Root Application

Add a second line to `config/routes.rb` to make zips the root application.

```
Rails.application.routes.draw do
  root 'zips#index'
```

Add Pagination

Add the `will_paginate` to the Gemfile

```
gem 'will_paginate', '~> 3.0.7'
```

Execute `bundle` to install the Gem

```
bundle
```

Add the `will_paginate` Command to the View

This command will add page properties to the displayed view and controls advance paging. Specifically, it can pass `:page` as a number >= 1. `will_paginate` also uses `:per_page` to express the row limit for a single page.

```
<table>
...
  <tbody>
    <% @zips.each do |zip| %>
      <% zip=toZip(zip) %>
      <tr>
        <td><%= zip.id %></td>
...
      </tr>
    <% end %>
  </tbody>
</table>
<%= will_paginate @zips %>
```

Add `will_paginate` Support for Paging in the Controller

The controller is passing a controlled set of parameters to the `Model.paginate` call. The page number is currently the only value passed but `:per_page` could be specified here as well.

```
  def index
    #@zips = Zip.all
    @zips = Zip.paginate(:page => params[:page])
  end
```

Add `will_paginate` Support for Paging in the Model

This method implements a facade around the `all()` method by translating the `will_paginate` inputs into `all()` query inputs and converts the document array results into a `will_paginate` result that contains such things as total number of documents.

```
  def self.paginate(params)
    Rails.logger.debug("paginate(#{params})")
    page=(params[:page] ||= 1).to_i
    limit=(params[:per_page] ||= 30).to_i
    offset=(page-1)*limit

    #get the associated page of Zips -- eagerly convert doc to Zip
    zips=[]
    all({}, {}, offset, limit).each do |doc|
      zips << Zip.new(doc)
    end

    #get a count of all documents in the collection
    total=all({}, {}, 0, 1).count

    WillPaginate::Collection.create(page, limit, total) do |pager|
      pager.replace(zips)
    end
  end
```

A reference on how to use will_paginate

Quick Test Drive

As a quick test to verify our current `will_paginate` implementation, the following URL should land us on the 3rd page of "Listing Zips"

`http://localhost:3000/?page=3`

Add Selection Criteria and Ordering

Given the following URL:

`http://localhost:3000/?page=38&per_page=10&sort=population:-1,city:1&state=MD`

Lets implement additional functionality that will allow our application to:

- Specify a particular page number: `page=38`
- Define a row limit per page: `per_page=10`
- Order the results by population DESC, city ASC: `sort=population:-1,city:1`
- Define zips for a particular state: `state=MD`

Create a Helper Method in the Controller to Convert the Sort Query Param to a MongoDB Query Sort Hash `app/controller/zips_controller.rb`

```
private
    #create a hash sort spec from query param
    #sort=state:1,city,population:-1
    #{state:1, city:1, population:-1}
    def get_sort_hash(sort)
      order={}
      if (!sort.nil?)
        sort.split(",").each do |term|
          args=term.split(":")
          dir = args.length<2 || args[1].to_i >= 0 ? 1 : -1
          order[args[0]] = dir
```

```
      end
    end
    return order
  end
```

Update the Controller Method to Pass the Query and Sort Terms into the will_paginate Call This passes right to our `Model.paginate` call where we can add a small amount of processing to pass it through to the `all()` method.

```
def index
  #@zips = Zip.all
  #@zips = Zip.paginate(params)

  args=params.clone                        #update a clone of params
  args[:sort]=get_sort_hash(args[:sort]) #replace sort with hash
  @zips = Zip.paginate(args)
end
```

Update the Model Method to Pass the Query and Sort Terms into `all()` `app/models/zip.rb`

```
def self.paginate(params)
  ...
  sort=params[:sort] ||= {}

  ...
  all(params, sort, offset, limit).each do |doc|

  ...
  total=all(params, sort, 0, 1).count

  ...
end
```

Final Test Drive

With our selection criteria and ordering logic now in place, the below URL:

`http://localhost:3000/?page=38&per_page=10&sort=population:-1,city:1&state=MD`

should render us results similiar to this:

Listing Zips

Id	City	State	Population			
21522	BITTINGER	MD	479	Show	Edit	Destroy
21156	Upper Falls	MD	464	Show	Edit	Destroy
20632	FAULKNER	MD	459	Show	Edit	Destroy
21677	WOOLFORD	MD	459	Show	Edit	Destroy
21816	CHANCE	MD	415	Show	Edit	Destroy
20630	DRAYDEN	MD	413	Show	Edit	Destroy
20779	TRACYS LANDING	MD	413	Show	Edit	Destroy

Id	City	State	Population			
20615	BROOMES ISLAND	MD	404	Show	Edit	Destroy
21672	TODDVILE	MD	361	Show	Edit	Destroy
21840	NANTICOKE	MD	358	Show	Edit	Destroy

```
<-- Previous 1 2 ...  34 35 36 37 38 39 40 41 42 Next -->
```

If you flip the value of `city:1` from `1` to `-1`, should will see the ordering of `DRAYDEN` and `TRACYS LANDING` switch places.

Heroku Deployment

Setup Database on MongoLabs

1. Create a MongoLabs Account

   `https://mongolab.com/home`

2. Create a (Free Sandbox) Database on MongoLabs

   - From your mongolab home page, select `Create New` MongoDB Deployments
   - For `Cloud Provider:` select `Amazon Web Services`
     - On the `Location:` Pull Down Menu, select the Region that is geographically closest to you
   - Under Plan: select the `Single-node` Option
     - Select the Free `Sanbox` option under `Standard Line`
   - Leave the `High Storage Line` options blank
   - In the `Database name:` input field, supply a name (i.e., zips_production)
   - Verify the `Price:` field calculator is `$0 / month`
   - Select the `Create new MongoDB deployment` button
   - On your mongolab home page, you should now see your database (i.e., zips_production) listed
   - Now select your newly created MongoDB deployment
   - The following URL template should be displayed with the details pertaining to your deployment

     ```
     To connect using a driver via the standard MongoDB URI:
         mongodb://<dbuser>:<dbpass>@<dbhost>/<dbname>
     ```

3. Create a Database User and Password on MongoLabs

   - Select the `Users` Menu
   - Select the `Add database user` button
   - In the `Add new database user` form, supply a username and password:
     - `<dbuser>`
     - `<dbpass>`
     - then select `Create`

4. Import zips.json from MongoDB using the database and user account created above

   ```
   $ wget http://media.mongodb.org/zips.json
   $ mongoimport -h dbhost -d dbname -c zips -u dbuser -p dbpass --file zips.json
   2015-12-07T18:03:34.015-0500    connected to: ds######.mongolab.com:<port>
   2015-12-07T18:03:36.416-0500    [################.........] <dbname>.<collection_name> 2.1 MB/3.0 MB (68
   2015-12-07T18:03:38.953-0500    imported 29353 documents
   ```

Setup Application on Heroku

1. Create a Heroku Account

   - If not yet installed, download and install the Heroku Toolbelt.
   - This client CLI will be used in later steps that rely on `heroku` commands

2. Register your application with Heroku by changing to the directory with a git repository and invoking `heroku apps:create (appname)`.

   Note that your application must be in the root directory of the development folder hosting the git repository.

   ```
   $ cd zips
   $ heroku apps:create appname
   Creating appname... done, stack is cedar-14
   https://appname.herokuapp.com/ | https://git.heroku.com/appname.git
   Git remote heroku added
   ```

   This will add an additional remote to your git repository.

   ```
   $ git remote --verbose
   heroku  https://git.heroku.com/appname.git (fetch)
   heroku  https://git.heroku.com/appname.git (push)
   ...
   ```

3. Add a `MONGOLAB_URI` environment variable where `dbhost` is both host and port# concatenated together, separated by a ":" (host:port) .

   ```
   $ heroku config:add MONGOLAB_URI=mongodb://dbuser:dbpass@dbhost/dbname
   ```

4. Add a `production` profile to the `config/mongoid.yml` file. The following Mongoid connection information was provided by the `MongoLab` page on the `Heroku` Dev Center page.

   ```
   production:
     clients:
       default:
         uri: <%= ENV['MONGOLAB_URI'] %>
         options:
           connect_timeout: 15
   ```

5. Update the `Gemfile` so that Heroku will accept and deploy our application.

   Restrict the `sqlite` gem in `Gemfile` to the development profile. Heroku does not support `sqlite` and this application does not use an `RDBMS`. However, this gem was put there by by `rails new` by default and required to stick around because we have not removed ActiveRecord from the application.

   ```
   gem 'sqlite3', group: :development
   ```

   Add the postgres gem to the production profile. We have not neutered the application of ActiveRecord and Heroku wants a supported database for that platform.

   ```
   group :production do
     #use postgres on heroku
     gem 'pg'
     gem 'rails_12factor'
   end
   ```

   Be sure to run bundle when complete and check the Gemfile.lock file into git.

```
$ bundle
```

6. Commit changes to application

```
$ git commit -am "ready for heroku deploy"
```

7. Deploy application

```
$ git push heroku master
```

Access Application

1. Access URL

```
http://appname.herokuapp.com
```

2. Access logs

```
$ heroku logs
```