Mongoid Relations and Queries

This document describes the highlights and assembly of the `Movies` application using a Mongoid-based backend. Much of the emphasis is placed on the data-tier and the controller, while leaving the view minimally integrated. This is to keep the focus on the data tier.

You will find the following topics discussed in this module:

- Model classes
- Relationships
  - Embedded
  - Linked
- Queries
- Geolocation
- Indexes, etc.

Highlights

Initialization

This section quickly covers the highlights relative to getting started. All cooked data is located within `db/*.json` and can be easily imported using the `rake` shell command.

1. Run `bundle` to ensure the necessary gems are installed
2. (In a separate terminal) Start the MongoDB server using `mongod`
3. Import the data using `rake db:seed` to create all collections. This will execute the code within `db/seed.rb` and use the JSON data also located in the `db/` directory.

   ```
   $ rake db:seed
   adding places
   adding actors
   adding writers
   adding directors
   adding movies
   ```

4. Add required indexes to the collections using rake `db:mongoid:create_indexes`. The source of these indexes are located within the associated Mongoid model classes in the `app/model` directory.

   ```
   $ rake db:mongoid:create_indexes
   D, [2015-11-30T17:26:06.796020 #34396] DEBUG -- : MONGODB | Adding localhost:27017 to the cluster.
   D, [2015-11-30T17:26:06.837599 #34396] DEBUG -- : MONGODB | localhost:27017 | movies_development.create
   D, [2015-11-30T17:26:06.873807 #34396] DEBUG -- : MONGODB | localhost:27017 | movies_development.create
   ```

One of the indexes created is a `2dsphere` index on the `place_of_birth.geolocation` property of an `Actor`.

```
class Actor
  include Mongoid::Document
  embeds_one :place_of_birth, as: :locatable, class_name: 'Place'
  ...
  index ({ :"place_of_birth.geolocation" => Mongo::Index::GEO2DSPHERE })
  ...
end
```

```
{:_id=>"nm0543215",
 :name=>"Sarah Manninen",
 :place_of_birth=>
  {:_id=>"Pasadena, CA, USA",
   :geolocation=>{:type=>"Point", :coordinates=>[-118.1445155, 34.1477849]},
   :city=>"Pasadena",
   :county=>"Los Angeles County",
   :state=>"CA",
   :country=>"US"}}
```

Model Types and Document Representation

This section describes the individual document and custom types contained within the example. Documents have `_id` properties. Custom types are anonymous compound data structures embedded within documents.

Measurement (Custom Type)

`Measurement` is an example of a custom type. Instead of modeling a property with just the `amount` (60) or a compound label with "amount (units)" ("60 min"), we have chosen to create a compound data structure with `amount` and `units`.

```
:runtime=>{:amount=>60, :units=>"min"}
```

The entire class has been supplied here for the first example of a custom class. All other custom classes have a similar concept to:

- provide an initialize method that is marshaling format independent
- provide one (1) instance method called `mongoize` required by Mongoid that will marshal the state of the instance into database form.
- provide three (3) class methods to:
  - transform any form of the object into database form (`mongoize`)
  - create an object instance from database form (`demongoize`)
  - support various other transformations required by the criteria queries (`evolve`).

```ruby
class Measurement
  attr_reader :amount, :units

  def initialize(amount, units=nil)
    @amount=amount
    @units = units
    #normalize
    case
    when @units == "meters" then @amount=(@amount/0.3048); @units="feet"
    end
  end

  #creates a DB-form of the instance
  def mongoize
    @units ? {:amount => @amount, :units => @units} : {:amount => @amount}
  end

  #creates an instance of the class from the DB-form of the data
  def self.demongoize(object)
    Measurement.new(object[:amount], object[:units])
  end
```

```
  #takes in all forms of the object and produces a DB-friendly form
  def self.mongoize(object)
    case object
    when Measurement then object.mongoize
    else object
    end
  end

  #used by criteria to convert object to DB-friendly form
  def evolve(object)
    case object
    when Measurement then object.mongoize
    else object
    end
  end
end
```

Point (Custom Type)

`Point` is a custom class that represents a geolocation point. It marshals itself in GeoJSON Point format, but is used to read in other formats commonly found. The coordinates array is in the order of [`longitude`, `latitude`]. The inner workings of the class function much like the `Measurement` custom type.

```
{:type=>"Point", :coordinates=>[-118.1445155, 34.1477849]},
```

Place (Document Model Class)

`Place` is an abstraction added to `Point` to hold location information about the gelocation point. It commonly contains official address information reported by the Google Maps Geocoding API.

```
{:_id=>"Pasadena, CA, USA",
  :geolocation=>{"type"=>"Point", "coordinates"=>[-118.1445155, 34.1477849]},
  :city=>"Pasadena",
  :county=>"Los Angeles County",
  :state=>"CA",
  :country=>"US"}
```

A few things to point out about this model class

- `_id` is mapped to `formatted_address`
- the `geolocation` property uses a custom type (`Point`)
- although you will find this document type embedded within other documents, it exists as a stand-alone document within its own `places` collection.
- more details about this class will be shown in the relationships section below.
- stating the field type is not required by Mongoid, but Mongoid will test for type if supplied. A few of the attributes have had their type removed just to demonstrate they are optional.

```
class Place
  include Mongoid::Document
  field :_id, type: String, default: -> { formatted_address }
  field :formatted_address, type: String
  field :geolocation, type: Point
  field :street_number
  field :street_name
```

```
  field :city
  field :postal_code, type: String
  field :county, type: String
  field :state, type: String
  field :country, type: String
  ...
end
```

Actor

`Actor` is a document model class that represents someone who plays a role in a `Movie`. They are independent of any one movie – thus a candidate to stand-alone in their own collection and be referenced. The information in this model class is unique to the `Actor`.

```
{:_id"=>"nm0993498",
 :bio=> "Arisa Cox was born on December 7, 1978 in Toronto, ...
 :birthName=>"Arisa Natalie Cox",
 :name=>"Arisa Cox",
 :urlPhoto=> "http://ia.media-imdb.com/images/M/MV5BM...7_AL_.jpg",
 :date_of_birth=>1978-12-07 00:00:00 UTC,
 :place_of_birth=> ... (see 1:1 embedded relationship example)
```

A few things to point out about this model class

- although not necessary since the names match the default choice – the instances of `Actor` will be stored in the `actors` collection.
- `created_at` and `updated_at` fields can be added and managed by Mongoid with the addition of the `Timestamps` mixin. This will be demonstrated later.
- The camelCase `birthName` from the document is mapped to the snake_case `birth_name` in the model class. This becomes important if you compress your JSON names or are simply trying to comply with Rails naming conventions for model property names (a good thing to do).
- `date_of_birth` has been represented as a `Date` type instead of a string representation of the date
- `height` is represented as a custom type (`Measurement`)
- more details about this class will be shown in the relationships section below.

```
class Actor
  include Mongoid::Document
#  include Mongoid::Timestamps
  store_in collection: "actors"

  field :name, type: String
  field :birthName, as: :birth_name, type: String
  field :date_of_birth, type: Date
  field :height, type: Measurement
  field :bio, type: String
  ... (see relationships for additional details)
end
```

One thing to note is that by moving our schema away from being UI display/String-based, we have introduced a complexity for the UI to make non-String updates. An attempt to map many of the String to data type conversions has been inserted into the examples, but there clearly are additional data type conversions to consider when you see the full end-to-end interactions.

Notice that both the phiscal document key and the model class alias key can be used when inspecting the model instance. In the following case, we are still able to access the raw `birthName` in the document, in addition to the mapped `birth_name` in the class.

```
> Actor.find("nm0993498").birth_name
 => "Arisa Natalie Cox"
> Actor.find("nm0993498").birthName
 => "Arisa Natalie Cox"
```

```
{:_id=>"nm0993498",
 :birthName=>"Arisa Natalie Cox",
```

Writer

`Writer` is one of the authors of a `Movie` and, like `Actor`, stored in a separate collection. The example does not include enough data to warrant a separate collection, but we are going to pretend that is true. One thing to note is that `Writer` has no refined role within a Movie – so it will be referenced directly from the `Movie` document.

```
{:_id=>"nm0905152", :name=>"Andy Wachowski" ... (please imagine more properties) }
```

```
class Writer
  include Mongoid::Document
  field :name, type: String
  ...
end
```

Director

`Director` is one of the directors of a `Movie` and, like `Writer`, requires some imagination on your part that we have enough data here to warrant a separate collection for the `Director`.

```
{:_id=>"nm0001081", :name=>"Cameron Crowe" ... (please imagine more properties) }
```

```
class Director
  include Mongoid::Document
  field :name, type: String
end
```

DirectorRef

`DirectorRef` is an annotated reference to a `Director`. It will be used to cache stable/core director information that the referencing document/view normally cares about and then contains a link to the details of the `Director` (where you are asked to use your imagination that more details exist).

```
{:name=>"F.W. Murnau", :_id=>"nm0003638"}
```

```
class DirectorRef
  include Mongoid::Document
  field :name, type: String
end
```

MovieRole

`MovieRole` is a character in a `Movie` played by an `Actor`. It contains descriptive information about the movie character and the id, name, and url of the actor image and profile (both just URLs). The information for this actor that is not relative to this movie or movie role (character) is located within the `actors` collection.

```
{:actorName=>"George O'Brien",
  :character=>"The Man",
  :main=>true,
  :urlCharacter=>"http://www.imdb.com/character/ch0131526",
  :urlPhoto=>
   "http://ia.media-imdb.com/images/M/MV5BMTI2MDg3NjYx...AL_.jpg",
  :urlProfile=>"http://www.imdb.com/name/nm0639563",
  :_id=>"nm0639563"}
```

```
class MovieRole
  include Mongoid::Document
  field :character, type: String
  field :actorName, as: :actor_name, type: String
  field :main, type: Mongoid::Boolean
  field :urlCharacter, as: :url_character, type: String
  field :urlPhoto, as: :url_photo, type: String
  field :urlProfile, as: :url_profile, type: String
  ...
end
```

Movie

`Movie` contains core information about the `Movie` and links to other information related to the movie.

```
{:_id=>"tt0018455",
 :actors=>
  [{:actorName=>"George O'Brien",
    :character=>"The Man",
    :main=>true,
    :urlCharacter=>"http://www.imdb.com/character/ch0131526",
    :urlPhoto=>
     "http://ia.media-imdb.com/images/M/MV5BMTI2MDg3NjYx...AL_.jpg",
    :urlProfile=>"http://www.imdb.com/name/nm0639563",
    :_id=>"nm0639563"},
    ...
   ],
 :countries=>["USA"],
 :directors=>[{:name=>"F.W. Murnau", :_id=>"nm0003638"}],
 :filmingLocations=>
  ["Big Bear Lake", "Big Bear Valley", "San Bernardino National Forest", "California", "USA"],
 :genres=>["Comedy", "Drama", "Romance"],
 :languages=>[],
 :metascore=>"",
 :originalTitle=>"Sunrise: A Song of Two Humans",
 :plot=> "In this fable-morality subtitled  ..."
 :rated=>"NOT RATED",
 :rating=>8.4,
 :runtime=>{:amount=>94, :units=>"min"},
 :simplePlot=> "A married farmer falls under the spell of ...
 :title=>"Sunrise",
 :type=>"Movie",
 :urlIMDB=>"http://www.imdb.com/title/tt0018455",
 :urlPoster=>
  "http://ia.media-imdb.com/images/M/MV5BMjIzNzg4....TE@._V1_SX214_AL_.jpg",
 :votes=>25165,
 :writers=>["nm0562346", "nm0837183"],
```

```
 :year=>1927,
 :release_date=>1927-01-04 00:00:00 UTC}
```

A few things to point out about this model class

- `year` and `votes` are mapped as type `Integer`
- `runtime` is mapped as a custom type `Measurement`
- camelCase `filmingLocations`, `simplePlot`, `urlIMDB`, and `urlPoster` names have been mapped to snake_case `filming_locations`, `simple_plot`, `url_IMDB`, and `url_poster` names.

```
class Movie
  include Mongoid::Document
  field :title, type: String
  field :type, type: String
  field :rated, type: String
  field :year, type: Integer
  field :release_date, type: Date
  field :runtime, type: Measurement
  field :votes, type: Integer
  field :countries, type: Array
  field :languages, type: Array
  field :genres, type: Array
  field :filmingLocations, as: :filming_locations, type: Array
  field :metascore, type: String
  field :simplePlot, as: :simple_plot,  type: String
  field :plot, type: String
  field :urlIMDB, as: :url_IMDB,  type: String
  field :urlPoster, as: :url_poster, type: String
  ...
end
```

Model Relationships and Realization

This section describes additional relationship properties for the model types introduced above.

1:1 Embedded (`Actor -> place_of_birth:Place`)

The `Actor` has one (1) `place_of_birth` property with an embedded `Place`. Please note that this same `Place` instance can show up in multiple `Actors` as well as other references to this specific place. That means that access to the `Place` properties are very efficient from its parent document, but must be updated everywhere as things change. In this case it would be rare to have to update these properties unless `Canada` became part of a different country.

```
{:_id=>"nm0543215",
 :name=>"Sarah Manninen",
 ...
 :date_of_birth=>1977-01-20 00:00:00 UTC,
 :place_of_birth=>
  {:_id=>"Waterloo, ON, Canada",
   :geolocation=>{"type"=>"Point", "coordinates"=>[-80.5204096, 43.4642578]},
   :city=>"Waterloo",
   :county=>"Waterloo Regional Municipality",
   :state=>"ON",
   :country=>"CA"}}
```

The embedded `Place` declares it is `embedded_in` the parent document. In this case there are many parent document types that could have a `Place`, so we take advantage of the `polymorphic` keyword and define the parent document as a `:locatable` instead of a specific model class.

```
class Place
  include Mongoid::Document
  ...
  embedded_in :locatable, polymorphic: true
end
```

The parent document declares `embeds_one` for the embedded type. However,

- since the name of the embedded class (`Place`) is not the same as the reference (`place_of_birth`), an additional property of `class_name: 'Place'` is supplied.
- since other types can also contain this embedded type, the polymorphic label of `locatable` is also supplied.

```
class Actor
  include Mongoid::Document
  ...
  embeds_one :place_of_birth, as: :locatable, class_name: 'Place'

class Writer
  include Mongoid::Document
  ...
  embeds_one :hometown, as: :locatable, class_name: 'Place'
end
```

To demonstrate, lets locate an `Actor` that does not yet have an address listed and make up an address for their place of birth.

```
> actor=Actor.where(:place_of_birth=>{:$exists=>0}).first
> pp Actor.collection.find(:_id=>actor.id).first
{"_id"=>"nm0828941",
 "bio"=> "Gerda Stevenson was born in ..."
 "height"=>{"amount"=>5.413386, "units"=>"feet"},
 "name"=>"Gerda Stevenson"}
```

Lets claim this person is from `Oakland` and get its address properties from our `places` collection.

```
> oakland=Place.where(:city=>"Oakland").first
```

Next we use the `parent.create_(embedded_relation)` method and a copy of the `place` properties to create an embedded instance of `Place` within `Actor`.

```
> actor.create_place_of_birth(oakland.attributes)
> pp Actor.collection.find(:_id=>actor.id).first
{"_id"=>"nm0828941",
 "bio"=> "Gerda Stevenson was born in ..."
 "height"=>{"amount"=>5.413386, "units"=>"feet"},
 "name"=>"Gerda Stevenson",
 "place_of_birth"=>
  {"_id"=>"Oakland, CA, USA",
   "geolocation"=>{"type"=>"Point", "coordinates"=>[-122.2711137, 37.8043637]},
   "city"=>"Oakland",
   "county"=>"Alameda County",
   "state"=>"CA",
   "country"=>"US"}}
```

To show we can insert another copy of the same embedded object in another collecton – we locate a `Writer` that does not yet have a `hometown` (we changed the relation name on purpose) and place an embedded copy of `Oakland` there too.

```
> writer=Writer.where(:hometown=>{:$exists=>0}).first
> pp Writer.collection.find(:_id=>writer.id).first
{"_id"=>"nm0000230", "name"=>"Sylvester Stallone"}

> writer.create_hometown(oakland.attributes)
> pp Writer.collection.find(:_id=>writer.id).first
{"_id"=>"nm0000230",
 "name"=>"Sylvester Stallone",
 "hometown"=>
  {"_id"=>"Oakland, CA, USA",
   "geolocation"=>{"type"=>"Point", "coordinates"=>[-122.2711137, 37.8043637]},
   "city"=>"Oakland",
   "county"=>"Alameda County",
   "state"=>"CA",
   "country"=>"US"}}
```

We can inspect the `writer` object methods for other things we can do with our embedded object `hometown`.

```
> writer.methods.grep /hometown/
 => [:hometown, :hometown=, :hometown?, :has_hometown?, :build_hometown, :create_hometown]
```

We can get the embedded object.

```
> writer.hometown.id
 => "Oakland, CA, USA"
```

We can inquire whether we have an instance of the hometown.

```
> writer.hometown?
 => true
> writer.has_hometown?
 => true
```

We can build a transient instance of the embedded object.

```
> place=writer.build_hometown
```

We can create a hollow embedded object. This still reports that the parent document has an instance of the embedded – even though it is hollow.

```
> writer.create_hometown
> pp Writer.collection.find(:_id=>writer.id).first
{"_id"=>"nm0000230", "name"=>"Sylvester Stallone", "hometown"=>{}}
> writer.has_hometown?
 => true
```

We can remove the embedded object.

```
> writer.hometown=nil
 => nil
> writer.has_hometown?
 => false
> pp Writer.collection.find(:_id=>writer.id).first
{"_id"=>"nm0000230", "name"=>"Sylvester Stallone"}
```

But notice that simple assignment does not embed the transient association.

```
> writer.hometown
 => nil
> writer.hometown=oakland
> writer.hometown
 => #<Place _id: Oakland, CA, USA...
> pp Writer.collection.find(:_id=>writer.id).first
{"_id"=>"nm0000230", "name"=>"Sylvester Stallone"}
> writer.save
 => true
> pp Writer.collection.find(:_id=>writer.id).first
{"_id"=>"nm0000230", "name"=>"Sylvester Stallone"}
```

M:1 Linked (`Director -> residence:Place`)

In a M:1 relationship, the one (1) side is the parent (with a primary key) and the many (M) side is the child (with a foreign key to the parent). The child must declare a `belongs_to` property to have the foreign key stored locally to realize the relationship. The parent optionally defines a `has_many` property. In this case we actually have a M:1 uni-directional relationship. Many directors can have one (1) residence:Place and can navigate that link. However, the parent does not know about the relationship and cannot navigate in the other direction. Since `Place` is being used in different contexts differently – we chose to keep this uni-directional and leave `Place` ignorant of what is related to it.

```
{:_id=>"nm0001081",
 :name=>"Cameron Crowe",
 :residence_id=>"Los Angeles, CA, USA"}
```

Remember that `Place` uses its `formatted_address` as its `_id`. Thus the foreign key in this case actually contains useful business information even before navigating the link.

```
class Director
  include Mongoid::Document
  ...
  belongs_to :residence, class_name: 'Place'
end
```

To provide an example of using this relationship... locate a `Director` that does not yet have a residence.

```
> director=Director.where(:residence=>{:$exists=>0}).first
 => #<Director _id: nm0001081, name: "Cameron Crowe", residence_id: nil>
> pp Director.collection.find(:_id=>director.id).first
{:_id=>"nm0001081", :name=>"Cameron Crowe"}
```

Assign the `director.residence` to an instance of `Place`.

```
> oakland=Place.where(:city=>"Oakland").first
 => #<Place _id: Oakland, CA, USA, >
> director.residence=oakland
 => #<Place _id: Oakland, CA, USA, >
```

Notice this did not immediately write anything to the database. The foreign key to the `Place` is writen when the `Director` is saved.

```
> pp Director.collection.find(:_id=>director.id).first
{:_id=>"nm0001081", :name=>"Cameron Crowe"}
 => {:_id=>"nm0001081", :name=>"Cameron Crowe"}

> director.save
> pp Director.collection.find(:_id=>director.id).first
{:_id=>"nm0001081", :name=>"Cameron Crowe", :residence_id=>"Oakland, CA, USA"}
```

There are many helper methods for this relationship, so I will pick out a few and shoot for some that do not overlap with the embedded ones covered earlier.

```
> director.methods.grep(/residence/).count
 => 17
```

Realize the related object is now in a separate collection and the `Director` only has a foreign key stored locally to that related document. Therefore, accessing any properties related to the `Place` causes a separate query to the database since there is no such thing as a join performed within the database within MongoDB.

```
> director=Director.find(director.id)
DEBUG | {"find"=>"directors", "filter"=>{"_id"=>"nm0001081"}}
 => #<Director _id: nm0001081, name: "Cameron Crowe", residence_id: "Oakland, CA, USA">

> director.residence.state
DEBUG | {"find"=>"places", "filter"=>{"_id"=>"Oakland, CA, USA"}}
 => "CA"
```

However, if all you want to access is the foreign key stored in the child document, you can call `(relation name)_id` and not have to access the parent collection.

```
> director.residence_id
 => "Oakland, CA, USA
```

If you have the foreign key for the parent document...

```
> director=Director.find(director.id)
 => #<Director _id: nm0001081, name: "Cameron Crowe", residence_id: "Oakland, CA, USA">
> la_id=Place.where(:city=>"LA").first.id
 => "Los Angeles, CA, USA"
```

...the assignment can be made using `(relation name)_id=` instead of getting an instance of the parent to form the relationship.

```
> director.residence_id=la_id
 => "Los Angeles, CA, USA"
> director.save
> pp Director.collection.find(:_id=>director.id).first
{:_id=>"nm0001081",
 :name=>"Cameron Crowe",
 :residence_id=>"Los Angeles, CA, USA"}
```

1:M Embedded (`Movie <-> roles:MovieRole`)

For the 1:M embedded we have a single document with parent properties and multiple embedded children and the children have identities.

```
{:_id=>"tt0075148",
 :title=>"Rocky",
 :roles=>
  [{:actorName=>"Sylvester Stallone",
    :character=>"Rocky",
    :main=>true,
    ...
    :_id=>"nm0000230"},
   {:actorName=>"Talia Shire",
    :character=>"Adrian",
    :main=>true,
    ...
    :_id=>"nm0001735"},
    ...]
}
```

The parent declares an `embeds_many`, listing the collection name and the `class_name` of the embedded child if it does not match the singular form of the collection name.

```
class Movie
  include Mongoid::Document
  field :title, type: String
  ...
  embeds_many :roles, class_name:"MovieRole"
  ...
end
```

The child class delcares an `embedded_in`, listing the parent property. In this case the name of the property and name of the parent model class match – so no `class_name` is necessary.

```
class MovieRole
  include Mongoid::Document
  field :character, type: String
  field :actorName, as: :actor_name, type: String
  field :main, type: Mongoid::Boolean
  ...
  embedded_in :movie
  ...
end
```

We can demonstrate the 1:M embedded by first creating a new `Movie`. The `create` method performs the combined function of `new/initialize` and `save` and the parent document must be saved before creating embedded objects (i.e., saving to database).

```
> rocky25=Movie.create(:_id=>"tt9000000", :title=>"Rocky XXV")
```

We find an actor to play a role in the movie and then create that role relative to the collection. That immediately updates the parent document with the embedded role. Since this is a 1:M relationship, the role is placed within an array.

```
> stallone=Actor.where(:name=>{:$regex=>"Stallone"}).first
> rocky=rocky25.roles.create(:_id=>stallone.id, :character=>"Rocky", :actorName=>"Sly", :main=>true)
> pp Movie.collection.find(:title=>"Rocky XXV").first
{"_id"=>"tt9000000",
 "title"=>"Rocky XXV",
 "roles"=>
  [{"_id"=>"nm0000230",
    "character"=>"Rocky",
    "actorName"=>"Sly",
    "main"=>true}]}
```

To add a second role – we locate another actor and fill in the details of the role using a transient instance and then add the instance to the collection. The addition to the collection immediately updates the state of the parent document.

```
> actor=Actor.first
> role=MovieRole.new
> role.id=actor.id
> role.character="Challenger"
> role.main=false
> role.actor_name=actor.name
> rocky25.roles << role

> pp Movie.collection.find(:title=>"Rocky XXV").first
{:_id=>"tt9000000",
 :title=>"Rocky XXV",
 :roles=>
  [{:_id=>"nm0000230", :character=>"Rocky", :actorName=>"Sly", :main=>true},
   {:_id=>"nm0084430",
    :character=>"Challenger",
    :main=>false,
    :actorName=>"Erwin Biswanger"}]}
```

We can repeat the example – except this time form the entire parent document prior to the intial save.

```
> rocky26=Movie.new(:_id=>"tt9000001", :title=>"Rocky XXVI")
> rocky=rocky26.roles.build(:_id=>stallone.id, :character=>"Rocky", :actorName=>"Sly", :main=>true)
> rocky26.roles << role
> pp Movie.collection.find(:title=>"Rocky XXVI").first
nil
```

At the point we have the transient object fully formed and now will save to the database.

```
> rocky26.save
 => true
> pp Movie.collection.find(:title=>"Rocky XXVI").first
{"_id"=>"tt9000001",
 "title"=>"Rocky XXVI",
 "roles"=>
  [{"_id"=>"nm0000230",
    "character"=>"Rocky",
    "actorName"=>"Sly",
    "main"=>true},
   {"_id"=>"nm0084430",
    "character"=>"Challenger",
    "main"=>false,
    "actorName"=>"Erwin Biswanger"}]}
```

Another interesting feature is that the embedded object can provide a reference to its parent document.

```
> role.movie.title
 => "Rocky XXVI"
```

M:1 Embedded Linked (`MovieRole <-> Actor`)

Although not technically an official relationship type, it is a common pattern when linking documents within MongoDB to maintain a certain amount of state about the link on the source side. This is sometimes called an `annotated link`. There are many actors in a movie, but each of those actors play a specific role that is important and specific to the movie. Elsewhere in this description you found that `MovieRole` was embedded in `Movie` to describe the role played by an `Actor`. That `MovieRole` also contained cached information about the `Actor`. We will now look at the M:1 relationship between the embedded `MovieRole` and `Actor`.

As described – the one side has a primary key and typically no reference to the child within the document. That is true about the following parent document (`Actor`).

```
{:_id=>"nm0000354",
 :birthName=>"Matthew Paige Damon",
 :height=>{:amount=>5.8398952, :units=>"feet"},
 :name=>"Matt Damon",
 :date_of_birth=>1970-10-08 00:00:00 UTC,
  ...
 :place_of_birth=>
  {:_id=>"Boston, MA, USA",
   :geolocation=>{:type=>"Point", :coordinates=>[-71.0588801, 42.3600825]},
   :city=>"Boston",
   :county=>"Suffolk County",
   :state=>"MA",
   :country=>"US"}}
```

The many side will typically host the foreign key. That is true with `MovieRole`, but there is some complexity in hosting the foreign key in the embedded document.

```
{:_id=>"tt0440963",
 :title=>"The Bourne Ultimatum",
 ...
 :roles=>
  [{:actorName=>"Matt Damon",
    :character=>"Jason Bourne",
    :main=>true,
    ...
    :_id=>"nm0000354"},
  ...
}
```

We are able to add the `belongs_to` relation to `MovieRole` and map that to the `_id` field (instead of the default `actor_id`).

```
class MovieRole
  include Mongoid::Document
  field :character, type: String
  field :actorName, as: :actor_name, type: String
  field :main, type: Mongoid::Boolean
  embedded_in :movie
  ...
```

```
  belongs_to :actor, :foreign_key => :_id
  ...
end
```

However, we are unable to declare a `has_many` to an embedded class. We are forced to write some application logic and take a peek at queries. In the `roles` method below, we first search for all movies for the actor, iterate through those movie documents and implement a follow-on queuy to locate a specific role within the movie. In watching the database interactions through debug – it actually looks like all work is done within Mongoid to locate the result of the second query.

```
class Actor
  include Mongoid::Document
  field :name, type: String

  #not supported
  #has_many roles:, class_name: 'MovieRole'
  #replaced with
  def roles
    Movie.where(:"roles._id"=>self.id)
         .map {|m| m.roles.where(:_id=>self.id).first}
  end
end
```

To demonstrate the relationship, lets locate an actor that we know should have many roles.

```
> damon=Actor.where(:name=>{:$regex=>"Matt Da"}).first
```

We can search for a nested `roles._id` field matching the actor.id value. That gives us a list of the movies the actor has played in.

```
> movie=Movie.where(:"roles._id"=>damon.id).first
```

From there we can locate a specific role having the actor's ID and show that the role has the foreign key to the actor.

```
> role=movie.roles.where(:id=>damon.id).first
> pp role.attributes
{:actorName=>"Matt Damon",
 :character=>"Jason Bourne",
 :main=>true,
 ...
 :_id=>"nm0000354"}
```

We can bundle that query information and application logic into the following `roles` instance method to give us all roles.

```
  def roles
    Movie.where(:"roles._id"=>self.id)
         .map {|m| m.roles.where(:_id=>self.id).first}
  end
```

With all roles, we can go from the actor to each movie.role to locate each movie and associated character they have played.

```
> damon.roles.map {|role| "#{role.movie.title} => #{role.character}" }
 => ["The Bourne Ultimatum => Jason Bourne", "Good Will Hunting => Will Hunting",
     "The Martian => Mark Watney", "Saving Private Ryan => Private Ryan"]
```

In forming the relationship, lets add the actor to a fake role in a fake movie.

```
> damon=Actor.where(:name=>{:$regex=>"Matt Da"}).first
> rocky26=Movie.create(:_id=>"tt9000001", :title=>"Rocky XXVI")

> rocky=rocky26.roles.create(:_id=>damon.id, :character=>"Rocky", :actorName=>"Matt", :main=>true)
> pp Movie.collection.find(:title=>"Rocky XXVI").first
{:_id=>"tt9000001",
 :title=>"Rocky XXVI",
 :roles=>
  [{:_id=>"nm0000354", :character=>"Rocky", :actorName=>"Matt", :main=>true}]}
```

Now we can go in an search for roles that are part of their newly assignment movie and return the movie `title` and `character` they will be playing.

```
> damon.roles.select{|role| /Rocky/=~role.movie.title} \
             .map {|role| "#{role.movie.title} => #{role.character}" }
 => ["Rocky XXVI => Rocky"]
```

1:1 Linked (`Movie -> sequel_to:Movie`)

The 1:1 linked relationship has the same parent/child relationship as the 1:M, except that there is only a single child. In this example we create a recursive relationship between movies where each `sequel Movie` contains the foreign key to its parent `Movie` it is a `sequel_to`. The child class storing the foreign key declares `belongs_to` and the parent class defines `has_one` if the relationship is bi-directional. Since neither property matches the name `Movie`, they both have to declare a `class_name` to indicate what is at the other end of the relationship.

```
{:_id=>"tt9000000",
 :title=>"Rocky XXV",
 :roles=> ... }
{:_id=>"tt9000001",
 :title=>"Rocky XXVI",
 :roles=> ...
 :sequel_to_id=>"tt9000000"}

class Movie
  include Mongoid::Document
  field :title, type: String
  ...
  has_one :sequel, class_name:"Movie"
  belongs_to :sequel_to, class_name:"Movie"
  ...
end
```

Using the definition above we would automatically get a foreign key document key of `sequel_to_id`. You can map that to any other field name except `sequel_to` using `foreign_key: :other_field_name`. Even when defining the `foreign_key: :sequel_to`, Mongoid will use `:sequel_to_id`. If we truely want that name to be the key in the document we can work around it by changing the model class relation name to be different from the document key.

To demonstrate the relationship we will locate a `Movie` and its `sequel`. The parent movie is assigned to the `sequel_to` property of the child. This updates the in-memory state of the `sequel`.

```
> rocky25=Movie.where(:title=>"Rocky XXV").first
> rocky26=Movie.where(:title=>"Rocky XXVI").first
> rocky26.sequel_to=rocky25
> pp Movie.collection.find(:title=>"Rocky XXVI").first
```

```
{:_id=>"tt9000001",
 :title=>"Rocky XXVI",
 :roles=> ...
```

The in-memory state of the sequel is written to the database during the next save. You can see the foreign key when the document is printed.

```
> rocky26.save
 => true
> pp Movie.collection.find(:title=>"Rocky XXVI").first
{:_id=>"tt9000001",
 :title=>"Rocky XXVI",
 :roles=> ...
 :sequel_to_id=>"tt9000000"}
```

No state is written to the parent document.

```
> pp Movie.collection.find(:title=>"Rocky XXV").first
{:_id=>"tt9000000",
 :title=>"Rocky XXV",
 :roles=> ...  }
```

Note that one can navigate the relationship in both directions.

```
> rocky26.sequel_to.title
 => "Rocky XXV"
> rocky26.sequel_to.sequel.title
 => "Rocky XXVI"
```

M:M (`Movie <-> writers:Writer`)

In a many-to-many bi-directional relationship, each side has an important primary key and the other side stores that as a foreign key in an collection. In the example, The `Movie` has primary key (`tt0091763`) and a collection of foreign keys to its `Writers` (collection with just one - `["nm0000231"]`).

```
movie:
{:_id=>"tt0091763", :title=>"Platoon", :writer_ids=>["nm0000231"]}
```

The `Writer` has a primary key (`nm0000231`) that matches one of the elements in the related movie. It too has a collection of foreign keys. The collection of foreign keys in the `Writer` are foreign keys to the `Movie` documents (`[..., "tt0091763"]`)

```
writer:
{:_id=>"nm0000231",
 :name=>"Oliver Stone",
 :movie_ids=>["tt0086250", "tt0091763"]}
```

Both sides of the relationship declare a `has_and_belongs_to_many` with the name of the relationship. The long word has two meanings; `has_many` and `belongs_to`. The `belongs_to` aspect stores the foreign key to the parent document and the `has_many` means it is the parent of many child documents. When we bring the two concepts together – the `has_many` and `belongs_to` are both feed off the local collection of document foreign keys.

```
class Writer
  include Mongoid::Document
  field :name, type: String
  embeds_one :hometown, as: :locatable, class_name: 'Place'
  ...
  has_and_belongs_to_many :movies
  ...
end

class Movie
  include Mongoid::Document
  field :title, type: String
  ...
  has_and_belongs_to_many :writers
  ...
end
```

To look at our bi-directional, many-to-many relationship, we can get the movie properties by navigating a relationship from the writer.

```
> stone=Writer.where(:name=>{:$regex=>"Stone"}).first
> stone.movies.map {|m| m.title}
 => ["Scarface", "Platoon"]
```

We can also go the other way. In this example we add a `hometown` property to the writer and get the hometown through a reference from the movie.

```
> stone=Writer.where(:name=>{:$regex=>"Stone"}).first
> nyc=Place.where(:_id=>{:$regex=>"^New York, NY"}).first
> stone.create_hometown(nyc.attributes)
> platoon=Movie.where(:title=>"Platoon").first
> platoon.writers.first.hometown.id
 => "New York, NY, USA"
```

To demonstrate creating a many-to-many relationship, lets create a new movie which we can verify has no writers at this point.

```
> rocky26=Movie.create(:_id=>"tt9000001", :title=>"Rocky XXVI")
> pp Movie.collection.find(:_id=>rocky26.id).first
{:_id=>"tt9000001", :title=>"Rocky XXVI"}
```

Lets go out and find a writer and notice the writer has already written two movies.

```
> stone=Writer.where(:name=>{:$regex=>"Stone"}).first
> pp Writer.collection.find(:_id=>stone.id).first
{:_id=>"nm0000231",
 :name=>"Oliver Stone",
 :movie_ids=>["tt0086250", "tt0091763"]}
```

Add the writer to the collection of movie writers.

```
> rocky26.writers << stone
 => [#<Writer _id: nm0000231, name: "Oliver Stone", movie_ids: ["tt0086250", "tt0091763", "tt9000001"]>]
```

Notice that Mongoid has updated both ends of the collection with the foreign key of the other.

```
> pp Movie.collection.find(:_id=>rocky26.id).first
{:_id=>"tt9000001", :title=>"Rocky XXVI", :writer_ids=>["nm0000231"]}
 => {:_id=>"tt9000001", :title=>"Rocky XXVI", :writer_ids=>["nm0000231"]}

> pp Writer.collection.find(:_id=>stone.id).first
{:_id=>"nm0000231",
 :name=>"Oliver Stone",
 :movie_ids=>["tt0086250", "tt0091763", "tt9000001"]}
 => {:_id=>"nm0000231", :name=>"Oliver Stone", :movie_ids=>["tt0086250", "tt0091763", "tt9000001"]}
```

Great! We can use both references to locate information about the other simply by walking the relationship.

```
> stone.movies.map{|m| m.title}
 => ["Scarface", "Platoon", "Rocky XXVI"]

> rocky26.writers.map{|w| w.name}
 => ["Oliver Stone"]
```

Notice that when the relationship is removed from one end – it is also removed from the other end.

```
> stone.movies.delete rocky26
> pp Writer.collection.find(:_id=>stone.id).first
{:_id=>"nm0000231",
 :name=>"Oliver Stone",
 :movie_ids=>["tt0086250", "tt0091763"]}
 => {:_id=>"nm0000231", :name=>"Oliver Stone", :movie_ids=>["tt0086250", "tt0091763"]}

> pp Movie.collection.find(:_id=>rocky26.id).first
{:_id=>"tt9000001", :title=>"Rocky XXVI", :writer_ids=>[]}
 => {:_id=>"tt9000001", :title=>"Rocky XXVI", :writer_ids=>[]}
```

Timestamps

Timestamp information is not added by default in Mongoid – as it is within ActiveRecord. To add `created_at` and `updated_at` fields and management, we add an additional `Mongoid::Timestamps` mixin.

```
class Movie
  include Mongoid::Document
  include Mongoid::Timestamps

class Writer
  include Mongoid::Document
  include Mongoid::Timestamps
```

Lets demonstrate the timestamp capability by re-doing the M:M relationship scenario and create our movie again with timestamps in place. Notice that two new elements have been introduced to the document for `created_at` and `updated_at`. We can also get these explicitly from the object.

```
> rocky26=Movie.create(:_id=>"tt9000001", :title=>"Rocky XXVI")
> pp Movie.collection.find(:_id=>rocky26.id)
{:_id=>"tt9000001",
 :title=>"Rocky XXVI",
 :updated_at=>2015-11-28 20:47:59 UTC,
 :created_at=>2015-11-28 20:47:59 UTC}
> rocky26.created_at
 => Sat, 28 Nov 2015 20:47:59 UTC +00:00
> rocky26.updated_at
 => Sat, 28 Nov 2015 20:47:59 UTC +00:00
```

updated_at will be modified as the document is changed and saved.

```
> rocky26.year=2015
 => 2015
> rocky26.save
 => true
> rocky26.updated_at
 => Sat, 28 Nov 2015 20:48:28 UTC +00:00
```

The writer was imported prior to timestamps being configured and does not have a created_at and updated_at to start with.

```
> stone=Writer.where(:name=>{:$regex=>"Stone"}).first
> pp Writer.collection.find(:_id=>stone.id).first
{:_id=>"nm0000231",
 :name=>"Oliver Stone",
 :movie_ids=>["tt0086250", "tt0091763"]}
```

If we add the writer to the movie, the writer receives an updated_at timestamp.

```
> rocky26.writers << stone
> pp Writer.collection.find(:_id=>stone.id).first
{:_id=>"nm0000231",
 :name=>"Oliver Stone",
 :movie_ids=>["tt0086250", "tt0091763", "tt9000001"],
 :updated_at=>2015-11-28 20:53:55 UTC}
```

The odd thing found was that the source of the collection we navigated from does not get an updated_at change – even when issuing save. This was found to be true no matter which end we started from.

```
> rocky26.updated_at
 => Sat, 28 Nov 2015 20:48:28 UTC +00:00

> pp Movie.collection.find(:_id=>rocky26.id).first
{:_id=>"tt9000001",
 :title=>"Rocky XXVI",
 :updated_at=>2015-11-28 20:48:28 UTC,
 :created_at=>2015-11-28 20:47:59 UTC,
 :year=>2015,
 :writer_ids=>["nm0000231"]}
```

The object, can of course, be manually updated with a touch command.

```
> rocky26.touch
> rocky26.updated_at
 => Sat, 28 Nov 2015 21:00:05 UTC +00:00
```

If we introduce a touch propagation property to the relationship we can get the touch command to cascade across the relationship. However, it appears that type of property is only available when the source of the touch has a single foreign key stored in its document. To demonstrate, we added the mixin to Actor and add the touch property to MovieRole.

```
class Actor
  include Mongoid::Document
  include Mongoid::Timestamps
  field :name, type: String
  ...
end
```

```
class MovieRole
  include Mongoid::Document
  field :character, type: String
  embedded_in :movie
  ...
  belongs_to :actor, foreign_key: :_id, touch: true
  ...
end
```

Our scenario starts off with an actor with no concept of when they were created or last updated. When we add a relationship to them from MovieRole (annotated with `touch: true`), the `updated_at` time of the actor is automatically updated.

```
rocky26=Movie.create(:_id=>"tt9000001", :title=>"Rocky XXVI")
> damon=Actor.where(:name=>{:$regex=>"Matt Da"}).first
> damon.created_at
 => nil
> damon.updated_at
 => nil
> rocky=rocky26.roles.create(:actor=>damon, :character=>"Rocky", :actorName=>"Matt", :main=>true)
> damon.updated_at
 => Sat, 28 Nov 2015 21:18:38 UTC +00:00
```

Constraints and Validation

Field Validation

ActiveModel validations can be added to Mongoid model classes. In the following example we declare that `name` must be present in the document.

```
class Director
  include Mongoid::Document
  field :name, type: String
  ...
  validates_presence_of :name
end
```

To test our validation, we can attempt to create a new `Director` without a name.

```
> director=Director.create(:_id=>"12345")
 => #<Director _id: 12345, name: nil, residence_id: nil>
```

Everything was quiet, but when we check for errors, we see that name cannot be blank.

```
> director.errors.count
 => 1
> director.errors
 => #<ActiveModel::Errors:0x000000088ad830 @base=#<Director _id: 12345, name: nil,
    residence_id: nil>, @messages={:name=>["can't be blank"]}>
```

If we want an exception thrown instead, we add the bang ("!") character to the `create()` method to have Mongoid immediately throw an exception when encountering a validation error.

```
> director=Director.create!(:_id=>"12345")
Mongoid::Errors::Validations:
message:
  Validation of Director failed.
summary:
  The following errors were found: Name can't be blank
resolution:
  Try persisting the document with valid data or remove the validations.
```

Relationship Constraints

You can add dependency constraints to parents to inform Mongoid what to do with a child document when the parent the child is referencing is removed from the database. There are five(5) options when you include the default case. Cardinality of relationship does make a difference in some of the behavior.

- (default): Orphans the child document
  - 1:1 and 1:M leaves the child with stale reference to the removed parent
  - M:M clears the child of the parent reference (acts like `:nullify`)
- nullify: Orphan the child document after setting the child foreign key to nil
- destroy: Remove the child document after running model callbacks on the child
- delete: Remove the child document without running model callbacks on the child
  - M:M does not remove the child document from database (acts like `:nullify`)
- restrict: Raise an error if a child references the parent being removed

The Mongoid documentation states `:nullify` is the default, but we have found a key difference in how `default` and `:nullify` leave the child orphaned for the non-M:M case as well as some of the activity performed with the database. For 1:1 annd 1:M cardinality relationships, the `default` case leaves the orphaned child untouched, with a stale foreign key reference to the removed parent. `:nullify` clears the foreign key from the child. Both will result in a `nil` parent returned when accessed, but there are differences in the database.

`:restrict` will throw an error in all cases if a child still exists with a foreign key reference to the parent being removed.

`:destroy` and `:delete` are functionally the same for 1:1 and 1:M relationships – in that they remove the children when the parent gets deleted and will run/not-run callbacks on the children at that time depending on the whether `:delete` or `:destroy` is selected. We have found the M:M cardinality case to be different in this area. M:M `:delete` acts like `:nullify` and does not remove the child. You must use `:destroy` if you wish the child of an M:M relationship to be removed.

Demonstration Lets demonstrate with the `Movie` `sequel/sequel_to` 1:1 relationship (mapped using a custom FK `sequel_of`) and the `Movie`/`Writer` M:M relationship.

```
class Movie
  has_and_belongs_to_many :writers
  has_one :sequel, foreign_key: :sequel_of, class_name:"Movie"
  belongs_to :sequel_to, foreign_key: :sequel_of, class_name:"Movie"

class Writer
  has_and_belongs_to_many :movies
```

Lets follow the same script in each case. We are re-finding the parent movie so that we can start the relationship from scratch at the point of the removal.

```
reload!
rocky30=Movie.create(:title=>"Rocky 30")
rocky31=Movie.create(:title=>"Rocky 31", :sequel_to=>rocky30)
writer=rocky30.writers.create(:name=>"A Writer")
rocky30.id
Movie.where(:id=>rocky30.id).first.writer_ids
Movie.where(:id=>rocky31.id).first.sequel_of
Writer.where(:id=>writer.id).first.movie_ids
rocky30=Movie.find(rocky30.id)
rocky30.destroy
Movie.where(:id=>rocky30.id).exists?
Movie.where(:id=>rocky30.id).first.writer_ids if Movie.where(:id=>rocky30.id).exists?
Movie.where(:id=>rocky31.id).exists?
Movie.where(:id=>rocky31.id).first.sequel_of  if Movie.where(:id=>rocky31.id).exists?
Writer.where(:id=>writer.id).exists?
Writer.where(:id=>writer.id).first.movie_ids if Writer.where(:id=>writer.id).exists?
```

To clean-up, we can follow-up with

```
p Movie.where(:title=>{:$regex=>"Rocky 3[0-1]"}).delete_all; \
Writer.where(:name=>{:$regex=>"A Writer"}).delete_all
```

Callbacks To add some clarity to relationship handling and the `delete` versus `destroy` case, we add some callbacks to the `Movie` and `Writer` model classes to be notified when the document is being removed from the database.

```
  before_destroy do |doc|
    puts "before_destroy Movie callback for #{doc.id}, "\
         "sequel_to=#{doc.sequel_to}, writers=#{doc.writer_ids}"
  end
  after_destroy do |doc|
    puts "after_destroy Movie callback for #{doc.id}, "\
         "sequel_to=#{doc.sequel_to}, writers=#{doc.writer_ids}"
  end

  before_destroy do |doc|
    puts "before_destroy Writer callback for #{doc.id}, "\
        "movies=#{doc.movie_ids}"
  end
  after_destroy do |doc|
    puts "after_destroy Writer callback for #{doc.id}, "\
        "movies=#{doc.movie_ids}"
  end
```

Mongoid document states callbacks should be reserved for cross-cutting (e.g., send messages) behavior and and not business behavior. The documentation also states that the amount of callbacks should be limited within an object tree for performance reasons.

Note the callbacks are called when `destroy` is called on the object.

```
> Movie.new.destroy
before_destroy Movie callback for 5680088fe301d06376000006, sequel_to=, writers=[]
D, | {"delete"=>"movies", "deletes"=>[{"q"=>{"_id"=>BSON::ObjectId('5680088fe301d06376000006')}...
after_destroy Movie callback for 5680088fe301d06376000006, sequel_to=, writers=[]
 => true

> Writer.new.destroy
```

```
before_destroy Writer callback for 56800908e301d06376000008, movies=[]
D, | {"delete"=>"writers", "deletes"=>[{"q"=>{"_id"=>BSON::ObjectId('56800908e301d06376000008')}...
after_destroy Writer callback for 56800908e301d06376000008, movies=[]
 => true
```

But not called when only `delete` is called.

```
> Movie.new.delete
D, | {"delete"=>"movies", "deletes"=>[{"q"=>{"_id"=>BSON::ObjectId('56800896e301d06376000007')}...
 => true

> Writer.new.delete
D, | {"delete"=>"writers", "deletes"=>[{"q"=>{"_id"=>BSON::ObjectId('5680090ce301d06376000009')}...
 => true
```

Relationship Constraints: `dependent: :destroy` To immediately demonstrate callbacks, lets look at the `:destroy` case.

```
  has_and_belongs_to_many :writers, dependent: :destroy
  has_one :sequel, foreign_key: :sequel_of, class_name:"Movie", dependent: :destroy
```

We start in our standard state.

```
> rocky30.id
 => BSON::ObjectId('56801498e301d06376000016')
> Movie.where(:id=>rocky30.id).first.writer_ids
 => [BSON::ObjectId('56801498e301d06376000018')]
> Movie.where(:id=>rocky31.id).first.sequel_of
 => BSON::ObjectId('56801498e301d06376000016')
> Writer.where(:id=>writer.id).first.movie_ids
 => [BSON::ObjectId('56801498e301d06376000016')]
> rocky30=Movie.find(rocky30.id)
> rocky30.destroy
```

The callbacks start with the parent object being called back.

```
before_destroy Movie callback for 56801498e301d06376000016,
   sequel_to=, writers=[BSON::ObjectId('56801498e301d06376000018')]
```

`Writer`s are retrieved by primary key (because the M:M relation stores the PK on the parent-side). The writer(s) are retrieved so that `destroy` callbacks could be issued and provided the document state during the callback. Each writer is removed from the database and the writer is also removed from the parent movies having that writer (including the one being removed in this case).

```
D, | {"find"=>"writers", "filter"=>
    {"$and"=>[{"_id"=>{"$in"=>[BSON::ObjectId('56801498e301d06376000018')]}}]}}
before_destroy Writer callback for 56801498e301d06376000018,
    movies=[BSON::ObjectId('56801498e301d06376000016')]
D, | {"delete"=>"writers", "deletes"=>[{"q"=>{"_id"=>BSON::ObjectId('56801498e301d06376000018')}, ...
D, | {"update"=>"movies", "updates"=>[{
    "q"=>{"$and"=>[{"_id"=>{"$in"=>[BSON::ObjectId('56801498e301d06376000016')]}}]},
    "u"=>{"$pull"=>{"writer_ids"=>BSON::ObjectId('56801498e301d06376000018')}}...
after_destroy Writer callback for 56801498e301d06376000018,
    movies=[BSON::ObjectId('56801498e301d06376000016')]
```

Sequel Movies are located by foreign key (sequel_of), retrieved, and callbacks invoked. There is an extra call to the database in this case to locate the sequel of a sequel.

```
D, | {"find"=>"movies", "filter"=>{"sequel_of"=>BSON::ObjectId('56801498e301d06376000016')}}
before_destroy Movie callback for 56801498e301d06376000017,
    sequel_to=#<Movie:0x00000005349130>, writers=[]
D, | {"find"=>"movies", "filter"=>{"sequel_of"=>BSON::ObjectId('56801498e301d06376000017')}}
D, | {"delete"=>"movies", "deletes"=>[{"q"=>{"_id"=>BSON::ObjectId('56801498e301d06376000017')},...
after_destroy Movie callback for 56801498e301d06376000017,
    sequel_to=#<Movie:0x00000005349130>, writers=[]
```

At this point, the parent Movie is removed and any writer still having a reference to this movie has their foreign key removed. This seems unnecessary, but remember this is all done outside of a transaction – so redundant calls may be useful.

```
D, | {"delete"=>"movies", "deletes"=>[{"q"=>{"_id"=>BSON::ObjectId('56801498e301d06376000016')},...
D, | {"update"=>"writers", "updates"=>[{
    "q"=>{"$and"=>[{"_id"=>{"$in"=>[BSON::ObjectId('56801498e301d06376000018')]}}]},
    "u"=>{"$pull"=>{"movie_ids"=>BSON::ObjectId('56801498e301d06376000016')}},...
after_destroy Movie callback for 56801498e301d06376000016,
    sequel_to=, writers=[BSON::ObjectId('56801498e301d06376000018')]
 => true
```

The parent and all children have been removed for the dependent: :destroy case.

```
> Movie.where(:id=>rocky30.id).exists?
 => false
> Movie.where(:id=>rocky30.id).first.writer_ids if Movie.where(:id=>rocky30.id).exists?
 => nil
> Movie.where(:id=>rocky31.id).exists?
 => false
> Movie.where(:id=>rocky31.id).first.sequel_of  if Movie.where(:id=>rocky31.id).exists?
 => nil
> Writer.where(:id=>writer.id).exists?
 => false
> Writer.where(:id=>writer.id).first.movie_ids if Writer.where(:id=>writer.id).exists?
 => nil
```

Relationship Constraints: dependent: :delete To further demonstrate callbacks, lets look at the :delete case. The two cases have primarily the same end functionality except that :delete does not invoke callbacks on the children. However, we do see a difference in how :delete and :destroy work for the M:M case.

```
  has_and_belongs_to_many :writers, dependent: :delete
  has_one :sequel, foreign_key: :sequel_of, class_name:"Movie", dependent: :delete
```

We start in our standard state. The destroy callbacks are still invoked on the parent because we have called destroy() on the parent. The impact of our change should be restricted to what occurs for the child.

```
> rocky30.id
 => BSON::ObjectId('56801c88e301d06376000019')
> Movie.where(:id=>rocky30.id).first.writer_ids
 => [BSON::ObjectId('56801c88e301d0637600001b')]
> Movie.where(:id=>rocky31.id).first.sequel_of
 => BSON::ObjectId('56801c88e301d06376000019')
> Writer.where(:id=>writer.id).first.movie_ids
```

```
 => [BSON::ObjectId('56801c88e301d06376000019')]
> rocky30=Movie.find(rocky30.id)
> rocky30.destroy
before_destroy Movie callback for 56801c88e301d06376000019,
    sequel_to=, writers=[BSON::ObjectId('56801c88e301d0637600001b')]
```

Even though callbacks are not being called for the `Writers`, each `Writer` to accessed by primary key using the values on the `Movie`-side.

```
D, | {"find"=>"writers", "filter"=>{"$and"=>[{
    "_id"=>{"$in"=>[BSON::ObjectId('56801c88e301d0637600001b')]}}]}}
```

Even though we are eventually removing the `Movie` and looking to remove dependents, it appears that the relationship is severed first. Each `Writer` is updated to have the `Movie` ID removed from its foreign key list. The `Movie` has its relationships cleared.

```
D, | {"update"=>"writers", "updates"=>[{
    "q"=>{"$and"=>[{"_id"=>{"$in"=>[BSON::ObjectId('56801c88e301d0637600001b')]}}]},
    "u"=>{"$pull"=>{"movie_ids"=>BSON::ObjectId('56801c88e301d06376000019')}}...
D, | {"update"=>"movies", "updates"=>[{"q"=>{"_id"=>BSON::ObjectId('56801c88e301d06376000019')},
    "u"=>{"$set"=>{"writer_ids"=>[]}}...
```

Sequel processing causes each child `Movie` to be located by a foreign key to the parent being removed. An extra set of `Writer` and `Movie` sequel processing occurs for the child `Movie` (since it is also a `Movie`). The child movie is finally removed after all cleanup has been done for that type.

```
D, | {"find"=>"movies", "filter"=>{"sequel_of"=>BSON::ObjectId('56801c88e301d06376000019')}}
D, | {"update"=>"writers", "updates"=>[{"q"=>{"$and"=>[{"_id"=>{"$in"=>[]}}]},
    "u"=>{"$pull"=>{"movie_ids"=>BSON::ObjectId('56801c88e301d0637600001a')}},...
D, | {"update"=>"movies", "updates"=>[{"q"=>{"_id"=>BSON::ObjectId('56801c88e301d0637600001a')},
    "u"=>{"$set"=>{"writer_ids"=>[]}},...
D, | {"find"=>"movies", "filter"=>{"sequel_of"=>BSON::ObjectId('56801c88e301d0637600001a')}}
D, | {"delete"=>"movies", "deletes"=>[{"q"=>{"_id"=>BSON::ObjectId('56801c88e301d0637600001a')}...
```

At this point the parent `Movie` is removed.

```
D, | {"delete"=>"movies", "deletes"=>[{"q"=>{"_id"=>BSON::ObjectId('56801c88e301d06376000019')}...
after_destroy Movie callback for 56801c88e301d06376000019,
    sequel_to=, writers=[]
 => true
```

However – one curious things is that the `Writer` was never removed. This is different behavior than the other relationship cardinalities and should be noted.

```
> Movie.where(:id=>rocky30.id).exists?
 => false
> Movie.where(:id=>rocky30.id).first.writer_ids if Movie.where(:id=>rocky30.id).exists?
 => nil
> Movie.where(:id=>rocky31.id).exists?
 => false
> Movie.where(:id=>rocky31.id).first.sequel_of  if Movie.where(:id=>rocky31.id).exists?
 => nil
> Writer.where(:id=>writer.id).exists?
 => true
> Writer.where(:id=>writer.id).first.movie_ids if Writer.where(:id=>writer.id).exists?
 => []
```

Relationship Constraints: `dependent: :nullify` To leave the children in place when removing the parent, we can switch the constraint to `:nullify`.

```
  has_and_belongs_to_many :writers, dependent: :nullify
  has_one :sequel, foreign_key: :sequel_of, class_name:"Movie", dependent: :nullify
```

We start in our standard state.

```
> rocky30.id
 => BSON::ObjectId('568022c9e301d0637600001c')
> Movie.where(:id=>rocky30.id).first.writer_ids
 => [BSON::ObjectId('568022c9e301d0637600001e')]
> Movie.where(:id=>rocky31.id).first.sequel_of
 => BSON::ObjectId('568022c9e301d0637600001c')
> Writer.where(:id=>writer.id).first.movie_ids
 => [BSON::ObjectId('568022c9e301d0637600001c')]
> rocky30=Movie.find(rocky30.id)
> rocky30.destroy
before_destroy Movie callback for 568022c9e301d0637600001c,
    sequel_to=, writers=[BSON::ObjectId('568022c9e301d0637600001e')]
```

The `Writers` are retrieved for some reason and followed up by the parent `Movie` being removed from `Writer` foreign key references. `Writer` references in the parent `Movie` to be removed are cleared as well. You will find that this M:M behavior is the same as in the `:delete` case.

```
D, | {"find"=>"writers", "filter"=>{"$and"=>[{
    "_id"=>{"$in"=>[BSON::ObjectId('568022c9e301d0637600001e')]}}]}}
D, | {"update"=>"writers", "updates"=>[{
    "q"=>{"$and"=>[{"_id"=>{"$in"=>[BSON::ObjectId('568022c9e301d0637600001e')]}}]},
    "u"=>{"$pull"=>{"movie_ids"=>BSON::ObjectId('568022c9e301d0637600001c')}}...
D, | {"update"=>"movies", "updates"=>[{"q"=>{"_id"=>BSON::ObjectId('568022c9e301d0637600001c')},
    "u"=>{"$set"=>{"writer_ids"=>[]}},...
```

Sequel `Movies` are retrieved for some reason and then the database is cleared of the foreign key to the parent. The extra query is there to locate the sequel of the sequel.

```
D, | {"find"=>"movies", "filter"=>{"sequel_of"=>BSON::ObjectId('568022c9e301d0637600001c')}}
D, | {"find"=>"movies", "filter"=>{"sequel_of"=>BSON::ObjectId('568022c9e301d0637600001d')}}
D, | {"update"=>"movies", "updates"=>[{"q"=>{"_id"=>BSON::ObjectId('568022c9e301d0637600001d')},
    "u"=>{"$set"=>{"sequel_of"=>nil, "updated_at"=>2015-12-27 17:41:29 UTC}}...
```

At this point the parent movie is removed from the database.

```
D, | {"delete"=>"movies", "deletes"=>[{"q"=>{"_id"=>BSON::ObjectId('568022c9e301d0637600001c')}...
after_destroy Movie callback for 568022c9e301d0637600001c,
    sequel_to=, writers=[]
 => true
```

None of the child documents are removed. They are orphaned with nil references to a parent.

```
> Movie.where(:id=>rocky30.id).exists?
 => false
> Movie.where(:id=>rocky30.id).first.writer_ids if Movie.where(:id=>rocky30.id).exists?
 => nil
> Movie.where(:id=>rocky31.id).exists?
```

```
 => true
> Movie.where(:id=>rocky31.id).first.sequel_of  if Movie.where(:id=>rocky31.id).exists?
 => nil
> Writer.where(:id=>writer.id).exists?
 => true
> Writer.where(:id=>writer.id).first.movie_ids if Writer.where(:id=>writer.id).exists?
 => []
```

Relationship Constraints: `dependent: (default)` To follow-up on the `default` case and how it is not exactly the same as the `:nullify` case in the database, we change to the following:

```
  has_and_belongs_to_many :writers
  has_one :sequel, foreign_key: :sequel_of, class_name:"Movie"
```

We start in our standard state.

```
> rocky30.id
 => BSON::ObjectId('56802765e301d0637600001f')
> Movie.where(:id=>rocky30.id).first.writer_ids
 => [BSON::ObjectId('56802765e301d06376000021')]
> Movie.where(:id=>rocky31.id).first.sequel_of
 => BSON::ObjectId('56802765e301d0637600001f')
> Writer.where(:id=>writer.id).first.movie_ids
 => [BSON::ObjectId('56802765e301d0637600001f')]
> rocky30=Movie.find(rocky30.id)
> rocky30.destroy
before_destroy Movie callback for 56802765e301d0637600001f,
    sequel_to=, writers=[BSON::ObjectId('56802765e301d06376000021')]
```

Note the amount and type of callbacks is quite a bit different from the `:nullify` case. The parent `Movie` is removed from the database prior to anything being queried. This is then followed up with the relationship being removed from the M:M `Writer` relationship but not the `Movie` sequel relationship.

```
D, {"delete"=>"movies", "deletes"=>[{"q"=>{"_id"=>BSON::ObjectId('56802765e301d0637600001f')},...
D, {"update"=>"writers", "updates"=>[{
    "q"=>{"$and"=>[{"_id"=>{"$in"=>[BSON::ObjectId('56802765e301d06376000021')]}}]},
    "u"=>{"$pull"=>{"movie_ids"=>BSON::ObjectId('56802765e301d0637600001f')}},...
after_destroy Movie callback for 56802765e301d0637600001f,
    sequel_to=, writers=[BSON::ObjectId('56802765e301d06376000021')]
 => true
```

The `Movie` sequel and `Writer` still exists and the `Writer` will end up in the `:nullify` state except thru much less work. The `Writer` ends up in an orphan state, like `:nullify` except that it is left referencing the parent that no longer exists.

```
2.2.2 :151 > Movie.where(:id=>rocky30.id).exists?
 => false
2.2.2 :152 > Movie.where(:id=>rocky30.id).first.writer_ids if Movie.where(:id=>rocky30.id).exists?
 => nil
2.2.2 :153 > Movie.where(:id=>rocky31.id).exists?
 => true
2.2.2 :154 > Movie.where(:id=>rocky31.id).first.sequel_of  if Movie.where(:id=>rocky31.id).exists?
 => BSON::ObjectId('56802765e301d0637600001f')
2.2.2 :155 > Writer.where(:id=>writer.id).exists?
 => true
2.2.2 :156 > Writer.where(:id=>writer.id).first.movie_ids if Writer.where(:id=>writer.id).exists?
 => []
```

Relationship Constraints: `dependent: :restrict` The restrict case is a bit different than the others in that it will stop the removal from occuring if there is an existing child reference to the parent being removed.

```
  has_and_belongs_to_many :writers, dependent: :restrict
  has_one :sequel, foreign_key: :sequel_of, class_name:"Movie", dependent: :restrict
```

We start in our standard state.

```
> rocky30.id
 => BSON::ObjectId('568029ede301d06376000022')
> Movie.where(:id=>rocky30.id).first.writer_ids
 => [BSON::ObjectId('568029ede301d06376000024')]
> Movie.where(:id=>rocky31.id).first.sequel_of
 => BSON::ObjectId('568029ede301d06376000022')
> Writer.where(:id=>writer.id).first.movie_ids
 => [BSON::ObjectId('568029ede301d06376000022')]
> rocky30=Movie.find(rocky30.id)
> rocky30.destroy
before_destroy Movie callback for 568029ede301d06376000022,
    sequel_to=, writers=[BSON::ObjectId('568029ede301d06376000024')]
```

However, in this case the removal goes nowhere because a child reference is found. In this case processing was stopped when the first child reference was found. The same error would result on the other relationship if Mongoid needed to go that far in checking relationships.

```
D, {"count"=>"writers", "query"=>{"$and"=>[{
   "_id"=>{"$in"=>[BSON::ObjectId('568029ede301d06376000024')]}}]}}
Mongoid::Errors::DeleteRestriction:
message:
  Cannot delete Movie because of dependent 'writers'.
summary:
  When defining 'writers' with a :dependent => :restrict, Mongoid will
  raise an error when attempting to delete the Movie when the child
  'writers' still has documents in it.
resolution:
  Don't attempt to delete the parent Movie when it has children, or
  change the dependent option on the relation.
```

Notice how all objects and relationships stay in-place.

```
> Movie.where(:id=>rocky30.id).exists?
 => true
> Movie.where(:id=>rocky30.id).first.writer_ids if Movie.where(:id=>rocky30.id).exists?
 => [BSON::ObjectId('568029ede301d06376000024')]
> Movie.where(:id=>rocky31.id).exists?
 => true
> Movie.where(:id=>rocky31.id).first.sequel_of  if Movie.where(:id=>rocky31.id).exists?
 => BSON::ObjectId('568029ede301d06376000022')
> Writer.where(:id=>writer.id).exists?
 => true
> Writer.where(:id=>writer.id).first.movie_ids if Writer.where(:id=>writer.id).exists?
 => [BSON::ObjectId('568029ede301d06376000022')]
```

Queries

Geolocation Query

Geolocation queries are synatically straight forward in Mongoid and every model class has the built-in ability to express and execute a geolocation query. The is nothing specifically required except for a set of geolocation coordinates in one of the fields and an index.

```
class Actor
  include Mongoid::Document
  embeds_one :place_of_birth, class_name: 'Place' , as: :locatable
  ...
  index ({ :"place_of_birth.geolocation" => Mongo::Index::GEO2DSPHERE })
```

We can grab a place from the `places` collection and directly express a query on the Actors class.

```
> silver_spring=Place.where(:city=>"Silver Spring", :state=>"MD").first

> Actor.near(:"place_of_birth.geolocation"=>silver_spring.geolocation)
       .limit(5).each {|actor| pp "#{actor.name}, pob=#{actor.place_of_birth.id}"}

DEBUG | {"find"=>"actors", "filter"=>{"place_of_birth.geolocation"=>{
  "$near"=>{:type=>"Point", :coordinates=>[-77.026088, 38.99066570000001]}}},
  "limit"=>5}

"Lewis Black, pob=Silver Spring, MD, USA"
"Jeffrey Wright, pob=Washington, DC, USA"
"Samuel L. Jackson, pob=Washington, DC, USA"
"Laura Cayouette, pob=Laurel, MD, USA"
"Mark Rolston, pob=Baltimore, MD, USA
```

We can add a custom method to the Actor class to make this slightly easier to call.

```
  def self.near_pob place, max_meters
    near(:"place_of_birth.geolocation" => place.geolocation)
    .max_distance(:"place_of_birth.geolocation" =>max_meters)
  end
end

> Actor.near_pob(silver_spring, 10*1609.34)
       .each {|actor| pp "#{actor.name}, pob=#{actor.place_of_birth.id}"}
"Lewis Black, pob=Silver Spring, MD, USA"
"Jeffrey Wright, pob=Washington, DC, USA"
"Samuel L. Jackson, pob=Washington, DC, USA"
```

Scaffolding

Assembly

The following describes some details of how the movies application example was assembled.

Create Application

1. Create a new Rails application.

```
$ rails new movies
$ cd movies
```

2. Add the `mongoid` gem to the `Gemfile` and run `bundle`.

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

```
$ bundle
```

3. Configure application with `Mongoid` by generating a `config/mongoid.yml` file and loading that from within `config/application.rb`.

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

```
module Movies
  class Application < Rails::Application
    ...
    #bootstraps mongoid within applications -- like rails console
    Mongoid.load!('./config/mongoid.yml')
    ...
    #mongoid gem configures mongoid as default model generator
    #this can make it explicit or switch back to active_record default
    config.generators {|g| g.orm :mongoid}
    #config.generators {|g| g.orm :active_record}
```

4. Start Rails server.

```
$ rails s
```

Custom Classes Assembly

Create custom classes that primarily represent anonymous blocks of information within the documents. Mongoid does not require that any information be mapped, but mapping it extends your ability to query the information and conveniently transform it if necessary (otherwise just leave it alone). In the case of this application there are two demonstration custom classes; `Measurement` and `Point`.

- Measurement
- Point

Each of the classes typically had an `initialize()` method that took specific arguments and not general purpose for different sources. The `to_s` is useful to output a formatted string of what the object represents.

- `initialize` - normalized form – independent of source formats
- `to_s` - useful in producing formatted output

Mongoid requires one instance method called `mongoize` to to create a hash, array, etc. of data that is in database form.

- `mongoize` - creates a DB-form of the instance

Mongoid requires the class provide three (3) class methods to be able to take an external object form and return either the database or object instance form. There is typically a lot of case statements checking for type and other signatures to determine how to get it in the required format.

- `self.demongoize(object)` - creates an instance of the class from the DB-form of the data
- `self.mongoize(object)` - takes in all forms of the object and produces a DB-friendly form
- `self.evolve(object)` - used by criteria to convert object to DB-friendly form

Create Scaffold

The examples in this section assume that `mongoid` is the default model generator. To be explicit at command time, add the `--orm mongoid` option to the command line.

Place Assembly

`Place` models a point and its descriptive address information.

1. Generate the core of the model class using the `rails generate` command.

   ```
   $ rails g model Place formatted_address geolocation:Point street_number \
       street_name city postal_code county state country
   ```

   This generates the following model class.

   ```
   class Place
     include Mongoid::Document
     field :formatted_address, type: String
     field :geolocation, type: Point
     field :street_number, type: String
     field :street_name, type: String
     field :city, type: String
     field :postal_code, type: String
     field :county, type: String
     field :state, type: String
     field :country, type: String
   end
   ```

2. Override the default field mapping for `_id` to use the `formatted_address`.

   ```
     field :_id, type: String, default: -> { formatted_address }
   ```

3. Some instances of this class will be embedded in other documents – but not always documents of the exact same model class type. Therefore we still declare the `embedded_in` property but state that it is embedded inside any model class that declares an `embeds` propert defined as type `locatable`.

   ```
     embedded_in :locatable, polymorphic: true
   ```

4. There is one more thing we need to add to classes that take custom types. If you intend to modify this object using the UI or external classes that want to deal mostly with strings, you should beef up your `demongoize` method to take additional forms that can convert the representation to instance form.

   ```
     def sanitize_for_mass_assignment(params)
       params ||= {}
       params.each_pair do |key, val|
         case
         when ["geolocation"].include?(key)
           params[key]=Point.demongoize(val)
   ```

```
      else
      end
    end
  end
```

Director Assembly

`Director` models the detailed information of a movie director.

1. Generate the core of the model class using the `rails generate` command.

   ```
   $ rails g model Director name
   ```

   This generates the following model class.

   ```
   class Director
     include Mongoid::Document
     field :name, type: String
   end
   ```

2. Add the `Timestamp` mixin.

   ```
   class Director
     include Mongoid::Document
     include Mongoid::Timestamps
   ```

3. Add a many-to-one, `belongs_to` relation from `Director` to `Place`. Note that, in this case, the `Place` is not embedded. It is linked using a foreign key.

   ```
     belongs_to :residence, class_name: 'Place'
   ```

4. Add an instance method that will return a query clause set to return the movies the director is asociated with. We have to do this using a custom query since the actual movie reference to the director is through an embedded class (`DirectorRef`)

   ```
     def movies
       Movie.where(:"directors._id"=>self.id)
     end
   ```

5. Add optional validation of the `name`. We went light on validation, but this is one of the few places used.

   ```
     validates_presence_of :name
   ```

DirectorRef Assembly

`DirectorRef` is an annotated reference to a director that gets embedded into the `Movie`. The class is populated with fields that describe the director and anything possibly related specifically to that movie. However, anything added to this class that is director-specific must be updated everywhere this director is used.

1. Generate the core of the model class using the `rails generate` command.

   ```
   $ rails g model DirectorRef name
   ```

   This generates the following model class.

   ```
   class DirectorRef
     include Mongoid::Document
     field :name, type: String
   end
   ```

2. Add the `embedded_in` relation to the `Movie`.

   ```
   embedded_in :movie
   ```

3. Add the child many-to-one relation from `DirectorRef` to the actual `Director` class. This class was referred to as an `annotated link`. This foreign key holds the link and the other properties are either cached from what is on the other side of that link of specific to this movie.

   ```
   belongs_to :director, foreign_key: :_id
   ```

Writer Assembly

`Writer` holds the detailed information about the writer of a movie. This class is directly associated with the movie without an annotated link.

1. Generate the core of the model class using the `rails generate` command.

   ```
   $ rails g model Writer name
   ```

   This generates the following model class.

   ```
   class Writer
     include Mongoid::Document
     field :name, type: String
   end
   ```

2. Add the `Timestamps` mixin.

   ```
   class Writer
     include Mongoid::Document
     include Mongoid::Timestamps
   ```

3. Add an `embeds_one` relation to the `Place`. Remember that the `Director` used a foreign key reference to the `Place` and this collection uses an embdded copy.

   ```
   embeds_one :hometown, as: :locatable, class_name: 'Place'
   ```

4. Add the many-to-many relationship with `Movie`. Since the relationship is directly between the `Movie` and `Writer`, we can delegate all of this to Mongoid.

   ```
   has_and_belongs_to_many :movies
   ```

Actor Assembly

`Actor` contains the information details of an actor in a `Movie`.

1. Generate the core of the model class using the `rails generate` command.

   ```
   $ rails g model Actor name birth_name date_of_birth:Date height:Measurement bio:text
   ```

   This generates the following model class.

   ```
   class Actor
     include Mongoid::Document
     field :name, type: String
     field :birth_name, type: String
     field :date_of_birth, type: Date
     field :height, type: Measurement
     field :bio, type: String
   end
   ```

2. Add a mapping of `birth_name` to the document value of `birthName`. The snake_case form of the name follows Rails naming standards, which will help us out when we get to the scaffold.

```
field :birthName, as: :birth_name, type: String
```

3. Add an `embeds_one` relationship with `Place`. Since multiple model types can embed a `Place`, we must declare the polymorphic label (`locatable`) defined in `Place`.

```
embeds_one :place_of_birth, class_name: 'Place' , as: :locatable
```

4. Add an index on the geolocation coordinates within the actor's birth place.

```
index ({ :"place_of_birth.geolocation" => Mongo::Index::GEO2DSPHERE })
```

5. Add a few custom queries that mimic what Mongoid would have implemented if it were allowed to declare a `has_many` to an embedded class or to the parent of an embedded class. Instead, we are forced to write a small amount of application code to make that happen.

```
#sort-of has_many :movies, class_name:"Movie"
def movies
  Movie.where(:"roles._id"=>self.id)
end
#sort-of has_many roles:, class_name: 'MovieRole'
def roles
  Movie.where(:"roles._id"=>self.id).map {|m| m.roles.where(:_id=>self.id).first}
end
```

6. Add a geolocation search helper method for `place_of_birth`. This primarily keeps the client from having to know the index name.

```
def self.near_pob place, max_meters
  near(:"place_of_birth.geolocation" => place.geolocation)
  .max_distance(:"place_of_birth.geolocation" =>max_meters)
end
```

7. Add conversion support for custom types added during mass assignment. This method will have to be beefed up with the various string or other object formats you throw at the custom type.

```
def sanitize_for_mass_assignment(params)
  params ||= {}
  params.each_pair do |key, val|
    case
    when ["height"].include?(key)
      params[key]=Measurement.demongoize(val)
    else
    end
  end
end
```

MovieRole Assembly

`MovieRole` holds the role-specific information and relation between the `Movie` and `Actor`.

1. Generate the core of the model class using the `rails generate` command.

```
$ rails g model MovieRole character actor_name main:boolean
    url_character url_photo url_profile
```

This generates the following model class.

```
class MovieRole
  include Mongoid::Document
  field :character, type: String
  field :actor_name, type: String
  field :main, type: Mongoid::Boolean
  field :url_character, type: String
  field :url_photo, type: String
  field :url_profile, type: String
end
```

2. Add document mappings for a few of the camelCase properties in the document.

```
  field :actorName, as: :actor_name, type: String
  field :urlCharacter, as: :url_character, type: String
  field :urlPhoto, as: :url_photo, type: String
  field :urlProfile, as: :url_profile, type: String
```

3. Add the `embedded` relationship to `Movie`.

```
  embedded_in :movie
```

4. Add the foreign key, many-to-one relationship to `Actor`.

```
  belongs_to :actor, foreign_key: :_id, touch: true
```

Movie Assembly

`Movie` holds the core information about the movie, its properties, and supporting members.

1. Generate the core of the model class using the `rails generate` command.

```
$ rails g model Movie title type rated year:integer release_date:date \
    runtime:Measurement votes:integer countries:array languages:array \
    genres:array filming_locations:array metascore simple_plot:text \
    plot:text url_imdb url_poster directors:array actors:array
```

   This generates the following model class.

```
class Movie
  include Mongoid::Document
  field :title, type: String
  field :type, type: String
  field :rated, type: String
  field :year, type: Integer
  field :release_date, type: Date
  field :runtime, type: Measurement
  field :votes, type: Integer
  field :countries, type: Array
  field :languages, type: Array
  field :genres, type: Array
  field :filming_locations, type: Array
  field :metascore, type: String
  field :simple_plot, type: String
  field :plot, type: String
  field :url_imdb, type: String
  field :url_poster, type: String
  field :directors, type: Array
  field :actors, type: Array
end
```

2. Add the `Timestamps` mixin.

```
class Movie
  include Mongoid::Document
  include Mongoid::Timestamps
```

3. Add document mappings for a few of the camelCase properties in the document.

```
  field :filmingLocations, as: :filming_locations, type: Array
  field :simplePlot, as: :simple_plot,  type: String
  field :urlIMDB, as: :url_imdb,  type: String
  field :urlPoster, as: :url_poster, type: String
```

4. Add the two `embedded` relationships to `MovieRole` and `DirectorRef`. These are two annotated links to `Actor` and `Director` and contain cached information on the `Movie` side about the document they link to. `MovieRole` also contains some movie-specific information relative to the movie role.

```
  embeds_many :roles, class_name:"MovieRole"
  embeds_many :directors, class_name:"DirectorRef"
```

5. Add the many-to-many relationship with `Writer`. This relationship was thinned down to just the foreign key on both sides of the relationship so the entire relation can be managed by Mongoid.

```
  has_and_belongs_to_many :writers

  has_one :sequel, foreign_key: :sequel_of, class_name:"Movie"
  belongs_to :sequel_to, foreign_key: :sequel_of, class_name:"Movie"
```

6. Add a method to address receiving updates for type-specific fields in simple string format from the web clients. This method and supporting converters need to be beefed up as new situations are encountered with non-native type strings. Notice, however, anything that has a built-in conversion to/from a string (e.g., Date and Integer) do not have to be converted.

```
  def sanitize_for_mass_assignment(params)
    params ||= {}
    params.each_pair do |key, val|
      case
      when ["runtime"].include?(key)
        params[key]=Measurement.demongoize(val)
      when ["countries",
            "languages",
            "genres",
            "filming_locations"].include?(key) && val.is_a?(String)
        params[key]=val.split(",").map {|v| v.strip }
      else
      end
    end
  end
```

Import Data

```
$ rake db:seed
```

Test Drive

1. Locate actors with a place of birth near Silver Spring, MD.

```
> Actor.near_pob(silver_spring, 10*1609.34)
       .each {|actor| pp "#{actor.name}, pob=#{actor.place_of_birth.id}"}; nil
"Lewis Black, pob=Silver Spring, MD, USA"
"Jeffrey Wright, pob=Washington, DC, USA"
"Roy Dotrice, pob=Washington, DC, USA"
"Samuel L. Jackson, pob=Washington, DC, USA"
```

```
 > damon.movies.map {|movie| movie.title}
  => ["The Bourne Ultimatum", "Good Will Hunting",
      "The Martian", "Saving Private Ryan"]
```

```
> damon=Actor.where(:name=>{:$regex=>"Matt Damon"}).first
> damon.roles.map {|role| "#{role.character} => #{role.movie.title}"}
 => ["Jason Bourne => The Bourne Ultimatum",
     "Will Hunting => Good Will Hunting",
     "Mark Watney => The Martian",
     "Private Ryan => Saving Private Ryan"]
```

```
 > pvt_ryan=damon.movies.where(:title=>"Saving Private Ryan").first
 > pvt_ryan.roles.each {|role| puts "#{role.actor_name} => #{role.actor.birth_name}"}; nil
 Tom Hanks => Thomas Jeffrey Hanks
 Tom Sizemore => Thomas Edward Sizemore Jr.
 Edward Burns => Edward Fitzgerald Burns
 Barry Pepper => Barry Robert Pepper
 Adam Goldberg => Adam Charles Goldberg
 Vin Diesel => Mark Sinclair
 Giovanni Ribisi => Antonino Giovanni Ribisi
 Jeremy Davies => Jeremy Davies Boring
 Matt Damon => Matthew Paige Damon
 Ted Danson => Edward Bridge Danson III
 Paul Giamatti => Paul Edward Valentine Giamatti
 Dennis Farina =>
 Joerg Stadler =>
 Max Martini => Maximilian Carlo Martini
 Dylan Bruno => Dylan A. Bruno
```

```
> pvt_ryan.directors.map {|d| d.name}
 => ["Steven Spielberg"]
> pvt_ryan.directors.map {|d| d.director.name}
 => ["Steven Spielberg"]
> pvt_ryan.directors.map {|d| d.director.movies.map {|m| m.title}}
 => [["Jurassic Park", "Jaws", "Saving Private Ryan"]]
> pvt_ryan.writers.map {|w| w.name }
 => ["Robert Rodat"]
> pvt_ryan.writers.map {|w| w.movies.map {|m| m.title}}
 => [["Saving Private Ryan"]]
```

Controller/View Assembly

Movies Contoller/View Assembly

1. Generate the scaffold for the controller and view.

```
$ rails g scaffold_controller Movie title type rated year:integer \
    release_date:date runtime:integer votes:integer countries:array \
    languages:array genres:array filming_locations:array metascore \
    simple_plot:text plot:text url_imdb url_poster directors:array actors:array
```

2. Add the `movies` resource to `config/routes.rb`

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

3. Basic page is displayable.

```
http://localhost:3000/movies
```

4. Bring the `url_poster`, `title`, and `url_imdb` elements to the beginning of the table along with their titles.

```
<th>Poster</th>
<th>Title</th>
<th>Url imdb</th>
...
<td><%= movie.url_poster %></td>
<td><%= movie.title %></td>
<td><%= movie.url_imdb %></td>
```

5. Change the `url_poster` to an `img` tag and use the value of `url_poster` as the URL to the image.

```
<td><img height="100px" width="80px" src= <%= movie.url_poster %>/></td>
```

6. Combine the `title` and `url_imdb` such that the `title` is the link text and the `url_imdb` is the URL of the link. Remove the header for the `url_imdb`.

```
<td><%= link_to movie.title, movie.url_imdb %></td>
```

7. Remove `plot`, `directors`, and `actors` form the index page. Feel free to remove or adjust anything else.

```
<th>Plot</th>
<th>Directors</th>
<th>Actors</th>
...
<td><%= movie.plot %></td>
<td><%= movie.directors %></td>
<td><%= movie.actors %></td>
```

8. Add general purpose partial view to dump contents of objects to a table. We will make use of this as we add relationships. You may have to create the `application` directory below `view`.

   `app/views/application/_member.html.erb`

```
<div class="field">
<table border='1'>
  <% member.attributes.sort.each do |r| %>
    <tr>
      <td><%= r[0] %></td>
      <td><%= r[1] %></td>
    </tr>
  <% end %>
</table>
</div>
```

9. Update the `show` page to use the partial just created. This should build a very simple, but functional view of the related information.

```
<p>
  <strong>Directors:</strong>
  <%= render partial: "member", collection: @movie.directors %>
</p>

<p>
  <strong>Writers:</strong>
  <%= render partial: "member", collection: @movie.writers %>
</p>

<p>
  <strong>Actors:</strong>
  <%= render partial: "member", collection: @movie.roles %>
</p>
```

10. It is endless what you can do from here.