

In this lecture, we will discuss...

- ✧ 1:1 Embedded
- ✧ Polymorphic Relationships
- ✧ Demo

Relationships – Embedded

- ✧ Parent document of the relation must declare the `embeds_one` macro to indicate it has one `embedded` child
- ✧ Document that is embedded uses `embedded_in`
- ✧ Actor → `place_of_birth:Place`

Relationships - Example

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

```
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
```



Polymorphic Relationships

- ✧ Embedding the same document type in to several different parent type
- ✧ Child → `polymorphic`

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

DEMO

Summary

- ✧ One to one relationships - children are embedded in the parent document
- ✧ Defined using Mongoid's `embeds_one` and `embedded_in`

What's Next?

- ✧ M:1 - Linked



In this lecture, we will discuss...

- ✧ M:1 Linked
- ✧ Foreign Key Relationship
- ✧ Demo



Relationships – belongs_to

- ✧ Children are stored in a **separate** collection from the parent (common but not necessarily)
- ✧ Child (*class using the FK*) uses `belongs_to` and parent **optionally** uses `has_many`
 - Without the "`has_many`" macro, the relationship becomes a uni-directional
- ✧ Multiple directors (**child**) can have the same place of residence (**parent**)



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

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



DEMO

Summary

- ✧ The parent uses the **optional** `has_many` macro to indicate it has n number of referenced children
- ✧ The document that is referenced uses `belongs_to`

What's Next?

- ✧ 1:M Embedded



In this lecture, we will discuss...

- ✧ 1:M Embedded `embeds_many` and `embedded_in`
- ✧ Accessing Relationships
- ✧ Modifying Relationships



Relationships – embeds_many

- ✧ Parent document of the relation should use the **embeds_many** macro to indicate it has **n** number of **embedded** children
- ✧ Document that is embedded uses **embedded_in**



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

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

```
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
```



DEMO

Summary

- ✧ One to many relationships - children are embedded in the parent document are defined using Mongoid's `embeds_many` and `embedded_in`

What's Next?

- ✧ M:1 – Embedded Linked



In this lecture, we will discuss...

- ✧ M:1 Embedded Linked
- ✧ annotated Link
- ✧ Demo

Relationships – Embedded Linked

- ✧ M:1 Embedded Link
- ✧ Many actors in a movie, but each play a specific role
- ✧ Movie Role \leftrightarrow Actor

```
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
```

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

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

    #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
end
```



DEMO

Summary

- ✧ annotated link
- ✧ “1” side has a primary key and typically has no reference to the child within the document.
- ✧ “M” side will typically host the foreign key

What's Next?

- ✧ 1 : 1 – Linked



In this lecture, we will discuss...

- ✧ 1:1 Linked - `has_one`
- ✧ Recursive Relationship
- ✧ Demo

Referenced 1-1 : `has_one`

- ✧ One to one relationships
- ✧ Children are referenced in the parent document
- ✧ References are defined using Mongoid's `has_one` and `belongs_to` macros



Linked (Movie -> sequel_to:Movie)

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



DEMO

Summary

- ✧ The parent document uses the `has_one` macro to indicate it has 1 referenced child
- ✧ The document that is referenced in it uses `belongs_to`

What's Next?

- ✧ **M:M** - `has_and_belongs_to_many`



In this lecture, we will discuss...

- ✧ **M:M** - has_and_belongs_to_many
- ✧ Bi-Directional Relationship
- ✧ Demo



References M-M : `has_and_belongs_to_many`

- ✧ Many to many relationships where the **inverse** documents are stored in a **separate** collection
- ✧ Both parent and child use Mongoid's `has_and_belongs_to_many` macro
- ✧ Foreign key IDs are stored as **arrays** on either side of the relation



Relationships – Writer Model

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



Relationships – Movie Model

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



DEMO

Summary

- ✧ Both sides of the relation use the same macro

What's Next?

- ✧ Constraints and Validations

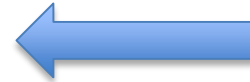
In this lecture, we will discuss...

- ✧ Constraints
- ✧ Validation
- ✧ Dependent behavior
 - `:delete`
 - `:destroy`
 - `:nullify`
 - `:restrict`

Field Validation

- ✧ ActiveRecord validations can be added to Mongoid model classes.

```
1 class Director
2   include Mongoid::Document
3   include Mongoid::Timestamps
4   field :name, type: String
5   .....
6   validates_presence_of :name
7 end
```



“name is mandatory”

Dependent Behavior

- ✧ Mongoid supports **dependent** options to manage **referenced** associations
- ✧ Will instruct Mongoid to handle **delete** situations
- ✧ `:delete, :destroy, :nullify, :restrict`



Relationship Constraints

- ✧ `(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` – **Orphans** the child document after setting the child foreign key to nil

Relationship Constraints

- ✧ `: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



delete vs destroy

- ✧ `:delete` – will only delete current object – straight delete in the database.
- ✧ `:destroy`– will delete current object and associated children record - analyzes the class you're deleting, determines what it should do for dependencies, runs through validations (callbacks - `:before_destroy`, `:after_destroy`.)



Callbacks – Movie Model

```
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
```



Callbacks – Writer Model

```
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
```



DEMO

Script – Demo Setup

- ✧ `reload!`
- ✧ `rocky30=Movie.create(:title=>"Rocky 30")`
- ✧ `rocky31=Movie.create(:title=>"Rocky 31",
:sequel_to=>rocky30)`
- ✧ `writer=rocky30.writers.create
(:name=>"A Writer")`



Script – Data Cleanup

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



Summary

- ✧ Constraints, Validations and Dependent Behavior – key in maintaining the state of your application data.

What's Next?

- ✧ Queries

