Desenvolvimento de Aplicações Web com **Ruby on Rails**

Arthur de Moura Del Esposte - esposte@ime.usp.br

By Arthur Del Esposte licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0)

Aula 04 - DRY, Validações e Associações

Arthur de Moura Del Esposte - esposte@ime.usp.br

By Arthur Del Esposte licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0)

Agenda

- DRY - Continuação
- Models - Validações
- Models - Associações

Aplicando o princípio **DRY** no MVC

Foto: Christopher Michel

DRY

- O princípio **DRY** (Don’t Repeat Yourself) é muito encorajado pelo Rails e devemos tentar aplicá-lo ao longo do nosso código para evitar duplicações
- Veja que temos duplicações tanto em MoviesController (show, edit, update) quanto nas Views (Movies#new e Movies#edit)
- Vamos modificar esse código para termos um melhor reuso

DRY

- O princípio **DRY** (Do ajado pelo Rails e devemos tentar apli ra evitar duplicações
- Veja que temos dup r (show, edit, update) quanto nas Views (M
- Vamos modificar es euso

Partials

- Partials são fragmentos **html.erb** que podem ser incluídos em nossas Views que são renderizadas junto com nossas páginas
- Partials permitem reutilização de nossa lógica de visualização
- As partials são arquivos criados nas mesmas pastas dos outros arquivos de visualização
- Os nomes dos arquivos de partials devem começar com **underline**:
  - **_form.html.erb**
- As Views **Movie#new** e **Movie#edit** possuem muita coisa em comum. Vamos extrair a parte duplicada para uma **partial**

Partials

- Crie o arquivo **app/views/movies/_form.html.erb** e copie para esse arquivo o código deste link
- Esse código possui basicamente a mesma estrutura do formulário das Views Movies#new e Movies#edit, porém as partes que variam são acessadas através de variáveis locais **method** e **action**
- Após salvar o arquivo, só precisamos remover os formulários das Views Movies#new e Movies#edit e renderizar a partial criada, passando as informações de **method** e **action**
- As variáveis são passadas através da chave **locals**

Partials

- **app/views/movies/new.html.erb:**

```
<h1 class="title">Create a new movie</h1>

<%= render partial: 'form', locals: {method: 'post', action: 'create'} %>
```

- **app/views/movies/edit.html.erb:**

```
<h1 class="title">
  Edit movie <%= @movie.title %>
</h1>

<%= render partial: 'form', locals: {method: "put", action: 'update'} %>
```

Partials

- Repare que não precisamos passar a variável **@movie**, pois ela já vem da controladora!
- Note também que os nossos arquivos de visão foram enxugados
- Consequentemente, só temos um lugar onde precisamos manter o formulário relacionado a filmes!

Filters

- Podemos usar **filtros** para fazer reúso dos códigos de controladoras
- Filtros definem formas para executarmos pedaços de código **antes** e **depois** de nossas **actions**

```
class MoviesController < ApplicationController
  before_action :set_movie

  private
  def set_movie
    id = params['id']
    @movie = Movie.find(id)
  rescue ActiveRecord::RecordNotFound
    render file: "#{Rails.root}/public/404.html", status: 404
  end
end
```

Filters em MoviesController

- No nosso caso, só queremos aplicar o filtro nas ações **show**, **edit** e **update**
- O código neste link contém a versão do **MoviesController** com as modificações usando filtros
- Note que os métodos estão bem mais enxutos e nosso código está com maior reúso
- O parâmetro **only** recebe uma lista de ações onde o filtro deve ser aplicado
- Da mesma forma, poderíamos utilizar o parâmetro **except** para aplicar o filtro em todas as ações exceto as especificadas por esse parâmetro

MyMovies - Commitando Modificações

- Registre uma nova versão com as modificações feitas até agora:

```
$ git add .

$ git commit -m “Initial commit”
```

- Atualize suas modificações no Github empurrando seus commits para lá:

```
$ git push origin master
```

Scaffold

Gerando tudo com o Scaffold

- Já vimos que o Rails pode gerar nossas **Modelos** com suas **migrações**, além de facilitar o trabalho gerando nossas **Controladoras** e **Visões**
- Entretanto, o Rails possui um gerador ainda mais poderoso: **scaffold**
- O gerador **scaffold** cria a estrutura completa para suas classes, incluindo a classe Modelo, a migração, as Rotas e Controladora com as ações correspondentes às que criamos, e as páginas **html.erb** necessárias
- Você pode chamar o script scaffold de forma semelhante às outras chamadas que utilizamos para gerar nosso códigos

```
$ rails generate scaffold MyNewModel attribute1:string attribute2:integer
```

Gerando tudo com o Scaffold

- Vamos utilizar o scaffold para gerar o restante do código necessário para **Actor**, uma vez que já criamos a classe Modelo e as migrações:

```
$ rails generate scaffold Actor name:string birthdate:datetime gender:string country:string type:string --migration=false --skip
```

- Veja que escolhemos não gerar as migrações e pulamos a geração de arquivos que já existem com a opção **--skip**
- Execute sua aplicação e acesse: http://localhost:3000/actors

Gerando tudo com o Scaffold

- Vamos fazer o mesmo com **Director**

```
$ rails generate scaffold Director name:string birthdate:datetime gender:string
country:string type:string --migration=false --skip
```

- Execute sua aplicação e acesse: http://localhost:3000/directors

Resources em Rotas

- Veja como o Rails definiu as rotas para as modelos geradas com scaffold no arquivo **config/routes.rb**
- O uso de **resources** nas rotas diz ao Rails para definir todas as rotas de acesso ao recurso especificado (**CRUD**), semelhante com o que fizemos com a entidade **Movie**
- A arquitetura baseada em **recursos** respeita as abstrações da arquitetura REST para serviços web
- Veja as rotas geradas:

```
$ rake routes
```

Validações

Validações

- Para preservar e manter a consistência dos registros do banco de dados é importante verificar se as entradas são **válidas**
  - Campos de preenchimento obrigatório
  - Campos que podem ficar vazios
  - Campos que só aceitam números
  - Campos que devem ser únicos (ex: username)
  - Campos que devem estar entre um tamanho máximo e mínimo
- Precisamos resolver:
  - Como **validar** a entrada
  - O que acontece ao tentar salvar um entrada **inválida**

Validações

- As validações verificam se um campo específico dentro de uma classe modelo está dentro das regras definidas
- Validações são executadas na criação ou atualização de um objeto no banco de dados
- **Caso um objeto tenha algum campo inválido, ele não poderá ser salvo**
- O Rails já tem algumas validações que podemos utilizar em nossas Model

Validações - Presença

Validando a presença dos campos **name** e **gender**

```
class Professional < ApplicationRecord
  validates :name, :gender, presence: true
end
```

- Podemos utilizar o método **valid?** para executar as validações e verificar se o objeto é válido
- **Implemente essas validações no projeto MyMovies**

Rails Console - Validações - Presença

Validando a presença dos campos **name** e **gender**

```
➜ reload!
➜ Actor.create(name: "Chris Evans", gender: "male").valid?
➜ Actor.new(name: "Chris Evans", gender: "male").valid?
➜ Actor.new(name: "Chris Evans").valid?
➜ Actor.new(name: "Chris Evans").invalid?
➜ actor = Actor.new
➜ actor.save # => false
➜ actor.errors.messages
➜ actor.save! # => ActiveRecord::RecordInvalid
```

Validações - errors

- Quando rodamos uma validação sobre um objeto, os erros relacionado aos atributos inválidos podem ser acessados pelo método **errors**
- Pode se usar **errors[:atributo]** para verificar se há algum erro relacionado à um atributo específico
  - Ele retorna um Array de erros relacionados a esse atributo
  - Se não houver nenhum erro, retorna um Array vazio
- Acessar o método errors só é útil depois de ter executando uma validação, uma vez que ele não faz isso!

```
➢ actor = Actor.new
➢ actor.valid? # => false
➢ actor.errors[:name].any?
➢ actor.save! # => ActiveRecord::RecordInvalid
```

Validações - Inclusão

Validando se o gênero digitado incluí um dos dois valores possíveis: **male** e **female**

```
class Professional < ApplicationRecord
  validates :name, :gender, presence: true
  validates :gender, inclusion: { in: ['male', 'female'],
    message: "%{value} is not valid - expected 'male' or 'female'" }
end
```

- Repare que agora temos duas validações diferentes sobre o atributo gender
- Note que colocamos uma mensagem personalizada
- **Implemente essas validações no projeto MyMovies**

Rails Console - Validações - Inclusão


```
➢ reload!
➢ Actor.new(name: "Will Smith", gender: "male").valid?
➢ actor = Actor.new(name: "Chris Evans", gender: "man")
➢ actor.valid? # => false
➢ actor.errors[:gender]
➢ actor.errors.full_messages
➢ actor.gender = "male"
➢ actor.save!
```

Validações - Tamanho do Campo

A validação de tamanho em Rails possui várias opções

```ruby
class Person < ApplicationRecord
  validates :name, length: { minimum: 2 }
  validates :bio, length: { maximum: 1000,
    too_long: "%{count} characters is the maximum allowed" }
  validates :password, length: { in: 6..20 }
  validates :registration_number, length: { is: 6 }
end
```

Validações - Campos numéricos

- Há uma grande variedade de validações disponíveis para campos numéricos:

```
class Player < ApplicationRecord
  validates :points, numericality: true
  validates :games_played, numericality: { only_integer: true }
end
```

- Assim como **:only_integer**, temos outras opções incluem:
  - :greater_than
  - :greater_than_or_equal_to
  - :equal_to
  - :less_than
  - :other_than
  - :odd
  - :even

Validações - Valores Únicos

- Muitas vezes queremos que os valores de uma coluna não se repitam ao longo dos vários registros
- Ou seja, queremos que eles sejam **únicos**. Por exemplo: username, email, nome do filme

```
class Movie < ActiveRecord::Base
  validates :title, presence: true
  validates :title, uniqueness: true
end
```

- **Implemente essas validações no projeto MyMovies**

Rails Console - Validações - Valores Únicos


```
➢ reload!
➢ Movie.create!(title: "Django")
➢ movie = Movie.new(title: "Django")
➢ movie.valid? # => false
➢ movie.errors.full_messages
➢ movie.title = "Django 2"
➢ movie.valid? # => true
➢ movie.errors.full_messages
```

Validações - Valores Únicos

- Podemos utilizar a opção **:scope** para que mais de um atributo seja verificado em relação à unicidade
- Assim, individualmente os atributos podem se repetir, mas um mesmo par de valores não pode existir!

```
class Movie < ActiveRecord::Base
  validates :title, presence: true
  validates :title, uniqueness: { scope: :release_date,
    message: "should not have two movies with same name and release_date" }
end
```

- **Implemente essas validações no projeto MyMovies**

Rails Console - Validações - Valores Únicos


```
➜ reload!
➜ Movie.create!(title: "A", release_date: "25-04-1977")
➜ Movie.create!(title: "B", release_date: "25-04-1977")
➜ Movie.create!(title: "A", release_date: "25-04-2001")
➜ Movie.create!(title: "A", release_date: "25-04-1977")
➜ movie = Movie.new(title: "A", release_date: "25-04-1977")
➜ movie.valid? # => false
➜ movie.errors.full_messages
```

Validações - Condicional

- Podemos utilizar uma validação somente quando uma condição for verdadeira:

```
class Order < ApplicationRecord
  validates :card_number, presence: true, if: :paid_with_card?

  def paid_with_card?
    payment_type == "card"
  end
end
```

Validações - Personalizadas

- Podemos querer implementar nossas próprias validações baseado em nossas regras de negócio
- Para isso, basta implementar um método de validação e adiciona os erros relacionados ao atributo avaliado

```ruby
class Invoice < ApplicationRecord
  validate :active_customer

  def active_customer
    errors.add(:customer_id, "is not active") unless customer.active?
  end
end
```

Tratando Validações na View

- Com validações, o usuário pode não conseguir salvar um objeto na página de criação e de edição
- Tente criar um filme violando a validação de se ter dois filmes na mesma data
- Como o usuário sabe que não foi possível salvar o filme?

Tratando Validações na View

- Sempre que um registro for inválido, temos que mostrar ao usuário quais erros ocorreram! Veja o código da partial app/views/movies/_form.html.erb

```
<% if @movie.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@movie.errors.count, "error") %> prohibited this movie from being saved:</h2>

    <ul>
    <% @movie.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
    </ul>
  </div>
<% end %>
```

Tratando Validações na View

- Veja as ações **MoviesController#create** e **MoviesController#update**
- Ambas usam **redirect_to** e **render**
- O **redirect_to** causa um redirecionamento onde mais uma requisição será feita automaticamente para completar a requisição original (HTTP 302). Assim, o Rails tratará a requisição normalmente utilizando a controladora e ação apropriada para responder à nova requisição baseado nas rotas
- O **render** simplesmente renderiza uma View diretamente sem fazer uma requisição adicional para isso. Dessa forma a variável **@movie** já fica disponível na View renderizada quando a validação falha

MyMovies - Commitando Modificações

- Registre uma nova versão com as modificações feitas até agora:

```
$ git add .

$ git commit -m “Add validations”
```

- Atualize suas modificações no Github empurrando seus commits para lá:

```
$ git push origin master
```

Associações

Associações

- Seja em Orientação a Objetos ou em Banco de Dados relacionais, definir relações entre entidades é fundamental para o **design** do software
- Em Rails, associações permitem que nossos registros mantenham referências para outros registros do banco de dados. Da mesma forma, em nível de programação, associações permitem que possamos manipular as relações entre os nossos objetos através de métodos
- No projeto **MyMovies** já pensamos sobre quais relações seriam interessantes, vamos implementar algumas delas agora
- Existem vários tipos de associações que podemos utilizar em Rails. Cada uma delas deve ser acompanhada de uma **migração** e declarações nas classes envolvidas

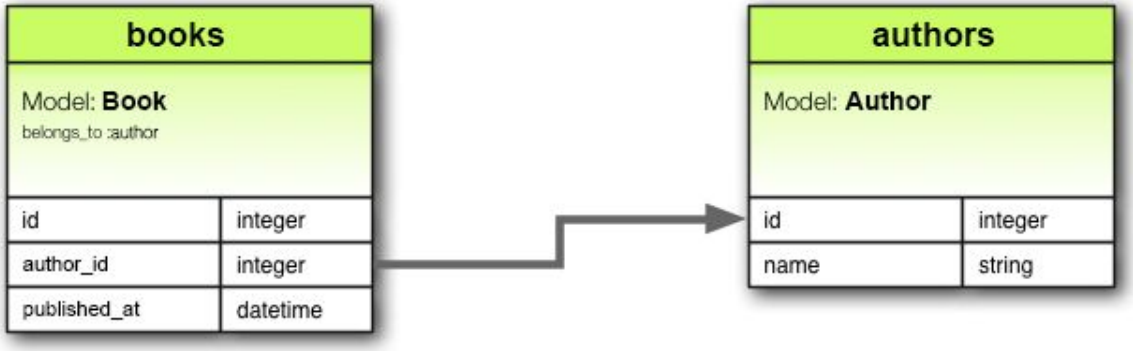
Associações

- Portanto, a migração cria as referências no banco de dados necessárias para persistir as associações usando **Chave Estrangeira**
- Por outro lado, declarar relações em código Ruby permite que usemos alguns métodos para manipular as relações, como por exemplo:

```
➢ movie = Movie.find_by_title("Star Wars")
➢ movie.actors # => Retorna a lista de atores do filme
➢ movie.director # => Retorna o diretor de um filme
➢ movie.ratings # => Retorna todas as avaliações de um filme
➢ Rating.where(movie_id: movie.id, person_id: person.id)
➢ actor = Actors.find_by_name("Will Smith")
➢ actor.movies # => Retorna a lista de filmes estrelado pelo ator
```

Associações - 1 para 1 - **belongs_to**

- Quando queremos ter uma associação de 1 para 1 falamos que um objeto da entidade A pertence à exatamente um objeto da entidade B
- No BD usamos uma **chave estrangeira** para referenciar outro registro

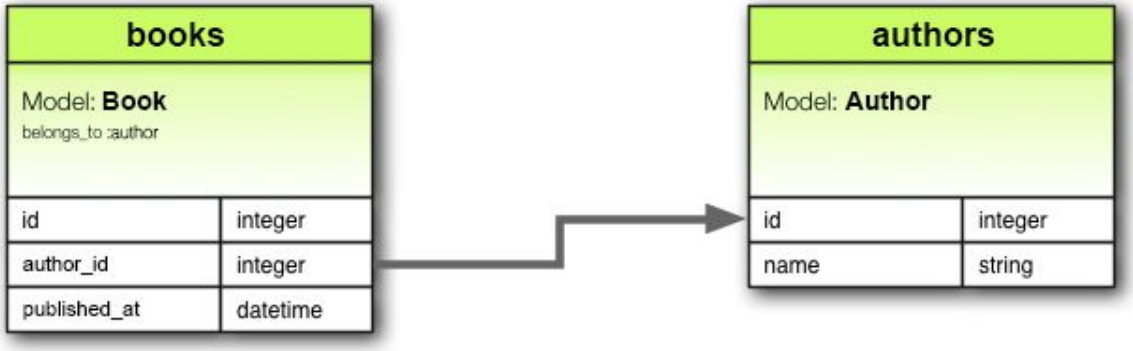



```
class Book < ActiveRecord: :Base
  belongs_to :author
end
```

Associações - 1 para 1 - **belongs_to**

- Quando queremos ter uma associação de 1 para 1 falamos que um objeto da entidade A pertence à exatamente um objeto da entidade B
- No BD usamos uma **chave estrangeira** para referenciar outro registro

Associação no BD



Associação na Classe

```
class Book < ActiveRecord: :Base
  belongs_to :author
end
```

Associações - 1 para 1 - **belongs_to**

- Para isso, temos que ter uma migração que adicione a chave estrangeira na tabela da classe que pertencente a outra entidade

```ruby
class CreateBooks < ActiveRecord::Migration[5.0]
  def change
    create_table :authors do |t|
      t.string :name
      t.timestamps
    end

    create_table :books do |t|
      t.belongs_to :author, index: true
      t.datetime :published_at
      t.timestamps
    end
  end
end
```

Associações - 1 para 1 - **belongs_to**

- Como declaramos belongs_to na classe Book, todos os objetos dessa classe passam a responder a métodos para acessar o autor: `book.author`

```
class Book < ActiveRecord: :Base
  belongs_to :author
end
```

- Se quisermos permitir que a classe Author também possua métodos semelhantes temos que explicitar a associação nessa classe também

Associações - **has_one** e **has_many**

- Se definirmos que cada ator possui exatamente um livro no sistema, estaremos estabelecendo uma relação 1 para 1. Usamos então **has_one**

```
class Author < ActiveRecord::Base
  has_one :book
end
```

- Porém, faz mais sentido definirmos uma relação 1 para N, dado que um autor pode escrever vários livros. Usamos então **has_many**

```
class Author < ActiveRecord::Base
  has_many :books
end
```

Associações - **has_one** e **has_many**

- Repare que nos dois casos não é preciso modificar a migração que já tínhamos, pois quem vai manter a chave estrangeira será a tabela **Books**
- Convenções são usadas para inferir o nome das tabelas baseado nas relações determinadas
- Em nosso projeto **MyMovies**, um filme possui exatamente um diretor. Por sua vez, um diretor pode ter dirigido vários filmes. Essa é uma relação 1 para N
- Vamos criar essa relação:
  - Gerar uma migração para adicionar a referência ao diretor na tabela movies
  - Definir as relações nas classes

```
$ rails generate migration add_director_to_movies
```

Migração para Associação

- Como temos uma tabela que não segue o nome da classe devido a Herança que definimos, nossa migração ficará da seguinte forma:

```
class AddDirectorToMovies < ActiveRecord::Migration[5.0]
  def change
    add_column :movies, :director_id, :integer
  end
end
```

```
$ rake db:migrate
```

Associação em Director e Movie

- Como não seguimos a convenção básica, temos que deixar claro nas classes Director e Movie alguns detalhes:

```ruby
class Director < Professional
  has_many :movies, class_name: 'Movie', foreign_key: 'director_id'
end
```

```ruby
class Movie < ActiveRecord::Base
  validates :title, presence: true
  validates :title, uniqueness: { scope: :release_date,
    message: "should not have two movies with same name and release_date" }

  belongs_to :director, class_name: "Director", foreign_key: "director_id"
end
```

Rails Console - Associação Director e Movie

- reload!
- movie = Movie.last
- director = Director.last
- movie.director = director
- movie.save!
- director.movies << Movie.find(1)
- director.movies # => Relation with movies
- movie.director # => A Director object
- movie.director = Actor.last

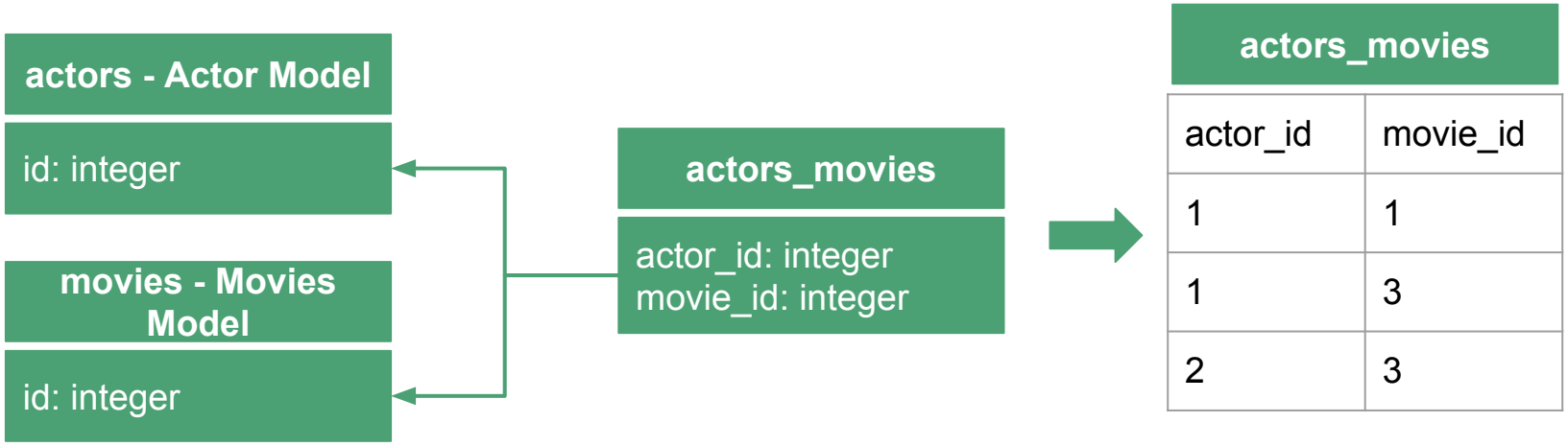
Associação em Director e Movie

- No Rails 5, a relação belongs_to torna a presença da associação obrigatória. Tente cadastrar qualquer filme pela interface gráfica para ver o erro!
- Por isso, vamos torná-la opcional

```
belongs_to :director, class_name: "Director", foreign_key: "director_id", , optional: true
```

Associações - N para N

- A nossa relação de Atores com Filmes é de muitos para muitos (N para N)
- Um ator atuou em um ou mais filmes, enquanto um filme possui vários atores
- Nesse caso, usamos uma relação **has_and_belongs_to_many**



Associações - N para N

- A nossa relação de Atores com Filmes é de muitos para muitos (N para N)
- Um ator atuou em um ou mais filmes, enquanto um filme possui vários atores
- Nesse caso, usamos uma relação has_and_belongs_to_many

```ruby
class Actor < Professional
  has_and_belongs_to_many :movies, association_foreign_key: 'movie_id', join_table: 'actors_movies'
end
```

```ruby
class Movie < ActiveRecord::Base
  validates :title, presence: true
  validates :title, uniqueness: { scope: :release_date,
    message: "should not have two movies with same name and release_date" }

  belongs_to :director, class_name: "Director", foreign_key: "director_id", optional: true
  has_and_belongs_to_many :actors, association_foreign_key: 'actor_id', join_table: 'actors_movies'
end
```

Associações - N para N

- Gere a migração para criar a tabela de união **actors_movies** (Join Table)

```
$ rails generate migration create_actors_movies
```

```ruby
class CreateActorsMovies < ActiveRecord::Migration[5.0]
  def change
    create_table :actors_movies, id: false do |t|
      t.integer :movie_id
      t.integer :actor_id
    end
  end
end
```

```
$ rake db:migrate
```

Rails Console - Associação Actor e Movie


```
➢ reload!
➢ movie_a = Movie.create!(title: "A", release_date: "25-02-1977")
➢ movie_b = Movie.create!(title: "B", release_date: "25-09-1987")
➢ actor = Actor.last
➢ actor.movies << movie_a
➢ movie_a.reload
➢ movie_a.actors
➢ movie_b.actors << actor
➢ actor.reload
➢ actor.movies # => Include A and B
```

Auto Associações

- Uma classe pode querer manter um auto-relacionamento em alguns casos
- Suponha que queremos manter as referências das sequências que um filme teve. Teríamos que ter um auto-relacionamento na classe Movie

```
class Movie < ApplicationRecord
  has_many :sequence, class_name: "Movie",
                          foreign_key: "sequence_id"

  belongs_to :original_movie, class_name: "Movie"
end
```

- Nesse caso teríamos que ter mais uma migração que adicione a coluna **sequence_id** na tabela **movies**
- **Não faremos isso por enquanto!**

MyMovies - Commitando Modificações

- Registre uma nova versão com as modificações feitas até agora:

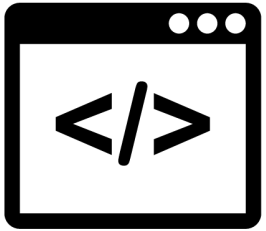
```
$ git add .

$ git commit -m “Add associations”
```

- Atualize suas modificações no Github empurrando seus commits para lá:

```
$ git push origin master
```

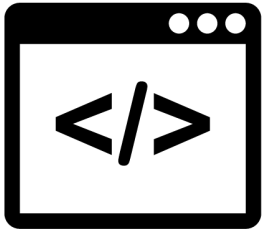
Atividades Sugeridas!

Exercício



- Continue a desenvolver a aplicação:
  - Adicione algumas associações no arquivo **db/seed.rb** para popular o banco de dados
  - Apresente na página de cada filme o diretor e a lista de atores. O nome de cada profissional deve ser um link para a página pessoal deles
  - Apresente a lista de filmes em que cada profissional já trabalhou na página do Ator e Diretor

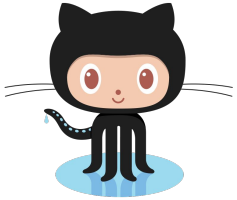
Exercício



- Na página do Ator, liste todos os atores com quem um ator já trabalhou, informando a quantidade de filmes que esses dois atores trabalharam juntos
- Obs:
  - Isso pode exigir uma auto-associação entre os atores diferentes utilizando a associação has_many :through. Veja a documentação dessa associação para utilizá-la

Contato

https://gitlab.com/arthurmde



https://github.com/arthurmde

http://bit.ly/2jvND12

http://bit.ly/2j0llo9

Centro de Competência em Software Livre - CCSL

esposte@ime.usp.br

Obrigado!