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

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

Aula 03 - Controller e Views

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

Agenda

- Gerando Modelos e Herança
- Controllers e Views
- Formulários
- Helpers do Rails
- DRY - Partials e Filtros

Gerando Modelos

Gerando Modelos com Rails

- O Rails possui alguns scripts que geram código automaticamente para agilizar o desenvolvimento
- Vamos utilizar o Rails para gerar a modelo **Professional**
- Como ambos **Actor** e **Director** possuem atributos semelhantes, ambos serão uma especialização da classe **Professional**
- Inicialmente, vamos gerar a model Professional:

```
$ rails generate model Professional name:string birthdate:datetime gender:string country:string type:string
```

Gerando Modelos com Rails

- Essa forma de criar models já gera:
  - Migração
  - Classe Modelo
  - Testes Unitários
- Rails usa convenção para gerar o nome da migração e das classes!
- Sempre abra a migração antes de aplicar para ver se tudo está correto

```
$ rake db:migrate
```

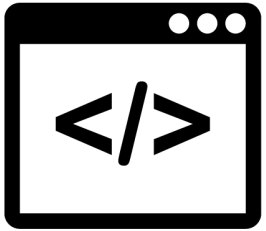
Herança com Rails

- Crie as classes **Actor** e **Director**. Ambas devem herdar de **Professional**
- Com isso, o Rails utiliza a mesma tabela para registrar tanto os atores quanto os diretores, porém usa a coluna **type** que criamos para distinguir o tipo específico de um registro

Console - Herança

- Actor.create!(name: "Angelina Jolie", gender: "female")
- Actor.create!(name: "Will Smith", gender: "male")
- Director.create!(name: "Quentin Tarantino", gender: "male")
- Actor.count
- Director.count
- Professional.count
- Professional.where(gender: "male")
- Actor.where(gender: "male")

Exercício



- Altere o arquivo db/seed.rb para alimentar alguns atores e diretores no banco de dados:
  - Atriz: Angelina Jolie
  - Ator: Will Smith
  - Atriz: Margot Robbie
  - Diretor: Quantin Tarantino
  - Diretor: Mel Gibson

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:

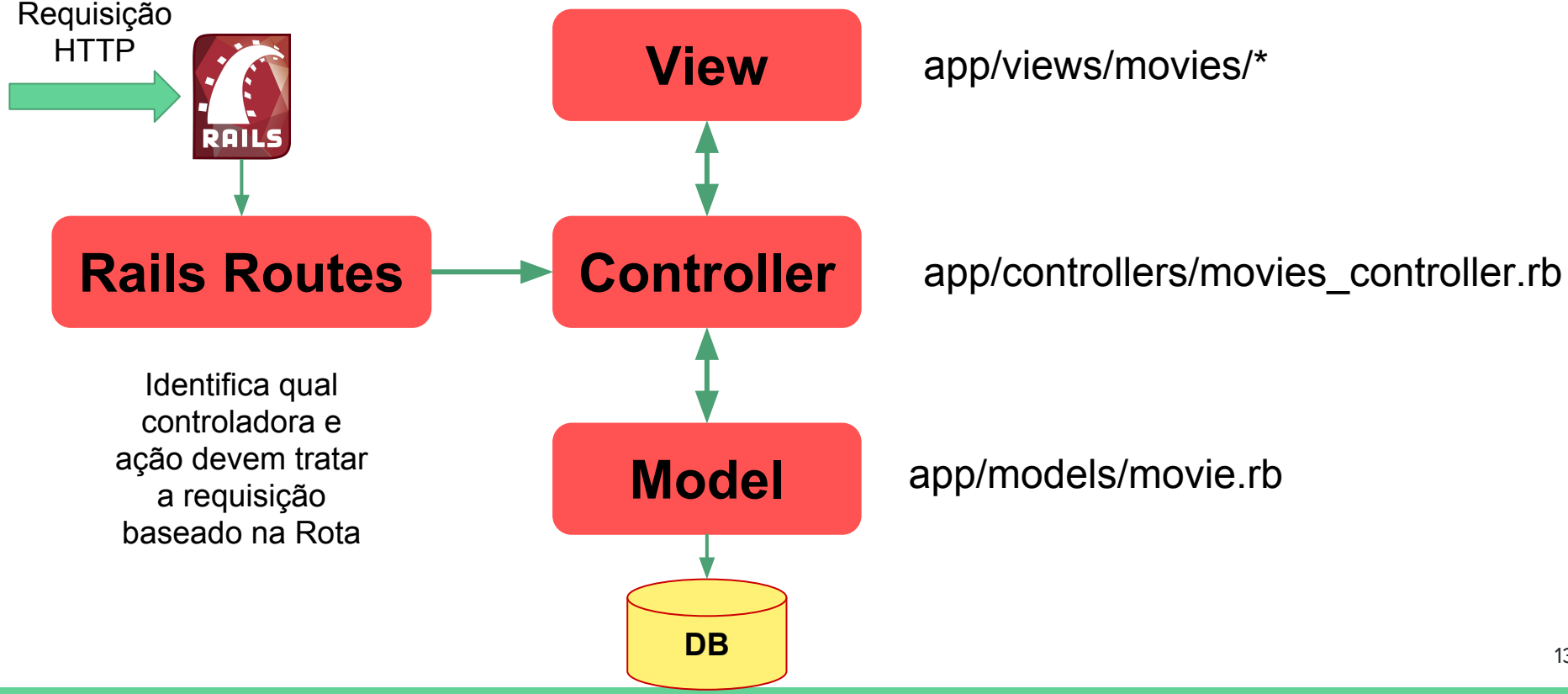
```
$ git push origin master
```

Acessando as Modelos através de Controladoras e Views

Acessando as Modelos

- No geral, queremos sempre fazer quatro operações com as classes Modelos - **CRUD**
- Para isso, temos que criar as páginas necessárias para que usuários possam interagir com a aplicação e manipular nossas classes
- Para uma mesmo Modelo, temos uma única Controladora que responde à todos as requisições comuns às operações CRUD e apresentação dos recursos
- Por outro lado, para um mesmo Modelo e Controladora, temos várias Visões diferentes que permitem:
    - Listar todos os filmes
    - Mostrar a página de um filme
    - Página para criar e editar um novo filme

MVC


Requisição
HTTP
RAILS
Rails Routes
Identifica qual
controladora e
ação devem tratar
a requisição
baseado na Rota
View
Controller
Model
DB
app/views/movies/*
app/controllers/movies_controller.rb
app/models/movie.rb

Gerando a Controladora e Visão

- Podemos facilitar a criação da Controladora e Visão utilizando o gerador de código do Rails!

```
$ rails generate controller Movies index show new create
```

- Veja os arquivos criados
- A ação **index** irá renderizar a lista de todos os filmes
- A ação **show** irá renderizar a página de um filme específico
- A ação **new** irá renderizar um formulário para criar um novo filme
- A ação **create** irá tratar a criação do filme e redirecionar para a página do novo filme cadastrado

Comunicação entre Controladores e Visão

- As Controladoras podem acessar as classes Modelos
- Implemente a ação **MoviesController#index** da seguinte forma:

```
def index
  @movies = Movie.all
end
```

- A Visão tem acesso as variáveis definidas na Controladora correspondente
- Para isso, os arquivos de Visão precisam fazer chamadas ao código Ruby
- Portanto, ao invés de trabalhar com arquivos HTML puros, vamos trabalhar com arquivos cuja extensão será **.html.erb**
- **ERB = Embedded Ruby**
- Ou seja, podemos ter código Ruby nesse tipo de arquivo que será posteriormente transformado em um HTML puro para responder ao cliente

Movies#index

- Adicione o seguinte código que está neste link no arquivo **app/views/movies/index.html.erb**
- Repare que nele temos a estrutura de um HTML comum, porém com adição de algumas chamadas a código Ruby, definidos por:
  - <% CÓDIGO RUBY %>
  - <%= CÓDIGO RUBY QUE RETORNA UM VALOR %>
- Além disso, veja que definimos uma **class** para o elemento **h1** e um **id** para o elemento **table**
- Nós usaremos esses atributos posteriormente para dar estilo para nossa página

Movies#index

- Para acessar a página criada, acesse: http://localhost:3000/movies/index
- Porém, geralmente omitimos o /index da URI de tal forma que o link http://localhost:3000/movies leve para a página que acabamos de criar
- Para fazer isso basta alterar a rota:

```
# get 'movies/index'
get 'movies' => "movies#index"
```

- Execute novamente a aplicação e acesse http://localhost:3000/movies

Movies#show

- Para a página **MoviesController#show** queremos apresentar um filme específico
- Precisamos que o usuário nos passe um parâmetro para sabermos exatamente qual recurso (filme) ele deseja acessar
- No geral utilizamos o **id** dos nossos objetos para identificar qual recurso específico o usuário deseja acessar,
- Portanto nossos links devem ser da seguinte forma:
  - /movies/**1** - Obtém a página do filme com id igual a 1
  - /movies/**10** - Obtém a página do filme com id igual a 10
- Precisamos alterar a nossa rota para seguir esse padrão! Veja como estão atualmente:

```
$ rake routes
```

Movies#show

- Comente a rota para o método **show** que havia antes e adicione uma rota nova que recebe o parâmetro **id**

```
# get 'movies/show'
get 'movies/:id' => "movies#show"
```

- Veja novamente as rotas:

```
$ rake routes
```

- Execute a aplicação e tente acessar a página http://localhost:3000/movies/1
- Verifique o console da aplicação e veja como o parâmetro **id** é passado!
- Qual arquivo foi renderizado?

Movies#show

- Na Controladora temos acesso aos parâmetros através da variável **params** que é uma Hash já definida pelo Rails

```
def show
  id = params['id']
  @movie = Movie.find id
end
```

- Modifique o arquivo app/views/movies/show.html.rb para renderizar o título do filme na tag **h1** dessa página
- Após isso acesse a página http://localhost:3000/movies/1

Movies#show

- Adicione o seguinte código no arquivo **app/views/movies/movie.html.erb**

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

<p>
  <strong>Release date:</strong>
  <%= @movie.release_date %>
</p>

<p>
  <strong>Description:</strong>
  <%= @movie.description %>
</p>
```

Movies#show

- Acesse: http://localhost:3000/movies/-10
- Por que obtemos um erro quando passamos um id que não existe?

Movies#show

- Acesse: http://localhost:3000/movies/-10
- Por que obtemos um erro quando passamos um id que não existe?
- Nós temos que tratar essa exceção!
- Quando um id não existir, vamos retornar o código de erro **404 (NotFound)** do **HTTP** e uma página informando que o filme procurado não existe!

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

Links

- Agora que já temos as páginas individuais de cada filme, podemos adicionar links para essas páginas dentro da tabela que apresenta todos os filmes do banco: http://localhost:3000/movies
- Altera o arquivo **app/views/movies/index.html.erb** para adicionar o link:

```
<% @movies.each do |movie| %>
  <tr>
    <td><%= link_to(movie.title, "/movies/#{movie.id}") %></td>
    <td><%= movie.release_date %></td>
  </tr>
<% end %>
```

- Verifique o HTML gerado na página

ActionView::Helpers

- O Rails possui vários métodos que ajudam a criar páginas dinâmicas que podem ser chamados dentro dos nossos arquivos **html.erb**
- O **link_to** é um exemplo de helper que nos ajuda a criar links
- Esses métodos estão definidos nos módulos chamados **ActionView::Helpers** cuja documentação pode ser encontrada nesse link
- Eventualmente, estarmos utilizando **helpers** do Rails para construir nossas páginas

Movies#new

- A página **MoviesController#new** deve renderizar um formulário para criar um novo filme que ainda **não existe**
- Acesse http://localhost:3000/movies/new
- O Rails entendeu que o **new** era o parâmetro **id** que definimos na rota **show**
- Por isso, temos que definir a rota para **MoviesController#new** antes da definição que fizemos de **MoviesController#show**

```
get 'movies/index'
get 'movies/new' => "movies#new"
get 'movies/:id' => "movies#show"
```

Movies#new

- Note que a página **new** só ira renderizar um formulário para criação de um novo registro. Quem irá criar de fato o novo registro no banco será a ação **MoviesController#create**

```
def new
  @movie = Movie.new
end
```

- Para criar um formulário, precisamos utilizar o objeto **@movie** criado para escolher os campos do formulário

Formulário

- Use o código disponível nesse link para criar a página de formulário no arquivo app/views/movies/new.html.erb
- Execute a aplicação e vá até http://localhost:3000/movies/new
- Veja o código que acabamos de usar e entenda o funcionamento do mesmo
- Esse formulário é outro exemplo de **helpers** que nos ajudam a criar páginas que interagem com nossos objetos. Nesse caso o método **form_for** irá criar um formulário cujos valores serão atribuídos para o objeto **@movie** que instanciamos em **MoviesController#new**.
- Além disso, também informamos qual será a url para qual o cadastro deve ser enviado. Inspecione a página renderizada para ver o HTML gerado
- Documentação do form_for e dos helpers da data e outros helpers

FormHelpers

- Alguns dos FormHelpers estão sempre associados a um **ActiveRecord**:
  - check_box
  - fields_for
  - file_field
  - form_for
  - hidden_field
  - label
  - password_field
  - radio_button
  - text_area
  - text_field

FormTagHelpers

- FormTagHelpers não estão ligados a um ActiveRecord e possuem um _tag no nome:

```
<%= form_tag :action => 'create' do %>
  Name: <%= text_field :restaurant, :name %>
  Address: <%= text_field :restaurant, :address %>
  <%= submit_tag 'Create' %>
<% end %>
```

HTTP POST

- Preencha o formulário e envie para criar um novo filme
- Repare que vamos ter um erro de rota!
- Isso acontece, pois formulários usam o método HTTP **POST** ao invés de **GET**
- Alguém sabe dizer o motivo?
- Vamos alterar a nossa rota de **MoviesController#create** para receber **POST**

```
# get 'movies/create'
post 'movies' => "movies#create"
```

- Acesse novamente http://localhost:3000/movies/new e preencha o formulário
- Veja no **console do Rails** como os parâmetros são enviados

Movies#create

- Para criarmos um novo filme:

```
def create
  @movie = Movie.new(movie_params)
  if @movie.save
    redirect_to action: :show, id: @movie.id
  else
    render :new
  end
end

private

def movie_params
  params.require(:movie).permit(:title, :release_date, :description)
end
```

Edição de Modelos

- Seria bem útil termos uma página que nos permita editar um filme
- Assim como a criação de um filme requer duas ações, a edição também precisará:
  - Renderizar formulário (HTTP GET)
  - Atualizar o filme com os dados enviados (HTTP PUT)
- Vamos ter que criar as ações **MoviesController#edit** e **MoviesController#update**
- Crie as seguintes rotas:

```
get 'movies/:id/edit' => "movies#edit"
put 'movies/:id/' => "movies#update"
```

Movies#edit

- A ação MoviesController#edit necessita recuperar o filme que será editado para que possamos apresentar o formulário de edição já com os campos atuais do filme

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

Movies#edit

- Note que é preciso criar o arquivo **app/views/movies/edit.html.erb**
- Adicione o código desse link nesse arquivo
- Execute a aplicação e acesse http://localhost:3000/movies/1/edit
- Note que ainda precisamos de criar a ação **MoviesController#update** para atualizar o filme no banco de dados

Movies#update

- Note as diferenças entre a ação **MoviesController#create** e **MoviesController#update**

```
def update
  id = params['id']
  @movie = Movie.find(id)
  @movie.update(movie_params)
  if @movie.save
    redirect_to action: :show, id: @movie.id
  else
    render :edit, id: @movie.id
  end
rescue ActiveRecord::RecordNotFound
  render file: "#{Rails.root}/public/404.html", status: 404
end
```

Movies#update

- Lembre-se que no caso do update não precisamos ter uma View específica, apenas redirecionamos para a página no filme atualizado
- Execute a aplicação e acesse http://localhost:3000/movies/1/edit
- Edite e o filme e verifique se o registro do filme foi atualizado corretamente

Estilizando nossa aplicação

- Podemos utilizar o CSS para melhorar o visual das nossas páginas
- Portanto, copie o código desse link e cole no final do arquivo **app/assets/stylesheets/application.css**
- Execute a aplicação e veja as modificações visuais nas páginas
- Veja como utilizamos os seletores baseados em classes, ids e elementos para estilizar nossas páginas HTML

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
```

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: **@movie**, **@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**

Partials

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

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

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

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

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

<%= render partial: 'form', 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** 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

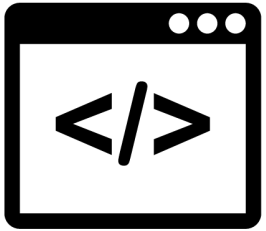
Resources em Rotas

- 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
```

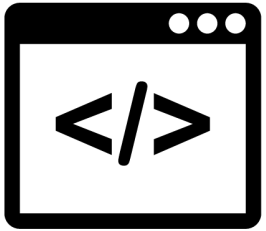
Atividades Sugeridas!

Exercício



- Continue a desenvolver a aplicação:
  - Crie um link na página do filme para permitir ao usuário editar o filme
  - Crie um link na página que lista todos os filmes para que o usuário possa criar um novo filme
  - Crie um link na página do filme para permitir ao usuário editar o filme que permita o usuário excluí-lo. Note que você terá que criar uma nova ação que responda à rota delete /movies/:id

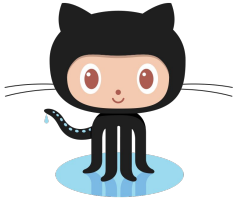
Exercício



- Os códigos gerados pelo scaffold para Actor e Director são muito parecidos. Há várias oportunidades para aplicarmos o princípio DRY para diminuir a duplicação no nosso código
- Obs:
  - Você pode usar herança nas classes Controllers também. As Controllers também herdam os filtros de suas superclasses

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!