PMR3304 - Sistemas de Informação - 04

Aula 04

Prof. Dr. Marcos de Sales Guerra Tsuzuki

30 de Agosto de 2019

PMR-EPUSP

Uploading de Arquivos

Uploading um Arquivo - Introdução

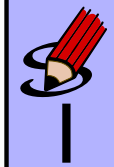
Vamos acrescentar um foto para a pessoa cadastrada.

- É necessário acrescentar um campo para realizar o upload de arquivo.
- O modelo repesentando os dados pessoais precisa manipular os dados do arquivo.
- Uma nova **migration** precisa ser acrescentada ao campo para a extensão do arquivo, pois fotos podem vir em formatos diversos.
- a **view** precisa exibir a imagem.

Uploading um Arquivo - Implementação

Vamos acrescentar um foto para a pessoa cadastrada. Será acrescentado o código abaixo no arquivo **app/view/people/_form.html.erb**:

```
<div class="field">
  <%= form.label :photo %><br />
  <%= form.file_field :photo %>
</div>
```



Para você fazer em casa

Insira o código e execute o servidor.

Uploading um Arquivo - Implementação

É possível identificar no cabeçalho o seguinte trecho de código:

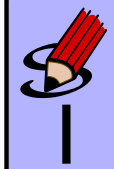
```
<form enctype="multipart/form-data" action="/people"
    method="post">
```

Ao acrescentar o tipo **enclosure** indica que o rails saberá como processar um arquivo anexo quando o corpo principal do formulário chegar.

Uploading um Arquivo - Implementação

Será necessário modificar a linha contendo o **form_for** do arquivo **app/view/people/_form.html.erb**::

```
<%= form_for(person, html: { multipart: true }) do |form| %>
```



Para você fazer em casa

Insira o código e execute o servidor.

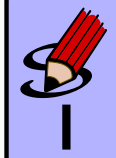
Este código suportará a inclusão e a edição de registros.

Uploading um Arquivo - Implementação

Devemos criar uma nova tabela que estará associada a **person**. Alguns bancos suportam a inclusão de imagens, mas não é o nosso caso, e as imagens serão armazenadas em um diretório do servidor. O arquivo será armazenado com o identificador do **person** e será armazenado o terminador do arquivo contendo a foto.

Vamos criar o **migration**:

```
rails generate migration add_photo_extension_to_person
```



Para você fazer em casa

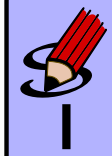
Execute o comando.

Uploading um Arquivo - Implementação

Foi criado um arquivo ***_add_photo_extension_to_person.rb**. Este arquivo será modificado para conter o código abaixo:

```
class AddPhotoExtensionToPerson < ActiveRecord::Migration[5.1]
  def change
    add_column :people, :extension, :string
  end
end
```

Este comando criará uma coluna **extension** do tipo **string** na tabela **people**. Executar o comando **rails db:migrate** para efetivar a mudança.



Para você fazer em casa

Execute o comando.

Uploading um Arquivo - Implementação

Será necessário modificar o controller **app/controller/people_controller.rb**, e acrescentar o campo **:photo**:

```
def person_params
  params.require(:person).permit(:name, :secret, :country, :email,
      :description, :can_send_email, :graduation_year, :
      body_temperature, :price, :birthday, :favorite_time, :photo)
end
```

Se esta modificação não for feita, surgirão erros como “Can’t mass-assign protected atributes: photo”.

Uploading um Arquivo - Implementação

É possível observar que o campo criado na tabela **person** possui nome **:extension** e o parâmetro do objeto **person** possui nome **:photo**.

Esta diferença possui o objetivo de evitar que o rails processe automaticamente. Será utilizado um mecanismo que garante que o código para armazenar a foto ocorra após a completa validação dos dados. O código abaixo deve ser acrescentado ao arquivo **app/models/person.rb**.

```
after_save :store_photo
```

Uploading um Arquivo - Implementação

O método **:store_photo** precisa ser criado:

```
private
  def store_photo
    if @file_data
      FileUtils.mkdir_p PHOTO_STORE
      File.open(photo_filename, wb) do |f|
        f.write(@file_data.read)
      end
      @file_data = nil
    end
  end
```

O método **:store_photo** é **private** indicando que está acessível apenas para a classe **Person**.

Uploading um Arquivo - Implementação

Se possui dado para ser armazenado, então deverá prosseguir (isto é o resultado do primeiro **if**. Em seguida o diretório será criado, caso ele já exista não deverá ser um problema. As linhas seguintes abrem um arquivo de nome **photo_filename** e escrevem os para ele. Em seguida, os dados de **@file_data** são zerados para evitar que eles sejam salvos novamente.

```
private
  def store_photo
    if @file_data
      FileUtils.mkdir_p PHOTO_STORE
      File.open(photo_filename, wb) do |f|
        f.write(@file_data.read)
      end
      @file_data = nil
    end
  end
```

Uploading um Arquivo - Implementação

Antes de salvar o arquivo é necessário processar a atribuição do atributo **:photo**. A listagem abaixo utiliza o método de atribuição que é executado quando o rails transfere o dado **file_data** vindo do HTML para o atributo **photo** de **Person**. O dado vindo do HTML é atribuído à variável **@file_data** (que é visível para o método **store_photo**.

```
def photo=(file_data)
  unless file_data.blank?
    @file_data = file_data
    self.extension = file_data.original_filename.split('.').last.
        downcase
  end
end
```

Uploading um Arquivo - Implementação

Finalmente, a extensão do arquivo é recuperada e salva no banco de dados.

```
def photo=(file_data)
  unless file_data.blank?
    @file_data = file_data
    self.extension = file_data.original_filename.split('.').last.
        downcase
  end
end
```

Uploading um Arquivo - Implementação

Os últimos passos processam o nome do arquivo:

```
# Como ele utilizara arquivos estaticos, o diretorio
# PHOTO_STORE sera criado na pasta "public".
PHOTO_STORE = File.join Rails.root, 'public', 'photo_store'
# retorna o path utilizado no HTML para a imagem
def photo_path
  "/photo_store/#{id}.#{extension}"
end
# onde escrever o arquivo imagem
def photo_filename
  File.join PHOTO_STORE, "#{id}.#{extension}"
end
# se o arquivo de foto existe, entao temos foto
def has_photo?
  File.exists? photo_filename
end
```

Uploading um Arquivo - Implementação

Pergunta, por que utilizar o **id** como nome do arquivo imagem a ser salvo?

Pergunta, o que aconteceria se nomes de arquivos coincidentes fossem utilizados?

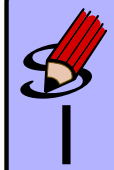
Uploading um Arquivo - Implementação

Finalmente, resta ajustar a **view** em **app/views/people/show.html.erb**

```
<p>
  <strong>Photo:</strong>
    <% if @person.has_photo? %>
      <%= image_tag @person.photo_path %>
    <% else %>
      No photo.
    <% end %>
</p>
```

Uploading um Arquivo - Implementação

Podemos executar e servidor e testar o código desenvolvido. Observe que o campo secreto está sendo exibido, isto será resolvido no futuro.



Para você fazer em casa

Teste o código desenvolvido.

Utilizando Form Builders

Utilizando Form Builders - Introdução

Será descrito como utilizar **form builders**, evitando o resultado do scaffold. Já vimos como suportar diversos tipos de dados.

Na aula passada vimos como criar um **helper** para criar uma lista com **combo boxes** ou **radio buttons**. O recurso exato será definido de acordo com o número de elementos presentes na lista.

Vamos criar um novo arquivo **app/helpers/tidy_form_builder.rb**. Será criado um método para criação específica de uma lista de países.

O arquivo **app/helpers/tidy_form_builder.rb** listado abaixo será criado.

```
class TidyFormBuilder < ActionView::Helpers::FormBuilder
  def country_select(method, options={}, html_options={})
    select(method, [['Canada', 'Canada'],
                    ['Mexico', 'Mexico'],
                    ['United Kingdom', 'UK'],
                    ['United States of America', 'USA']],
                   options, html_options)
  end
end
```

A classe **TidyFormBuilder** herda propriedades da Classe **ActionView::Helpers::FormBuilder**. Os métodos definidos aqui estarão disponíveis às **views**.

É necessário conectar a **view** para referenciar o **TidyFormBuilder**. Isto será feito modificando a linha contendo o **form_for** no arquivo **app/views/people/_form.html.erb**.

```
<%= form_for(person, html: { multipart: true }, builder:
      TidyFormBuilder) do |form| %>
```

Agora, a construção do combo box deve ser modificada para:

```
<div class="field">
  <%= form.label :country %><br>
  <%= form.country_select :country %>
</div>
```

Esta ação remove código do **viewer** e transfere para o interior.

Vamos modificar também a criação dos demais campos:

```
<div class="field">
  <%= form.label :name %>
  <%= form.text_field :name %>
</div>
```

Verificando a documentação do rails é possível observar que existe um método que é utilizado para criar os diversos tipos de recursos. Aqui nova caso é para o tipo **text**:

```
def text_field(method, options={})
  ..
end
```

O método a ser acrescentado ao arquivo **app/helpers/tidy_form_builder.rb** é:

```
def text_field(method, options={})
  label = label_for(method, options) + super(method, options)
end
```

A chamada ao método **super** indica a chamada ao método original **text_field**. O método **label_for** também deve ser criado:

```
private
def label_for(method, options={})
  (label(options.delete(:label) || method).safe_concat("<br />"))
end
```

```
private
def label_for(method, options={})
  (label(options.delete(:label) || method).safe_concat("<br />"))
end
```

Este método parece um pouco estranho, acionando o método **delete**. Esta chamada executa dois principais efeitos: 1. remove o valor **:label** e ele não será transferido para o método **super**; e, 2. ele também retorna o valor do parâmetro **:label**, caso ele exista. Caso o valor de **:label não exista**, o **metod** será acionado com o valor padrão (o nome interno do campo).

A chamada para criar o campo no arquivo **_form.html.erb** ficará muito mais simples. Originalmente, como abaixo:

```
<div class="field">
  <%= form.label :name %>
  <%= form.text_field :name %>
</div>
```

E ficará como:

```
<div class="field">
  <%= form.text_field :name %>
</div>
```

Utilizando Form Builders - Implementação

Os demais métodos, com assinaturas mais complexas, precisam de mais código, mas utilizam a mesma lógica:

```
def date_select(method, options = {}, html_options = {})
  label_for(method, options) + super(method, options, html_options)
end
def select(method, choices, options = {}, html_options = {})
   label_for(method, options) + super(method, choices, options, html_options)
end
def time_select(method, options = {}, html_options = {})
  label_for(method, options) + super(method, options, html_options)
end
def check_box(method, options = {}, checked_value = "1", unchecked_value = "0")
  label_for(method, options) + super(method, options, checked_value,
        unchecked_value)
end
```

Integrando Form Builders e Styles

Será descrito como utilizar **form builders** para acrescentar uma opção **:required** nos campos do formulário. O arquivo **app/views/people/_form.html.erb** ficará na forma:

```
<div class="field">
  <%= f.text_field :name, required: true %>
</div>
<div class="field">
  <%= f.password_field :secret, required: true %>
</div>
<div class="field">
  <%= f.country_select :country, required: true %>
</div>
```

Isto será a conexão com o estilo definido na CSS.

Utilizando Form Builders - Implementação

O arquivo **app/helpers/wrapping_tidy_form_builder.rb** listado abaixo será criado.

```
class WrappingTidyFormBuilder < ActionView::Helpers::FormBuilder


end
```

Vamos incluir um método **private** no arquivo recém criado:

```
def wrap_field(text, options={})
  field_class = "field"
  if options[:required]
    field_class = "field required"
  end
  "<div class='#{field_class}'>".html_safe.safe_concat(text).safe_concat("
      </div>")
end
```

Todo o novo código ficará neste arquivo.

O método atual está na forma:

```
def text_field(method, options={})
  label = label_for(method, options) + super(method, options)
end
```

E deverá ficar como:

```
def text_field(method, options={})
  wrap_field(label = label_for(method, options) + super(method, options)
      , options)
end
```

Este método deverá ser colocado no novo arquivo recém criado.

Ainda lembrando que alguns browsers não aceitam estilos (como o *Lynx*), vamos inserir um "*" no label para indicar que o campo precisa ser fornecido. Para isto, o método **label_for** será modificado:

```
def label_for(method, options={})
  label = label(options.delete(:label) || method)
  if options[:required]
    label.safe_concat(" <span class='required_mark'>*</span>")
  end
  label.safe_concat("<br />")
end
```

Este método deverá ser colocado no novo arquivo recém criado, em **private**.

A classe **WrappingTidyFormBuilder** herda propriedades da Classe **ActionView::Helpers::FormBuilder**. Os métodos definidos aqui estarão disponíveis às **views**.

É necessário conectar a **view** para referenciar o **WrappingTidyFormBuilder**. Isto será feito modificando a linha contendo o **form_for** no arquivo **app/views/people/_form.html.erb**.

```
<%= form_for(@person, html: { multipart: true }, builder:
    WrappingTidyFormBuilder) do |form| %>
```

O último trecho de código necessário é o CSS contido no arquivo **app/assets/stylesheets/people.scss**.

```
div.field {
        margin-top: 0.5em; margin-bottom: 0.5em; padding-left: 10px;
}
div.field label {
        font-weight: bold;
}
div.field span.required_mark {
        font-weight: bold; color: red;
}
div.required {
        padding-left: 6px; border-left: 4px solid #DD0;
}
```

Perguntas para Testar o Conhecimento

Perguntas para Testar o Conhecimento

- Qual é a diferença entre **<%** e **<%=**?
- Quanto de lógica colocamos nos arquivos com extensão **erb**?
- O que é **PUMA** e porque ele vem incluído com o Rails?
- Como enviamos dados de uma **view** - **app/views/*** para o **layout** - **app/views/layouts/***?
- Em que arquivo devemos inserir código para que todos os **controllers** tenham acesso?
- Como verificar se um campo de um objeto está em branco?

The End!