PMR3304 - Sistemas de Informação - 03

Aula 03

Prof. Dr. Marcos de Sales Guerra Tsuzuki

23 de Agosto de 2019

PMR-EPUSP

Models com Forms

Models com Forms - Introdução

Na aula anterior demonstramos como criar aplicações simples utilizando scaffoldings, mas um aspecto importante do Rails scaffolding é que eles não devem ser permanentes, e devem aproveitados no desenvolvimento de uma aplicação.

Para demonstrar um conjunto completo de possibilidades em HTML, a aplicação precisará suportar mais que um campo de dados e precisará suportar uma boa variedade de tipos de dados. Os tipos de dados suportados pelo *ActiveRecord* são: **:string**, **:text**, **:integer**, **:float**, **:decimal**, **:datetime**, **:timestamp**, **:time**, **:date**, **:binary** e **:boolean**.

Para demonstrar um conjunto completo de possibilidades em HTML, a aplicação precisará suportar mais que um campo de dados e precisará suportar uma boa variedade de tipos de dados. Os tipos de dados suportados pelo *ActiveRecord* são: **:string**, **:text**, **:integer**, **:float**, **:decimal**, **:datetime**, **:timestamp**, **:time**, **:date**, **:binary** e **:boolean**.

O HTML oferece uma variedade de formas de entrada, mas o mapeamento não ocorre na forma de um para um. As possíveis são: campos de texto (normais, escondidos, e senha), áreas de texto, checkboxes, radio buttons, listas de seleção, upload de arquivos, e outros botões.

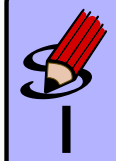
Como exemplo para o desenvolvimento, os seguintes campos serão criados e as seguintes formas de entrada serão utilizadas:

- Strings regulares: Nome, secreto país, e e-mail;
- Strings longas: descrição;
- Boolean: “podemos enviar um e-mail?”
- Números: ano de graduação, um número real para temperatura do corpo e um decimal para preço;
- Datas e Horas: data de nascimento e um horário favorito do dia.

Vamos iniciar uma nova aplicação: **rails new guestbook**.

Vá para o diretório recém criado: **cd guestbook**.

Em seguida execute o seguinte comando: **rails generate scaffold Person name:string secret:string country:string email:string description:text can_send_email:boolean graduation_year:integer body_temperature:float price:decimal birthday:date favorite_time:time**.



Para você fazer em casa

Execute os comandos descritos.

Foi criado um arquivo de migration, observe que Person é o singular e People é o plural.

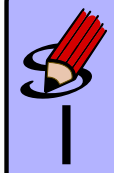
Models com Forms - Criando HTML com Scaffold

```ruby
class CreatePeople < ActiveRecord::Migration[5.1]
  def change
    create_table :people do |t|
      t.string :name
      t.string :secret
      t.string :country
      t.string :email
      t.text :description
      t.boolean :can_send_email
      t.integer :graduation_year
      t.float :body_temperature
      t.decimal :price
      t.date :birthday
      t.time :favorite_time
      t.timestamps
    end
  end
end
```

Devemos executar o migration para que a mudança surta efeito (criação da tabela): **rails db:migrate**.

O modelo criado pelo Rails é simples:

```
class Person < ApplicationRecord
end
```



Para você fazer em casa

Execute o comando migrate.

Vamos observar o conteúdo do arquivo **app/views/people/new.html.erb**, que é o form para visualizar um novo registro de *people*.

```
<h1>New Person</h1>

<%= render 'form', person: @person %>

<%= link_to 'Back', people_path %>
```

O comando **render 'form'. person: @person**, indica que um outro arquivo deve ser acessado: **_form.html.erb**. Isto deriva da convenção de nomes adotada pelo Rails. Esta forma de inclusão de arquivo por referência é utilizada para evitar repetições.

Models com Forms - Criando HTML com Scaffold

Vamos observar o conteúdo do arquivo **_form.html.erb**, de início apenas o sua parte superior:

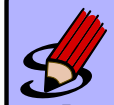
```
<%= form_for(@person) do |form| %>
  <% if person.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(person.errors.count, "error") %> prohibited this
          person from being saved:</h2>
      <ul>
      <% person.errors.full_messages.each do |message| %>
        <li><%= message %></li>
      <% end %>
      </ul>
    </div>
  <% end %>
  ...
<% end %>
```

Esta parte do arquivo exibe todos os erros obtidos por validação nos campos de dados. Alguns podem ter obtido **form_with** ao invés de **form_for**. Foi criada uma variável **form** (em alguns casos, poderá ser **f**).

Continuando o conteúdo do arquivo **_form.html.erb**:

```
<%= form_with(model: person, local: true) do |form| %>
  ...
    <%= form.text_area :description, id: :person_description %>
  ...
    <%= form.date_select :birthday, id: :person_birthday %>
  ...
    <%= form.check_box :can_send_email, id: :person_can_send_email %>
  ...
    <%= form.time_select :favorite_time, id: :person_favorite_time %>

<% end %>
```

A descrição será fornecida por um conjunto de texto longo. O campo “Can send e-mail” será fornecido por um check box. O aniversário será fornecido pela seleção de data e a hora preferida será fornecida como um horário.



Para você fazer em casa

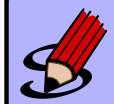
Use **fom_for**. Execute o servidor **rails server**, e em seguida observe no browser a saída: `http://localhost:3000/people/new`.

O **form_for** cria um cabeçalho adequado para o HTML conforme o conteúdo da variável **@person**. Caso ela esteja vazia, será preparada para a criação de uma nova *person*. Isto pode ser observado no browser, ao solicitar o source code.

```
<form class="new_person" id="new_person" action="/people" accept-
      charset="UTF-8" method="post"><input name="utf8" type="hidden"
      value="&#x2713;" /><input type="hidden" name="authenticity_token"
      value="hneWmbZIceoOm8g4GgbclMZ9WhL1szqrp+2qnL/5+0
      NptTOtAHICbx1/s/i728o93Sc+Cv7pKEeaCRGNlGgwug==" />
```

Caso seja fornecido um conteúdo em **@person**, o **form_for** criará um formulário para edição.

```
<form class="edit_person" id="edit_person_1" action="/people/1" accept-
    charset="UTF-8" method="post"><input name="utf8" type="hidden"
    value="&#x2713;" /><input type="hidden" name="_method" value="
    patch" /><input type="hidden" name="authenticity_token" value="n+
    W0X2/8IKf53TpWiUEu8TguN5wFzgOiO7v+
    TbxzBFtoYhEy54x4E0P4PdlvXjkiCe7Ae3Buh3RFQsLX+3+nOw==" />
```



Para você fazer em casa

Verifique o código em seu browser. Em seguinda, insira um registro, e logo em seguida edite o registro. Verifique o código em seu browser novamente, e identifique os dois casos acima.

Models com Forms - Criando campos e área de texto

O Rails scaffolding incluiu apenas dois comandos distintos para a criação de texto: **text_field** e **text_area**. O comando **text_field** criará um campo com uma única linha:

```
<%= form.text_field :name, id: :person_name %>
```

O comando **text_area** criará um campo com várias linhas:

```
<%= form.text_area :description, id: :person_description %>
```

Ambos possuem um campo **id** que é útil para o caso de stylesheets. É possível definir a dimensão do campo:

```
<%= form.text_area :description, id: :person_description, cols:30, rows:
    10 %>
```

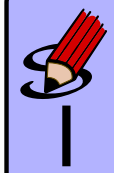
O Rails permite a criação de campos escondidos (ou *hidden*):

```
<%= form.hidden_field :graduation_year %>
```

Este campo estará presente na página, mas não estará visível.
Outro tipo de campo está reservado para senhas:

```
<%= form.password_field :secret %>
```

As entradas para este campo serão exibidas com asterisco, escondendo o valor da entrada digitada.



Para você fazer em casa

Teste as modificações efetuadas.

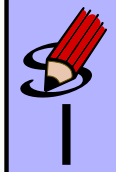
A criação de labels é simples. Observe a saída de um scaffolding.

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

Este campo possui um pouco de material extra de informação que o browser utilizará para realizar a associação.

```
<div class="field">
  <label for="person_name">Name </label><br>
  <input type="text" name="person[name]" id="person_name" />
</div>
```

Se você clicar sobre a palavra “Name” o foco irá para o campo para digitar o campo imediatamente abaixo.



Para você fazer em casa

Teste a afirmação.

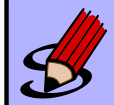
Models com Forms - Criando Labels

O label pode ser um texto, bastando acrescentar um campo como em:

```
<div class="field">
  <%= form.label :name, 'Your name' %>
  <%= form.text_field :name, id: :person_name %>
</div>
```

Isto criará um código HTML como descrito abaixo:

```
<div class="field">
  <label for="person_name">Your Name</label><br>
  <input type="text" name="person[name]" id="person_name" />
</div>
```



Para você fazer em casa

Teste as modificações efetuadas.

Checkboxes são simples.

```
<%= form.check_box :can_send_email %>
```

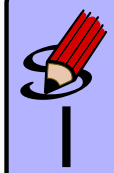
Isto criará um código HTML como descrito abaixo:

```
<div class="field">
  <label for="person_can_send_email">Can send email</label><br>
  <input name="person[can_send_email]"
    type="hidden" value="0" />
  <input type="checkbox" value="1" name="person[can_send_email]"
    id="person_can_send_email" />
</div>
```

Models com Forms - Criando Checkboxes

```
<div class="field">
  <label for="person_can_send_email">Can send email</label><br>
  <input name="person[can_send_email]"
    type="hidden" value="0" />
  <input type="checkbox" value="1" name="person[can_send_email]"
    id="person_can_send_email" />
</div>
```

Porque existe um segundo input to tipo hidden? É o valor default para o caso em que o checkbox não foi selecionado. A especificação HTML informa que checkbox não selecionado não são enviados.



Para você fazer em casa

Verifique a afirmação.

Models com Forms - Criando Checkboxes

Como o caso do label, é possível fornecer um parâmetro adicional, por exemplo uma classe para estilo CSS.

```
<%= form.check_box :can_send_email, class: email %>
```

Isto criará um código HTML como descrito abaixo:

```
<div class="field">
  <label for="person_can_send_email">Can send email</label><br>
  <input name="person[can_send_email]"
    type="hidden" value="0" />
  <input class="email" type="checkbox" value="1" name="person[
      can_send_email]"
    id="person_can_send_email" />
</div>
```

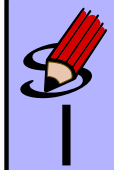
Models com Forms - Criando Checkboxes

Também é possível especificar o valor de retorno, ao invés de ser apenas 0 ou 1.

```
<%= form.check_box :can_send_email, {class: 'email'}, "yes", "no" %>
```

Isto criará um código HTML como descrito abaixo:

```
<div class="field">
  <label for="person_can_send_email">Can send email</label><br>
  <input name="person[can_send_email]"
    type="hidden" *value="no" />
  <input class="email" type="checkbox" value="yes" name="person[
      can_send_email]"
    id="person_can_send_email" />
</div>
```



Para você fazer em casa

Verifique a afirmação.

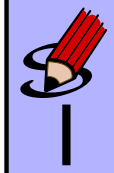
A criação de radio buttons não é automática a partir do scaffold. Radio buttons são utilizados para selecionar a partir de um pequeno conjunto de opções.

```
<fieldset>
  <legend>Country</legend>
    <%= form.radio_button :country, 'USA' %>
    <%= form.label "person_country_usa", "USA" %><br>
    <%= form.radio_button :country, 'Canada' %>
    <%= form.label "person_country_canada", "Canada" %><br>
    <%= form.radio_button :country, 'Mexico' %>
    <%= form.label "person_country_mexico", "Mexico" %><br>
</fieldset>
```

Isto criará um código HTML como descrito abaixo:

```
<fieldset>
  <legend>Country</legend>
    <input id="person_country_usa" name="person[country]"
      type="radio" value="USA"/>
    <label for="person_country_usa">USA</label><br>
    <input id="person_country_canada" name="person[country]"
      type="radio" value="Canada"/>
    <label for="person_country_canada">Canada</label><br>
    <input id="person_country_mexico" name="person[country]"
      type="radio" value="Mexico"/>
    <label for="person_country_mexico">Mexico</label><br>
</fieldset>
```

Caso o valor do campo tenha algum dos valores de país, também seria acrescentado um código indicando **checked="checked"**.



Para você fazer em casa

Teste as modificações efetuadas.

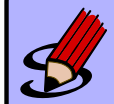
Models com Forms - Criando Radio Buttons

Criando radio buttons a partir de uma lista hash, ordenando os itens.

```
<% nations = { 'United States of America' => 'USA',
               'Canada' => 'Canada',
               'Mexico' => 'Mexico',
               'United Kingdom' => 'UK' }%>

<fieldset>
  <legend>Country</legend>
  <% list = nation.sort list.each do |x| %>
    <%= form.radio_button :country, x[1] %>
    <%= label for="<% ("person_country_" + x[1].downcase %>"
    <%= x[0] %><\label><br>
  <% end %>
</fieldset>
```

Isto criará um código HTML como descrito abaixo:

```
<fieldset>
  <legend>Country</legend>
    <input type="radio" value="Canada"
      name="person[country]" id="person_country_canada"/>
    <label for="person_country_canada">Canada</label><br>
    <input type="radio" value="Mexico"
      name="person[country]" id="person_country_mexico"/>
    <label for="person_country_mexico">Mexico</label><br>
    <input type="radio" value="UK"
      name="person[country]" id="person_country_uk"/>
    <label for="person_countryuk">United Kingdom</label><br>
    <input type="radio" value="USA"
      name="person[country]" id="person_country_usa"/>
    <label for="person_country_usa">United States of America
      </label><br>
</fieldset>
```

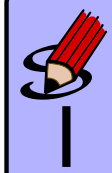


Para você fazer em casa

Teste as modificações efetuadas.

Listas de seleção (ou combo boxes) são muito semelhantes a radio buttons. Utilizando um vetor de strings

```
<%= form.label :country %><br>
<%= form.select :country, [ ['Canada', 'Canada'],
                            ['Mexico', 'Mexico'],
                            ['United Kingdom', 'UK],
                            ['United States of America', 'USA'] ]%>
```



Para você fazer em casa

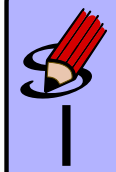
Teste as modificações efetuadas.

Isto criará um código HTML como descrito abaixo:

```
<div class="field">
<fieldset>
  <label for="person_country">Country</label><br>
  <select name="person[country]" id="person_country">
    <option value="Canada">Canada</option>
    <option value="Mexico">Mexico</option>
    <option value="UK">United Kingdom</option>
    <option value="USA">United States of America</option></select>
</fieldset>
</div>
```

É possível selecionar uma opção padrão incluindo a opção **selected**.

```
<%= form.label :country %><br>
<%= form.select :country, [ ['Canada', 'Canada'],
                            ['Mexico', 'Mexico'],
                            ['United Kingdom', 'UK],
                            ['United States of America', 'USA'] ],
                            :selected => 'USA'%>
```



Para você fazer em casa

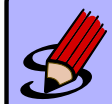
Teste as modificações efetuadas.

Isto criará um código HTML como descrito abaixo:

```
<fieldset>
  <label for="person_country">Country</label><br>
  <select name="person[country]" id="person_country">
    <option value="Canada">Canada</option>
    <option value="Mexico">Mexico</option>
    <option value="UK">United Kingdom</option>
    <option selected="selected" value="USA">United States of America</
        option></select>
</fieldset>
```

Criando listas de seleção a partir de uma lista hash, ordenando os itens.

```
<% nations = { 'United States of America' => 'USA',
               'Canada' => 'Canada',
               'Mexico' => 'Mexico',
               'United Kingdom' => 'UK' }%>

<p>
  <%= Form.label :country %><br>
  <% list = nations.sort %>
  <%= form.select :country, list %>
</p>
```



Para você fazer em casa

Teste as modificações efetuadas.

Models com Forms - Datas e Horários

O rails suporta a entrada de datas e horários.

```
<div class="field">
  <%= form.label :birthday %>
  <%= form.date_select :birthday, id: :person_birthday %>
</div>

<div class="field">
  <%= form.label :favorite_time %>
  <%= form.time_select :favorite_time, id: :person_favorite_time %>
</div>
```

Além dos métodos **date_select** e **time_select**, existem também **:start_year**, **:end_year**, **:use_month_numbers**, **:discard_day**, **:disabled**, **:include_blank**, **:include_seconds** e **:order**.

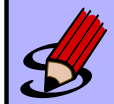
A criação de helpers permite que menos repetição ocorra (similar ao DRY), que o código fique mais legível (observando que o mesmo código é acionado de diferentes locais, facilitando a sua interpretação), mais consistência (o mesmo código uilizado em lugares disintos raramente permanecerá o mesmo) e compartilhamento em views (múliplas views podem acessar o mesmo helper).

Considere a seguinte criação de radio buttons:

```
<% nations = { 'United States of America' => 'USA',
               'Canada' => 'Canada',
               'Mexico' => 'Mexico',
               'United Kingdom' => 'UK' }%>

<%= buttons(:person, :country, nations) %>
```

O método buttons deve ser inserido no arquivo **people_helper.rb**.

```
module PeopleHelper
def buttons(model_name, target_property, button_source)
  html=''
  list = button_source.sort
  html << '<fieldset><legend>Country</legend>'
  list.each do |x|
    html << radio_button(model_name, target_property, x[1])
    html << (x[0])
    html << '<br>'
  end
  html << '</fieldset>'
  return html.html_safe
end
end
```



Para você fazer em casa

Teste as modificações efetuadas.

Models com Forms - Helpers

```
def buttons_select(model_name, target_property, button_source)
  html=''
  list = button_source.sort
  if list.length < 4
    html << '<fieldset><legend>Country</legend>'
    list.each do |x|
      html << radio_button(model_name, target_property, x[1])
      html << (x[0])
      html << '<br>'
    end
    html << '</fieldset>'
  else
    html << '<label for="person_country">Country</label><br />'
    html << select(model_name, target_property, list)
  end
  return html.html_safe
end
```

Para você fazer em casa

Teste as modificações efetuadas.

Validação

O arquivo original **person.rb** foi criado conforme abaixo:

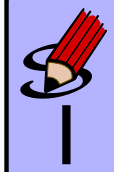
```
class Person < ApplicationRecord
end
```

Ao invés de testar se um campo está presente, você pode deixar que o rails faça isto por você.

```
class Person < ApplicationRecord
  # the name is mandatory
  validates_presence_of :name
end
```

Ao tentar salvar um nome em branco aparecerá uma mensagem de erro, conforme o detalhe gerado em HTML abaixo:

```
<div id="error_explanation">
      <h2>1 error prohibited this person from being saved:</h2>
      <ul>
        <li>Name can't be blank</li>
      </ul>
    </div>
```

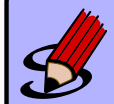


Para você fazer em casa

Verifique a ocorrência do caso descrito.

A mensagem de erro surgiu devido ao trecho de código existente na *view*:

```
<% if person.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(person.errors.count, "error") %>
      prohibited this person from being saved:</h2>
    <ul>
    <% person.errors.full_messages.each do |message| %>
      <li><%= message %></li>
    <% end %>
    </ul>
  </div>
<% end %>
```



Para você fazer em casa

Qual arquivo possui este trecho de código?

O controller também participou na exibição do erro:

```ruby
# POST /people
# POST /people.json
def create
  @person = Person.new(person_params)

  respond_to do |format|
    if @person.save
      format.html { redirect_to @person, notice: 'Person was successfully created
          .' }
      format.json { render :show, status: :created, location: @person }
    else
      format.html { render :new }
      format.json { render json: @person.errors, status: :unprocessable_entity }
    end
  end
end
```

Models com Forms - Validação

```ruby
# POST /people
# POST /people.json
def create
  @person = Person.new(person_params)

  respond_to do |format|
    if @person.save
      format.html { redirect_to @person, notice: 'Person was successfully created
          .' }
      format.json { render :show, status: :created, location: @person }
    else
      format.html { render :new }
      format.json { render json: @person.errors, status: :unprocessable_entity }
    end
  end
end
```

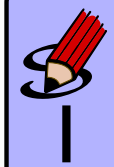
Se um erro surgir, **@person.save** falhará, retornando **false**.

Customização da mensagem de erro

```
class Person < ApplicationRecord
  # the name is mandatory
  validates_presence_of :name
  validates_presence_of :secret,
  message: "must provided so we can recognize you"
  # password length 6–24
  validates_length_of :secret, in: 6..24


  ...
end
```

Models com Forms - Validação

```
class Person < ApplicationRecord
  ...
  validates_format_of :secret, with: /[0–9]/,
    message: "must contain at least one number"
  # at least one upper case
  validates_format_of :secret, with: /[A–Z]/,
    message: "must contain at least one upper case"
  # at least one lower case
  validates_format_of :secret, with: /[a–z]/,
    message: "must contain at least one lower case"
end
```

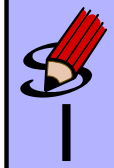


Para você fazer em casa

Confirme os erros que aparecem.

Limitando as escolhas:

```
validates_inclusion_of :country, in: ['Canada','UK'],
    message: "must be one of Canada, UK"
```



Para você fazer em casa

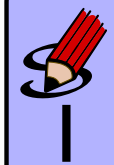
Confirme os erros que aparecem.

Verificando o e-mail:

```
# email should read like an email address
validates_format_of :email,
   with: /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\Z/i,
        message: "doesn't look like a proper email address"
```

O ruby permite que expressões sejam processadas facilmente.

Veremos um pouco mais sobre isto em outra aula.



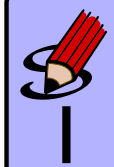
Para você fazer em casa

Confirme os erros que aparecem.

A unicidade de um campo pode ser verificada:

```
validates_uniqueness_of :email, case_sensitive: false,
  message: "has already been entered"

validates_uniqueness_of :email, case_sensitive: false,
  scope: [:name,:secret],
  message: "has already been entered"
```

Estamos com duas possibilidades, um campo que possui dados sem repetição, ou um conjunto de campos cuja combinação não deve ser repetida (campos múltiplos). As duas opções acima não podem existir em conjunto.

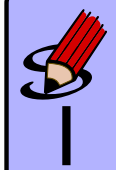


Para você fazer em casa

Confirme os erros que aparecem.

Validação numérica:

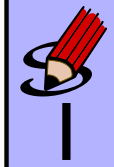
```
validates_numericality_of :graduation_year, allow_nil: true,
  greater_than: 1920, less_than_or_equal_to: Time.now.year,
      only_integer: true

validates_numericality_of :body_temperature, allow_nil: true,
  greater_than_or_equal_to: 60, less_than_or_equal_to: 130,
      only_integer: false

validates_numericality_of :price, allow_nil: true,
      only_integer: false
```



Para você fazer em casa Confirme os erros que aparecem.

Calendário

```
validates_inclusion_of :brithday,
  in: Date.civil(1900, 1, 1) .. Date.today,
      message: "must be between January 1st, 1900 and today"
```



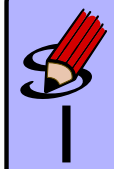
Para você fazer em casa

Confirme os erros que aparecem.

Verifique apenas se:

```
validates_presence_of :description, if: :require_description_presence?

def require_description_presence?
  self.can_send_email
end
```

É possível observar que o metodo **require_description_presence?** termina com “?” indicando que ele retorna uma variável Booleana. Ele aparentemente não retorna nada, mas o rails retorna o último valor, assim **self.can_send_email** é retornado. O **self** não precisaria estar aqui, e foi colocado apenas para facilitar a leitura.



Para você fazer em casa

Confirme os erros que aparecem.

Exemplo de validação customizada para a contagem de palavras:

```
def description_length_words
  unless self.description.blank? then
    num_words = self.description.split.length
    if num_words < 5 then
      self.errors.add(:description, "must be at least 5 words long")
    elsif num_words > 50 then
      self.errors.add(:description, "must be at most 50 words long")
    end
  end
end
```

Para você fazer em casa

Teste esta rotina com algum dos campos.

The End!