Programação de Sistemas para Internet

Jalerson Lima

Curso técnico nível médio subsequente em Informática para Internet

Programação de Sistemas para Internet

Jalerson Lima

Curso técnico nível médio subsequente em Informática para Internet

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte.

Natal-RN

2022

Presidente da República
Luiz Inácio Lula da Silva

Ministro da Educação
Camilo Sobreira De Santana

Secretário de Educação
Profissional e Tecnológica
Getúlio Marques Ferreira

Reitor
José Arnóbio de Araújo Filho

Pró-Reitor de Pesquisa e Inovação
Avelino Aldo de Lima Neto

Caderno elaborado em parceria entre o Instituto Federal de Educação, Ciência e Tecnologia e o Sistema Escola Técnica Aberta do Brasil – e-Tec Brasil.

Comitê Editorial da Diretoria de Educacão a Distância e Tecnologias Educacionais - Campus Avançado Natal Zona Leste/IFRN

Presidente
Wagner de Oliveira

Membros
José Roberto Oliveira dos Santos
Albérico Teixeira Canario de Souza
Glácio Gley Menezes de Souza
Wagner Ramos Campos

Suplentes
João Moreno Vilas Boas de Souza Silva
Allen Gardel Dantas de Luna
Josenildo Rufino da Costa
Leonardo dos Santos Feitoza

Equipe | Produção de Material Didático

Equipe de Elaboração
Cognitum

Coordenação Institucional
COTED

Projeto Gráfico
Eduardo Meneses e Fábio Brumana

Revisão ABNT
Francisco de Assis Noberto

Revisão linguística
Maria Tânia Florentino de Sena Nascimento

Revisão Pedagógica
Kalina Alessandra Rodrigues de Paiva

Revisão tipográfica
Wagner Ramos Campos

Diagramação
Alexandre Baccelli
Georgio Nascimento
Joaci De Paula
Leonardo dos Santos Feitoza
Luã Santos
Luanna Canuto
Rômulo França

Ficha catalográfica

l43 Lima, Jalerson.
Programação de Sistemas para Internet. / Organização: Jalerson Lima, -- 2022.
122 f. ; 30cm.

Guia (Curso técnico nível médio subsequente em Informática para Internet). Campus Zona Leste -
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte, Natal (RN), 2022.

ISBN: 978-65-84831-47-6

1. Educação 2. Guia 3. Curso Técnico. 4. Médio Subsequente. I. Título.

CDU: 004.4’27

Catalogação na publicação pelo Bibliotecário-Documentalista
Ezequiel da Costa Soares Neto CRB15/613
Biblioteca Sebastião Názaro do Nascimento (BSNN) / Campus Natal-Zona Leste – IFRN

Sumário

Aula 1 - Introdução ao Framework Ruby on Rails 7

Aula 2 - Ambiente de Desenvolvimento 17

Aula 3 - Criando seu primeiro projeto 23

Aula 4 - Criando o Scaffold Mensagem 31

Aula 5 - O Padrão MVC 37

Aula 6 - Acessando o Banco de Dados 49

Aula 7 - Melhorando a Apresentação - Parte I 57

Aula 8 - Melhorando a Apresentação - Parte II 67

Aula 9 - DRY - *Don't Repeat Yourself* 77

Aula 10 - Validação de Dados 81

Aula 11 - Associações em Rails 87

Aula 12 - Desenvolvendo e associando os modelos 95

Aula 13 - Desenvolvendo o controlador pessoas 101

Aula 14 - Desenvolvendo o formulário de cadastro de pessoas 111

Aula 1 - Introdução ao Framework Ruby on Rails

Objetivos

Ao final desta aula, você deverá ser capaz de:

Compreender o padrão arquitetural *Model-View-Controller*;

Entender alguns princípios, práticas e técnicas do *frameworks;*

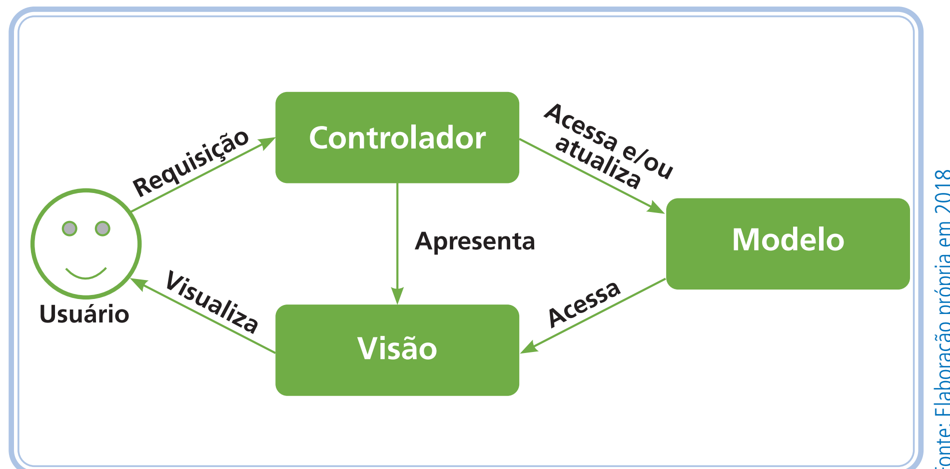
Compreender os conceitos de rotas, migrações, Gems e métodos HTTP.

Introdução

Ruby on Rails (ou simplesmente Rails) é um framework construído em Ruby para desenvolvimento de aplicações Web. Mas o que é um framework? Segundo o TechTerms (2018), um framework é uma plataforma para desenvolvimento de aplicações de software. Ele provê uma base para que desenvolvedores de software possam construir aplicações para uma plataforma específica. Ou seja, um framework é uma estrutura de software que fornece recursos e funcionalidades genéricas que facilitam o desenvolvimento de aplicações.

As aplicações Web desenvolvidas em Rails são baseadas no padrão arquitetural Model-View-Controller (MVC). Esse padrão divide a aplicação em três partes interconectadas: o Model (Modelo), a View (Visão) e o Controller (Controlador). O MVC é um padrão muito famoso usado por diversos frameworks, como Django, Apache Struts, JSF, CakePHP, CodeIgniter, Laravel e muitos outros. Portanto, compreender o funcionamento do MVC é fundamental para programar em frameworks que seguem esse padrão, além de tornar mais fácil o aprendizado de outros frameworks MVC.

1 Padrão MVC



Fonte: Elaboração própria em 2018

Figura 1: Estrutura do padrão MVC

A visão (*view*) é o componente de apresentação do sistema, ou seja, tudo aquilo que o usuário consegue visualizar pertence à visão. Vale salientar que a visão não é um componente único, mas sim um conjunto. Por exemplo: no contexto de um sistema acadêmico, pode haver uma visão para apresentar os dados de um aluno, uma visão para apresentar os dados do professor, uma visão com um formulário de matrícula em disciplina etc. A visão que será apresentada para o usuário é definida pelo controlador (controller) e ela também pode utilizar os modelos da aplicação, conforme explicaremos a seguir.

O modelo (*model*) é o componente que permite que a aplicação armazene dados de forma persistente (permanente), além de possuir as regras de negócio do sistema. Portanto, quando falamos em modelos, estamos falando em dados e lógica da aplicação. No contexto de um sistema acadêmico, podemos ter um modelo chamado Aluno, o qual possui nome, matrícula, e-mail etc.; e um modelo chamado Professor, o qual possui nome, salário, área de atuação etc. Cada modelo da aplicação corresponde a uma tabela (entidade) no banco de dados. Os controladores podem acessar e modificar modelos, enquanto a visão deve apenas acessá-los.

Os controladores (*controllers*) recebem a requisição do usuário, acessam e/ou atualizam os modelos (se necessário) e apresentam uma visão para o usuário. Portanto, o controlador é o componente que coordena o recebimento de requisições do usuário e interage, quando necessário, com os modelos e com as visões.

1.1 Princípios, práticas e técnicas

O framework Ruby on Rails foi construído seguindo um conjunto de princípios e práticas que visam facilitar o desenvolvimento de aplicações Web. São eles: o DRY (*Don't Repeat yourself*); o CoC (*Convention over Configuration*); e o ORM (*Object-Relational Mapping*).

O **DRY** é um princípio de desenvolvimento de *software* que visa reduzir a repetição de informação de qualquer tipo. Formulado por Andy Hunt e Dave Thomas, no livro **O Programador Pragmático**, o DRY propõe que cada porção de conhecimento (informação, código etc.) deve ter uma única representação e deve ser livre de ambiguidades em todo o sistema. Seguindo esse princípio, seu sistema será mais fácil de ser mantido, mais extensível e menos propenso a erros.

O **CoC** é um princípio que visa reduzir a quantidade de configurações necessárias durante o desenvolvimento, consequentemente, reduzindo o esforço do programador. Introduzido por David Heinemeier Hansson (DHH), o criador do framework Rails, esse princípio define uma configuração padrão (convenção), a qual servirá para a maioria dos casos, mas também permite que o desenvolvedor altere essa configuração, caso seja necessário.

O **ORM** é uma técnica para conversão de dados entre sistemas desenvolvidos com Programação Orientada a Objetos (POO) e bancos de dados relacionais. A representação de dados na Programação Orientada a Objetos é diferente da representação usada nos bancos de dados relacionais, portanto, o papel do ORM é realizar a conversão entre essas duas formas de representação diferentes.

É importante conhecermos esses conceitos, no entanto, não se preocupe em compreendê-los a fundo neste momento, pois eles se tornarão mais evidentes à medida em que nos aprofundarmos nos estudos do framework.

1.2 Rotas

As rotas (*routes*) definem qual controlador irá atender a cada requisição HTTP que chega a sua aplicação Web. Por exemplo: supondo que você tenha uma loja virtual e seu cliente acesse o endereço www.minhalojavirtual.com.br/produtos de sua loja, as rotas definem qual controlador irá atender às requisições de clientes que acessarem esse endereço. Assim, poderia

haver uma rota que definisse que, quando alguém acessar tal endereço, o controlador *ProdutosController* irá atender a essa requisição.

1.3 Gems, Rubygems e Bundler

Se você tem um *smartphone* com Android ou iOS, provavelmente, conhece o conceito de aplicativos: programas que dão novos recursos e funcionalidades para seu celular. Você também deve estar familiarizado com o conceito de loja de aplicativos, como o Google Play ou a Apple Store, nas quais você pode buscar e instalar novos aplicativos no seu celular. Dessa forma, se você quiser um novo aplicativo de câmera fotográfica para o seu *smartphone*, basta ir na loja de aplicativos, buscar, baixar e instalar. Tudo de forma simples, fácil e rápida, graças à loja de aplicativos.

O conceito de Gem se assemelha ao conceito de um aplicativo para *smartphone*. Uma Gem é um pacote de *software* que pode ser "plugado" ao seu projeto Rails, fornecendo novos recursos e funcionalidades à sua aplicação. Nesse caso, se você precisa fazer submissão de arquivos (permitir que os usuários anexem arquivos num formulário), você pode usar uma Gem chamada *paperclip* disponível em https://github.com/thoughtbot/paperclip; ou, se sua aplicação precisa trabalhar com autenticação de usuários, você pode usar uma Gem chamada *devise* (https://github.com/plataformatec/devise), dentre muitas outras disponíveis.

Dessa forma, para recursos e funcionalidades genéricas, comumente necessárias em diversas aplicações, é provável que já exista uma Gem que fornece esse recurso/funcionalidade, dispensando que você tenha de implementá-la e, consequentemente, economizando tempo e esforço.

Entretanto, se os aplicativos para celular têm uma loja que facilita a busca e instalação, teriam também as Gems uma loja? Sim, o RubyGems. Contudo, não chamamos o RubyGems de "loja de Gems", mas de gerenciador de pacotes da linguagem Ruby. O RubyGems (https://rubygems.org/) é um serviço *on-line* que nos permite distribuir, publicar, buscar e instalar Gems de forma fácil e rápida.

Então, temos as Gems, pacotes de *software* que podem ser instalados no nosso projeto, fornecendo novas funcionalidades a ele. Temos o RubyGems, um gerenciador de pacotes (Gems), o qual nos permite buscar e instalar Gems facilmente no nosso projeto. No entanto, a pergunta que você pode

estar se fazendo é: onde você deve especificar quais Gems seu projeto irá utilizar? Fazemos isso num arquivo chamado Gemfile, um arquivo de texto no qual especificamos todas as Gems que deverão ser instaladas no nosso projeto. O trecho de código ilustrado abaixo mostra um exemplo de Gemfile.

```
source 'https://rubygems.org'

gem 'paperclip'
gem 'devise'

gem 'cancan'
```

No Gemfile apresentado acima, foi especificado que o **source** (fonte) das Gems será o RubyGems. Ou seja, todas as Gems especificadas nesse arquivo serão buscadas no RubyGems. Em seguida, especificamos as Gems necessárias para o nosso projeto: **paperclip**, **devise** e **cancan**.

O Bundler (http://bundler.io/) é um programa utilizado pelos desenvolvedores para facilitar a instalação das Gems especificadas no Gemfile. O Bundler lê o Gemfile, busca e instala as Gems e as dependências necessárias, facilitando bastante esse processo e economizando tempo. Para instalar todas as Gems especificadas no Gemfile, usamos o comando **bundle install**.

1.4 *Migrations*

Durante o desenvolvimento de um projeto de *software*, é muito comum haver a evolução da estrutura do banco de dados, através da criação de novas entidades, associações, restrições etc. O framework Ruby on Rails utiliza o conceito de migrações (*migrations*) para evoluir a estrutura do banco de dados do projeto. Portanto, quando estamos desenvolvendo em Rails, não devemos mudar a estrutura da base de dados usando SQL, e sim usando migrações. O interessante das migrações é que elas podem ser executadas para evoluir o banco de dados e, ainda, podem ser desfeitas para que o banco volte ao estado anterior.

Uma migração é uma classe Ruby que herda da classe **ActiveRecord::Migration** e que possui dois métodos: **up** e **down**. Quando uma migração é executada para evoluir a estrutura do banco de dados, o método **up** é executado. Quando desejamos desfazer as alterações realizadas por uma migração, o método **down** é executado. Observe o Exemplo de Código 1, o qual ilustra uma migração chamada **CreateProducts**.

Fonte: Elaboração própria em 2018.

```
1   class CreateProducts < ActiveRecord::Migration
2     def up
3       create_table :products do |t|
4         t.string :name
5         t.text :description
6
7         t.timestamps null: false
8       end
9     end
10
11    def down
12      drop_table :products
13    end
14  end
```

Fonte: http://guides.rubyonrails.org/active_record_migrations.html

Figura 2: Exemplo de código 1 - migração

Observe que a migração **CreateProducts**, ilustrada no Exemplo de Código 1, possui os métodos **up** e **down**. O método **up** cria uma tabela chamada **products** (linha 3) com dois campos: **name** do tipo **string** (linha 4) e **description** do tipo **text** (linha 5). Na linha 7, o método **timestamps** cria mais dois campos (**created_at** e **updated_at**), os quais registram, automaticamente, o horário de criação e o horário da última atualização do objeto. O método **down**, por sua vez, realiza apenas a exclusão da tabela **products**. Novamente, não se preocupe em compreender cada detalhe das migrações agora, pois elas serão alvo de estudo em breve.

1.5 Métodos HTTP

O protocolo HTTP, *Hypertext Transfer Protocol*, que em português significa Protocolo de Transferência de Hipertexto, é um dos principais protocolos de comunicação da Internet e é responsável pela transferência transparente quando estamos navegando na Internet através do nosso navegador

Para melhor compreender alguns conceitos a serem abordados em aulas futuras, é preciso compreendermos o conceito de métodos HTTP. Os métodos HTTP, também conhecidos como verbos HTTP, determinam qual ação deve ser executada em um recurso na Internet. A requisição precisa especificar qual é o seu método, que pode ser GET, POST, DELETE, PUT, HEAD, OPTIONS, TRACE e CONNECT. Dentre eles, os mais utilizados são os quatro primeiros, detalhados na Quadro 1.

Quadro 01: Principais métodos HTTP

Método	Descrição
GET	Solicita os detalhes de um determinado recurso.
POST	Os dados enviados no corpo da requisição são usados para criar um novo recurso.
DELETE	Apaga um recurso.
PUT	Atualiza um recurso.

Fonte: Elaboração própria em 2018.

Os métodos GET, POST, DELETE e UPDATE são utilizados pelo framework Ruby on Rails para definir se um determinado recurso da aplicação está sendo acessado, criado, apagado ou atualizado

Leitura complementar

Aprenda um pouco mais sobre o protocolo HTTP e seus métodos através do artigo **Entendendo um pouco mais sobre o protocolo HTTP**, do blog Nando Vieira: https://nandovieira.com.br/entendendo-um-pouco-mais-sobre-o-protocolo-http.

Avaliando seus conhecimentos

1) Qual é a utilidade do comando **bundle install**?

a) Instala o Bundler no sistema.
b) Instala os bundlers especificados no Bundler.
c) Instala as rotas especificadas no routes.rb.
d) Instala as gems especificadas no Gemfile.
e) Instala o banco de dados especificado no database.yml

2) Qual é a melhor definição de Gems?

a) As Gems definem qual controlador irá atender a cada requisição HTTP que chega à sua aplicação Web.
b) As Gems recebem a requisição do usuário, acessam e/ou atualizam os modelos (se necessário) e apresentam uma visão para o usuário.
c) As Gems são componentes de apresentação do sistema, ou seja, tudo aquilo que o usuário consegue visualizar pertence às Gems.
d) Uma Gem é um pacote de software que pode ser "plugado" ao seu projeto, fornecendo novos recursos e funcionalidades à sua aplicação.
e) As Gems permitem que a aplicação armazene dados de forma persis-

tente (permanente), além de possuir as regras de negócio do sistema.

3) Indique a alternativa que melhor explica o que é o Gemfile.

a) O Gemfile é o arquivo de configuração no qual devemos especificar as views do nosso sistema.
b) O Gemfile é o arquivo de configuração no qual devemos especificar as Gems usadas no projeto.
c) O Gemfile é o arquivo de configuração no qual devemos especificar os models do nosso sistema.
d) O Gemfile é o arquivo de configuração no qual devemos especificar as rotas do nosso sistema.
e) O Gemfile é o arquivo de configuração no qual devemos especificar as definições de acesso ao banco de dados.

4) Qual é o significado de MVC?

a) *Migration-Vision-Control.*
b) *Model-View-Controller.*
c) *Model-Vision-Control.*
d) *Model-View-Control.*
e) *Model-Vision-Controller*.

5) Dentre as alternativas abaixo, indique a que melhor descreve a utilidade das migrations?

a) As *migrations* definem as rotas do sistema, especificando qual controlador irá atender a cada requisição HTTP recebida pelo sistema.
b) As *migrations* são usadas para migrar o sistema de um computador para outro e permitir que o mesmo projeto seja executado em diferentes máquinas.
c) As *migrations* são usadas para manipulação do banco de dados do sistema, especialmente para criação, alteração e exclusão de entidades e relacionamentos no banco de dados.

6) Qual é a melhor definição de rotas no contexto de aplicações MVC?

a) As rotas são as definições de acesso ao banco de dados, realizadas no *database.yml*.
b) As rotas recebem a requisição do usuário, acessam e/ou atualizam os

modelos (se necessário) e apresentam uma visão para o usuário.
c) As rotas são componentes de apresentação do sistema, ou seja, tudo aquilo que o usuário consegue visualizar pertence às rotas.
d) As rotas definem qual controlador irá atender a cada requisição HTTP que chega à sua aplicação Web.
e) As rotas permitem que a aplicação armazene dados de forma persistente (permanente), além de possuírem as regras de negócio do sistema.

Resumindo

Nesta nossa aula, estudamos alguns importantes conceitos que serão necessários para o aprendizado de como desenvolver aplicações utilizando o framework Ruby on Rails. Iniciamos com a definição de frameworks e o padrão arquitetural MVC. Em seguida, estudamos alguns princípios, conceitos e técnicas relativas ao framework Ruby on Rails. Por fim, conhecemos como funcionam as rotas, migrações, Gems e métodos HTTP. Até a próxima unidade didática.

Referências

BUNDLER. Bundler. **Bundler**. Disponível em: http://bundler.io/.

CAELUM. **Desenvolvimento ágil para web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails:** coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBYGEMS. RubyGems. **RubyGems.** Disponível em: https://rubygems.org/.

RUBYONRAILS.ORG. Ruby on Rails Guides. **Ruby on Rails**. Disponível em: http://guides.rubyonrails.org/.

SILVA, M. S. Rails Gilrs Tutorial. **Site do Maujor**. Disponível em: http://www.maujor.com/railsgirlsguide/.

VIEIRA, N. Entendendo um pouco mais sobre o protocolo HTTP. **Nando Vieira**. Disponível em: https://nandovieira.com.br/entendendo-um-pouco-mais-sobre-o-protocolo-http.

TECHTERMS. Framework Definition. TechTerms. Disponível em: https:// https://techterms.com/definition/framework.

Aula 2 - Ambiente de Desenvolvimento

Objetivos

Ao final desta aula, você deverá ser capaz de:

Conhecer a plataforma de desenvolvimento Codeanywhere.

Aprender a criar uma conta gratuitamente.

Criar um container usando uma *stack.*

Conhecer as funções e recursos do Codeanywhere.

Conhecer alguns comandos do terminal.

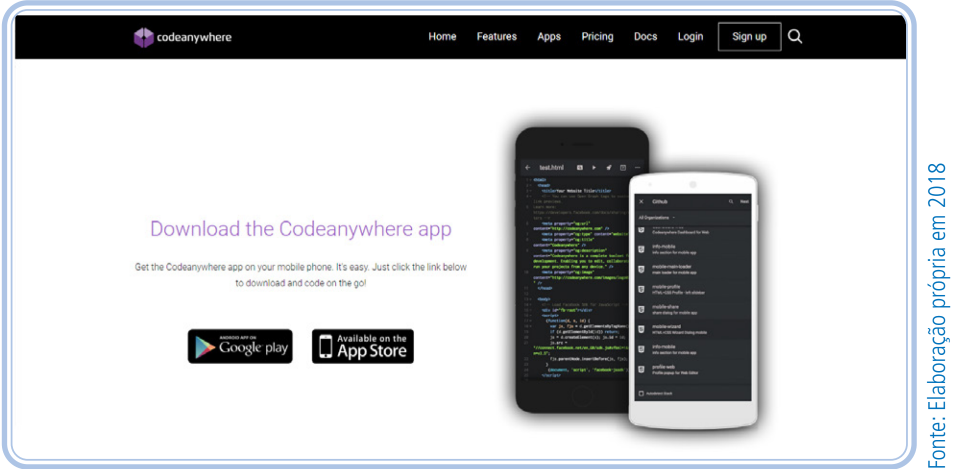
Compartilhar um *container* com outras pessoas.

Conhecer o *dashboard* do Codeanywhere

Introdução

Para que possamos desenvolver aplicações usando Ruby on Rails, inicialmente precisamos preparar o ambiente de desenvolvimento, que consiste na instalação e na configuração do sistema operacional e de outros softwares necessários, como o Ruby, o banco de dados, o sistema de controle de versões, o próprio framework Rails, o editor de código, entre outros softwares.

Para facilitar, usaremos uma plataforma de desenvolvimento na nuvem conhecida como Codeanywhere (https://codeanywhere.com), que é um ambiente de desenvolvimento on-line, o qual nos permitirá construir aplicações usando diversas linguagens e frameworks sem a necessidade de configurar e instalar cada software que iremos precisar, pois tudo já estará pronto para ser usado. A Figura 1 apresenta a página inicial do sítio do Codeanywhere.



Fonte: Elaboração própria em 2018

Figura 1: Página inicial do Codeanywhere

Observe que temos um diretório chamado **mural** dentro do diretório **workspace**, que é a pasta onde nos encontramos atualmente. Para entrar nesse diretório, devemos usar o comando **cd** seguido do nome do diretório, conforme apresentado a seguir.

```
cabox@box-codeanywhere:~/workspace$ cd mural
cabox@box-codeanywhere:~/workspace/mural$
```

Agora, nosso diretório atual é **mural**, que está dentro do diretório **workspace**, o qual está dentro da pasta *home* do nosso usuário (**~**). Lembrando que **mural** é o diretório onde se encontra o nosso projeto Rails, criado pelo comando **rails new mural**. Caso você, por algum motivo, não consiga ver o diretório atual, use o comando **pwd**, conforme ilustrado abaixo.

```
cabox@box-codeanywhere:~/workspace/mural$ pwd
/home/cabox/workspace/mural
```

A resposta do comando **pwd** é o caminho completo de onde estamos atualmente, pois operamos no diretório **mural**, que está dentro de **workspace**, que está dentro de **cabox**, que está dentro de **home**.

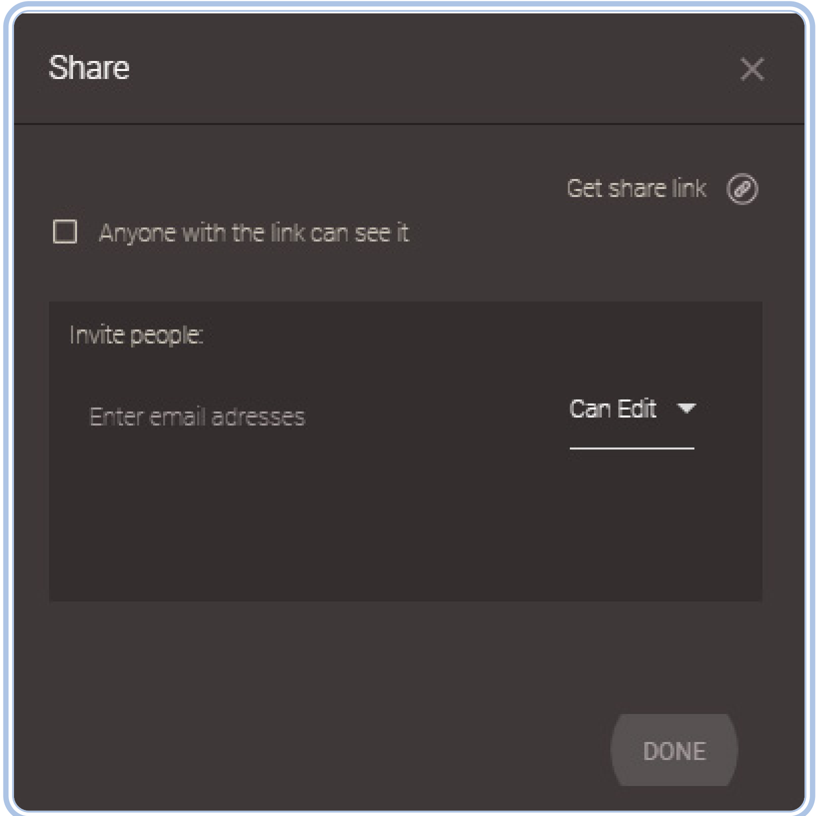
No momento, estamos no diretório **mural** que está dentro da pasta **workspace**. Assim, podemos dizer que **workspace** é o diretório pai do diretório **mural**. O comando **cd** também pode ser usado para voltarmos para o diretório pai, ou seja, para o diretório **workspace**. Para isso, use o comando **cd**, conforme apresentado a seguir

```
cabox@box-codeanywhere:~/workspace/mural$ cd..
cabox@box-codeanywhere:~/workspace$
```

Usando o comando **cd ..**, voltamos para o diretório **workspace**. Lembre-se de que muitos dos comandos, os quais iremos usar durante nossas aulas devem ser executados dentro da pasta do projeto, ou seja, dentro do diretório **mural**.

Compartilhando o *container*

Uma função muito importante do Codeanywhere é a possibilidade do compartilhamento de *containers*, ou seja, permitir que outras pessoas acessem o seu container. Isso pode ser muito útil quando você precisar da ajuda de alguém para superar algum problema em seu projeto. Basta compartilhar o *container* com o seu professor ou um amigo, e, dessa forma, ele terá mais facilidade em te dar suporte.



Fonte: Elaboração própria em 2018

Figura 2: Compartilhado o *container*

Para compartilhar seu container, basta clicar na opção *Share*, a qual aparece no menu do container apresentado na Figura 1. Feito isso, o formulário ilustrado na Figura 2 será apresentado.

Há duas formas de compartilhar seu container: através de um *link* ou através de convite. Para compartilhar através de um *link*, habilite a opção *Anyone with the link can see it* (qualquer um com o *link* pode ver) e clique em *Get share link*. Depois disso, um *link* será gerado e você deverá enviá-lo para as pessoas com as quais você quer compartilhar seu container.

A segunda forma de compartilhamento é através de convite. Para isso, na seção *Invite people* (Figura 2), preencha o campo *Enter email address* com o endereço de *e-mail* da pessoa com a qual você quer compartilhar o *container.* No campo ao lado, selecione se essa pessoa poderá editar (*Can Edit*) ou poderá apenas visualizar (*Can View*) seu *container.*

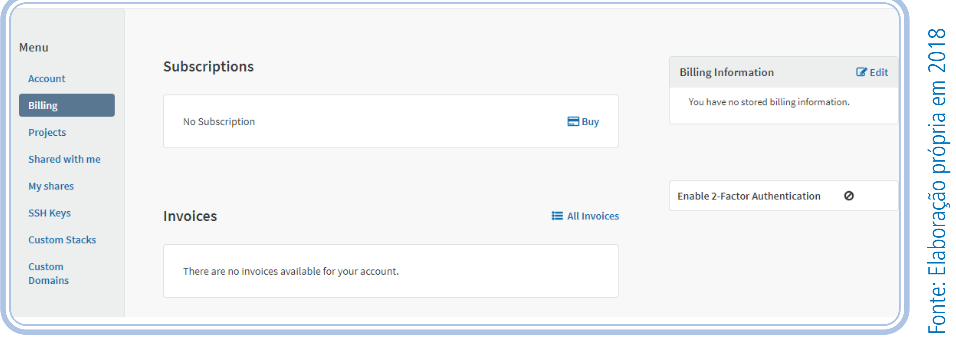
Atividade

Siga as orientações da aula e compartilhe, através de um *link*, seu container com o professor

Dashboard

O dashboard, disponível através do endereço https://codeanywhere.com/dashboard, é um painel que nos permite ter acesso a várias funções de configuração do Codeanywhere. A página inicial do dashboard, ilustrada na Figura 3, é a página de Billing, ou seja, de cobrança. Estamos usando uma conta gratuita, portanto não precisamos fornecer dados de cobrança para o Codeanywhere.

De todas as opções disponíveis no menu ao lado esquerdo do dashboard, podemos destacar algumas que serão úteis para nossas aulas.



Fonte: Elaboração própria em 2018

Figura 3: Dashboard do Codeanywhere

- *Account*: permite alterar configurações da conta como nome, nome de usuário, endereço de e-mail, senha, apagar a sua conta, entre outras funções.

- *Projects*: permite que você crie, abra, edite ou apague projetos (containers).

- *Shared with me*: permite visualizar os containers compartilhados com você.

- *My shares*: permite visualizar os containers que você está compartilhando com outras pessoas.

Resumindo

Nesta aula, tivemos uma visão geral do Codeanywhere como ambiente de desenvolvimento de software. Aprendemos a criar uma conta gratuitamente, criamos um container para desenvolver nossos projetos, conhecemos algumas funções e recursos do Codeanywhere, aprendemos a usar o terminal de comandos e, para finalizar, compartilhamos *containers* e conhecemos o dashboard. Na próxima unidade didática, prosseguimos com esses estudos. Até lá!

Leitura Complementar

Conheça um pouco mais sobre o Codeanywhere através do artigo **Codeanywhere: Uma IDE para desenvolvimento baseado na *cloud*** (em Português de Portugal): https://pplware.sapo.pt/internet/codeanywhere-ide-desenvolvimento-cloud/.

Referências

CODEANYWHERE. **Cross platform cloud ide**. Disponível em: https://codeanywhere.com/. Acesso em: 23 fev. 2018.

GORAILS. **Install Ruby on Rails on Ubuntu 16.10 Yakkety Yak**: a guide to setting up a ruby on rails development environment. Disponível em: https://gorails.com/setup/ubuntu/16.10. Acesso em: 12 dez. 2017.

RUBY. **Ruby programming language**. Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. **Ruby on rails**: a web-application framework that includes everything needed to create database-backed web applications according to the model-view-

Aula 3 - Criando seu primeiro projeto

Objetivos

Ao final desta aula, você deverá ser capaz de:

Criar seu próprio projeto Rails;

Compreender a organização da estrutura de um projeto Rails;

Executar e acessar seu primeiro projeto através do navegador;

Utilizar o Rails Console.

Introdução

Para apresentar o desenvolvimento de aplicações usando o framework Ruby on Rails, utilizaremos o projeto Mural de Mensagens como exemplo. O Mural de Mensagens é uma aplicação simples que permite o cadastro, a listagem, a edição, o detalhamento e a exclusão de mensagens. Cada mensagem tem um título, um corpo, o nome e o endereço de e-mail do autor. Veja o Mural de Mensagens em funcionamento através do vídeo: https://youtu.be/aL1yynzQsH4.

Para criar um novo projeto com Ruby on Rails é bem simples, abra o terminal de comandos e execute o comando **rails new <nome-do-projeto>**, substituindo **<nome-do-projeto>** pelo nome do projeto que você deseja criar. Vamos criar o projeto Mural de Mensagens executando o comando abaixo no terminal.

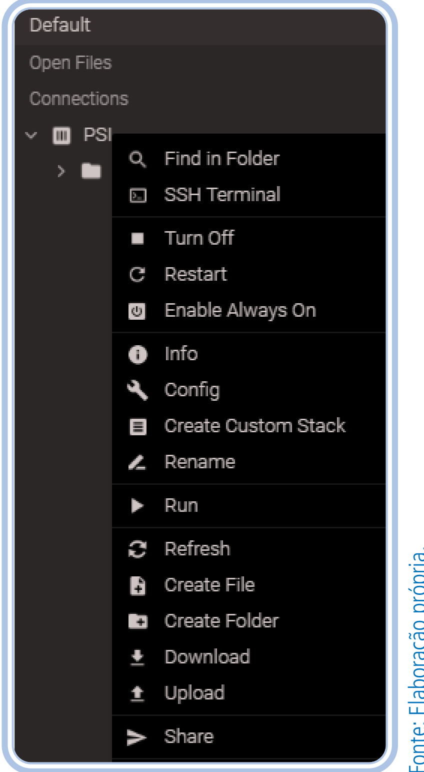
```
rails new mural
```

Ao executar o comando, uma série de mensagens serão exibidas na tela, conforme ilustrado a seguir.

```
$ rails new mural
      create  README.rdoc
      create  Rakefile
      create  config.ru
      create  .gitignore
      create  Gemfile
      create  app
      create  app/assets/javascripts/application.js
      create  app/assets/stylesheets/application.css
      create  app/controlers/application_controller.rb

[...]

      run bumdle install
Fetching gem metadata from https://rubygems.org/........
Fetching version metadata from https://rubygems.org/...
Resolving
dependencies.........................................
Installing rake 11.1.2
Using i18n 0.7.0
Installing json 1.8.3 with native extensions

[...]

Bundle complete! 12 Gemfile dependencies, 55 gems now
installed.
Use bundle show [gemname] to see where a bundled gem is
installed.
```

As primeiras mensagens que aparecem, precedidas de **create**, indicam que um diretório ou um arquivo está sendo criado pelo Rails para compor a estrutura de diretórios e artefatos do seu novo projeto. Além disso, um comando é executado automaticamente: **bundle install**. Conforme explicado, esse comando lê o **Gemfile**, baixa e instala as Gems necessárias para o projeto. Quando a execução do comando for concluída, significa que o seu novo projeto já está criado

Seu projeto foi criado numa nova pasta chamada **mural**, contudo, essa nova pasta não é visível no gerenciador de arquivos do Codeanywhere. Para exibi-lo no gerenciador de arquivos, clique com o botão direito do mouse no item que representa seu container (primeiro item abaixo de *Connections*). O menu ilustrado na Figura 1 deverá ser apresentado.



Fonte: Elaboração própria.

Figura 1: Menu do Container

Em seguida, clique na opção *Refresh* para atualizar a estrutura de arquivos e diretórios do gerenciador de arquivos. Depois disso, um novo diretório chamado **mural** deverá ser exibido abaixo do item que representa seu *container*.

Ao criar um projeto Rails usando o comando rails **new**, será criado um projeto com o banco de dados SQLite3, o banco de dados padrão quando não especificamos qual banco desejamos usar. Para especificar um banco de dados, devemos utilizar o parâmetro **-d** após o nome do projeto.

```
rails new meuprojeto -d postgresql
```

O comando acima irá criar um projeto Rails chamado **meuprojeto**, o qual usa o banco de dados PostgreSQL. As opções de banco de dados compatíveis com o framework Rails são: **mysql**, **oracle**, **postgresql**, **sqlite3**, **frontbase**, **ibm_db**, **sqlserver**, **jdbcmysql**, **jdbcsqlite3**, **jdbcpostgresql**, **jdbc**.

Atividade

Siga as orientações da aula para criar seu primeiro projeto Rails chamado **mural**. Esse projeto deve utilizar o banco de dados SQLite3.

Estrutura do projeto

Ao criar um projeto em Ruby on Rails, o próprio framework constrói uma estrutura de pastas e arquivos para facilitar a organização e o desenvolvimento da aplicação. A Tabela 1 apresenta as principais pastas e arquivos de um projeto Rails.

Tabela 01: Principais pastas e arquivos de um projeto Rails

Arquivo/Pasta	Descrição
app/	É a principal pasta da aplicação. Contém os *models*, *views* e *controllers* do MVC, além de outras pastas e arquivos.
app/assets	Pasta que agrupa alguns artefatos da aplicação: imagens, *scripts* JavaScript e folhas de estilos.
app/assets/images	Armazena as imagens da aplicação.
app/assets/javascripts	Armazena os *scripts* JavaScript da aplicação.
app/assets/stylesheets	Armazena as folhas de estilo (CSS) da aplicação.
app/controllers	Contém os *controllers* da aplicação.
app/helpers	Contém os *helpers* da aplicação. *Helpers* são módulos que permitem definir métodos que extraem a lógica da *view*.
app/mailers	Contém os *mailers*. Os *mailers* são classes que auxiliam a aplicação no envio de mensagens de *e-mail*.
app/models	Armazena os *models* da aplicação.
app/views	Armazena as *views* da aplicação.
config/	Contém os arquivos de configuração do projeto.
config/routes.rb	Esse é o arquivo de rotas discutido na Aula 1.
db/	Contém todos os arquivos relativos à persistência de dados do projeto.
db/migrate	Esse diretório armazena todas as *migrations* do projeto, estudadas na Aula 1.
db/seeds.rb	O *seeds.rb* é um script que nos permite carregar os dados iniciais da aplicação no banco de dados.
lib/	Contém as bibliotecas externas que serão usadas por sua aplicação.
log/	Armazena os *logs*.
public/	Essa é a única pasta visível aos usuários da aplicação.
test/	Contém todos os artefatos relativos a testes automatizados da aplicação.
tmp/	Armazena os arquivos temporários (*cache*, *pid* e arquivos de sessão).
vendor/	Armazena os códigos de terceiros.
Gemfile	Arquivo que especifica as *Gems* utilizadas pela aplicação.

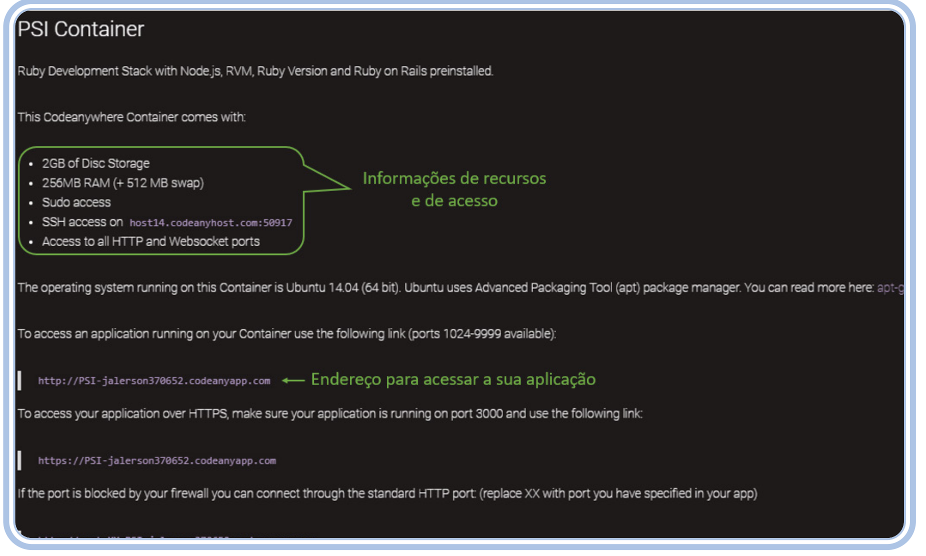
Fonte: RUBYONRAILS.ORG, 2016.

Executando o projeto

Para executar o seu projeto e poder visualizá-lo no navegador, primeiramente, devemos garantir que estamos no diretório do nosso projeto Rails, ou seja, no diretório **mural**. Em seguida, execute o comando **rails server**, ou apenas **rails s**, no terminal de comandos, conforme apresentado a seguir.

```
cabox@box-codeanywhere:~/workspace/mural$ rails server
=> Booting WEBrick
=> Rails 4.2.4 application starting in development on
http://localhost:3000
=> Run rails server -h`for more startup options
=> Ctrl-C to shutdown server
[2016-05-14 17:38:58] INFO WEBrick 1.3.1
[2016-05-14 17:38:58] INFO ruby 2.2.3 (2015-08-18) [i686-
-linux]
[2016-05-14 17:38:58] INFO WEBrick::HTTPServer#start:
pid=4180 port=3000
```

Ao executar esse comando, um servidor Web de desenvolvimento chamado WEBrick será inicializado na porta 3000. Contudo, ainda precisamos descobrir em qual endereço nosso projeto está disponível. Para isso, clique com o botão direito no item do gerenciador de arquivos que representa seu container (primeiro item abaixo de *Connections*). No menu que for apresentado (Figura 1), clique na opção *Info*. Feito isso, uma nova aba deverá ser aberta com as informações apresentadas na Figura 6.

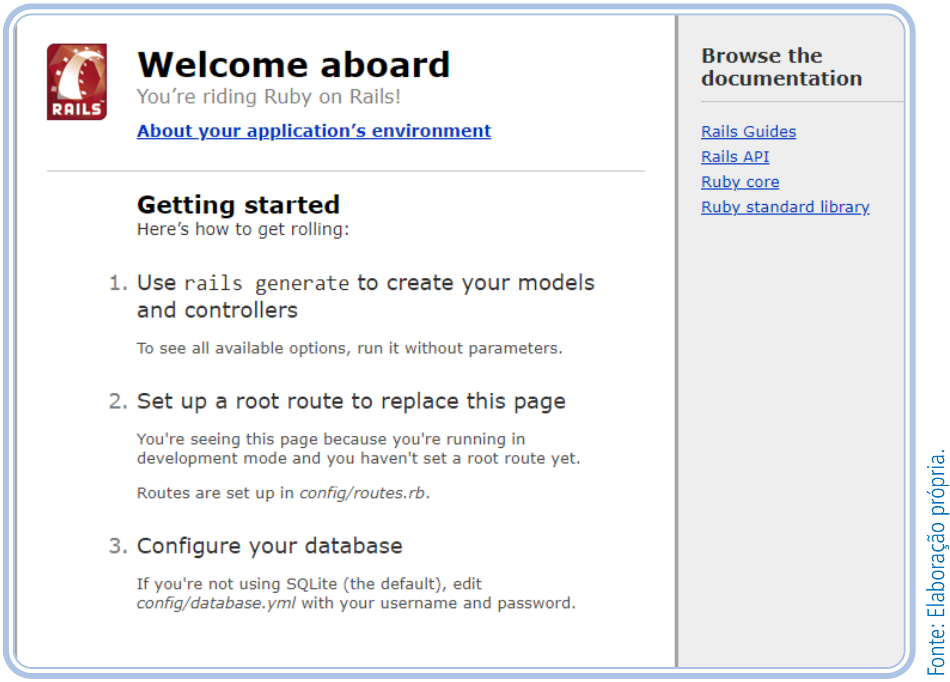


Fonte: Elaboração própria.

Figura 2: Compartilhado o *container*

A Figura 2 indica qual é o endereço que você deve observar no seu container para acessar sua aplicação. Copie o endereço indicado no seu container, e, em seguida, abra seu navegador de Internet. Cole o endereço que você copiou, e, ao final do endereço, adicione **:3000**, a porta na qual o servidor web WEBrick está disponibilizando nossa aplicação.

Por exemplo, supondo que o Codeanywhere esteja disponibilizando sua aplicação através do endereço http://psi-aluno.codeanyapp.com, para acessar sua aplicação através do navegador, você deverá usar http://psi-aluno.codeanyapp.com:3000. Ao acessar sua aplicação, através do navegador, a página ilustrada na Figura 3 deverá ser apresentada.



Fonte: Elaboração própria.

Figura 3: Página inicial padrão do Rails

Parabéns! Você acabou de criar o seu primeiro projeto em Ruby on Rails! É claro que ele ainda está vazio, mas esse é o primeiro passo para criar aplicações mais complexas. Para encerrar a execução do seu projeto, volte para o terminal de comandos e use a sequência de teclas CTRL+C para interromper a execução do WEBrick.

Atividade

Siga as orientações da aula para executar seu projeto Rails. Em seguida, abra seu navegador de Internet e acesse seu projeto através dele.

Rails Console

O Rails Console é um utilitário presente em qualquer projeto Rails. Nele, podemos executar códigos em Ruby, utilizar classes de nosso projeto, como modelos e controladores, acessar o banco de nossa aplicação, entre outras utilidades. Para

acessar o Rails Console, digite o comando **rails console**, no terminal de comandos, e tecle ENTER. Para sair do Rails Console, digite o comando **exit** e tecle ENTER. Outro comando que nos permite entrar no Rails console é o **rails c**, que é apenas uma abreviação do comando **rails console**.

O Rails Console se parece muito com o IRB, logo, você pode estar se perguntando qual é a diferença entre eles. Na realidade, o Rails Console é o IRB, mas com algumas funcionalidades adicionais. Conforme mencionado anteriormente, através do Rails Console você poderá ter acesso às classes e aos recursos do seu projeto Rails, algo que não acontece no IRB.

Infelizmente, ainda não temos modelos, controladores e nem *views* no nosso projeto, portanto, por enquanto, ainda não vamos usar esses recursos. Eles serão usados no Rails Console em breve, quando estivermos estudando esses assuntos.

Atividade

No Rails Console, execute o comando print **'Hello world!'** e observe a mensagem **"Hello World!"** apresentada na tela.

Resumindo

Nesta aula, aprendemos a criar nosso primeiro projeto Rails. Além disso, abordarmos sobre a estrutura de diretórios que organiza um projeto. Também aprendemos como executar e acessar nosso projeto através do navegador de Internet, e, por fim, conhecemos o Rails Console. Esperamos você na próxima aula!

Leitura complementar

Saiba mais sobre como construir suas próprias aplicações usando Rails, lendo os tutoriais do Rails Girls: http://guides.railsgirls.com/guides-ptbr/guide-to-the-guide.

Referências

BUNDLER. Bundler. **Bundler**.Disponível em: http://bundler.io/. Acesso em: 14 maio 2016.

CAELUM. Desenvolvimento Ágil para **Web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails** - Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBYGEMS. RubyGems. **RubyGems**. Disponível em: https://rubygems.org/. Acesso em: 14 maio 2016.

RUBYONRAILS.ORG. Ruby on Rails Guides. **Ruby on Rails**. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. Rails Gilrs Tutorial. **Site do Maujor**. Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

VIEIRA, N. Entendendo um pouco mais sobre o protocolo HTTP. **Nando Vieira**. Disponível em: https://nandovieira.com.br/entendendo-um-pouco-mais-sobre-o-protocolo-http. Acesso em: 14 maio 2016.

WIKIPÉDIA. Framework. **Wikipédia**. Disponível em: https://pt.wikipedia.org/wiki/Framework. Acesso em: 14 maio 2016.

Aula 4 - Criando o Scaffold Mensagem

Objetivos

Ao final desta aula, você deverá ser capaz de:

Compreender a diferença entre convenção e configuração.

Configurar o projeto para pluralizar palavras em português.

Configurar o projeto com palavras e expressões em português.

Conhecer o comando para criação de scaffold.

Utilizar o comando scaffold para criar um cadastro completo de mensagens.

Introdução

Você se lembra de quando discutimos *Convention over Configuration* na Aula 01? Essa prática define que as convenções devem ter preferência sobre as configurações, ou seja, usamos convenções para resolver os casos gerais e as configurações para resolver as exceções. Caso você ainda não tenha entendido muito isso, vamos ver um exemplo adiante..

O Rails é um framework construído usando-se a língua inglesa como referência. Essa é uma prática muito comum e vale para quase todas as linguagens de programação e frameworks existentes. Uma das convenções do Rails é que as aplicações serão construídas usando-se a língua inglesa como referência. Essa é uma convenção, e, portanto, todas as aplicações que usam a língua inglesa não precisam fazer nenhuma configuração de idioma.

Contudo, agora pretendemos criar uma aplicação que irá usar o idioma Português do Brasil, e não inglês. Nesse aspecto, a nossa aplicação é uma exce-

ção à regra, e, dessa forma, devemos configurá-la para que ela use o nosso idioma. Assim, utilizaremos uma configuração para fugir da convenção.

Para ilustrar melhor esse cenário, vamos abrir o Rails Console do nosso projeto. Para isso, vá para o terminal e digite o comando **rails console**. Em seguida, vamos usar o método **pluralize** sobre uma String. O método **pluralize** é utilizado para pluralizar uma palavra. Aqui, utilizaremos esse método para mostrar que o nosso projeto consegue pluralizar palavras em inglês, mas não em português.

Digite a String **"mensagem"** e use o método **pluralize** sobre ela, conforme ilustrado abaixo.

```
"mensagem". pluralize
```

Observe como o Rails pluralizou a palavra **"mensagem"**. Ele, simplesmente, adicionou um **"s"** no final, mas sabemos que o plural de **"mensagem"** é **"mensagens"** com **"ns"** no final. Agora, digite **"message"**, em inglês, e novamente use o método pluralize:

```
"message". pluralize
```

Observe que, dessa vez, o Rails pluralizou a palavra **"message"** corretamente (**"messages"**). Você pode estar pensando que o Rails sempre adiciona um **"s"** no final, independentemente da palavra, mas isso não é verdade. Vamos usar um verbo irregular em inglês:

```
"person". pluralize
```

Observe que ele pluralizou corretamente a palavra **"person"** para **"people"**.

Configurando nosso idioma

Vamos configurar o nosso projeto para o idioma Português do Brasil, de forma que o Rails pluralize corretamente as palavras em nosso idioma. Para isso, precisamos modificar o arquivo **config/initializers/inflections.rb**. É nesse arquivo que configuramos as regras de pluralização de palavras.

Vamos utilizar um **inflections.rb**, que já está configurado para o nosso idioma. Para isso, abra uma aba em seu navegador e acesse o endereço

https://gist.github.com/mateusg/924574. Para copiar o conteúdo desse arquivo, clique no botão Raw, copie todo o conteúdo desse script e cole no **inflections.rb** do nosso projeto.

Pronto, essa é a configuração necessária no **inflections.rb**. Agora, o nosso projeto já é capaz de pluralizar palavras em Português do Brasil. No entanto, essa não é toda a configuração que vamos fazer em nosso projeto. Neste momento, vamos configurar o idioma para Português do Brasil. Para isso, abra o script **config/aplication.rb**, descomente a linha 21 e configure o valor para **"pt-BR"**, como mostramos abaixo.

```
config.i18n.default_locals = "pt-BR"
```

Logo, baixe e adicione ao nosso projeto o arquivo de locale para Português do Brasil. Esse arquivo define algumas palavras em nosso idioma, como os dias da semana, os meses do ano e outras expressões em português. Acesse o endereço https://github.com/svenfuchs/rails-i18n/blob/master/rails/locale/pt-BR.yml, clique no botão Raw e copie todo o conteúdo desse arquivo.

Depois disso, crie um arquivo chamado **pt-BR.yml** na pasta **config/locales**. Abra esse arquivo e cole o conteúdo que você copiou. Pronto! Concluímos as configurações básicas do nosso projeto Rails.

Atividade

Siga as orientações para configurar seu projeto **Mural de Mensagens** para Português do Brasil.

O comando Scaffold

O scaffold é um comando do framework Rails que gera os arquivos necessários para cadastro, detalhamento, edição, exclusão e listagem de um determinado elemento. Nesta nossa aula, iremos criar o scaffold para Mensagem. Isso significa que iremos gerar os arquivos para termos as funcionalidades de cadastro, detalhamento, edição, exclusão e listagem de mensagens.

Para gerar o scaffold de Mensagem, abra o terminal e execute o comando a seguir:

```
rails generate sacaffold mensagem titulo:string corpo:text
autor:string email:string
```

O comando começa com **rails generate scaffold** seguido do nome do modelo que desejamos: **mensagem** (no singular). Os sucessivos parâmetros são os atributos do modelo mensagem: **titulo**, do tipo **string**; **corpo**, do tipo **text** e **autor** do tipo **string**. Após a execução desse comando, serão gerados modelos, rotas, controlador, *views*, a migração, entre outros arquivos e diretórios.

Abra a pasta **app/controllers** e observe que o controlador **mensagens_controller.rb** foi gerado. Além disso, foi criado o modelo **Mensagem** dentro da pasta **models**, e as *views* dentro da pasta **views/mensagens**.

Volte para o terminal e execute o comando **rake db:create**. Esse comando irá criar o banco de dados que será usado pela nossa aplicação. Depois disso, vamos executar o **rake db:migrate**. Tal comando irá executar todas as *migrations* que ainda não foram executadas. No nosso caso, como só temos uma *migration*, ele irá executar apenas a migration para criar a tabela **mensagens**.

Feito isso, já é possível executar o projeto para ver o scaffold de mensagens funcionando. Para isso, vamos executar o servidor de desenvolvimento da nossa aplicação usando o comando **rails s** ou **rails server**. Em seguida, abra seu navegador de Internet e acesse o endereço da sua aplicação, conforme estudamos na aula passada. Ao acessar esse endereço, você deve visualizar a página ilustrada na Figura 1.

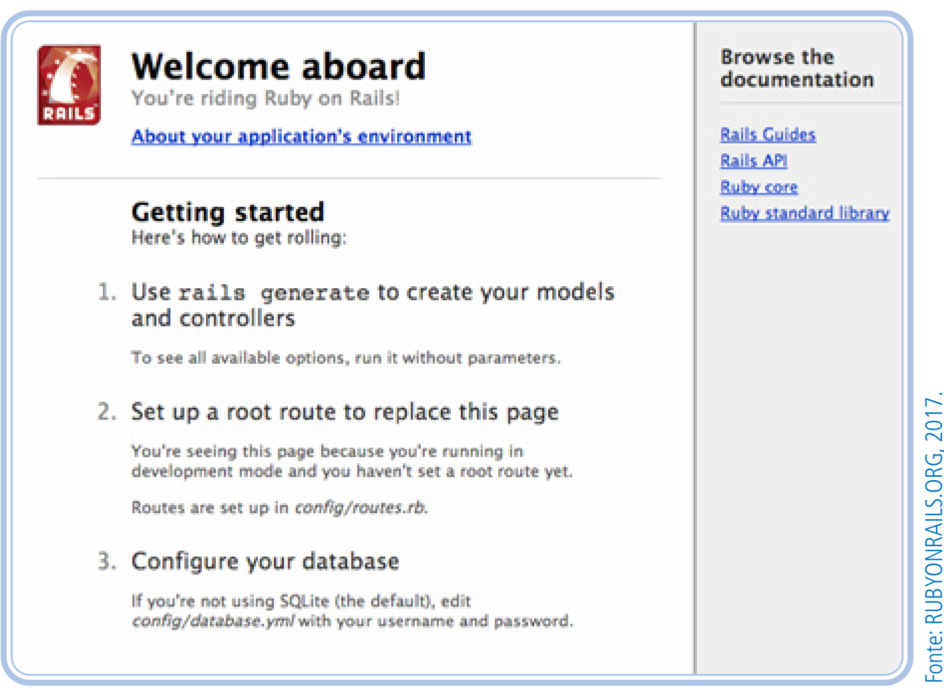


Figura 1: Página inicial padrão de um projeto Rails

A Figura 1 apresenta a página padrão de um projeto Rails. Para ver a página de listagem de mensagens, ilustrada na Figura 2, insira **/mensagens** no final do endereço da sua aplicação. Por exemplo, se você acessou sua aplicação através do endereço http://psi-aluno.codeanyapp.com:3000, você deverá acessar a listagem de mensagens através do endereço http://psi-aluno.codeanyapp.com:3000/mensagens.

Listing Mensagens

Titulo Corpo Autor Email

New Mensagem

Fonte: Autoria própria

Figura 2: Listagem de mensagens

Podemos ver que a listagem não contém mensagem, pois ainda não a criamos. Veja, também, que a listagem é apresentada através de uma página bem simples, mas que contém todas as funcionalidades de cadastro, edição, exclusão e detalhamento implementadas. Fique à vontade para usar e testar a sua primeira aplicação desenvolvida em Rails.

Atividade

Siga as orientações para criar o scaffold mensagem no seu projeto **Mural de Mensagens**

Resumindo

Nesta unidade didática, apresentamos as configurações iniciais necessárias para o desenvolvimento do nosso **Mural de Mensagens**. Exemplificamos o *Convention over Configuration*, configuramos o projeto para usar o idioma Português do Brasil, e, por último, utilizamos o comando scaffold para gerar um cadastro completo de mensagens. Até a próxima aula.

Leitura complementar

Para aprender mais sobre scaffolds em Rails, leia a seção **Criando Idea scaffold do Rails Girls**, disponível em: http://guides.railsgirls.com/guides-ptbr/guide-to-the-guide#2_criando_idea_scaffold.

Referências

CAELUM. **Desenvolvimento Ágil para Web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails**: Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBY. **Ruby programming language**. Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. **Ruby on rails**: a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern. Disponível em: http://rubyonrails.org/. Acesso em: 12 dez. 2017.

RUBYONRAILS.ORG. **Ruby on Rails Guides**. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. **Rails Gilrs Tutorial**. Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

Aula 5 - O Padrão MVC

Objetivos

Ao final desta aula, você deverá ser capaz de:

Compreender o que são e qual o funcionamento das rotas.

Conhecer os controladores e como eles se integram ao padrão MVC.

Compreender a camada de apresentação da aplicação.

Aprender como funcionam os modelos da aplicação.

Introdução

Você O MVC (*Model-View-Controller*) é um padrão arquitetural de *software* que separa a aplicação em três partes: modelos (*models*), visões (*views*) e controladores (*controllers*). Os modelos são componentes responsáveis pela lógica de negócio, pelas validações e pelo armazenamento de dados. As *views* são responsáveis pela apresentação dos dados da aplicação para o usuário, os quais podem ser apresentados em HTML, JSON, XML ou outros formatos. Os controladores são responsáveis por atender e responder às requisições HTTP enviadas pelos usuários.

O framework Ruby on Rails segue o padrão MVC, portanto, todas as aplicações desenvolvidas em Rails serão organizadas de acordo com esse padrão arquitetural.

Rotas

Quando uma requisição HTTP chega à nossa aplicação Rails, o framework consulta o arquivo de rotas (**config/routes.rb**) para determinar qual método e qual controlador irá atender essa requisição. Assim, é no arquivo de

rotas que estabelecemos todas as rotas da nossa aplicação. Mas… o que é uma rota? Uma rota é uma definição de qual método e de qual controlador será executado quando uma requisição HTTP chegar até a nossa aplicação.

Para ver um exemplo de definição de rotas, abra o arquivo de **rotas config/routes.rb** do projeto **Mural de Mensagens**. Observe, na linha 2, a presença da seguinte definição:

```
resources :mensagens
```

Essa configuração define um conjunto de rotas para a nossa aplicação. Para ver as rotas definidas por essa configuração, execute o comando **rake routes** no terminal de comandos. O resultado abaixo deverá ser apresentado.

```
       Prefix Verb   URI Pattern                  Controller#Action
    mensagens GET    /mensagens(.:format)         mensagens#index
              POST   /mensagens(.:format)         mensagens#create
  new_mensagem GET   /mensagens(.:format)         mensagens#new
 edit_mensagem GET   /mensagens/id/edit(.:format) mensagens#edit
     mensagem GET    /mensagens(.:format)         mensagens#show
              PATCH  /mensagens(.:format)         mensagens#update
              PUT    /mensagens(.:format)         mensagens#update
              DELETE /mensagens(.:format)         mensagens#destroy
```

Esse resultado apresentado pelo comando no terminal é uma tabela com quatro colunas (**Prefix**, **Verb**, **URI Pattern** e **Controller#Action**) e nove linhas. Cada linha apresenta uma rota definida na sua aplicação (exceto a primeira linha, a qual mostra os cabeçalhos).

A primeira coluna (**Prefix**) apresenta nomes de métodos, os quais poderão ser usados na sua aplicação para se referir a essa rota. Por exemplo, caso desejemos nos referir à rota **/mensagens**, que apresenta a listagem de mensagens, podemos simplesmente usar o método **mensagens_path**. Caso desejemos nos referir à rota **/mensagens/new**, podemos usar o método **new_mensagem_path**. Observe que sempre é necessário introduzir **_path** no final do nome do método.

A segunda coluna (**Verb**) indica qual método HTTP deve ser utilizado para acessar cada rota. A terceira coluna (**URI Pattern**) mostra os caminhos (*paths*) associados a cada rota. E a quarta coluna (**Controller#Action**) apresen-

ta o nome do controlador e o nome do método que irá atender às requisições enviadas para cada uma das rotas.

Por exemplo, quando um usuário acessar o endereço **/mensagens** da nossa aplicação utilizando o método HTTP GET, o método **index** do controlador **MensagensController** (**app/controllers/mensagens_controller.rb**) irá atender essa requisição. Quando um usuário acessar o endereço **/mensagens/new** da nossa aplicação, utilizando o método HTTP GET, o método **new** do controlador **MensagensController** (**app/controllers/mensagens_controller.rb**) irá atender essa requisição.

Quando um usuário acessar o endereço **/mensagens** da nossa aplicação, usando o método HTTP POST, o método **create** do controlador **MensagensController** (**app/controllers/mensagens_controller.rb**) irá atender essa requisição.

Dessa forma, há um conjunto de rotas que a nossa aplicação é capaz de atender. Cada rota é definida por um endereço e indica qual método de qual controlador será responsável por atender às requisições enviadas para esse endereço.

Atividade

Execute o comando **rake routes** para visualizar as rotas já configuradas em seu projeto.

Controladores

Os controladores são os componentes do padrão MVC, responsáveis por receber e responder às requisições HTTP enviadas pelos usuários. Em um projeto Rails, os controladores estão localizados na pasta **app/controllers**.

Para ilustrar um controlador, abra o controlador **MensagensController**, presente no *script* **app/controllers/mensagens_controller.rb** do nosso projeto **Mural de Mensagens**. Esse controlador possui um conjunto de métodos que serão responsáveis por atender às requisições HTTP enviadas pelos usuários.

Observe abaixo, por exemplo, a implementação do método **index**, responsável por atender requisições enviadas para o endereço **/mensagens**, que apresenta a listagem de mensagens para o usuário.

```
def index
  @mensagens = Mensagem.all
 end
```

O método **index** possui uma única linha de código. O método **all**, chamado sobre o modelo **Mensagem**, retorna um array de objetos da classe Mensagem cadastrados no banco de dados. Esse **array** é então atribuído à variável **@mensagens**, uma variável de instância. As variáveis de instância definidas nos métodos do controlador estarão disponíveis nas *views*. Dessa forma, será possível usar a variável **@mensagens** na *view* que será apresentada para o usuário.

Abaixo, observe a implementação do método **new**, responsável por atender requisições enviadas para o endereço **/mensagens/new**.

```
def new
  @mensagens = Mensagem.new
 end
```

O método **new** do controlador também tem apenas uma linha de código: um novo objeto da classe **Mensagem** é criado e armazenado na variável **@mensagem**.

Agora, observe a implementação dos métodos **show** e **edit**, responsáveis por detalhar e editar mensagens, respectivamente.

```
def show
ends

def edit
end
```

Perceba que o corpo desses métodos está vazio. Isso ocorre quando não é necessário realizar nenhum processamento no controlador, ou seja, deseja-se apenas apresentar a *view*. Apesar desses métodos estarem vazios, um outro método é executado quando eles forem executados. Observe, na linha 2 do controlador **MensagensController**, a seguinte linha de código:

```
before_action :set_mensagem, only [:show, :edit, :update,
:destroy]
```

O **before_action** é um método o qual define que o método **set_mensagem** será executado antes da execução dos métodos **show**, **edit**, **update** e **destroy** (veja o array passado como parâmetro em **only**). O método **set_mensagem** pode ser localizado no final do script, contendo a implementação a seguir.

```
def set_mensagem
  @mensagens.find(params[:id])
end
```

A variável **params** contém um hash com os parâmetros enviados na requisição HTTP. Ao usar **params[:id]**, estamos acessando o ID do objeto no qual estamos interessados. Esse ID está sendo fornecido como parâmetro para o método **find** do modelo **Mensagem**. O método **find** é usado quando se deseja localizar um objeto específico a partir de um ID fornecido como parâmetro. Esse objeto é então armazenado na variável **@mensagem**, a qual estará disponível na view que será apresentada para o usuário.

Views

As *views* são os componentes de apresentação de qualquer aplicação Rails, ou seja, é aquilo que o usuário irá ver efetivamente. Em um projeto Rails, as *views* podem ser localizadas no diretório **app/views**, que são organizadas em subdiretórios (um para cada controlador). Dessa forma, as *views* do controlador **MensagensController** estão presentes em **app/views/mensagens**.

Outra convenção importante do framework Rails é quando um método qualquer do controlador é executado, a *view* que será apresentada ao usuário será aquela com o mesmo nome do método. Por exemplo, quando o usuário acessa o endereço **/mensagens**, o método **index** do controlador **MensagensController** será executado, e, em seguida, a *view* **app/views/mensagens/index.html.erb** será apresentada para o usuário. Quando o usuário acessa o endereço **/mensagens/new**, o método **new** do controlador **MensagensController** será executado, e, em seguida, a *view* **app/views/mensagens/new.html.erb** será apresentada para o usuário.

Existem três métodos no controlador **MensagensController** que não seguem a essa convenção: são os métodos **create**, **update** e **destroy**, responsáveis por criar, atualizar e apagar objetos, respectivamente. Após a criação ou atualização de um objeto, o usuário é redirecionado para

o detalhamento do objeto criado (método **show**). Após a deleção de um objeto, o usuário é redirecionado para a página de listagem (método **index**).

Usando código Ruby nas Views

Abra a *view* **app/views/mensagens/index.html.erb** e verifique que o código presente nela é, em sua maior parte, código HTML, mas também é possível ver códigos em Ruby delimitados por **<%** e **%>** ou **<%=** e **%>**.

Todo código em Ruby presente nas *views* devem estar dentro de delimitadores. Existem dois delimitadores de abertura, que são **<%** e **<%=**, e um delimitador de fechamento, que é o **%>**. Usando o delimitador de abertura **<%=**, o resultado da expressão em Ruby presente dentro dos delimitadores será apresentado. Por exemplo, na linha 1 da *view* **app/views/mensagens/index.html.erb**, temos o seguinte.

```
<p id="notice"><%= notice %></p>
```

O delimitador de abertura **<%=** garante que o valor da variável **notice** será apresentado no código HTML. Os delimitadores **<%** e **%>**, por sua vez, não possuem esse comportamento, pois o código Ruby presente dentro deles é apenas executado, mas seu resultado não será apresentado no código HTML. Por exemplo, observe os códigos Ruby presentes nas linhas 17 e 27 da *view* **app/views/mensagens/index.html.erb**.

```
17 <% @mensagens.each do |mensagem| %>
18   <tr>
19     <td><%= mensagem.titulo %></td>
20     <td><%= mensagem.corpo %></td>
21     <td><%= mensagem.autor %></td>
22     <td><%= mensagem.email %></td>
23     <td><%= link_to 'Show', mensagem %></td>
24     <td><%= mensagem'Edit', edit_mensagem_path(mensagem)
   (mensagem) %></td>
25     <td><%= link_to 'Destroy, mensagem, methodo: :delete,
   data: {confirm: 'Are you sure?' } %></td>
26   <tr>
27 <% end %>
```

Na linha 17, temos o método **each** sendo executado sobre a variável **@mensagens**. Essa variável foi criada no método **index** do controlador **MensagensController** (**app/controllers/mensagens_controller.rb**). O método **each** irá executar uma vez para cada objeto presente na variável **@mensagens**. Dessa forma, uma linha da tabela HTML (*tag* **<tr>**) será apresentada para cada mensagem.

A pasta *layouts*

Dentro do diretório **app/views**, há um subdiretório chamado **layouts**, e dentro desse diretório há um único arquivo: **application.html.erb**. Essa é uma importante *view*, pois ela contém o código que é compartilhado entre todas as outras *views* da nossa aplicação.

Para melhor compreender a utilidade dessa *view*, abra a *view* **app/views/mensagens/show.html.erb** e observe que ela não é uma página HTML completa. Não há *tags* **<html>**, **<head>**, **<title>**, **<body>** etc. A *view* **show.html.erb** (e as demais) não têm essas *tags*, porque elas estão presentes na *view* **application.html.erb**, que, conforme explicamos, contém o código comum a todas as *views*.

Veja o código da *view* **app/layouts/application.html.erb**:

```
1   <!DOCTYPE html>
2   <html>
3   <head>
4     <title>Workspace</title>
5     <%= stylesheet_link_tag 'application', media: 'all', 'data-
    turbolinks-track' => true %>
6     <%= javascript_include_tag 'application', 'data-turbolinks-
    track' => true %>
7     <%= csrf_meta_tags %>
8   </head>
9   <body>
10
11  <%= yield %>
12
13  </body>
14  </html>
```

Quando uma *view* qualquer é apresentada para o usuário, seu código será mostrado substituindo o **<%= yield %>**, o qual está presente na *view* **app/layouts/application.html.erb**, formando a *view* completa que será apresentada para o usuário. Portanto, sempre que você quiser inserir um código que deve estar presente em todas as *views* da sua aplicação, basta inseri-lo na *view* **app/layouts/application.html.erb**.

Essa é uma boa prática de programação conhecida como DRY (*Don't Repeat Yourself*), traduzido como "Não se repita", a qual nos orienta a não repetir um mesmo trecho de código em mais de um local da nossa aplicação, pois, caso esse trecho de código precise ser alterado, ele precisará ser alterado em vários locais. Ao invés de repetir o trecho de código, devemos centralizá-lo num local e usá-lo quando necessário.

Partials

Partials são *views* que podem ser reaproveitadas em outras *views*. Nossa aplicação já possui um exemplo de *partial*, o **app/views/mensagens/_form.html.erb**. Essa *partial* contém o formulário de cadastro/edição de mensagens, o qual serve para dois propósitos distintos: cadastro e edição. Por isso, ele está sendo reaproveitado em duas outras *views*: **new.html.erb e edit.html.erb**.

Observe, por exemplo, o código da *view* app/views/mensagens/new.html.erb.

```
<hi>New mensagem</hi>

<%= render 'form' %>

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

A *view* **new.html.erb** inclui o *partial* **_form.html.erb** na linha que contém **<%= render 'form' %>**. Dessa forma, o código do formulário que está presente em _form.html.erb será apresentado substituindo a linha **<%= render 'form' %>**. Aqui, **'form'** é o nome do *partial* que está sendo incluído. Nesse caso, estamos incluindo o *partial* **_form.html.erb**. Vale salientar, também, que o nome do **partial** deve sempre começar com *underline* (**_**).

Modelos

Os modelos (*models*) são classes, presentes na pasta **app/models**, que herdam de **ActiveRecord::Base** e são responsáveis pela lógica de negócio da aplicação, validações, associações etc. Cada modelo corresponde a uma tabela no banco de dados. Por exemplo, o nosso projeto **Mural de Mensagens** possui um modelo chamado Mensagem (**app/models/mensagem.rb**), e esse modelo possui uma tabela correspondente chamada mensagens, a qual é responsável pelo armazenamento das **mensagens**. O nome do modelo é sempre no singular, enquanto o nome da tabela é sempre no plural.

Observe o código do modelo **Mensagem** do nosso projeto.

```
class Mensagem < ActiveRecord::Base
end
```

Confira que a classe **Mensagem** está vazia, mas isso não significa que ela não tenha métodos disponíveis. Ela herda de **ActiveRecord::Base** e, portanto, herda vários métodos que serão muito úteis no desenvolvimento da nossa aplicação. Alguns desses métodos são apresentados na Tabela 1.

Tabela 1: Métodos disponíveis em modelos Rails.

Método	Exemplo	Descrição
all	*Mensagem.all*	Retorna um array com todos os objetos armazenados no banco.
count	*Mensagem.count*	Retorna a quantidade de objetos armazenados no banco.
find	*Mensagem.find(10)*	Retorna o objeto que tem o **id** igual a 10.
where	*Mensagem.where(autor: 'Pedro)*	Retorna um array de objetos que possui o atributo **autor** igual a **'Pedro'**.
first	*Mensagem.first*	Retorna o primeiro objeto armazenado no banco.
last	*Mensagem.last*	Retorna o último objeto armazenado no banco.
find_by	*Mensagem.find_by (autor: 'Pedro')*	Retorna um objeto que possui o atributo **autor** igual a **'Pedro'**.

Fonte: Elaboração própria.

Como criamos o modelo **Mensagem** usando o comando **rails generate scaffold**, também foi gerado, além do modelo, a *migration* (vide abaixo), a qual cria a tabela **mensagens** no banco de dados

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Workspace</title>
5    <%= stylesheet_link_tag 'application', media: 'all',
   'data-turbolinks-track' => true %>
6    <%= javascript_include_tag 'application', 'data-
   turbolinks-track' => true %>
7    <%= csrf_meta_tags %>
8  </head>
9  <body>
10
11 <%= yield %>
12
```

Essa *migration* é uma classe em Ruby que herda da classe **ActiveRecord::Migration**. A classe **CreateMensagens** possui um único método, **change**, que é executado quando a *migration* é executada. O método cria a tabela mensagens no banco de dados com os campos **titulo**, **corpo**, **autor** e **email**, que foram os campos especificados no comando **rails generate scaffold**.

Leitura complementar

Aprenda mais sobre MVC, assistindo ao vídeo O que é MVC?, do canal Space Rails, disponível em https://www.youtube.com/watch?v=h9ZIPIdXFuY.

Avaliando seus conhecimentos

1) O que significa MVC?

a) *Movel-View-Controller*

b) *Model-View-Controller*

c) *Model-View-Controler*

d) *Movel-View-Controler*

e) Nenhuma das alternativas anteriores.

2) Onde ficam os controladores de um projeto Rails?

a) app/controllers

b) config/controllers

c) controllers/app
d) db/controllers
e) application/controllers

3) Não é um método disponível nos modelos em Rails:
a) all
b) first
c) count
d) findby
e) where

3) O que é um partial?
a) É uma classe em Ruby responsável por atender e responder às requisições HTTP.
b) É uma classe em Ruby, que herda de **ActiveRecord::Base**, e é responsável pela lógica de negócio, validações e associações.
c) É uma *view* que encapsula todas as outras *views* da aplicação, portanto, ela contém todo o código que é compartilhado entre todas as outras *views*.
d) É uma view que pode ser incluída em outras *views* da aplicação.
e) Nenhuma das alternativas anteriores.

Gabarito: 1) b; 2) a; 3) d; 4) d; 5) a

Resumindo

Esta aula apresentou uma visão geral sobre o padrão arquitetural do *software* conhecido como MVC. Além de introduzir como diferentes componentes, modelos, *views* e controladores se integram entre si nesse padrão, bem como funcionam cada um deles. Esperamos que você tenha compreendido bem esse conteúdo. Até mais!

Referências

CAELUM. **Desenvolvimento Ágil para Web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails**: Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBY. **Ruby programming language.** Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. **Ruby on rails:** a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern. Disponível em: http://rubyonrails.org/. Acesso em: 12 dez. 2017.

RUBYONRAILS.ORG. **Ruby on Rails Guides**. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. **Rails Gilrs Tutorial.** Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

Aula 6 - Acessando o Banco de Dados

Objetivos

Ao final desta aula, você deverá ser capaz de:

Aprender como configurar o banco de dados do projeto;

Compreender como acessar o banco de dados através do cliente no terminal de comandos;

Compreender como acessar o banco de dados através do Rails Console.

Introdução

Em outro momento, constatamos que o nosso banco de dados foi criado quando executamos o comando **rake db:migrate**. Porém, não vimos nada sobre esse banco de dados. Qual banco de dados estamos usando? Quais suas configurações? Como podemos acessá-lo? Essas são as perguntas que responderemos nesta Unidade X/Aula 10.

Quando criamos uma aplicação Rails, o banco de dados padrão usado pelo framework Rails é o SQLite. O SQLite é um banco de dados relacional bem simples, contido em um único arquivo, sendo ideal para o desenvolvimento de pequenas aplicações como a nossa.

Em qualquer aplicação Rails, as configurações de banco de dados são definidas no arquivo **config/database.yml** (mostrado abaixo). Nesse arquivo, encontram-se as configurações do banco de dados SQLite para a nossa aplicação Rails. Caso estivéssemos usando o banco de dados diferente, como MySQL ou PostgreSQL, esse arquivo seria ligeiramente diferente.

```
1  default: & default
2    adapter: sqlite3
3    pool: 5
4    timeout: 5000
5
6  development:
7    <<: *default
8    database: db/development. sqlite3
9
10 test
11   <<: *default
12   database: db/test.sqlite3
13
14 production:
15   <<: *default
16   database: db/production.sqlite3
```

Nesse arquivo, estão definidas três bases de dados diferentes: **development**, **test** e **production**. A base **development** (linha 6) é a base de dados de desenvolvimento, é ela que usamos enquanto estamos desenvolvendo a nossa aplicação. A base **test** (linha 10) é a base de dados de teste, e é ela que usamos quando executamos testes automatizados. A base **production** (linha 14) é a base de dados de produção, ou seja, utilizada quando a aplicação está em produção, isto é, *on-line* para os usuários acessarem.

Observe, ainda, que existe uma base chamada **default** (linha 1), a qual, na realidade, não é mais uma base de dados, mas um conjunto de configurações que são compartilhadas entre todas as configurações de bases de dados nesse arquivo. Dessa forma, configurações que sejam comuns às bases de dados devem ser feitas em **default**.

Quando executamos o comando **rake db:migrate**, duas dessas bases de dados foram criadas na pasta **db:** o **development.sqlite3** e o **test.sqlite3**, representando, respectivamente, as bases de desenvolvimento e de testes automatizados.

Acessando o banco de dados através do cliente SQLite

Para acessar a base de desenvolvimento, abra o seu terminal e execute o comando **sqlite3 db/development.sqlite3**. Feito isso, você estará acessando o banco de dados de desenvolvimento da sua aplicação através

do cliente SQLite no terminal de comandos. A partir desse ponto, qualquer comando SQL que você aprendeu na disciplina de Banco de Dados é válido. Além deles, você pode executar alguns comandos próprios do SQLite. Para mais detalhes sobre eles, use o comando **.help**.

Para sair do cliente SQLite, use o comando **.quit** ou **.exit**. O comando **.tables** exibe as tabelas existentes nessa base de dados. Observe a execução desse comando abaixo.

```
sqlite> .tables
mensagens          schema_migrations
```

O resultado apresentado indica que temos duas tabelas presentes na base de dados: **mensagens** e **schema_migrations**. A tabela **mensagens** foi criada quando executamos o comando **rake db:migrate**, sendo responsável por armazenar dados de mensagens cadastradas na nossa aplicação. A tabela **schema_migrations** é gerada automaticamente pelo framework Rails, e ela é responsável por armazenar dados de *migrations* que já foram executadas. Assim, o framework é capaz de determinar quais *migrations* já foram executadas e quais ainda não foram.

Atividade

Acesse o banco de dados do seu projeto usando o cliente do SQLite3 no terminal de comandos, e, em seguida, execute o comando **.tables** para visualizar as tabelas.

O comando **.schema**, seguido do nome de uma tabela, **mensagens** por exemplo, exibe o comando CREATE de uma tabela específica. Esse comando é muito útil para podermos ver a estrutura da tabela. Observe abaixo o resultado da execução do comando **.schema mensagens**.

```
CREATE TABLE "mensagens" ("id" INTEGER PRIMARY KEY AUTOINCREMENT NOT
NULL, "titulo" varchar, "corpo" text, "autor" varchar, "email" varchar,
"created_at" datetime NOT NULL, "updated_at" datetime NOT NULL);
```

Atividade

Acesse o banco de dados do seu projeto, usando o cliente do SQLite3 no terminal de comandos, e, em seguida, execute o comando **.schema** sobre uma tabela para visualizar sua estrutura.

Vamos executar o seguinte comando **SELECT** para listar as mensagens cadastradas na nossa tabela mensagens.

```
select *from mensagens;
```

Não se esqueça de colocar o ponto-e-vírgula no final da instrução SQL.

Atividade

Acesse o banco de dados do seu projeto, usando o cliente do SQLite3 no terminal de comandos, e, em seguida, execute um **SELECT** sobre uma das tabelas.

Vamos sair do cliente SQLite, usando o comando **.quit**.

Acessando o banco de dados através do Rails Console

Lembra-se do Rails Console abordado na Unidade III/Aula 03? Também podemos acessar o nosso banco de dados através do Rails Console. Entre no Rails Console, digitando o comando **rails c**. Lembre-se de que, no Rails Console, podemos usar comandos Ruby, classes e recursos existentes no nosso projeto Rails. Vamos começar usando o *model* **Mensagem**, e chamar o método **.all**, que retorna todos os objetos daquele modelo persistidos no banco de dados.

```
Mensagem.all
```

Através do Rails Console também podemos persistir novos objetos no banco. Para isso, use o *model* **Mensagem** e chame o método **create**, passando uma lista de parâmetros com os dados do nosso novo objeto.

```
Mensagem.create(titulo: "Ola", corpo: "Ola mundo!", autor: 
"Pedro", email: "pedro@email.com")
```

Não precisamos fornecer valores para **id**, **created_at** e **updated_at**, pois esses campos serão preenchidos automaticamente pelo framework. Agora, execute o comando **Mensagem.all** e veja o novo objeto persistido. Outra forma de persistir objetos é usando o método **save**. Vamos criar um objeto da classe **Mensagem** usando o método **new** e guardá-lo numa variável chamada **mensagem**.

```
mensagem = Mensagem.new(titulo: "Aula 6", corpo: "Como
está a aula?", autor: "Jalerson", email: "jalerson.lima@
email.com")
```

Perceba que esse objeto ainda não foi persistido na tabela **mensagens**. Ele apenas foi criado e guardado na variável **mensagem**. Para salvá-lo no banco de dados, devemos chamar o método **save** sobre a variável que guarda o objeto, que, nesse caso, se chama **mensagem**.

```
mensagem.save
```

Pronto! Agora, esse objeto está salvo na tabela **mensagens**. Também, podemos atualizar objetos já persistidos. Para isso, usamos o método **update_attributes** sobre o objeto persistido. Por exemplo, sabemos que, na variável **mensagem** há um objeto da classe Mensagem que está persistido. Portanto, podemos atualizá-lo usando o nome da variável que guarda o objeto, **mensagem**, e chamando o método **update_attributes**, passando os nomes e valores dos atributos que desejamos atualizar.

```
mensagem.update_attributes(titulo: "Aula 6" - Acessando o
banco de dados", autor: "Jalerson Lima")
```

Observe que estamos atualizando os campos **título** e **autor** do objeto. Chamando a variável **mensagem**, você pode ver que o objeto foi atualizado.

```
=> #<Mensagem id: 1, titulo: "Aula 6 - Acessando o banco
de dados", corpo: "Como está a aula?", autor: "Jalerson
Lima", email: "jalerson.lima@email.com", created_at:
"2018-02-03 13:48:32", updated_at: "2018-02-03 13:48:32">
```

Outro método usado para atualização de objetos é o **update_attribute** (no singular). A diferença entre eles é que o **update_attributes** é capaz de atualizar vários campos de uma única vez, enquanto o **update_attribute** atualiza um único atributo, conforme ilustrado abaixo.

```
Mensagem.update_attribute(:corpo, “Como está a aula? Estão
gostando?”)
```

Observe que o comando **update_attribute**, usado acima, atualiza apenas o atributo corpo do objeto guardado na variável mensagem. Outro método

muito útil presente em qualquer modelo Rails é o método **find**, o qual recebe um **id** como parâmetro e retorna o objeto que possui aquele **id** específico.

```
Mensagem.find(1)
=> #Mensagem id: 1, titulo: "Aula 6 - Acessando o banco de
dados", corpo: "Como está a aula? autor: "Jalerson Lima",
email: "jalerson.lima@email.com", created_at: "2018-02-03
13:48:32", updated_at: "2018-02-03 13:48:32">
```

Para apagar um objeto da tabela **mensagens**, basta chamar o método **delete** sobre a variável que guarda o objeto. Portanto, podemos chamar **mensagem.delete** para apagar esse objeto do banco de dados.

```
mensagem.delete
```

Para sair do Rails Console, basta executar o comando **exit**.

Atividade

Acesse o banco de dados do seu projeto, usando o Rails Console. Em seguida, crie um objeto da modelo **Mensagem**, salve-o no banco de dados, localize-o usando o método **find**, atualize um (ou mais) de seus campos e, por fim, apague-o.

Resumindo

Nesta aula, aprendemos como configurar o banco de dados em um projeto Rails. Ainda estudamos como acessar o banco de dados do seu projeto utilizando um cliente no terminal de comandos, e como acessá-lo também utilizando o Rails Console. Até a próxima!

Referências

CAELUM. **Desenvolvimento Ágil para Web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails:** Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBY. **Ruby programming language**. Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. **Ruby on rails**: a web-application framework that includes everything

needed to create database-backed web applications according to the model-view-controller (mvc) pattern. Disponível em: http://rubyonrails.org/. Acesso em: 12 dez. 2017.

RUBYONRAILS.ORG. **Ruby on Rails Guides**. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. **Rails Gilrs Tutorial**. Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

SQLITE. **Sqlite home page.** Disponível em: https://www.sqlite.org/. Acesso em: 04 fev. 2018.

Aula 7 - Melhorando a Apresentação Parte I

Objetivos

Ao final desta aula, você deverá ser capaz de:

Aprender como instalar e configurar o Bootstrap no projeto;

Compreender como aplicar os estilos e usar os componentes do Bootstrap;

Estilizar a *view* de listagem de mensagens;

Configurar a rota raiz do projeto.

Introdução

O Bootstrap é uma poderosa biblioteca para construção de interfaces gráficas para a Web. Com ela, você tem a sua disposição diversos componentes, estilos e efeitos prontos para serem utilizados. Dessa forma, pode-se desenvolver uma aplicação com uma interface gráfica mais agradável sem ter de criar tudo do zero.

Acesse o *site* do Bootsrap, através do endereço http://getbootstrap.com, e clique no *link* Example, presente no menu que aparece na parte superior do *site*. Nessa página, você pode visualizar exemplos de vários componentes e de páginas construídas com o Bootstrap.

Instalando o Bootstrap

Para instalar o Bootstrap no nosso projeto, abra a *view* **app/views/layouts/application.html.erb**, e, nela, vamos incluir as referências necessárias para o perfeito funcionamento do Bootstrap. Mas, primeiro, vale perguntar: você entendeu porque vamos incluir as referências nessa view? Queremos que o Bootstrap fique disponível em todas as views do nosso projeto, dessa manei-

ra, precisamos incluir as referências nessa view, a qual possui o código que é compartilhado entre todas as outras views.

O código da sua *view* **application.html.erb** deve ser similar ao código apresentado abaixo.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Workspace</title>
5    <%= stylesheet_link_tag    'application', media: 'all',
     'data-turbolinks-track' => true %>
6    <%= javascript_include_tag 'application', 'data-turbo-
     links-track' => true %>
7    <%= csrf_meta_tags %>
8  </head>
9  <body>
10
11 <%= yield %>
12
13 </body>
14 </html>
```

Vamos, inicialmente, incluir a referência do arquivo de folhas de estilos CSS do Bootstrap. Essa referência deve ser incluída entre as tags de abertura e de fechamento (**<head>** e **</head>**) do cabeçalho da página, conforme ilustrado abaixo.

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
com/bootstrap/4.0.0/css/bootstrap.min.css">
```

O próximo passo é incluir três referências para scripts JavaScript antes da tag de fechamento **</body>**, como a seguir.

```
<script src="https://code.jquery.com/jquery-3.2.1.slim.min.
js"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/
umd/popper.min.js">
</script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/
js/bootstrap.min.js"></script>
```

O último script incluído (**bootstrap.min.js**) é o JavaScript do próprio Bootstrap, responsável por nos fornecer as funcionalidades dessa biblioteca. Os dois primeiros scripts incluídos, o JQuery (**jquery-3.2.1.slim.min.js**) e o Popper (**popper.min.js**), scripts dos quais o Bootstrap depende para seu perfeito funcionamento. O JQuery (https://jquery.com) é outro exemplo de biblioteca JavaScript muito famosa entre os desenvolvedores de software, pois ela fornece muitos outros recursos de desenvolvimento bastante úteis.

Incluída a folha de estilos CSS e os scripts JavaScript, sua view **application.html.erb** deve se parecer com o código mostrado abaixo.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Workspace</title>
5    <%= stylesheet_link_tag    'application', media: 'all',
     'data-turbolinks-track' => true %>
6    <%= javascript_include_tag 'application', 'data-turbo-
     links-track' => true %>
7    <%= csrf_meta_tags %>
8    <link rel="stylesheet"
   href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/
   bootstrap.min.css">
9  </head>
10 </body>
11
12 <%= yield %>
13
14 <script src="https://code.jquery.com/jquery-3.2.1.slim.min.
   js"></script>
15 <script
   src="https://cdnjs.cloudflare.com/ajax/libs/popper.
   js/1.12.9/umd/popper.min.js"></script>
   <script
   src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/
   bootstra
16 p.min.js"></script>
   </body>
   </html>
17
18
```

Feito isso, o Bootstrap está instalado no seu projeto, contudo, ainda precisamos aplicar os estilos e utilizar os componentes do Bootstrap.

Atividade

Siga as orientações da aula para instalar e configurar o Bootstrap no seu projeto.

Usando o Bootstrap

Antes de começar a utilizar os estilos do Bootstrap, vamos apagar alguns estilos gerados automaticamente pelo framework Rails quando executamos o comando **rails generate scaffold**. Para isso, apague o arquivo **app/assets/stylesheets/scaffolds.scss**. Os estilos presentes nesse arquivo não nos interessam e podem entrar em conflito com os estilos do próprio Bootstrap.

Em seguida, vamos incluir no nosso projeto alguns estilos particulares para a nossa aplicação. Abra a folha de estilos **app/assets/stylesheets/application.css**, e, após os comentários (não apague os comentários), vamos usar o seguinte código:

```
1  body {
2    background-color: #EBEBEB !important;
3  }
4
5  .container {
6    background-color: #FFF;
7    border: 1px solid #D1D1D1;
8    margin: 1em;
9    padding: 1em;
10 }
11
12 .rounded-corners {
13   -moz-border-radius:7px;
14   -webkit-border-radius:7px;
15   border-radius:7px;
16 }
17
18 .card {
19   margin-bottom: 1em;
20 }
```

Nesse código, alteramos a cor de fundo para cinza na *tag* **body** (linha 1). Além disso, estamos aplicando alguns outros estilos para a classe **container**

(linha 5): alterando a cor de fundo, definindo uma borda e alterando as margens externa e interna. Por fim, estamos criando a classe **rounded-corners** (linha 12), para que possamos arredondar bordas, e alternado a classe **card** (linha 18) do Bootstrap definir a margem externa em **1em**. Feito isso, salve o arquivo.

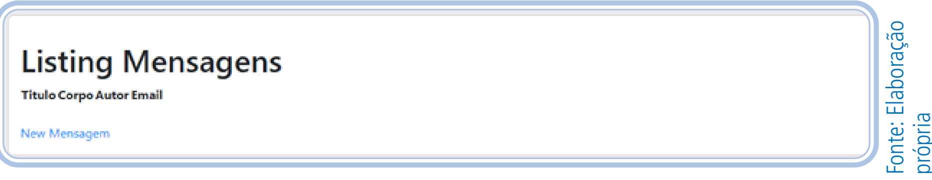
Abra novamente a *view* **app/views/layouts/application.html.erb** e coloque o yield dentro de uma **div** com as classes **container e rounded-corners**. Dessa forma, podemos garantir que todo o conteúdo que for apresentado no **yield** estará dentro dessa **div**.

```
<div class="container rounded-corners">
  <%= yield %>
</div>
```

Atividade

Siga as orientações destw aula para criar os estilos em **app/assets/stylesheets/application.css** e estilizar a *view* **app/views/layouts/application.html.erb**.

Feito isso, vamos ver como essas mudanças já melhoraram a apresentação do nosso projeto. Salve a view, e, no terminal de comandos, execute **rails s**. Com a aplicação em execução, abra seu navegador de Internet e acesse o endereço http://localhost:3000/mensagens. Você deverá visualizar uma página semelhante a ilustrada na Figura 1.



Fonte: Elaboração própria

Figura 1: Página de listagem de mensagens.

Aplicando estilos na listagem de mensagens

Vamos dar continuidade abrindo a *view* **app/views/mensagens/index.html.erb**, a *view* responsável pela apresentação da listagem de mensagens. Na primeira linha dessa *view*, temos a variável **notice**. A variável **notice** contém uma mensagem referente à última operação realizada pelo usuário.

Substitua a linha 1 dessa *view* pelo seguinte código:

```
<% if notice.present? %>
  <div id="notice" class="alert alert-success" role="alert"><%= notice %></div>
<% end %>
```

Dessa forma, estamos, inicialmente, verificando se a variável **notice** existe, usando o método **present?**. Se a variável **notice** existir, significa que há uma mensagem a ser exibida para o usuário. Essa mensagem será exibida dentro de uma **div** com as classes **alert** e **alert-success** para dar à **div** um visual de alerta. Para mais detalhes sobre o componente *alert* do Bootstrap, visite a documentação referente a esse componente, através do endereço http://getbootstrap.com/docs/4.0/components/alerts/.

Em seguida, abaixo desse código, vamos adicionar um *link* que permita adicionar uma nova mensagem ao mural. Para isso, use o seguinte código

```
<div class="float-md-right">
  <%= link_to 'Nova Mensagem', new_mensagem_path, class: "btn btn-primary" %>
</div>
```

Estamos usando uma **div** com a classe **float-md-right**. Essa classe pertence ao Bootstrap, e ela move a **div** para a direita. Dentro da **div**, temos um fragmento código em Ruby. Estamos usando o método auxiliar **link_to** para criar um *link*. Esse método recebe os seguintes parâmetros: o texto que irá aparecer no *link*, o endereço para o qual o usuário será redirecionado e a classe CSS que será aplicada ao *link*.

No segundo parâmetro, no qual devemos informar o endereço para onde o usuário será redirecionado ao clicar no *link*, estamos usando **new_mensagem_path**. Esse é um outro método auxiliar, o qual define um path, ou caminho, da nossa aplicação. Esse método em particular contém o endereço do formulário para criar uma nova mensagem. Métodos desse tipo foram criados na nossa aplicação quando definimos o **resources:mensagens** no arquivo de rotas (Unidade IV/Aula 05).

O terceiro parâmetro fornecido para o **link_to** são nomes de classes CSS para estilizar o *link* e deixá-lo com o visual de um botão. As classes **btn** e **btn-primary** também são fornecidas pelo Bootsrap. Para mais detalhes

sobre elas, visite a documentação através do endereço http://getbootstrap.com/docs/4.0/components/buttons/.

Em seguida, na tag **<h1>**, substitua **Listing Mensagens** por **Mural de Mensagens**.

A seguir, observe que há uma tabela com a listagem de mensagens. Cada mensagem cadastrada no banco de dados está sendo exibida em uma linha da tabela. Entretanto, iremos utilizar um componente visual do Bootstrap chamado Card (http://getbootstrap.com/docs/4.0/components/card). Um card é um cartão que possui título, corpo e rodapé, e cada mensagem será apresentada no seu próprio *card*. Vamos substituir todo o resto do conteúdo da *view* pelo código abaixo.

```
1  <% if @mensagens.empty? %>
2    <p>Nenhuma mensagem encontrada.</p>
3  <% else %>
4    <% @mensagens.each do |mensagem| %>
5      <div class="card">
6        <div class="card-header">
7          <h5><%= mensagem.titulo %></h5>
8        </div>
9        <div class="card-body">
10         <%= mensagem.corpo %>
11       </div>
12       <div class="card-footer">
13         <div class="btn-group">
14           <%= link_to 'Detalhar', mensagem, class: 'btn btn-
   secondary btn-sm' %>
15           <%= link_to 'Editar', edit_mensagem_
   path(mensagem), class: 'btn btn-secondary btn-sm' %>
16           <%= link_to 'Excluir', mensagem, method: :delete,
   data: { confirm: 'Tem certeza?' }, class: 'btn btn-secondary
17 btn-sm' %>
18         </div>
19         <div class="float-md-right">
             <small>Enviada por <%= mail_to mensagem.email,
   mensagem.autor %> em <%=
20 mensagem.created_at.strftime("%d/%m/%Y") %></small>
21         </div>
22       </div>
23     </div>
24   <% end %>
   <% end %>
```

Inicialmente, na linha 1, verificamos se a variável **@mensagens** contém alguma mensagem usando o método **empty?**, que retorna verdadeiro caso não haja mensagens. Se ela estiver vazia, o parágrafo **<p>Nenhuma mensagem encontrada.</p>** será exibido (linha 2). Caso haja pelo menos uma mensagem, vamos iterar na variável **@mensagens** e guardar cada mensagem na variável **mensagem** usando o método **each** (linha 4).

Para cada mensagem, iremos apresentar uma **div** com a classe **card** (linha 5). Cada **card** possui um título (**div** com a classe **card-header** na linha 6), um corpo (div com a classe **card-body** na linha 9) e um rodapé (**div** com a classe **card-footer** na linha 12). No título do **card**, estamos exibindo o título da mensagem (linha 7). No corpo do **card**, estamos exibindo o corpo da mensagem (linha 10), e, no rodapé do **card** (linha 12), estamos exibindo uma **div** (linha 13) a qual contém alguns *link*s que permitem o detalhamento (linha 14), a edição (linha 15) e a exclusão (linha 16) da mensagem.

Na linha 14, o *link* detalhar está recebendo como parâmetro o objeto **mensagem**. Quando fornecemos um objeto, ao invés de um *path*, para o método **link_to**, o *link* gerado irá redirecionar o usuário para a página de detalhamento do objeto.

No segundo **link_to**, presente na linha 15, estamos usando **edit_mensagem_path**, que é o path o qual irá redirecionar o usuário para o formulário de edição da mensagem. Observe também que estamos passando o objeto **mensagem** como parâmetro para esse path. Isso é necessário, porque o *path* precisa saber qual mensagem será editada pelo usuário.

O terceiro **link_to**, presente na linha 16, também recebe o objeto **mensagem** como parâmetro. Contudo, ele não irá redirecionar o usuário para o detalhamento da mensagem, e sim para o método **delete** do controlador mensagens, responsável por apagar essa mensagem. Isso ocorre, porque estamos especificando **method: :delete**. O quarto parâmetro, **data**, apresenta uma mensagem de confirmação para o usuário, para que ele possa ter a chance de confirmar se deseja realmente apagar a mensagem.

Por fim, é apresentada uma **div** (linha 18) com algumas informações gerais da mensagem. Nesse contexto estamos usando outro método auxiliar, chamado **mail_to**, que gera um *link* para envio de e-mail (linha 19). Nesse método, passamos como parâmetro o endereço de e-mail cadastrado na mensagem e o nome do autor. Em seguida, apresentamos a data de criação da mensagem

(**created_at**), e usamos o método **strftime** para formatar a data em dia, mês e ano. Não se esqueça de salvar o arquivo antes de fechá-lo.

Atividade

Siga as orientações da aula estilizar a *view* **app/views/mensagens/index.html.erb**.

Configurando a rota raiz

Para finalizar, vamos fazer uma configuração na nossa aplicação, para que a listagem de mensagens seja apresentada quando acessarmos o endereço raiz (**http://localhost:3000**), sem a necessidade de digitar **/mensagens**. Para isso, abra o arquivo de rotas **config/routes.rb** e inclua a linha

```
root 'mensagens#index'
```

Essa configuração está definindo que, quando o usuário acessar o endereço raiz da nossa aplicação, ou seja, o **root**, quem irá atender essa requisição é o método **index** do controlador **mensagens**.

Salvo o arquivo de rotas, e, no seu navegador de Internet, recarregue a página para que você possa ver as alterações. Observe que a nossa aplicação já está bem diferente! Cada mensagem dentro do seu próprio *card*, com título, corpo e rodapé. No rodapé, temos acesso aos *link*s para detalhar, editar e excluir cada uma das mensagens. Naturalmente as outras *views* da nossa aplicação ainda não estão estilizadas, pois isso será feito nas próximas aulas.

Atividade

Siga as orientações da aula para configurar a rota raiz do projeto para **'mensagens#index'**.

Resumindo

Esta aula apresentou como instalar e configurar a biblioteca Bootstrap no seu projeto Rails. Além disso, aprendemos como aplicar os estilos e como usar os componentes do Bootstrap para melhorar a apresentação da *view* de listagem de mensagens. Estudamos, ainda, como configurar a rota raiz do projeto para facilitar o acesso à listagem de mensagens. Esperamos que tenha compreendido bem. Até a próxima aula!

Referências

BOOTSTRAP. **Bootstrap:** the most popular html, css and js library in the world. Disponível em: http://getbootstrap.com/. Acesso em: 09 fev. 2018.

CAELUM. **Desenvolvimento Ágil para Web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails**: Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBY. **Ruby programming language**. Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. **Ruby on rails**: a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern. Disponível em: http://rubyonrails.org/. Acesso em: 12 dez. 2017.

RUBYONRAILS.ORG. **Ruby on Rails Guides**. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. **Rails Gilrs Tutoria**l. Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

Aula 8 - Melhorando a Apresentação Parte II

Objetivos

Ao final desta aula, você deverá ser capaz de:

Aplicar os estilos e usar os componentes do Bootstrap na página de detalhamento das mensagens;

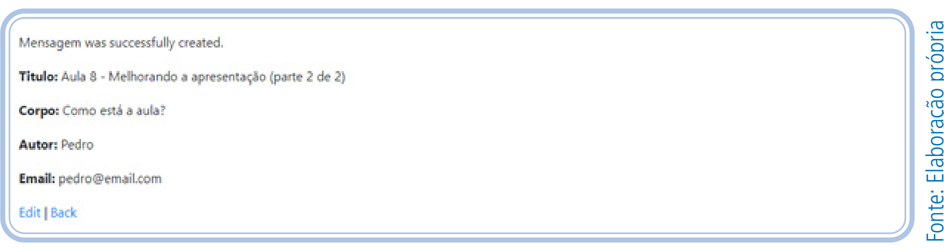
Instalar e configurar a *gem Simple Forms* no projeto;

Usar o *Simple Forms* para construir o formulário de cadastro/edição de mensagens;

Traduzir as mensagens de alerta da aplicação.

Introdução

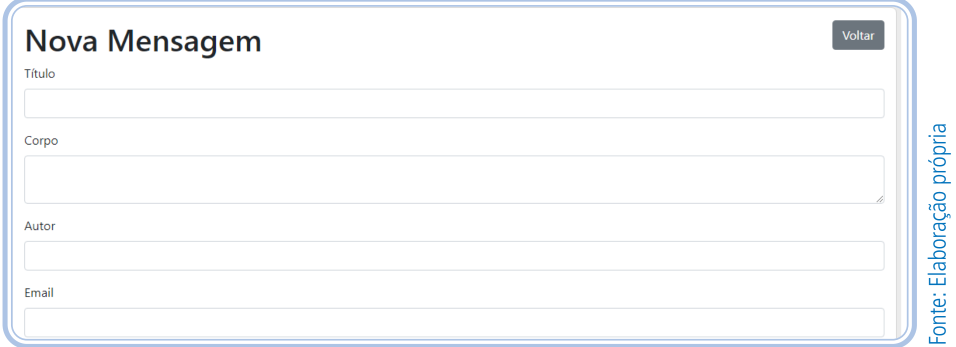
Nesta aula, iremos melhorar o design da *view* de detalhamento das mensagens, os formulários de edição e cadastro de novas mensagens. Atualmente, a nossa página de detalhamento de mensagens deve ser similar à ilustrada na Figura 1.



Fonte: Elaboração própria

Figura 1: Detalhamento da mensagem

Os formulários de cadastro e edição de mensagens devem ser similares ao ilustrado na Figura 2.



Fonte: Elaboração própria

Figura 2: Formulário de cadastro de novas mensagens

Aplicando estilos

Nesta seção, vamos estudar como estilizar a página de detalhamento de mensagens e como estilizar os formulários de cadastro e edição de mensagens.

Estilizando o detalhamento de mensagens

Abra a *view* de detalhamento das mensagens **app/views/mensagens/show.html.erb** e substitua todo o código dessa *view* pelo código abaixo.

```
1  <% if notice.present? %>
2    <div id="notice" class="alert alert-success" role="alert"><%=
   notice %></div>
3  <% end %>
4
5  <div class="float-md-right btn-group">
6    <%= link_to 'Voltar', mensagens_path, class: 'btn btn-
7  secondary' %>
     <%= link_to 'Editar', edit_mensagem_path(@mensagem), class:
8  'btn btn-secondary' %>
     <%= link_to 'Excluir', @mensagem, method: :delete, data: {
9  confirm: 'Tem certeza?' }, class: 'btn btn-secondary' %>
10 </div>
11
12 <h1>Detalhar Mensagem</h1>
13
14 <div class="card">
15   <div class="card-header">
16     <h5><%= @mensagem.titulo %></h5>
17   </div>
18   <div class="card-body">
19     <%= @mensagem.corpo %>
```

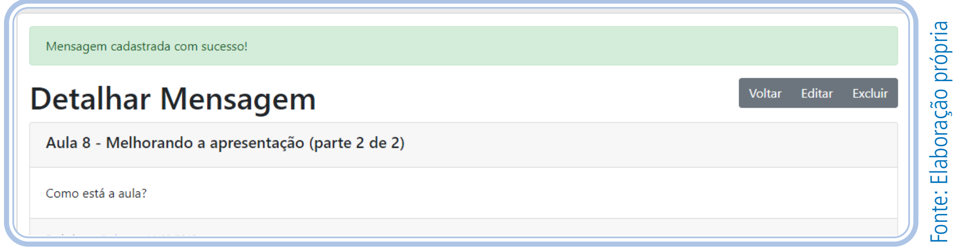
```
20    </div>
21    <div class="card-footer">
        <small>Enviada por <%= mail_to @mensagem.email,
22  @mensagem.autor %> em <%=
23  @mensagem.created_at.strftime("%d/%m/%Y") %></small>
      </div>
    </div>
```

No início do código, entre as linhas 1 e 3, temos o mesmo trecho que usamos na *view* **index.html.erb**, no qual verificamos se existe uma variável chamada **notice** (linha 1) para exibir uma mensagem para o usuário. Logo, temos uma **div** com a classe **float-md-right** (linha 5), que desloca todo esse conteúdo para a direita da tela. Ainda nessa **div**, temos a classe **btn-group**, usada para agrupar os botões. Para mais detalhes sobre a classe **btn-group**, consulte a documentação do Bootstrap em: http://getbootstrap.com/docs/4.0/components/button-group.

Dentro dessa **div** (linha 5), temos três *link*s: o primeiro é usado para voltar para a listagem de mensagens (linha 6). Nesse aspecto, é usado **mensagens_path** para se referir ao endereço de listagem de mensagens. O segundo *link* direciona o usuário para a *view* de edição da mensagem (linha 7), e o terceiro *link* para a exclusão da mensagem (linha 8).

Em seguida, temos uma *tag* **h1** com o conteúdo Detalhar Mensagem (linha 11). E, por fim, temos novamente um **card** que exibe os detalhes da mensagem (linha 13). No cabeçalho do **card** (linha 14), temos o título da mensagem (linha 15). No corpo do **card** (linha 17), temos o corpo da mensagem (linha 18). No rodapé do **card** (linha 20), temos algumas informações gerais sobre a mensagem (linha 21).

Salve o arquivo e acesse a página de detalhamento de uma mensagem qualquer para ver as mudanças. A página que você deve visualizar deve ser semelhante à da Figura 3.



Fonte: Elaboração própria

Figura 3: Página de detalhamento de mensagens estilizada

Atividade

Siga as orientações da aula para aplicar os estilos do Bootstrap à página de detalhamento de mensagens.

Aperfeiçoando os formulários

No caso dos formulários de cadastro e edição de mensagens, iremos não apenas melhorá-los visualmente, mas também simplificar sua implementação. Isso será possível por meio da *gem Simple Forms*. Lembra-se do conceito de *gems* estudado na Unidade I/Aula 01? As *gems* introduzem novas funcionalidades, recursos e/ou facilidades ao projeto Rails. Nesse caso, o S*imple Forms* simplifica a implementação de formulários.

Para instalar uma *gem* ao seu projeto, devemos consultar, no *site* da gem, suas instruções de instalação e de uso. No caso do *Simple Forms*, o endereço do seu *site* é https://github.com/plataformatec/simple_form. Acesse esse endereço no seu navegador e observe que as instruções de instalação (*Installantion*) solicitam que você inclua a seguinte linha no **Gemfile** do seu projeto.

```
gem 'simple_form'
```

Salve seu **Gemfile** e, em seguida, realize a instalação da gem, executando o comando **bundle install** no terminal de comandos. Feito isso, as instruções de instalação da gem orientam que você deve executar o seguinte comando no terminal:

```
rails generate simple_form:install --bootstrap
```

Esse comando irá instalar os arquivos do *Simple Forms* ao seu projeto, bem como integrar essa instalação ao Bootstrap (para usar seus estilos e componentes). Se o servidor de desenvolvimento estiver em execução, será preciso reiniciá-lo para a *gem* ser carreada ao projeto.

A página da *gem* também oferece instruções sobre como utilizá-la. Vamos dar início as mudanças no nosso formulário, abrindo o *partial* **app/views/mensagens/_form.html.erb**. Lembre-se de que esse *partial* é a *view*, a qual contém o formulário usado para cadastro e edição de mensagens. Desse modo, alterando-o, estaremos alterando os formulários de cadastro e edição. Substitua o código dessa *view* pelo código:

```
1  <%= simple_form_for @mensagem do |f| %>
2    <%= f.input :titulo, label: 'Título' %>
3    <%= f.input :corpo, as: :text %>
4    <%= f.input :autor %>
5    <%= f.input :email %>
6    <%= f.button :submit, 'Enviar', class: 'btn-primary' %>
7  <% end %>
```

O framework Rails oferece um método auxiliar chamado **form_for** para construção de formulários. O Simple Forms, por sua vez, oferece o **simple_form_for** (linha 1), que recebe a variável a qual contém o objeto (**@mensagem**). Usamos o método **input** para gerar cada campo no formulário. O primeiro parâmetro do **input** é o nome do atributo, no qual o campo deve ser gerado. Portanto, para gerar um campo para o atributo **titulo**, usamos: **<%= f.input :titulo, label: 'Título' %>**.

No método **input**, apenas o primeiro parâmetro é obrigatório, contudo, outros parâmetros podem ser fornecidos. O parâmetro **label** indica o rótulo do campo. Fornecemos o **label** ao primeiro campo (**titulo**) para que o rótulo seja escrito com o acento. O parâmetro **as: :text**, fornecido no segundo campo, indica que esse deve ser um campo de texto (com múltiplas linhas).

Por fim, usamos o método **button** (**<%= f.button :submit, 'Enviar', class: 'btn-primary' %>**), na linha 6, para gerar o botão de envio do formulário. O segundo parâmetro (**'Enviar'**) especifica o texto que deve aparecer no botão. O terceiro parâmetro (**class: 'btn-primary'**) pertence ao Bootstrap e estiliza o botão em azul.

Feito isso, salve esse arquivo e abra o formulário, usando seu navegador de Internet e introduzindo **/mensagens/new** no final do endereço. Você deverá visualizar um formulário semelhante ao apresentado na Figura 4.

Fonte: Elaboração própria

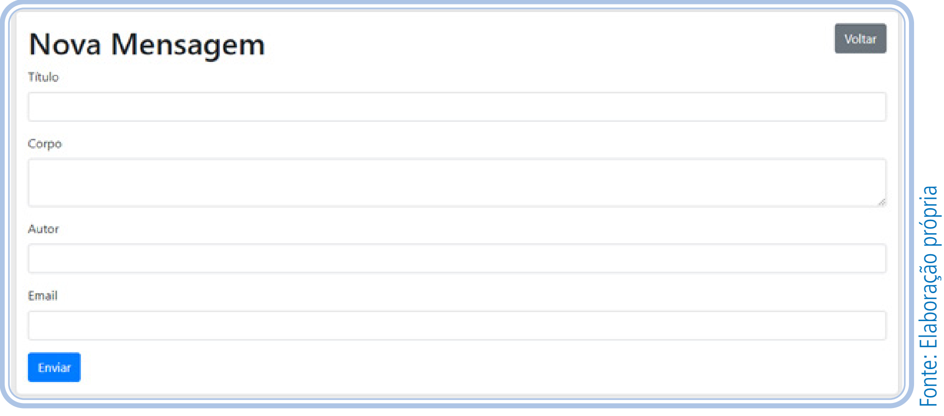
Figura 4: Formulário estilizado, usando Simple Forms e Bootstrap

Observe que, agora, o formulário está bem mais agradável visualmente e, além disso, seu código está muito mais simples e enxuto. Para melhorar ainda mais, precisamos traduzir o ***New* Mensagem** para **Nova Mensagem**, transformar o *link* Back num botão e posicioná-lo melhor.

Para isso, abra a *view* **app/views/mensagens/new.html.erb** e substitua pelo código abaixo.

```
1  <div class="float-md-right">
2    <%= link_to 'Voltar', mensagens_path, class: "btn btn-
3  secondary" %>
4  </div>
5
6  <h1>Nova Mensagem</h1>
7
8  <h1>Editar Mensagem</h1>
   <%= render 'form' %>
```

Inicialmente, posicionamos uma **div** no lado direito da tela, usando a classe **float-md-right** do Bootstrap (linhas 1 a 3). Essa **div** contém o *link* para voltar à listagem de mensagens. Em seguida, alteramos a *tag* **<h1>** (linha 5) para **Nova Mensagem**, e, por fim, usamos o **render 'form'** (linha 7) para carregar o *partial* **_form.html.erb** nesse local. Salve essa *view* e recarregue o formulário no navegador para visualizar as alterações. Seu formulário de cadastro deve estar parecido com o ilustrado na Figura 5.



Fonte: Elaboração própria

Figura 5: Novo formulário de cadastro de mensagens

Se você tiver a curiosidade de olhar o formulário de edição de uma mensagem, vai observar que nem todas as mudanças que fizemos estarão visíveis nele, conforme ilustrado na Figura 6.

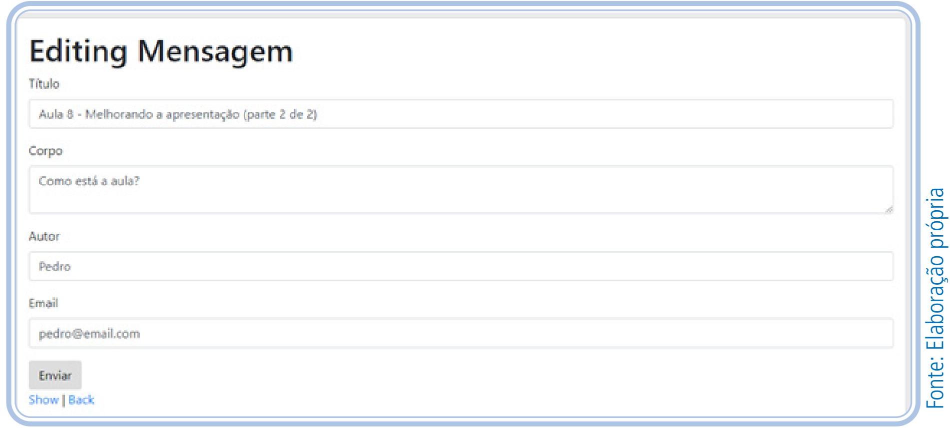


Figura 6: Formulário de edição da mensagem

Na Figura 6, o título da página ainda consta como ***Editing* Mensagem**, e os *links* *Show* e *Back* estão presentes após o botão **Enviar.** Isso ocorre, porque ainda não realizamos as mudanças na *view* **app/views/mensagens/edit.html.erb**.

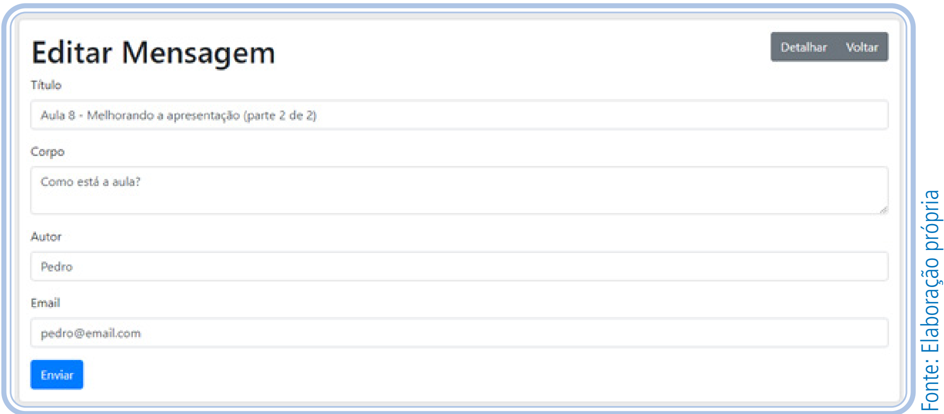
Abra a *view* **app/views/mensagens/edit.html.erb** e substitua todo o código da *view* pelo código

```
1  <div class="float-md-right btn-group">
2    <%= link_to 'Detalhar', @mensagem, class: "btn btn-seconda-
   ry"
3  %>
4    <%= link_to 'Voltar', mensagens_path, class: "btn btn-
5  secondary" %>
6  </div>
7
8  <h1>Editar Mensagem</h1>
   <%= render 'form' %>
```

Inicialmente, temos uma **div** com as classes **float-md-right** e **btn-group** (linha 1), ambas pertencentes ao Bootstrap. A classe **float-md-right** posiciona o conteúdo da **div** ao lado direito da tela, enquanto a classe **btn-group** agrupa os botões presentes dentro da **div**. Dentro dessa **div**, existem dois *links*: o primeiro direciona o usuário para o detalhamento da mensagem (linha 2) e o segundo direciona o usuário de volta para a listagem de mensagens (linha 3).

Em seguida, traduzimos ***Editing* Mensagem** para **Editar Mensagem** (linha 6), e, por fim, usamos o método **`render`** (linha 8) para apresentar o *partial* **app/views/mensagens/_form.html.erb** que contém o formulário de mensa-

gens. Feito isso, o formulário de edição de mensagens deve ser apresentado como ilustrado na Figura 7.



Fonte: Elaboração própria

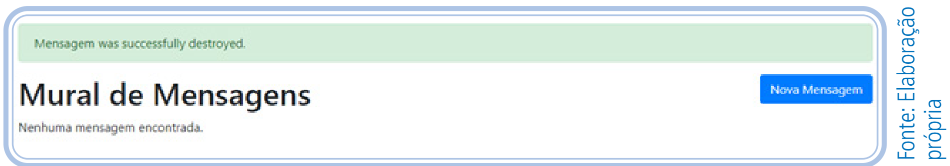
Figura 7: Novo formulário de edição de mensagens

Atividade

Siga as orientações da aula para instalar e configurar a *gem Simple Forms*, bem como aplicar os estilos do Bootstrap ao formulário de cadastro e edição de mensagens.

Traduzindo os alertas

Você deve ter percebido que, quando realizamos uma operação de cadastro, edição ou exclusão de mensagens, um alerta é exibido no topo da página para indicar que a operação foi realizada com sucesso (ver Figura 8).



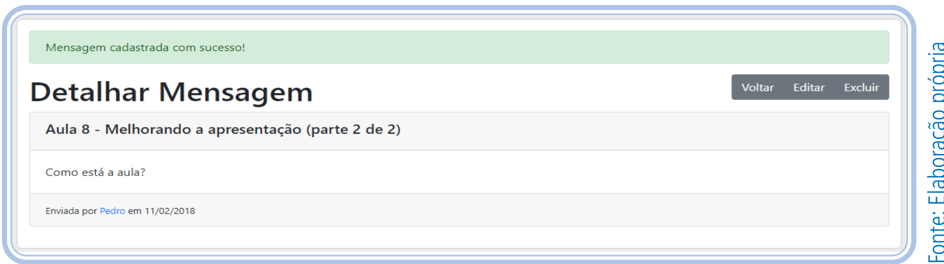
Fonte: Elaboração própria

Figura 8: Alerta em inglês

Observe que a mensagem de alerta ainda está em inglês. Vamos, agora, traduzir essas mensagens de alerta para português. Para isso, abra o controlador **app/controllers/mensagens_controller.rb** e traduza as mensagens de **notice** nas linhas 31, 45 e 59.

```
31  format.html { redirect_to @mensagem, notice: 'Mensagem
    cadastrada com sucesso!' }
45  format.html { redirect_to @mensagem, notice: 'Mensagem atualizada
    com sucesso!' }<%= f.input :titulo, label: 'Título' %>
59  format.html { redirect_to mensagens_url, notice: 'Mensagem
    apagada com sucesso!' }<%= f.input :corpo, as: :text %>
```

Feito isso, ao realizar qualquer uma das operações mencionadas anteriormente, a mensagem de alerta deve ser exibida em português, como constatamos na Figura 9.



Fonte: Elaboração própria

Figura 9: Mensagem de alerta traduzida

Atividade

Siga as orientações da aula para traduzir as mensagens de alerta presentes no controlador **app/controllers/mensagens_controller.rb**.

Resumindo

Nesta aula, concluímos as melhorias na apresentação da nossa aplicação, aplicando os estilos e os componentes do Bootstrap na página de detalhamento de mensagens. Além disso, aprendemos a instalar e a configurar a *gem Simple Forms* para construção do formulário de cadastro e edição de mensagens, e, por fim, traduzimos as mensagens de alerta da nossa aplicação.

Referências

BOOTSTRAP. **Bootstrap**: the most popular html, css and js library in the world. Disponível em: http://getbootstrap.com/. Acesso em: 09 fev. 2018.

CAELUM. **Desenvolvimento Ágil para Web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails**: Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

PLATAFORMATEC/SIMPLE_FORM. **Forms made easy for rails! it's tied to a simple dsl, with no opinion on markup**. Disponível em: https://github.com/plataformatec/simple_form. Acesso em: 09 fev. 2018.

RUBY. **Ruby programming language**. Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. **Ruby on rails**: a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern. Disponível em: http://rubyonrails.org/. Acesso em: 12 dez. 2017.

RUBYONRAILS.ORG. **Ruby on Rails Guides**. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. **Rails Gilrs Tutorial**. Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

Aula 9 - DRY - *Don't Repeat Yourself*

Objetivos

Ao final desta aula, você deverá ser capaz de:

Compreender o conceito de DRY;

Reconhecer o problema de repetição de código em nossa aplicação;

Aplicar o DRY para resolver o problema de repetição.

Introdução

Lembra do conceito de DRY (*Don't Repeat Yourself*) apresentado na Unidade I/Aula 01? Caso não se lembre, DRY é uma boa prática de programação que orienta a não repetir um trecho de código na aplicação. Essa recomendação existe porque, se precisarmos alterar esse trecho de código, teremos de fazê-la em vários locais, correndo o risco de esquecermos de fazer a alteração em um desses locais.

A solução proposta pelo DRY é a seguinte: um trecho de código, o qual é necessário em vários locais, precisa ser colocado em um único local onde ele possa ser chamado (requerido) nos locais em que ele é necessário. Caso esse trecho de código precise ser alterado, ele será modificado num único local, e sua mudança terá efeito em todos os locais onde ele for chamado.

Aqui, vamos apresentar um pequeno problema de repetição de código existente na nossa aplicação e como aplicar a ideia de DRY para resolvê-lo. Conforme vimos na aula passada, a nossa aplicação usa mensagens de alerta (Figura 1) no topo da página para indicar ao usuário quando uma operação de cadastro, edição ou exclusão foi realizada com sucesso.

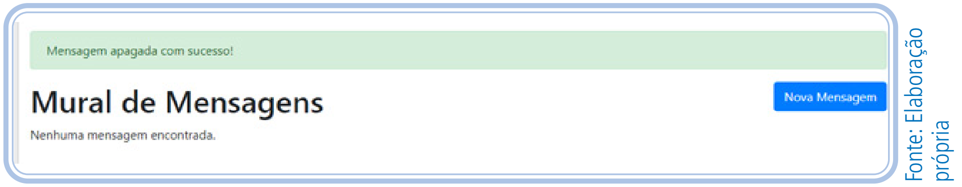


Figura 1: Alerta de exclusão de mensagem**.**

Suponha que não gostamos das cores desse alerta e queremos alterar a cor de fundo e as letras para azul, como ilustrado na Figura 2.

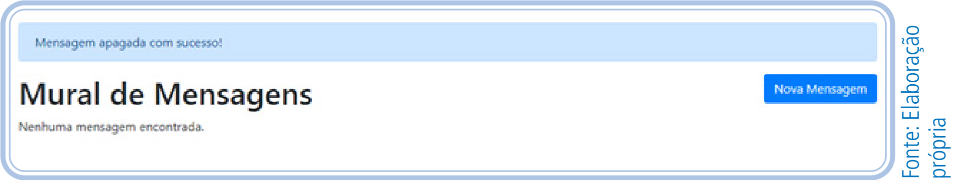


Figura 2: Nova mensagem de alerta.

Para fazer isso, precisamos abrir a *view* **app/views/mensagens/index.html.erb** e alterar, na linha 2, a classe **alert-success** para **alert-primary**. Lembrando que essas são classes CSS fornecidas pelo Bootstrap. Para mais detalhes sobre elas, consulte a documentação, através do endereço http://getbootstrap.com/docs/4.0/components/alerts/.

```
1  <% if notice.present? %>
2    <div id="notice" class="alert alert-primary"
   role="alert"><%= notice %></div>
3  <% end %>
```

Feito isso, você irá perceber que, ao excluir uma mensagem, o alerta será exibido (Figura 2). Contudo, ao cadastrar ou editar uma mensagem, o alerta ainda está sendo exibido com suas cores anteriores, como ilustrado na Figura 1. Por que isso está ocorrendo?

Isso está ocorrendo porque o trecho de código que exibe o alerta está repetido em dois locais diferentes: na *view* **app/views/mensagens/index.html.erb** e na *view* **app/views/mensagens/show.html.erb.** Fizemos a alteração apenas na *view* **index.html.erb**, que é exibida após a exclusão de uma mensagem, mas a view **show.html.erb** também exibe o alerta quando cadastramos ou editamos uma mensagem. Por isso, o alerta é exibido com o design novo quando excluímos uma mensagem, mas é apresentado com o design antigo quando cadastramos ou apagamos uma mensagem.

Aplicando DRY

De acordo com as aplicações anteriores, nenhum trecho de código pode se repetir em toda a aplicação. Se um trecho de código for necessário em mais de um local, ele precisa ser chamado nesses locais. Para que o trecho de código do alerta possa ser chamado em várias *views*, precisamos colocar esse trecho de código num *partial*.

Para isso, crie um arquivo chamado **_notice.html.erb** na pasta **app/views/mensagens**. Não se esqueça de colocar o *underline* (_) antes do nome do arquivo, pois todos os *partials* começam com *underline*. Abra esse novo arquivo e digite o seguinte código.

```
  <% if notice.present? %>
  <div id="notice" class="alert alert-primary" role="alert"><%= notice %></div>
<% end %>
```

Esse é o trecho de código do alerta, o qual é necessário nas *views* **index.html.erb** e **show.html.erb**. O próximo passo é substituir, nas *views* **index.html.erb** e **show.html.erb**, o código ilustrado acima pelo código abaixo.

```
1  <% if notice.present? %>
2    <%= render partial: 'notice' %>
3  <% end %>
```

O método **render partial: 'notice'** é responsável por incluir o código presente no **partial _notice.html.erb** na *view* em questão.

Atividade

Siga as orientações da aula para aplicar o conceito de DRY, isolando o trecho de código que apresenta o alerta num *partial*.

Resumindo

Nesta aula, pudemos ver um exemplo real de um problema provocado pela repetição de código em nossa aplicação. Para resolver esse problema, aplicamos o conceito de DRY (Don't Repeat Yourself) para isolar o código que estava repetido. Até mais, estudante!

Referências

CAELUM. **Desenvolvimento Ágil para Web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails**: Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBY. **Ruby programming language**. Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. **Ruby on rails**: a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern. Disponível em: http://rubyonrails.org/. Acesso em: 12 dez. 2017.

RUBYONRAILS.ORG. **Ruby on Rails Guides**. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. **Rails Gilrs Tutorial**. Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

Aula 10 - Validação de Dados

Objetivos

Ao final desta aula, você deverá ser capaz de:

Compreender o problema gerado pela ausência de validação dos dados;

Aprender quais são as validações disponíveis em Rails;

Aplicar validações no modelo Mensagem.

Introdução

Os nossos formulários de cadastro e edição de mensagens estão funcionando, mas você já tentou cadastrar uma mensagem sem preencher um ou mais campos do formulário? Se fez, você deve ter observado que, mesmo sem preencher os campos, o formulário é submetido e uma nova mensagem é cadastrada com sucesso.

Para corrigir isso, precisamos começar a utilizar validação de dados, que consiste em uma verificação dos dados preenchidos pelos usuários para determinar se eles atendem a determinadas especificações. Por exemplo: no nosso caso, podemos determinar que o preenchimento dos campos título, corpo, autor e *e-mail* sejam obrigatórios.

Validações em Rails

No Rails, as validações de dados são feitas nos modelos, usando *validation helpers*, métodos que especificam quais campos devem ser validados e como devem ser validados. Existem vários tipos de validação. Aqui, vamos trabalhar com validações de aceitação, confirmação, tamanho de dados, campos numéricos, preenchimento e unicidade.

Aceitação

Alguns formulários requerem que o usuário marque um determinado campo para que aquele seja submetido. Um exemplo disso é quando precisamos marcar um campo indicando que aceitamos os termos de serviço, de uso ou de privacidade. Para validar se um determinado campo foi marcado pelo usuário, usamos **validates** com o parâmetro **acceptance: true**.

```
validates :termos_servico, acceptance: true
```

No exemplo acima, estamos validando se o campo **termos_servico** foi marcado.

Confirmação

Também é comum que, por questões de segurança, alguns campos precisem ser preenchidos mais de uma vez. Exemplos disso são campos de e-mail e senha. A confirmação de um dado, ou seja, a necessidade de preenchê-lo duas vezes, visa diminuir as chances de erro de digitação.

Para validar se um determinado campo foi confirmado, ou seja, se ele foi preenchido e depois preenchido novamente para confirmação, usamos **validates** com o parâmetro **confirmation: true**.

```
validates :email, confirmation: true
```

No exemplo acima, estamos validando se o campo **email** foi preenchido duas vezes.

Tamanho

Outra situação é a dos campos que exigem um tamanho mínimo ou máximo de caracteres. Essa exigência também é normalmente feita para campos de senha. Para validar se um campo tem um tamanho mínimo ou máximo de caracteres, usamos validates com o parâmetro **length**. Nesse parâmetro, podemos especificar um tamanho mínimo usando **minimum**, um tamanho máximo usando **maximum**, um intervalo usando **in**, ou um tamanho específico usando **is**.

```
validates :nome, length: { minimum: 2 }
validates :descricao, length: { maximum: 500}
```

```
validates :registro, length: { is: 6 }
```

```
validates :nome, length: { minimum: 2 }
```

O primeiro exemplo verifica se o campo **nome** tem, no mínimo, 2 caracteres. O segundo exemplo verifica se o campo **descricao** tem, no máximo, 500 caracteres. O terceiro exemplo verifica se o campo **senha** tem entre 6 e 20 caracteres. O último exemplo verifica se o campo **registro** tem exatamente 6 caracteres de tamanho.

Campos numéricos

Para validar um campo numérico, ou seja, se um usuário digitou um número inteiro ou real, usamos **validates** com o parâmetro **numericality: true**. Para validar se foi digitado apenas números inteiros, usamos, ao invés de **true**, **only_integer: true**.

```
validates :pontos, numericality: true
validates :jogadores, numericality: { only_integer: true}
```

O primeiro exemplo verifica se o valor digitado no campo **pontos** é um número. O segundo exemplo verifica se o valor digitado no campo **jogadores** é um número inteiro.

Preenchimento

Talvez a validação mais comum seja a verificação do preenchimento de determinados campos obrigatórios. Para validar se um campo foi preenchido, usamos validates com o parâmetro presence: true.

```
validates :nome, :email, :senha, presence: true
```

Observe que, no exemplo acima, estamos validando três campos ao mesmo tempo: **nome**, **email** e **senha**. Isso também pode ser feito com os outros validadores.

Unicidade

Também podemos nos deparar com campos que exigem valores únicos, ou seja, que o valor preenchido por um usuário não seja preenchido por outro usuário. Essa restrição normalmente é aplicada para campos de *e-mail* ou

nome de usuário. Para validar se o valor digitado pelo usuário é único, usamos **validates** com o parâmetro **uniqueness: true**.

```
validates :email, uniqueness: true
```

O exemplo acima verifica se os valores digitados no campo **email** são únicos (exclusivos para cada usuário).

Validando o Modelo Mensagem

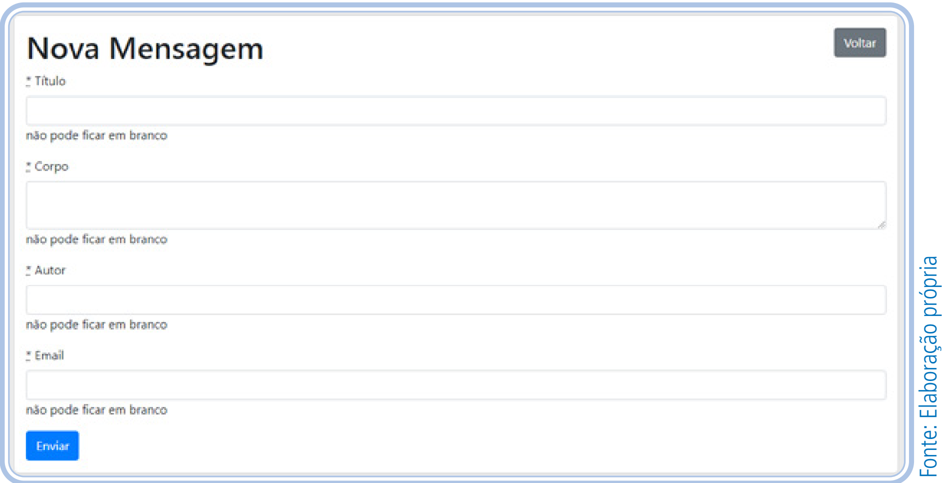
No projeto **Mural de Mensagens**, vamos validar o preenchimento dos campos título, corpo, autor e e-mail. Para isso, abra o modelo Mensagem em **app/models/mensagem.rb** e adicione os seguintes validadores:

```
class Mensagem < ActiveRecord::Base
  validates :titulo, :corpo, :autor, :email, presence: true
end
```

Feito isso, salve o script, execute o projeto e acesse o formulário de cadastro de mensagens. Tente submeter o formulário com um ou mais campos não preenchidos. Você deve visualizar uma mensagem indicando que o preenchimento dos campos é obrigatório, conforme ilustrado na Figura 1.

Atividade

Siga as orientações da aula para validar o preenchimento dos campos **titulo**, **corpo**, **autor** e **email** no modelo Mensagem.



Fonte: Elaboração própria

Figura 1: Formulário com validação

Na Figura 1, os campos de preenchimento obrigatório agora possuem um asterisco (*) antes do rótulo. Caso você não goste dessa aparência, pode remover esses asteriscos, usando o parâmetro **required: false** nos campos do formulário presentes em **app/views/mensagens/_form.html.erb**.

```
<%= simple_form_for @mensagem do |f| %>
  <%= f.input :titulo, label: 'Título', required: false %>
  <%= f.input :corpo, as: :text, required: false %>
  <%= f.input :autor, required: false %>
  <%= f.input :email, required: false %>
  <%= f.button :submit, 'Enviar', class: 'btn-primary' %>
<% end %>
```

Feito isso, concluímos o desenvolvimento do nosso primeiro projeto em Rails, o **Mural de Mensagens**. Parabéns!

Resumindo

Nesta aula, aprendemos sobre a importância da validação de dados em aplicações, bem como os tipos de validações disponíveis em Rails. Por fim, aplicamos um grupo de validações no modelo Mensagem para verificar o preenchimento de campos obrigatórios.

Referências

CAELUM. **Desenvolvimento Ágil para Web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails**: Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBY. **Ruby programming language**. Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. **Ruby on rails**: a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern. Disponível em: http://rubyonrails.org/. Acesso em: 12 dez. 2017.

RUBYONRAILS.ORG. **Ruby on Rails Guides**. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. **Rails Gilrs Tutorial**. Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

Aula 11 - Associações em Rails

Objetivos

Ao final desta aula, você deverá ser capaz de:

Apresentar o novo projeto Agenda de Contatos;

Introduzir o conceito de associações entre modelos;

Apresentar, através de exemplos, os seis tipos de associações disponíveis no framework.

Introdução

Nas últimas aulas, desenvolvemos um Mural de Mensagens para ilustrar a construção de uma aplicação simples utilizando scaffold, que é o mecanismo de geração de código o qual constrói o cadastro completo de um elemento do sistema, que, no nosso caso, foram as mensagens.

Contudo, a geração de código através do scaffold não nos ajuda a compreender bem como a aplicação foi construída, dificultando, assim, um pouco o aprendizado. Agora, daremos início ao desenvolvimento de uma Agenda de Contatos, sem usar o scaffold. Vamos codificar cada view, cada model e cada controller necessário. Para isso, vamos desenvolver um novo projeto Rails chamado Agenda de Contatos.

A Agenda de Contatos será uma aplicação a qual permitirá aos seus usuários cadastrarem pessoas, que será um dos modelos do nosso sistema com os campos nome (**string**) e data de nascimento (**date**). Associado a esse modelo, teremos outro chamado Contato, que, naturalmente, será responsável por armazenar formas de contato de cada pessoa. Esse contato poderá ser um número de telefone fixo, celular ou endereço de e-mail.

Dado esse contexto, fica fácil perceber um relacionamento 1 para N entre os modelos Pessoa e Contato, já que uma pessoa pode possuir mais de um

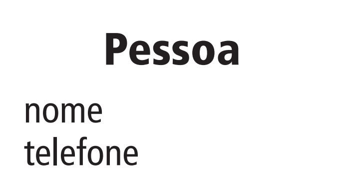
contato, mas cada contato pertence a uma única pessoa. O objetivo final da nossa aplicação será permitir que seus usuários possam cadastrar, listar, editar, detalhar e excluir pessoas e seus contatos.

Atividade

Para iniciar o desenvolvimento da Agenda de Contatos, será necessário criar novo projeto Rails chamado **agenda**. Para isso, siga as orientações para criar um novo projeto apresentadas na Unidade III/Aula 03. Além disso, você precisará realizar as configurações de idioma no seu novo projeto, seguindo as orientações da seção **Configurando nosso idioma**, da Aula 04.

Associações

Uma associação é um relacionamento entre dois modelos. As associações são importantes, pois nos permitem realizar operações entre os modelos, de forma simples e fácil.

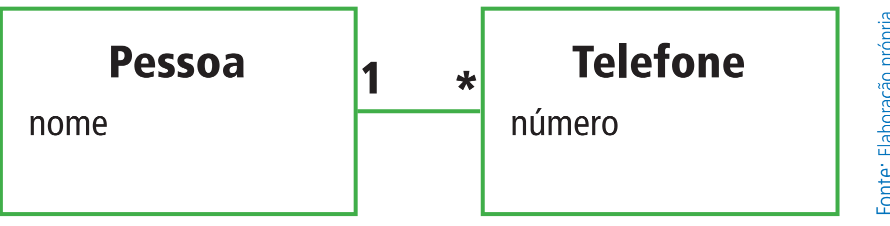


Fonte: Elaboração própria

Figura 1 - Modelo Pessoa com os atributos nome e telefone.

Por exemplo, considere o modelo Pessoa (vide Figura 1) com os atributos 'nome' e 'telefone'. Contudo, observe que uma pessoa pode ter mais de um número de telefone atualmente e, usando apenas o modelo Pessoa, nossa aplicação não permite que cadastremos mais de um número de telefone por pessoa.

A solução para esse problema é criar um modelo chamado Telefone (vide Figura 2), o qual irá armazenar os números de telefone, e usar uma associação 1 para N para indicar a qual pessoa cada número de telefone pertence.



Fonte: Elaboração própria

Figura 2 – Associação entre os modelos Pessoa e Telefone.

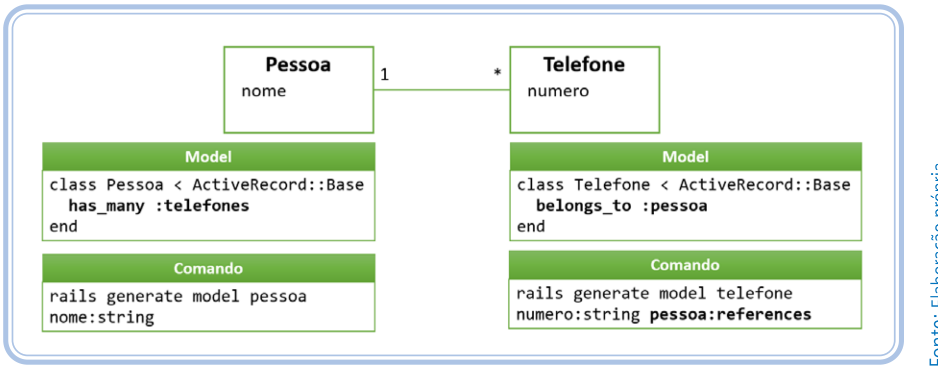
Dessa forma, serão geradas duas tabelas no nosso banco de dados: 'pessoas', que irá armazenar informações relativas às pessoas, e 'telefones', que irá armazenar os números de telefone cada pessoa. Atente para o aprendizado na disciplina de Banco de Dados: para que essa associação funcione corretamente, será preciso criar uma chave estrangeira na tabela 'telefones' (gerada a partir do modelo Telefone), a qual referencia a chave primária da tabela 'pessoas' (gerada a partir do modelo Pessoa).

Existem seis tipos de associações em Rails: belongs_to, has_one, has_many, has_many :through, has_one :through e has_and_belongs_to_many.

Associações has_many e belongs_to

Vamos utilizar o exemplo dos modelos Pessoa e Telefone para explicar como criar uma associação 1 para N entre esses dois modelos. Esse é, de longe, o tipo de associação mais comum nos bancos de dados relacionais atualmente.

Para criar uma associação 1 para N entre os modelos Pessoa e Telefone, usaremos o **has_many :telefones** no *model* Pessoa e o **belongs_to :pessoa** no *model* Telefone, conforme ilustra a Figura 3. No comando **rails generate model telefone**, estamos usando pessoa:references. Isso irá gerar uma chave estrangeira chamada **pessoa_id** na tabela **telefones**, e essa chave será usada para indicar a qual pessoa pertence cada número de telefone.



Fonte: Elaboração própria

Figura 3 - Implementando associação has_many e belongs_to.

Associação has_one

Suponha que estamos desenvolvendo um sistema de controle de estoque e temos dois modelos chamados Fornecedor e Conta, como ilustrado na Figura 4. Cada fornecedor possui apenas uma conta, e cada conta pertence a um único fornecedor.

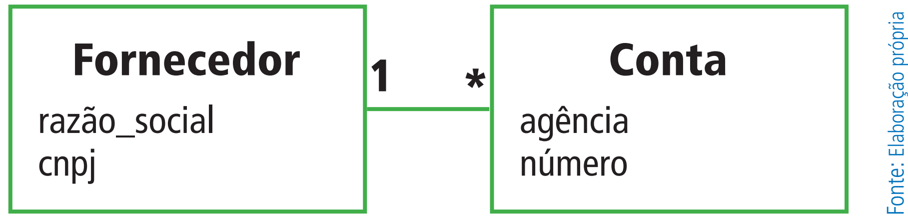


Figura 4 - Modelos Fornecedor e Conta.

Para criar uma associação 1 para 1 entre Fornecedor e Conta, usaremos **has_one :conta** no modelo Fornecedor e **belongs_to :fornecedor** no modelo Conta, cono visualizamos na Figura 5. Novamente, estamos usando **fornecedor:references** no comando para criar a chave estrangeira **fornecedor_id** na tabela contas.

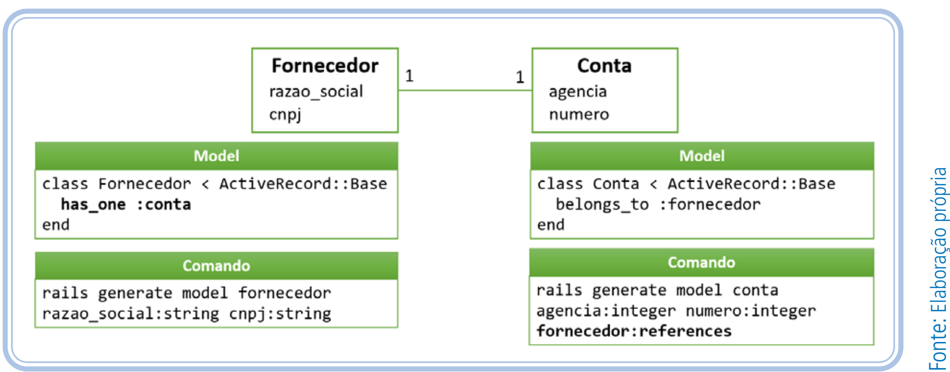


Figura 5 - Implementação da associação has_one.

Associação has_many :through

Considere que estamos desenvolvendo um sistema de atendimento clínico e temos os seguintes modelos: Paciente, Consulta e Médico. Um paciente pode ter várias consultas, um médico pode ter feito várias consultas, e cada consulta sempre tem um paciente e um médico. Portanto, temos uma associação 1 para N entre paciente e consulta, uma associação 1 para N entre consulta e médico, e uma associação N para N entre paciente e médico (vide Figura 6).

Figura 6 - Modelos Paciente, Consulta e Médico.

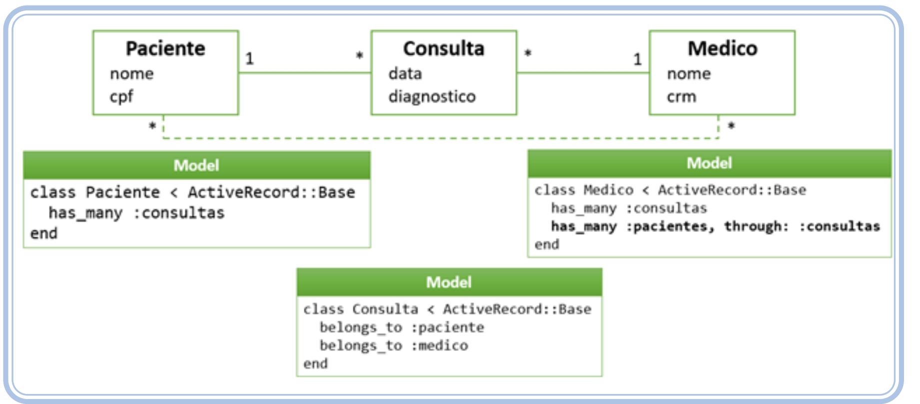
Em determinado ponto no desenvolvimento do sistema, pode ser necessário identificar quem são os pacientes de um determinado médico. Isso pode ser obtido iterando entre todas as consultas do médico e, então, buscando o paciente em cada consulta, conforme ilustrado abaixo.

```
pacientes = Array.new
medico.consultas.each do |consulta|
  pacientes << consulta.paciente
end
```

No entanto, existe uma solução mais simples, a associação **has_many :through**, que simula uma associação 1 para N entre os modelos Paciente e Médico, através do modelo Consulta. Usando essa associação entre os modelos, o código acima poderia ser simplificado a uma única linha de código a seguir.

```
medico.pacientes
```

Para usarmos o **has_many :through**, primeiro precisamos criar uma associação 1 para N entre os modelos Médico e Consulta. Para isso, usamos **has_many :consultas** no modelo Médico. Em seguida, ainda no modelo Médico, usamos o **has_many :pacientes**, **through: :consultas** para criar uma associação entre os modelos Médico e Paciente através do modelo Consulta, conforme ilustra a Figura 7.



Fonte: Elaboração própria

Figura 7 - Implementação da associação has_many :through.

has_one :through

Da mesma forma que a associação **has_many :through**, a associação **has_one :through** também permite simular uma associação entre dois modelos através de outro modelo. Na Figura 8, temos três modelos: Gerente, Fornecedor e Conta, todos associados entre si através de uma relação

1 para 1. Nesse caso, podemos utilizar o **has_one :through** no modelo Conta para identificar quem é o gerente através do modelo Fornecedor.

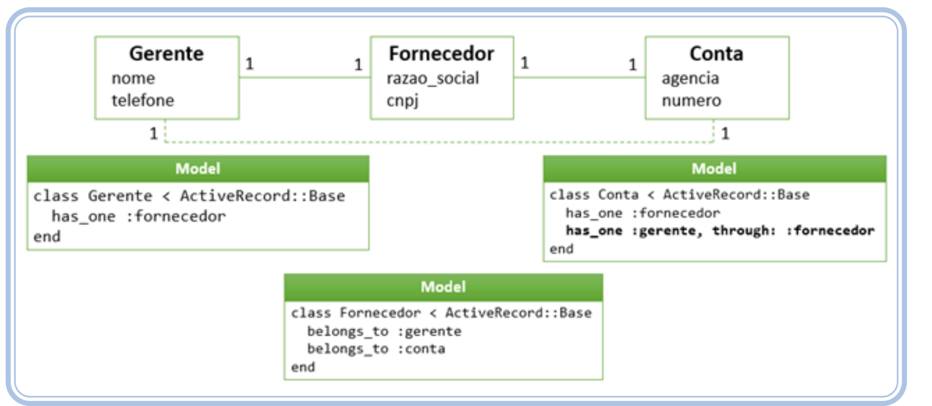


Figura 8 - Implementação da associação has_one :through.

A associação **has_one :gerente, through: :fornecedor** no modelo Conta permitiria identificar o gerente de uma conta específica da seguinte forma.

```
conta.gerente
```

Associação has_and_belongs_to_many

Considere que, novamente, estamos desenvolvendo o sistema de atendimento clínico e, num determinado ponto do desenvolvimento, precisamos relacionar quais especialidades médicas podem solicitar determinados exames médicos. Nesse aspecto, se uma especialidade médica pode solicitar vários exames e um mesmo exame pode ser solicitado por mais de uma especialidade, temos uma associação N para N entre os modelos Especialidade e Exame, como mostra a Figura 9.

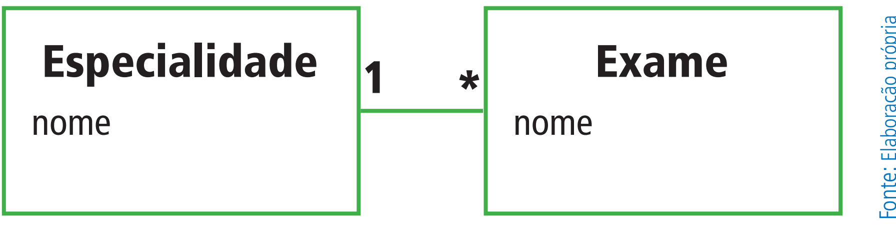
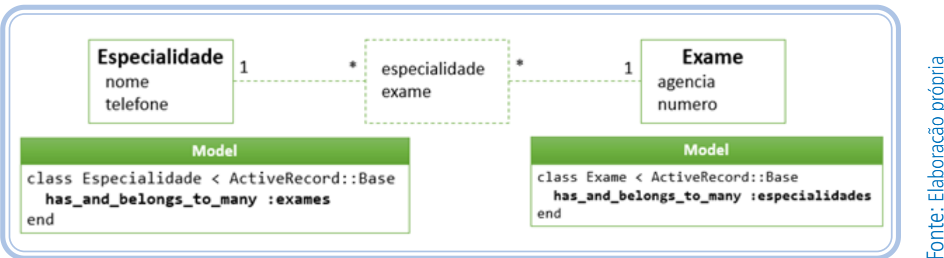


Figura 9 - Modelos Especialidade e Exame.

Como você deve ter estudado na disciplina Banco de Dados, para criarmos uma associação N para N entre duas entidades, precisamos criar uma entidade intermediária. Entretanto, a associação **has_and_belongs_to_many** permite criar uma associação N para N entre dois modelos sem a necessida-

de de criar um modelo intermediário, pois isso será gerenciado automaticamente pelo framework no banco de dados.



Fonte: Elaboração própria

Figura 10 - Associação has_and_belongs_to_many.

Na Figura 10, utilizamos a associação **has_and_belongs_to_many** nos modelos Especialidade e Exame. Isso irá criar uma associação N para N entre esses dois modelos, sem a necessidade de criar um terceiro modelo. Dessa forma, podemos consultar quais exames podem ser solicitados por uma especialidade, usando **especialidade.exames**, e também quais especialidades podem solicitar um determinado exame **exame.especialidades**.

Autoavaliação

1) Imagine um sistema de uma editora de livros com os modelos Livro e Autor. Considerando que um livro pode ter vários autores, e um autor pode ter escrito vários livros. Quais são as associações indicadas para esse caso?

a) Uma associação has_many e belongs_to seria suficiente para lidar com esse caso.

b) Duas associações has_one seria a opção mais adequada.

c) Uma associação has_many é a opção mais indicada para esse caso.

d) Duas associações has_many ou duas has_and_belongs_to_many seriam suficientes.

e) Nenhuma das alternativas anteriores.

2) Justifique sua resposta da questão anterior.

3) Qual é a diferença entre as associações has_many e has_and_belongs_to_many?

Gabarito: 1) d

Resumindo

Nesta nossa aula, apresentamos brevemente o novo projeto a ser desenvolvido, a Agenda de Contatos, bem como revisamos o conceito de associações entre modelos, as quais representam entidades no banco de dados. Além disso, aprendemos como realizar associações em modelos Rails utilizando seis diferentes tipos de associações.

Referências

CAELUM. Desenvolvimento Ágil para Web com Ruby on Rails 4. São Paulo: Caelum.

FUENTES, V. B. Ruby on Rails: Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBY. Ruby programming language. Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. Ruby on rails: a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern. Disponível em: http://rubyonrails.org/. Acesso em: 12 dez. 2017.

RUBYONRAILS.ORG. Ruby on Rails Guides. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. Rails Gilrs Tutorial. Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

Aula 12 - Desenvolvendo e associando os modelos

Objetivos

Ao final desta aula, você deverá ser capaz de:

Aprender como criar modelos em aplicações Rails;

Criar os modelos Pessoa e Contato;

Compreender os arquivos gerados pelo comando;

Associar os modelos Pessoa e Contato;

Aplicar as validações nos modelos criados.

Introdução

Vamos dar início ao desenvolvimento da nossa Agenda de Contatos criando os modelos Pessoa, responsável por representar uma pessoa na agenda, e Contato, responsável por representar uma forma de contato para cada pessoa. O modelo Pessoa deverá conter os campos nome e data_nascimento, enquanto o modelo Contato deverá conter o campo valor, responsável por armazenar o número de telefone, caso o contato seja um telefone, ou um endereço de *e-mail*, caso o contato seja um *e-mail*.

Além disso, precisaremos associar esses dois modelos, pois eles, claramente, precisam estar relacionados para indicar que cada contato pertence a uma pessoa, e uma pessoa possui vários contatos.

Criando modelos

Para gerar um modelo em Rails, devemos usar o comando **rails generate model** ilustrado na Figura 1.

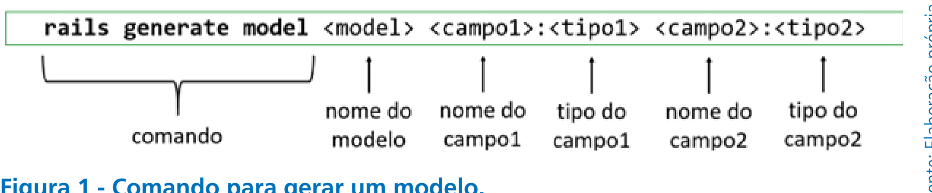


Figura 1 - Comando para gerar um modelo.

Devemos substituir o parâmetro <model> pelo nome do modelo que desejamos no singular. Para especificar os campos desejados, usamos o nome e o tipo do campo separados por dois pontos. Podemos adicionar quantos campos forem necessários. Os tipos de campo disponíveis no framework Rails estão apresentados na Tabela 1.

Tabela 1: Tipos de campos

Tipo	Descrição
binary	Binário
boolean	Booleano
date	Data
datetime	Data e hora
decimal	Número decimal
float	Número real
integer	Número inteiro
primary_key	Chave primária
references	Referência outro modelo
string	String (até 255 caracteres)
text	Texto
time	Registro de tempo em data, hora, minutos, segundos e fuso horário
timestamp	Registro de tempo em data, hora, minutos, segundos e milissegundos

Fonte: Elaboração própria

Criando o modelo Pessoa

Para gerar o modelo Pessoa com os campos nome, do tipo string, e data_nascimento, do tipo date, devemos usar o comando abaixo.

```
rails generate model pessoa nome:string data_nascimento:date
```

Atividade

Siga as orientações da aula para gerar o modelo Pessoa com os campos **nome**, do tipo string, e **data_nascimento**, do tipo date.

O comando **rails generate model** irá gerar três arquivos automaticamente: a *migration*, o teste unitário e a *fixture.* A *migration* é um script Ruby, que irá

criar a tabela no banco de dados, localizado na pasta **db/migrate**. Os nomes das *migrations* sempre seguem um padrão: uma sequência de números contendo a data atual (ano, mês e dia) e hora, a palavra **create** e o nome do model no plural. Por exemplo, ao executar o comando **rails generate model** para criar o model Pessoa, a migration abaixo será gerada automaticamente.

```
db/migrate/20160608171956_create_pessoas.rb
```

Essa migration terá o seguinte conteúdo.

```
1  class CreatePessoas < ActiveRecord::Migration
2    def change
3      create_table :pessoas do |t|
4        t.string :nome
5        t.date :data_nascimento <%= f.input :email %>
6
7        t.timestamps null: false
8      end
9    end
10 end
```

As *migrations* contém um método **change** (linha 2) ou dois métodos **up** e **down**, que são executados quando a *migration* é executada. No código acima, o método **change** irá criar a tabela **pessoas**, usando o método **create_table** (linha 3). Essa tabela contém as colunas **nome** (linha 4), do tipo **string**, e **data_nascimento** (linha 5), do tipo **date**.

Além dessas colunas, três outras serão adicionadas automaticamente ao seu modelo: a) o **id**, um campo inteiro, chave primária padrão da sua tabela, preenchido automaticamente com um valor sequencial; b) o **created_at**, campo do tipo *timestamp* (ou seja, que armazena a data e a hora), nesse caso, guarda o *timestamp* da criação do objeto; c) **updated_at**, campo do tipo *timestamp*, nesse caso, guarda a data e a hora da última atualização do objeto.

Outro arquivo gerado pelo comando **rails generate model** é o teste unitário, um *script* produzido para testar o *model* que você está criando. Referente ao *model* Pessoa, será gerado o script **test/models/pessoa_test.rb**, o qual será responsável por realizar os testes no modelo Pessoa. As *fixtures* são arquivos de texto que guardam os dados a serem usados nos testes.

Criando e associando o modelo Contato

Conforme explicado anteriormente, criaremos o modelo contato para representar uma forma de contato com uma pessoa. Nesse modelo, teremos o campo valor, que irá armazenar o número de telefone ou o endereço de e-mail. Para gerar esse modelo, vamos usar o comando

```
rails generate model contato valor:string pessoa:references
```

Observe que, no comando acima, estamos usando **pessoa:referentes**. Esse parâmetro cria, na tabela **contatos**, uma chave estrangeira (**pessoa_id**) referente à tabela **pessoas**, a qual irá nos permitir associar essas duas tabelas com um relacionamento 1 para N.

Atividade

Siga as orientações desta aula para gerar o modelo Contato com o campo **valor**, do tipo *string*, e uma referência para o modelo Pessoa.

Para associar os modelos Pessoa e Contato, abra o modelo Pessoa (**app/models/pessoa.rb**) e use has_many :contatos para indicar que uma pessoa pode ter vários contatos.

```
class Pessoa < ActiveRecord::Base
  has_many :contatos
end
```

Para concluir a associação, abra o modelo Contato (**app/models/contato.rb**) e use **belongs_to :pessoa** para indicar que cada contato pertence a uma pessoa. Caso essa linha já exista no modelo, nenhuma modificação é necessária.

```
class Contato < ActiveRecord::Base
  belongs_to :pessoa
end
```

Feito isso, a associação entre os modelos Contato e Pessoa estará concluída.

Atividade

Siga as orientações desta aula para criar uma associação 1 para N entre os modelos Pessoa e Contato. Lembre-se de que cada pessoa tem várias formas de contato, e cada contato pertence a uma única pessoa.

Validação de dados

Como estamos trabalhando nos modelos, vamos realizar as devidas validações de dados que estudamos na Unidade X/Aula 10. No modelo Pessoa, precisamos garantir que o usuário preencha pelo menos o campo **nome**. Desse modo, abra o modelo Pessoa em **app/models/pessoa.rb** e insira a linha 4 apresentada no código abaixo.

```
1  class Pessoa < ActiveRecord::Base
2    has_many :contatos
3
4    validates :nome, presence: true
5  end
```

No modelo Contato, é preciso que o usuário preencha o campo **valor** obrigatoriamente. Para isso, abra o modelo Contato em **app/models/contato.rb** e insira a linha 4 apresentada no código abaixo.

```
1  class Contato < ActiveRecord::Base
2    belongs_to :pessoa
3
4    validates :valor, presence: true
5  end
```

Atividade

Siga as orientações desta aula para validar os dados nos modelos Pessoa e Contato. No modelo Pessoa, valide o preenchimento do campo **nome**, e, no modelo Contato, valide o preenchimento do campo **valor**.

Resumindo

Nesta aula, estudamos como criar modelos por meio de comandos em aplicações Rails, através do desenvolvimento dos modelos Pessoa e Contato. Também associamos esses modelos e, por fim, aplicamos regras de validação de dados em ambos.

Referências

CAELUM. **Desenvolvimento Ágil para Web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails**: Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBY. **Ruby programming language**. Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. **Ruby on rails**: a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern. Disponível em: http://rubyonrails.org/. Acesso em: 12 dez. 2017.

RUBYONRAILS.ORG. **Ruby on Rails Guides**. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. **Rails Gilrs Tutorial**. Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

Aula 13 - Desenvolvendo o controlador pessoas

Objetivos

Ao final desta aula, você deverá ser capaz de:

Criar o controlador pessoas;

Gerar os métodos e *views* associados;

Desenvolver o método index e *view* para listagem de pessoas;

Melhorar a apresentação da *view* com Bootstrap.

Criando o controlador

Para criar um controlador em Rails, devemos usar o comando **rails generate controller**, conforme ilustrado na Figura 1.

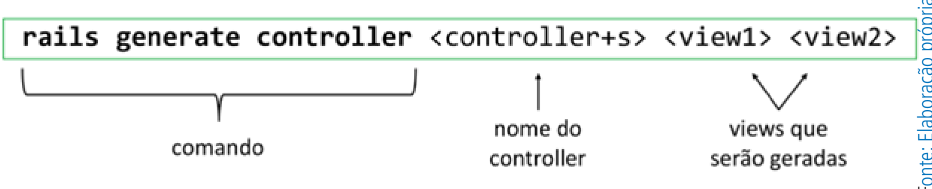


Figura 1: Comando para gerar controladores

Devemos substituir o parâmetro **<controller+s>** pelo nome do controlador (no plural) que desejamos gerar. Opcionalmente, podemos especificar nomes de métodos e *views* para serem gerados com o controlador. Basta substituir **<view1>**, **<view2>**, **<view3>**, e quantas outras views você desejar, pelos nomes dos métodos e das *views*.

Por exemplo, para criar um controlador chamado **pessoas_controller**, com os métodos/*views* **index**, **new** e **edit**, devemos usar o comando abaixo.

```
rails generate controller pessoas index new edit
```

No comando acima, especificamos que o nome do controlador será pessoas (não é necessário escrever **pessoas_controller**) e que desejamos que sejam gerados também os métodos/*views* **index**, **new** e **edit**.

Após a execução do comando acima, o controlador **pessoas_controller.rb** será gerado na pasta **app/controller**; serão geradas as views **index.html.erb**, **new.html.erb** e **edit.html.erb** na pasta **app/views/pessoas**; será gerado, automaticamente, um teste unitário para realização de testes automatizados no controlador.

Além disso, será gerada uma classe helper (**app/helpers/pessoas_helper.rb**), usada como classe auxiliar na qual podemos definir métodos para reduzir a complexidade (a quantidade de código) nas views. Também será gerado um script CoffeeScript[1] (**app/assets/javascripts/pessoas.coffee**), no qual você poderá usar JavaScript, e uma folha de estilos SASS[2] (**app/assets/stylesheets/pessoas.scss**), na qual você poderá usar CSS.

Atividade

Siga as orientações desta aula para criar o controlador **pessoas** e os métodos/*views* **index**, **new** e **edit**.

Ajustando as rotas

Além do controlador e das views, o comando **rails generate controller** cria uma rota para cada método/*view* especificada no comando. Portanto, o comando **rails generate controller pessoas index new edit** deverá gerar três rotas no *script* **config/routes.rb**.

A rota **get '/pessoas'** especifica que, quando o endereço **/pessoas** for acessado através do método HTTP GET, quem irá atender essa requisição é o método **index** do controlador **pessoas_controller**.

A rota **get 'pessoas/new'** especifica que, quando o endereço **/pessoas/new** for acessado através do método HTTP GET, quem irá atender essa requisição é o método **new** do controlador **pessoas_controller**.

1 http://coffeescript.org/
2 https://sass-lang.com/

A rota get **'pessoas/edit'** especifica que, quando o endereço **/pessoas/edit** for acessado através do método HTTP GET, quem irá atender essa requisição é o método **edit** do controlador **pessoas_controller**.

No arquivo de rotas (**config/routes.rb**), vamos redefinir as rotas para o controlador **pessoas_controller**. Para isso, substitua as linhas abaixo:

```
get 'pessoas'
get 'pessoas/new'
get 'pessoas/edit'
```

Substitua por:

```
resources :pessoas
root 'pessoas#index'
```

Usando o **resources :pessoas**, definimos automaticamente um conjunto de sete rotas para o controlador **pessoas_controller**. Para mais detalhes sobre esse conjunto de rotas, releia a seção **Rotas** da Unidade V/Aula 05 **- O padrão MVC**. Usando o **root 'pessoas#index'**, estamos configurando que, quando o usuário acessar o endereço raiz da nossa aplicação (**/**), quem irá atender essa requisição é o método **index** do controlador **pessoas**.

Atividade

Siga as orientações desta aula para ajustar as rotas da aplicação, definindo o conjunto de rotas **resources :pessoas** e a rota raiz (**root 'pessoas#index'**).

Desenvolvendo a *view* de listagem

Agora, implementaremos a listagem de pessoas. Para isso, abra o controlador app/controllers/pessoas_controller e implemente o método index, como ilustrado abaixo.

```
def index
 @pessoas = Pessoa.all
end
```

O papel do **index** é fornecer o conjunto de objetos do model Pessoa cadastrados no banco de dados. O método **all** da classe Pessoa retorna um

array com todos os objetos da classe persistidos no banco de dados, então devemos usar esse método para guardar o retorno dele numa variável chamada **@pessoas**.

Conforme estudamos na disciplina Programação Estruturada e Orientada a Objetos, variáveis que começam com **@** são variáveis de instância. No Rails, todas as variáveis de instância ficam disponíveis na *view*, e é exatamente isso o que precisamos: chamamos o método **all** da classe Pessoa para ter um *array* com os objetos Pessoa persistidos no banco de dados, em seguida guardamos esse *array* na variável de instância **@pessoas**, que estará disponível na *view* **index** para podermos apresentar os dados numa tabela em HTML. Vale salientar que, variáveis locais, ou seja, sem o **@**, não ficam disponíveis na *view*, apenas dentro do método no qual elas foram criadas.

De acordo com nossas configurações de rotas, quando um usuário acessar o endereço **/pessoa**s, o método **index** será executado, e em seguida, a *view* **app/views/pessoas/index.html.erb** será carregada. Portanto, agora precisamos alterar o código dessa *view*. Abra a *view* em **app/views/pessoas/index.html.erb**, apague o código existente nessa *view* e substitua pelo código apresentado abaixo.

```
1  <h1>Pessoas</h1>
2
3  <% if @pessoas.empty? %>
4   <p>Nenhuma pessoa encontrada.</p>
5  <% else %>
6   <table border="1">
7    <thead>
8     <th>Nome</th>
9     <th>Data de nascimento</th>
10   </thead>
11   <tbody>
12    <% @pessoas.each do |pessoa| %>
13     <tr>
14      <td><%= pessoa.nome %></td>
15      <td><%= pessoa.data_nascimento %></td>
16     </tr>
17    <% end %>
18   </tbody>
19  </table>
20 <% end %>
```

Na linha 1, há um título de nível 1 (*tag* **<h1>**) com o corpo Pessoas para indicar que essa página apresenta uma listagem de pessoas. Na linha 3, observe como podemos utilizar código Ruby dentro da nossa *view*. Para isso, basta o código Ruby estar dentro dos delimitadores **<%** e **%>**. Dentro dos delimitadores, estamos usando a variável de instância **@pessoas** definida no método index do controlador **pessoas_controller**. Perceba que essa variável não foi criada aqui na *view*, mas sim no controlador.

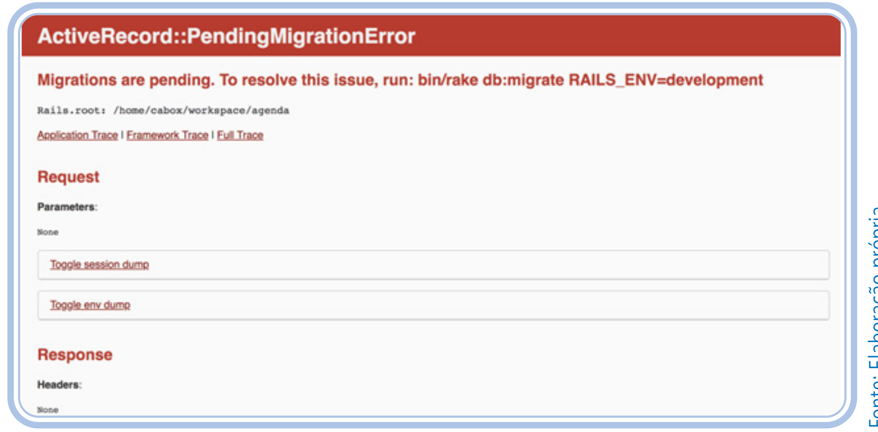
Ainda na linha 3, usamos um if para verificar, com o método **empty?**, se o *array* da variável **@pessoas** está vazio. Esse método retorna **true** quando o *array* está vazio, e **false** caso contrário. Portanto, caso o *array* da variável @pessoas esteja vazio, será apresentado o parágrafo **<p>Nenhuma pessoa encontrada</p>** (linha 4). Caso o *array* contenha objetos, o **else** (linha 5) será executado, apresentando a tabela entre as linhas 6 e 19. A tabela possui duas colunas com dois cabeçalhos: nome (linha 8) e data de nascimento (linha 9).

Em seguida, novamente usamos código Ruby na nossa view, mas, dessa vez, usamos o método **each** na variável **@pessoas** (linha 12) para iterar entre os objetos guardados no array **@pessoas**, exatamente como fizemos na disciplina Programação Estruturada e Orientada a Objetos.

Esse laço irá executar uma vez para cada objeto armazenado no array guardado em **@pessoas**, e, a cada execução, um objeto será guardado na variável **pessoa**. Observe que, dentro do laço, criamos uma linha da tabela usando a tag **<tr>** (linha 13), e para cada linha definimos duas células usando a tag **<td>** (linhas 14 e 15).

Perceba que, novamente, estamos usando os delimitadores Ruby, contudo, dessa vez, há uma diferença: a presença do sinal de igual logo após **<%**. Os delimitadores **<%= e %>** também executam código Ruby nas views, no entanto, o retorno da execução do código é apresentado na tela. Portanto, o código **<%= pessoa.nome %>** (linha 14) irá apresentar o valor do atributo **nome** do objeto da classe Pessoa armazenado na variável **pessoa**.

Feito isso, execute o seu projeto e acesse a sua aplicação. Caso tenha dúvidas sobre como fazer isso, consulte a seção **Executando seu projeto** da Unidade III/Aula 03 – Criando seu primeiro projeto. Ao acessar sua aplicação, você deverá se deparar com o erro ilustrado na Figura 2.



Fonte: Elaboração própria

Figura 2: Erro de migrações pendentes

O erro ilustrado na Figura 2 é o **PendingMigrationError**, o qual significa que existem migrations no seu projeto que ainda não foram executadas. E de fato há: na aula passada criamos os modelos Pessoa e Contato, usando o comando **rails generate model**. Ao criar os modelos usando esse comando, duas migrations foram criadas para criar as tabelas no banco de dados (uma para cada modelo).

Desse modo, para o nosso projeto ser executado com sucesso, precisamos executar as *migrations* antes. Para isso, interrompa a execução do servidor, usando CTRL + C no terminal de comandos do Codeanywhere, e, em seguida, execute o comando **rake db:migrate**.

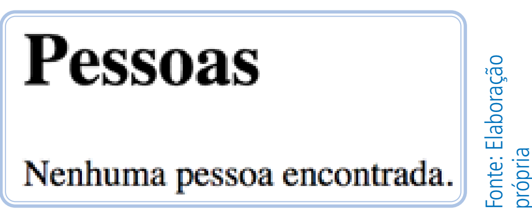
Depois, com as *migrations* executadas, você pode voltar a executar seu projeto para acessar sua aplicação.

Atividade

Siga as orientações desta aula para desenvolver a view de listagem de pessoas (**app/views/pessoas/index.html.erb**), e, em seguida, execute as migrations criadas na aula passada.

Instalando e aplicando estilos do Bootstrap

Ao acessar sua aplicação, você deverá ver algo similar à Figura 3.



Fonte: Elaboração própria

Figura 3: Erro de migrações pendentes

Como percebemos, nossa aplicação está funcionando, mas ainda não temos nenhuma pessoa cadastrada e muito menos um formulário que nos permita cadastrar uma. Para contornar esse problema, vamos cadastrar uma pessoa por meio do Rails Console. Para isso, interrompa a execução do servidor novamente no terminal de comandos, e, em seguida, use o comando **rails c** para entrar no Rails Console. Uma vez no Rails Console, execute o comando abaixo.

```
Pessoa.create(nome: 'Pedro Silva', data_nascimento: '14-10-2000')
```

O comando acima persiste uma nova pessoa no banco de dados com o nome Pedro Silva, e a data de nascimento 14/10/2000. Feito isso, saia do Rails Console com o comando **exit**. Execute o servidor mais uma vez e acesse sua aplicação novamente. Dessa vez, você deverá visualizar algo similar à Figura 4.

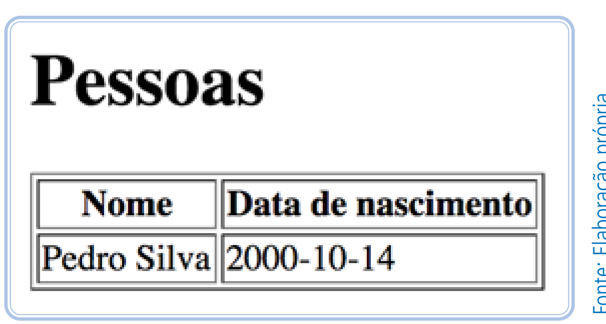


Figura 4: Listagem de pessoas.

A nossa página funciona, mas ainda está muito simples. Vamos melhorar o design novamente, utilizando o Bootstrap: entre no endereço http://getbootstrap.com/getting-started/ e copie o código de importação do script e das folhas de estilo. Em seguida, abra a *view* **app/views/layouts/application.html.erb** e cole o código copiado na página do Bootstrap. Caso tenha dúvidas sobre como fazer isso, consulte a seção **Instalando o Bootstrap** da Unidade VII/Aula 07 – Melhorando a apresentação (Parte I).

Lembrando que o **application.html.erb** é uma *view* compartilhada que será carregada quando qualquer *view* da nossa aplicação for carregada. Todas as *views* do nosso projeto serão carregadas no **yield**, portanto, qualquer código que coloquemos nessa *view* irá valer para todas as outras *views*.

Em seguida, vamos colocar o **yield** dentro de uma **div** com a classe **container**, conforme ilustrado a seguir.

```
<div class="container">
  <%= yield %>
</div>
```

A classe **container** é definida pelo Bootstrap e é apropriada para que o conteúdo das nossas *views* sejam apresentadas dentro dessa **div**. Vamos voltar para a view **app/views/pessoas/index.html.erb** e utilizar outras classes CSS, definidas pelo Bootstrap, para estilizar a nossa tabela. Para isso, substitua o código na linha 6.

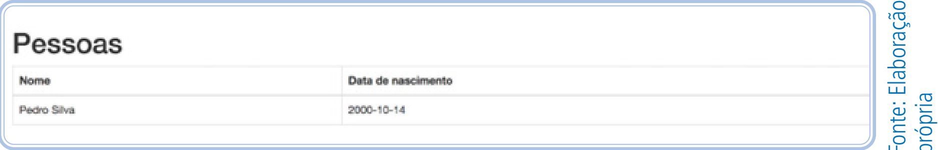
```
<table border="1">
```

pelo seguinte código:

```
<table class="table table-bordered table-hover">
```

A classe **table** estiliza a nossa tabela para deixá-la mais apresentável. A classe **table-bordered** inclui as bordas da tabela, e classe **table-hover** muda a cor da linha quando passamos o mouse sobre ela.

Feito isso, volte a acessar a sua aplicação (Figura 5).



Fonte: Elaboração própria

Figura 5: Listagem de pessoas com Bootstrap.

Atividade

Siga as orientações desta aula para melhorar a apresentação da *view* de listagem de pessoas (**app/views/pessoas/index.html.erb**), instalando e aplicando os estilos do Bootstrap.

Formatando datas

A data apresentada na tabela de listagem de pessoas está formatada como ano/mês/dia, que não é o formato de data o qual estamos acostumados no Brasil. Para resolver esse problema, vamos criar um método auxiliar (*helper method*) para formatar a data em dia/mês/ano.

Para isso, abra o script **app/helpers/application_helper.rb**. Nele, devemos incluir métodos auxiliares que ajudam a reduzir a quantidade de código nas *views*. Os métodos definidos nesse script em particular estarão disponíveis em qualquer *view* da aplicação. Por outro lado, os métodos definidos em **app/helpers/pessoas_helper.rb** estarão disponíveis apenas nas *views* associadas ao controlador pessoas.

No script **application_helper.rb**, insira o método **data** ilustrado abaixo.

```
1  module ApplicationHelper
2   def data(data)
3    if data.present?
4     data.strftime('%d/%m/%Y')
5    else
6     '-'
7    end
8   end
9  end
```

O método data, ilustrado acima, recebe uma data como parâmetro e, usando o método **strftime**, formata a data fornecida para o formato **'%d/%m/%Y'**, que é formato dia/mês/ano. Mais detalhes sobre a formatação de datas com esse método estão disponíveis em: https://apidock.com/ruby/DateTime/strftime.

Para finalizar, basta usarmos o método **data** na *view* **app/views/pessoas/index.html.erb**. Abra a *view* e substitua a linha 15 pela linha abaixo.

```
<td><%= data(pessoa.data_nascimento) %></td>
```

Feito isso, acesse a sua aplicação novamente e você deverá visualizar algo similar à Figura 6.

Pessoas

Nome	Data de nascimento
Pedro Silva	14/10/2000

Fonte: Elaboração própria

Figura 6: Listagem de pessoas com a data formatada.

Atividade

Siga as orientações da aula para formatar a data apresentada na listagem de pessoas para o formato dia/mês/ano.

Resumindo

Nessa aula, aprendemos a criar o controlador **pessoas** e os métodos/*views* associados **index**, **new** e **edit**. Além disso, desenvolvemos a *view* de listagem de pessoas (**app/views/pessoas/index.html.erb**) e, por fim, instalamos e aplicamos os estilos do Bootstrap para tornar a view mais visualmente agradável.

Referências

BOOTSTRAP. **Bootstrap**: the most popular html, css and js library in the world. Disponível em: http://getbootstrap.com/. Acesso em: 09 fev. 2018.

CAELUM. **Desenvolvimento Ágil para Web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails**: Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBY. **Ruby programming language**. Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. **Ruby on rails**: a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern. Disponível em: http://rubyonrails.org/. Acesso em: 12 dez. 2017.

RUBYONRAILS.ORG. **Ruby on Rails Guides**. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. **Rails Gilrs Tutorial**. Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

Aula 14 - Desenvolvendo o formulário de cadastro de pessoas

Objetivos

Ao final desta aula, você deverá ser capaz de:

Alterar a página de listagem de pessoas;

Instalar e configurar a gem Simple Form ao projeto;

Construir o formulário de cadastro de pessoas;

Implementar os métodos no controlador.;

Ajustar helper methods e estilos CSS.

Introdução

Na aula passada, desenvolvemos o controlador **app/controllers/pessoas_controller.rb**, responsável por gerenciar requisições relativas às pessoas da nossa Agenda de Contatos. Também construímos a view de listagem de pessoas (**app/views/pessoas/index.html.erb**). Agora, daremos início ao desenvolvimento alternado dessa view, para incluir um *link* o qual redirecione o usuário para o formulário de cadastro de pessoas.

Abra a *view* **app/views/pessoas/index.html.erb** e use o helper method **link_to** para adicionar um *link* para nossa *view*, conforme ilustrado abaixo. Adicione apenas o código em negrito. O código sem negrito já se encontra na sua *view*.

```
<h1>Pessoas</h1>
<%= link_to 'Cadastrar', new_pessoa_path, class: 'btn btn-default' %>
<% if @pessoas.empty? %>
 <p>Nenhuma pessoa encontrada.</p>
```

Um helper method é um método auxiliar o qual nos permite fazer algo de forma simples e fácil: nesse caso, criar um *link*. Poderíamos utilizar a *tag* **<a>** do HTML, mas a ideia aqui é demonstrar os recursos do framework Ruby on Rails.

O helper method **link_to** está recebendo três parâmetros: o primeiro é o texto que será exibido (**'Cadastrar'**); o segundo é o método **new_pessoa_path** que retorna o *path*, ou seja, o caminho para o formulário de cadastro de pessoas; e o terceiro é o **class**, o qual será usado no *link*. Você pode estar se perguntando de onde veio o método **new_pessoa_path**. Ele foi gerado automaticamente quando definimos o **resources :pessoas** no arquivo de rotas (**config/routes.rb**). Para mais detalhes sobre o arquivo de rotas, consulte a seção **Rotas** da Unidade V/Aula 05 – O Padrão MVC.

Execute o servidor (**rails s**), caso ele ainda não esteja em execução. Recarregue a página e observe que o nosso *link* foi criado e está parecido com um botão graças ao **class** o qual aplicamos no código (vide Figura 1). Porém, o botão ficou junto com a tabela, o que não é visualmente agradável. Vamos corrigir isso usando CSS.

Fonte: Elaboração própria

Figura 1: *View* de listagem de pessoas com botão.

Abra o arquivo **app/assets/stylesheets/application.css** e adicione, abaixo dos comentários, o código em negrito.

```
*= require_tree .
 *= require_self
 */

 .btn {
   margin-bottom: 1em !important;
}
```

Isso vai garantir que qualquer elemento com a classe **btn** tenha uma margem inferior de **1em**. Salve o arquivo, recarregue a página e você deverá visualizar algo similar à Figura 2.

Fonte: Elaboração própria

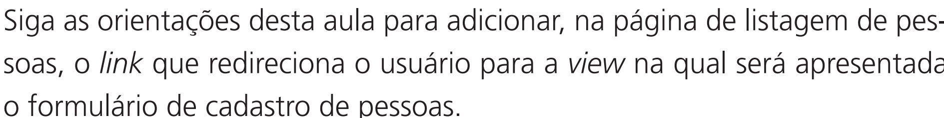
Figura 2: Tabela com margem superior.

Clique no botão e observe que já existe uma *view* sendo apresentada, conforme ilustrado na Figura 3. É nela que vamos construir o formulário de cadastro.

Fonte: Elaboração própria

Figura 3: *View* para cadastro de pessoas.

Atividade



Siga as orientações desta aula para adicionar, na página de listagem de pessoas, o *link* que redireciona o usuário para a *view* na qual será apresentada o formulário de cadastro de pessoas.

Construindo o formulário

Para nos auxiliar e facilitar a construção de formulários, vamos, mais uma vez, utilizar a gem Simple Form (https://github.com/plataformatec/simple_form). Para isso, adicione a seguinte linha de código ao seu **Gemfile**.

```
gem 'simple_form'
```

Em seguida, execute o comando **bundle install** no terminal de comandos para baixar e instalar a gem no seu projeto. Para concluir a instalação do Simple Form ao nosso projeto, execute o comando abaixo no terminal de comandos.

```
rails generate simple_form:install --bootstrap
```

Esse comando irá instalar os arquivos necessários para o funcionamento do Simple Form no nosso projeto, além de integrá-lo com o Bootstrap, conforme já apresentado na Unidade VIII/Aula 08 – Melhorando a apresentação (Parte II).

Atividade

Siga as orientações desta aula para instalar a gem Simple Form no seu projeto.

Concluída a instalação do Simple Form, precisamos modificar o método **new** do controlador **app/controllers/pessoas_controller.rb**. O que precisamos fazer no método new é criar o objeto que será preenchido com os dados informados pelo usuário no formulário. Portanto, adicione o código em negrito a seguir.

```
def new
  @pessoa = Pessoa.new
end
```

Criamos um objeto da classe **Pessoa** e guardamos na variável **@pessoa**, disponível na view **app/views/pessoas/new.html.erb** e apresentada após a execução desse método.

Em seguida, abra a view **app/views/pessoas/new.html.erb**, apague todo o seu conteúdo e adicione o código abaixo.

```
1  <h1>Cadastrar Pessoa</h1>
2
3  <%= link_to 'Voltar', pessoas_path, class: 'btn btn-
   default' %>
4
5  <%= simple_form_for @pessoa do |f| %>
6   <%= f.input :nome, required: true %>
7   <%= f.input :data_nascimento, as: :date, label: 'Data de
   nascimento', html5: true %>
8   <%= f.button :submit, 'Enviar', class: 'btn-default' %>
9  <% end %>
```

Iniciamos o código com **<h1>Cadastrar Pessoa</h1>** (linha 1), para apresentar ao usuário que ele se encontra no formulário de cadastro de pessoas. Na linha 3, adicionamos um link o qual permite ao usuário voltar para a listagem de pessoas.

Na linha 5, usamos o método **simple_form_for**, fornecido pela gem Simple Form, para iniciar a construção do formulário. Esse método recebe o objeto **@pessoa**, o qual foi criado no método **new** do controlador **app/control-**

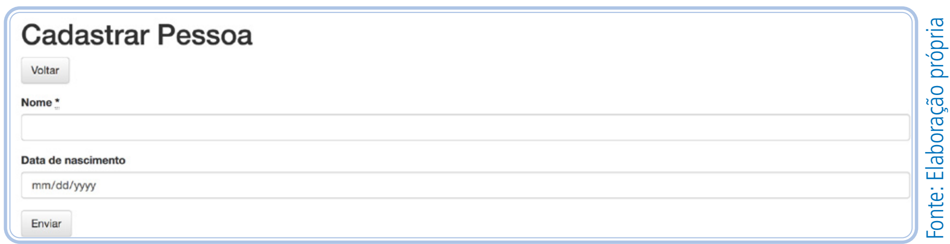
lers/pessoas_controller.rb. O objeto **@pessoa** será preenchido com os dados, fornecidos pelo usuário, através dos campos do formulário.

Na linha 6, usamos **f.input :nome** para adicionar o primeiro campo ao formulário. Nesse caso, estamos adicionando o campo que permitirá ao usuário preencher o nome da pessoa. O parâmetro **required: true** obriga que esse campo seja preenchido pelo usuário.

Na linha 7, usamos **f.input :data_nascimento** para adicionar o campo, o qual irá permitir ao usuário preencher a data de nascimento da pessoa. Para esse campo, estamos incluindo mais parâmetros: **as: :date** define que esse é um campo de data, e não um campo de texto simples; **label: 'Data de nascimento'** define o rótulo do campo; e **html5: true** habilita o seletor de datas padrão do navegador fornecido pelo HTML5.

Na linha 8, adicionamos um botão usando **f.button :submit**, passamos o parâmetro **'Enviar'** (o rótulo a ser exibido no botão), e, em seguida, passamos o parâmetro **class: 'btn-default'** para estilizar o botão com a classe **btn-default** do Bootstrap. Finalmente, na linha 9, concluímos a construção do formulário.

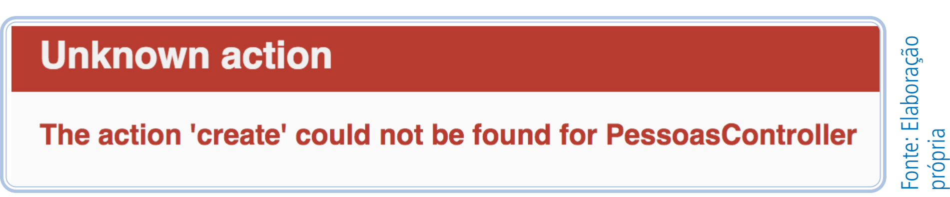
Ao recarregar a página no navegador, você deverá visualizar algo parecido com o formulário ilustrado na Figura 4.



Fonte: Elaboração própria

Figura 4: Formulário de cadastro de pessoas

Preencha o formulário com os dados de uma pessoa e clique em 'Enviar'. Ao submeter o formulário, o erro será apresentado, como na Figura 5.



Fonte: Elaboração própria

Figura 5: Erro ao submeter formulário

A mensagem de erro diz: "A ação '*create*' não pôde ser encontrada no controlador *PessoasController*". Tal erro ocorreu porque os dados foram enviados para o método **create** do controlador **app/controllers/pessoas_controller.rb**, mas esse método ainda não existe.

Atividade

Siga as orientações desta aula para implementar o método **new** do controlador **app/controllers/pessoas_controller.rb** e o formulário de cadastro de pessoas na *view* **app/views/pessoas/new.html.erb**.

Implementando o método create

Abra o controlador **app/controllers/pessoas_controller.rb** e adicione o método **create** apresentado abaixo.

```
1  def create
2   @pessoa = Pessoa.new(pessoa_params)
3  if @pessoa.save
4    redirect_to pessoas_path, notice: 'Pessoa cadastrada com sucesso!'
5  else
6    render :new
7   end
8  end
```

Iniciamos criando um novo objeto da classe Pessoa e guardando-o na variável **@pessoa**. A **pessoa_params** está sendo passada como parâmetro para o método **new**. Nesse aspecto, pessoa_params é um método não existente (será implementado em breve), responsável por indicar quais campos do modelo Pessoa podem ser preenchidos pelo usuário. É uma medida de segurança, a qual visa prevenir ataques de *mass assignment*.

Portanto, um objeto da classe Pessoa será criado e guardado na variável **@pessoa** (linha 2). Esse objeto ainda não foi armazenado no banco de dados. Para persisti-lo no banco, usamos o método **save** sobre o objeto **@pessoa** (linha 3). O método **save** retorna um valor booleano para indicar se o objeto foi salvo com sucesso ou não.

Caso o objeto seja salvo com sucesso, o **redirect_to** será executado (linha 4), redirecionando o usuário para **pessoas_path** (a listagem de pessoas implementadas

na aula passada). Além disso, passamos como parâmetro um **notice**, que é uma mensagem para indicar o cadastro com sucesso de uma pessoa.

Caso o objeto não tenha sido salvo no banco, o **render :new** será executado (linha 6). O **render** é um método que nos permite apresentar alguma *view* para o usuário, nesse caso, será apresentada a *view* a**pp/views/pessoas/new.html.erb** para o usuário poder verificar os problemas ocorridos no salvamento e preencher o formulário novamente.

Você pode estar se perguntando qual é a diferença entre o **redirect_to** e o **render**. Quando usamos o **redirect_to**, estamos redirecionando o usuário para um novo endereço, gerando uma nova requisição e, portanto, o método do controlador será executado e uma *view* será apresentada. Quando usamos o **render**, estamos apenas apresentando uma *view* para o usuário.

Em seguida, precisamos implementar o método **pessoa_params**, o qual está sendo usado pelo método **create** e será usado por outros métodos em breve. Para isso, adicione o código abaixo no final da classe **PessoasController** (**app/controllers/pessoas_controller.rb**).

```
private
 def pessoa_params
   params.require(:pessoa).permit(:nome, :data_nascimento)
end
```

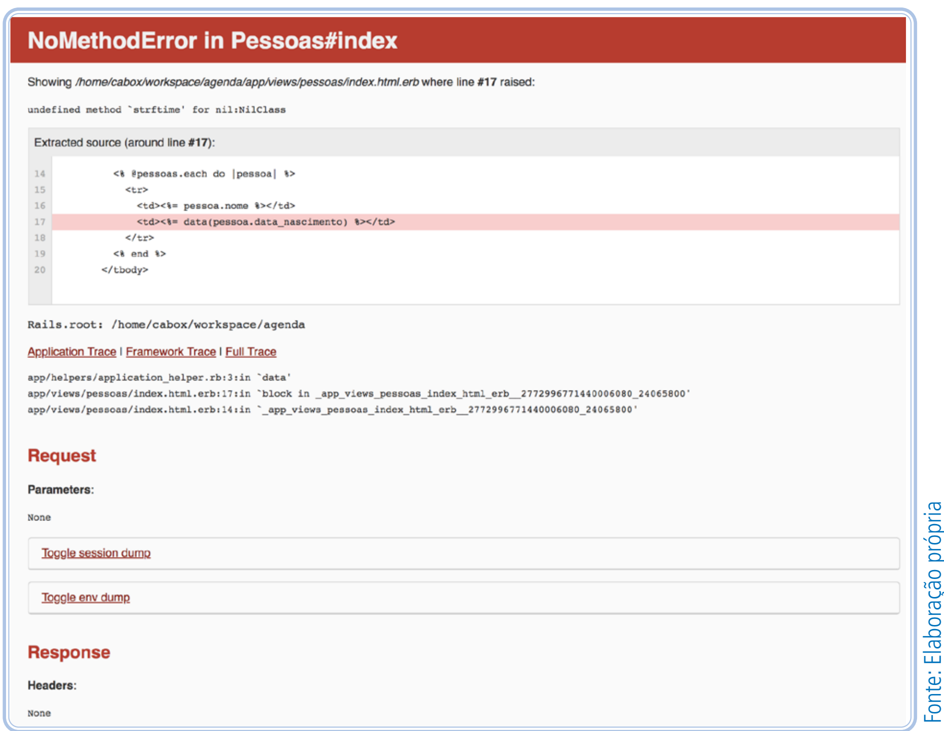
Como ele é um método que não será usado externamente, podemos defini-lo como privado. **params** é uma variável que guarda todos os parâmetros enviados na requisição, e, entre esses parâmetros, estão os dados preenchidos pelo usuário no formulário. Executamos o método **require** para obrigar que a requisição tenha dados sobre uma pessoa, e, em seguida, executamos o método **permit** para indicar quais são os atributos que poderão ser preenchidos com os dados enviados pelo usuário no formulário.

Atividade

Siga as orientações desta aula para implementar os métodos **create** e **pessoa_params** no controlador **app/controllers/pessoas_controller.rb**.

Concluída a implementação dos métodos **create** e **pessoa_params**, volte para o navegador, acesse o formulário de cadastro de pessoas, e,

dessa vez, preencha apenas o campo 'nome', deixando o campo 'data de nascimento' em branco. Submeta o formulário e, então, você deverá visualizar o erro abaixo.



Fonte: Elaboração própria

Figura 6: Erro ao submeter o formulário incompleto

A página de erro, na Figura 6, indica que ocorreu um problema na linha 17 da *view* **app/views/pessoas/index.html.erb**. A mensagem de erro diz "*undefined method 'strftime' for nil:NilClass*", ou seja, o método **strftime** não existe para o valor nulo (**nil**). Dessa forma, acidentalmente nós executamos o método **strftime** numa variável com valor nulo.

Isso ocorreu porque criamos uma nova pessoa sem uma data de nascimento, portanto o atributo **data_nascimento** dessa pessoa é nulo. Isso é possível, pois o campo 'data de nascimento' não é de preenchimento obrigatório. Nessa perspectiva, a nova pessoa foi salva com sucesso no banco de dados.

Contudo, na *view* de listagem de pessoas (**app/views/pessoas/index.html.erb**), nós estamos passando a data de nascimento de cada pessoa cadastrada como parâmetro para o método **data**, conforme ilustrado a seguir

```
<% @pessoas.each do |pessoa| %>
  <tr>
    <td><%= pessoa.nome %></td>
    <td><%= data(pessoa.data_nascimento) %></td>
  </tr>
<% end %>
```

O método **data**, o qual criamos em **app/helpers/application_helper.rb**, chama o método **strftime** sobre a variável passada como parâmetro sem antes verifica-la se ela é nula, conforme apresentado abaixo.

```
def data(data)
  data.strftime('%d/%m/%Y')
end
```

Portanto, quando um parâmetro nulo é passado para o método **data**, o erro apresentado na Figura 6 ocorre. Para corrigi-lo, precisamos alterar o código do método **data** (**app/helpers/application_helper.rb**) para verificar se o valor do parâmetro é nulo, conforme apresentado abaixo.

```
def data(data)
  if data.present?
    data.strftime('%d/%m/%Y')
  else
    '-'
  end
end
```

Essa nova implementação do método **data**, inicialmente, verifica se o parâmetro está presente (não é nulo) usando o método **present?**. Caso o parâmetro esteja presente, executa-se o método **strftime** sobre ele, caso contrário, o método retorna a String **'-'**.

Recarregue a página de listagem de pessoas, dessa vez, você não deve receber erros. Mas o *notice* com a mensagem 'Pessoa cadastrada com sucesso!' não foi exibido. Isso aconteceu porque nós ainda não definimos nas nossas *views* onde os *notices* devem ser exibidos.

Atividade

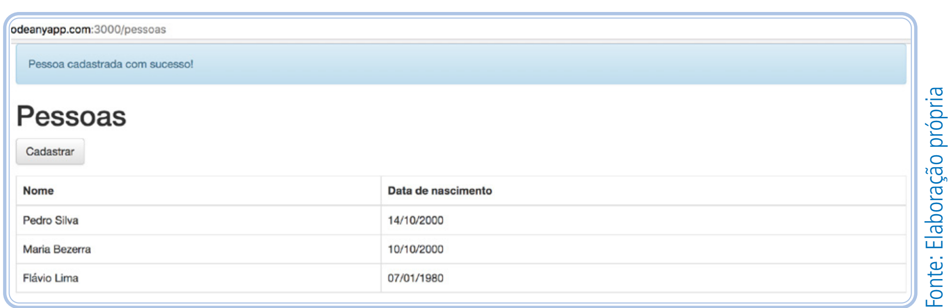
Siga as orientações desta aula para alterar a implementação do método **data**, presente em **app/helpers/application_helper.rb**.

Trabalhando nas views

Para que os *notices* possam ser exibidos em qualquer *view* da nossa aplicação, adicione o código dos notices na *view* **app/views/layouts/application.html.erb**. Abra essa *view* e adicione o código em negrito apresentado abaixo.

```
1  <div class="container">
2   <% if notice.present? %>
3    <div class="alert alert-info" role="alert"><%= notice %></div>
4   <% end %>
5
6  <% if alert.present? %>
7    <div class="alert alert-danger" role="alert"><%= alert %></div>
8   <% end %>
9
10  <%= yield %>
11 </div>
```

Na linha 2, utilizamos um **if** para verificar se existe um **notice** para ser exibido. Se existir, a linha 3 será exibida para o usuário. A linha 3 apresenta uma **div** com as classes **alert** e **alert-info**, ambas fornecidas pelo Bootstrap, para exibir uma mensagem para o usuário. O mesmo foi feito para o **alert** a partir da linha 6, mas qual é a diferença entre o **notice** e o alert? O notice normalmente é usado para dar mensagens de sucesso ou confirmação. Já o **alert**, para dar mensagens de alerta ou erro.



Fonte: Elaboração própria

Figura 7: Mensagem de sucesso

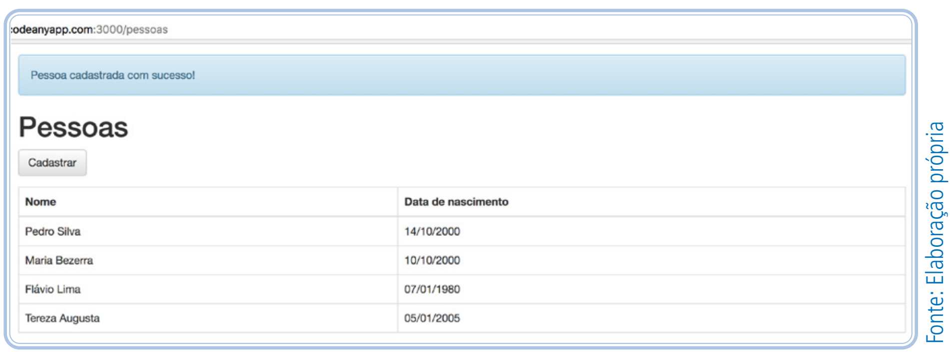
Volte para o formulário de cadastro de pessoas, preencha os campos e envie. Ao fazer isso, você deverá visualizar o *notice* (Figura 7).

Na Figura 7, a mensagem ficou junta com a margem superior da página, o que não é visualmente agradável. Para corrigir isso, usando CSS, abra a folha de estilos **app/assets/stylesheets/application.css** e adicione o seguinte código:

```
.container {
  margin-top: 1em;
}
```

O código CSS acima define quais elementos, com a classe **container**, vão ter uma margem superior de **1em**. Como todo conteúdo da nossa aplicação é apresentado dentro da **div** com a classe **container** na *view* **app/views/layouts/application.html.erb**, isso deverá resolver esse problema.

Ao fazer um novo teste preenchendo e submetendo o formulário de cadastro de pessoas, você deverá visualizar algo similar à Figura 8.



Fonte: Elaboração própria

Figura 8: *Notice* corrigido

Atividade

Siga as orientações desta aula para implementar as mensagens de sucesso e de alerta na *view* **app/views/layouts/application.html.erb** e fazer os devidos ajustes no CSS.

Resumindo

Nesta aula, construímos o formulário de cadastro de pessoas em nosso projeto. Para isso, foi necessária a instalação da gem Simple Form, a implementação dos métodos **create** e **pessoa_params** no controlador

app/controllers/pessoas_controller.rb, bem como a realização de ajustes em helper methods e nos estilos CSS da nossa aplicação. Desejamos que tenha aprendido todo o conteúdo das aulas e tenha feito um bom proveito deste material.

Referências

BOOTSTRAP. **Bootstrap**: the most popular html, css and js library in the world. Disponível em: http://getbootstrap.com/. Acesso em: 09 fev. 2018.

CAELUM. **Desenvolvimento Ágil para Web com Ruby on Rails 4**. São Paulo: Caelum.

FUENTES, V. B. **Ruby on Rails**: Coloque a sua aplicação web nos trilhos. São Paulo: Casa do Código, 2012.

RUBY. **Ruby programming language**. Disponível em: http://ruby-lang.org/. Acesso em: 12 dez. 2017.

RUBY ON RAILS. **Ruby on rails**: a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern. Disponível em: http://rubyonrails.org/. Acesso em: 12 dez. 2017.

RUBYONRAILS.ORG. **Ruby on Rails Guides**. Disponível em: http://guides.rubyonrails.org/. Acesso em: 14 maio 2016.

SILVA, M. S. **Rails Gilrs Tutorial**. Disponível em: http://www.maujor.com/railsgirlsguide/. Acesso em: 14 maio 2016.

INSTITUTO FEDERAL
Rio Grande do Norte
Campus Avançado Natal - Zona Leste

editoraifrn

Didáticos