treinadev

Ruby on Rails: do zero ao CRUD

CAMPUS CODE

O framework Rails

Introdução ao Rails

Gems e instalação

Quando falamos de Ruby, uma das expressões mais comuns é **Ruby on Rails**, que significa utilizar Ruby unido com o *framework* Rails.

O Rails é um *framework* de código aberto que prioriza a produtividade na criação de aplicações web. Com ele possuímos recursos para todos os níveis de uma aplicação web: recebemos solicitações de URLs, consultamos e inserimos dados em um banco, retornamos dados renderizados através de páginas HTML, enviamos e-mails, criamos uma API etc.

Apesar de toda essa complexidade, o Rails é basicamente um conjunto de código Ruby. Para facilitar a distribuição de códigos como o do Rails, o Ruby utiliza as **RubyGems**.

RubyGems é um formato padronizado e autossuficiente da linguagem para que desenvolvedores empacotem trechos de código e distribuam/compartilhem com outros desenvolvedores. Com isso, podemos incluir códigos externos no nosso projeto de forma simples. Ao instalar o Ruby, você ganha o comando `gem` no seu *shell*. Esse comando permite consultar e instalar gems pelo seu terminal.

Para consultar as gems já instaladas em nosso computador:

```
$ gem list
```

Podemos instalar o Rails utilizando alguns comandos no nosso terminal:

```
$ gem install rails
```

ou

```
$ gem install rails --no-document # podemos instalar sem
documentação, o que é mais rápido

$ rails --version # para verificar a versão instalada :)
```

Execute um `gem list` novamente e veja quantas gems foram instaladas. O Rails, mesmo sendo uma gem, depende de outras gems que foram instaladas automaticamente.

Dica

Esse conteúdo foi produzido usando a versão 5.2.3 do Rails. Para instalar essa versão especificamente você pode executar:

```
gem install rails --version 5.2.3
```

Você pode conhecer mais sobre as gems no site oficial RubyGems ou no RubyToolbox, que oferece um catálogo de gems organizado por categoria e classificado de acordo com alguns critérios (idade da gem, frequência de atualização, número de colaboradores, etc.).

Dica

Se quiser saber mais sobre o Rails, acesse o guia do Ruby on Rails: Rails Guides - O que é o Rails.

Criando um projeto

Vamos tirar aquele projeto da gaveta e colocá-lo em prática! Quando instalamos o Rails, ele já disponibiliza alguns comandos de terminal. Um dos principais comandos é o `rails new`, usado para criar um projeto Rails. Esse comando tem um parâmetro obrigatório: o nome da aplicação.

```
$ rails new task_list # cria uma estrutura de pastas e arquivos
```

Ao criar um projeto, você pode notar vários comandos sendo executados no terminal, mas podemos dividir em 2 etapas principais:

- **São criados vários arquivos e pastas que compõem a estrutura da aplicação**

Dentro de qualquer aplicação Rails, os arquivos e pastas que você vai criar devem seguir uma estrutura pré-estabelecida e essa estrutura é criada durante o `rails new`. São criados também alguns arquivos de configuração e o `Gemfile`.

- **É executado o comando bundle install**

O arquivo `Gemfile` citado acima descreve uma lista de gems que devem estar instaladas para garantir o funcionamento de uma aplicação Rails. O comando `bundle install` lê o `Gemfile` e garante que todas as gems descritas estão instaladas.

Dica

O `Gemfile` e o comando `bundle install` estão ligados a uma gem que faz parte do Ruby: o Bundler. Você pode conhecer mais lendo o site oficial: bundler.io.

Rodando nosso projeto

Criamos o projeto e entendemos melhor como tudo funciona no `rails new`, mas o que realmente queremos ver é a aplicação em execução. Para isso, precisamos entrar na pasta do projeto e rodar a aplicação. No terminal, rode os comandos abaixo:

```
$ cd task_list # entra na pasta da aplicação
$ rails server # roda a aplicação
```

Agora, no navegador, entre na sua aplicação através da URL: http://localhost:3000/.

Se tudo estiver correto, você deve ver uma página com a mensagem "**Yay! You're on Rails!**". Caso ela não abra corretamente, pode ser necessário habilitar a `gem mini_racer`. Para isso, vá no seu `Gemfile` e descomente a gem `mini_racer`.

```
gem 'mini_racer', platforms: :ruby
```

Em seguida, rode o comando `bundle install` para instalar essa gem e, em seguida, `rails server` novamente. Se você preferir, pode instalar um interpretador mais completo como o Node.js. Ele também soluciona o problema e retira a necessidade da `gem mini_racer`.

Se quiser parar o servidor, assim como qualquer aplicação Ruby, você pode usar a combinação de teclas `Ctrl+c` (no MacOS também é `Ctrl`).

Dica

Para saber mais sobre o processo de criação de um projeto Rails, leia o guia: Rails Guides - Creating a New Rails Project.

Como funciona uma aplicação Rails

Começamos a desbravar as terras do Rails e até recebemos boas-vindas! Apesar de amigável, essa tela não é o nosso foco. Queremos ver uma aplicação web construída por

nós! No entanto, antes de começar é preciso entender como a web funciona por trás dos panos.

Desde 1990, a internet usa um protocolo para distribuir informações por aí: o **HTTP** (*HyperText Transfer Protocol*). Ele é um protocolo de **requisição e resposta** utilizado para requisitar uma informação de um servidor por um cliente. Servidor, em resumo, é a máquina de onde estamos pedindo a informação. Esse protocolo surgiu da necessidade de padronizar como as máquinas que usam a internet se comunicam e está aí até hoje (com algumas melhorias).

Sempre que abrimos uma página em nosso navegador, estamos enviando requisições e recebendo respostas HTTP. Abaixo vamos resumir o funcionamento dessa troca de informações fazendo um paralelo com os principais componentes do Ruby on Rails.

Ao abrir o site da Campus Code, você digitou www.campuscode.com.br e pressionou `Enter`. Isso gerou uma requisição HTTP que pode ser decomposta, de forma simplificada, em:

- **Verbo ou Método:** GET
- **Domínio:** campuscode.com.br
- **Caminho Relativo:** `'/'`

Dentro do protocolo HTTP, o verbo representa que tipo de ação queremos tomar, e no caso do GET queremos obter um conteúdo. Outros exemplos são o POST, usado para criar novos dados, e o DELETE, usado para apagar dados.

Dica

Você pode ler mais sobre os verbos HTTP disponíveis aqui.

O domínio é a identificação para que nossa aplicação seja encontrada na internet (ou numa rede local em caso de redes privadas de computadores). Já o caminho relativo é tudo

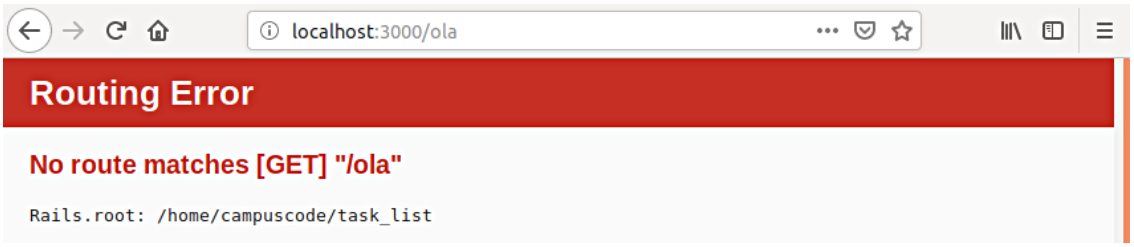
o que vem depois do domínio em nossa URL. Por padrão, ao acessar a tela inicial de um site o caminho relativo é sempre `'/'`, como em nosso exemplo.

Para o Rails (e outros *frameworks*), o verbo e o caminho relativo serão usados para determinar qual ação o usuário solicitou. A tabela abaixo mostra exemplos de requisições HTTP. Você consegue entender que a junção **verbo + caminho relativo** é suficiente para tomarmos decisões no código?

Domínio	Verbo	Caminho Relativo
globo.com	GET	/noticias
loja.com.br	DELETE	/produtos/12
sistema.com.br	POST	/inscricao

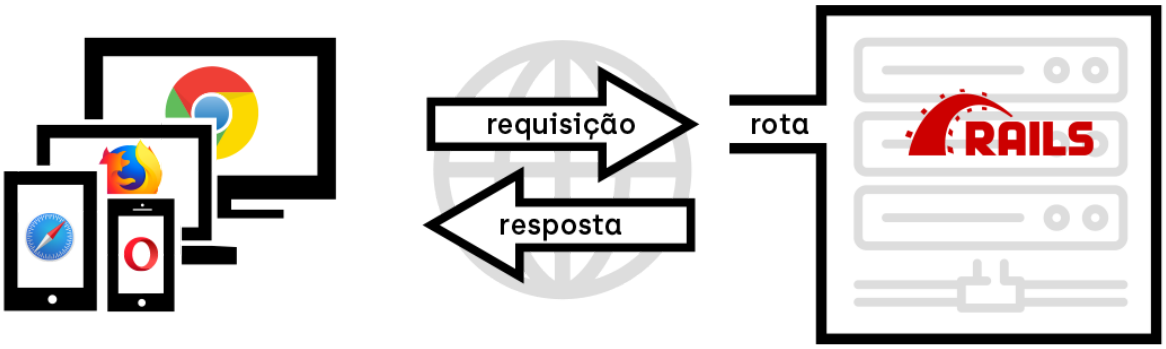
É importante lembrar que estamos falando de um protocolo, então independente de estar em Rails, Node, PHP ou Django, as requisições seguem sempre essa estrutura.

Voltando à nossa aplicação, ao acessar http://localhost:3000 você disparou um GET no caminho relativo `'/'`. Por se tratar da tela inicial, o Rails já traz uma mensagem de boas vindas automática, mas tente acessar outro endereço, por exemplo: http://localhost:3000/ola.



Bum! Tomamos um erro!

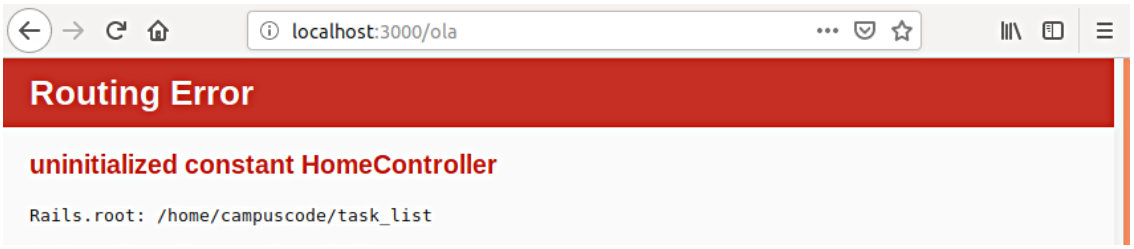
Lendo essa mensagem do Rails entendemos que não existe a rota `'/ola'`. Em Rails cada combinação de um verbo HTTP com um caminho relativo compõe uma rota. Uma rota leva a um trecho de código que ao final de sua execução devolve, na maioria das vezes, uma página HTML.



Vamos abrir o arquivo `routes.rb` na pasta `config` e então definir nossa primeira rota:

```
Rails.application.routes.draw do
  get '/ola', to: 'home#welcome'
end
```

Agora conseguimos receber um GET no caminho `'/ola'`. Ao tentar acessar http://localhost:3000/ola, teremos um erro diferente:



A primeira linha indica que ocorreu um erro de rota. A segunda, nos diz que não foi inicializado o `HomeController`, ou seja, ele ainda não existe na aplicação. Mas o que é um *controller*?

Dica

Para saber mais sobre rotas, acesse o guia: Rails Guides - Rails Routing from the Outside In.

Primeiro *Controller*

Vimos que em Rails cada rota relaciona uma requisição HTTP (verbo + caminho relativo) a um trecho de código que deve ser executado. Esse trecho de código, por convenção, deve ser sempre um método dentro de um *controller*.

Cada *controller* é uma classe criada dentro da pasta `app/controllers` e cada método dessa classe está vinculado a uma ação da aplicação. Enviar um e-mail, exibir uma página HTML, exibir um formulário e receber os dados de um formulário são exemplos de ações cujos códigos estarão nos *controllers*. Pareceu complicado? Vamos por partes.

Como temos uma rota para `'/ola'`, quando entramos na URL inicial (`'localhost:3000/ola'`) na verdade estamos "chamando" essa rota. Quando essa rota foi declarada, definimos algo muito importante na aplicação: que a rota `'/ola'` deve mandar o *controller* `Home` chamar a *action* `welcome`. Isso é o que queremos dizer quando escrevemos `home#welcome` no arquivo `routes.rb`.

Agora precisamos criar o `HomeController` para processar essa requisição. Na pasta `app/controllers` vamos criar um novo arquivo chamado `home_controller.rb`. O Rails tem essa convenção de todos os *controllers* terem `controller` no nome.

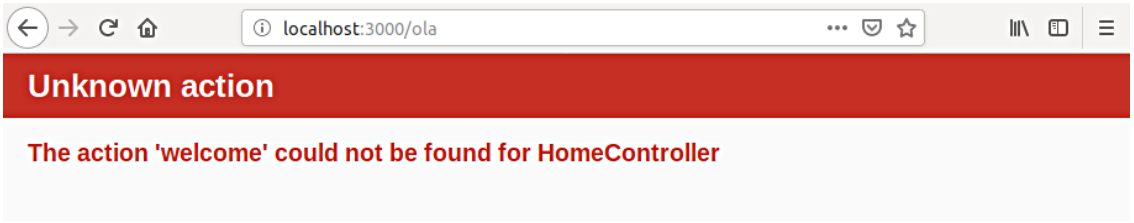
Se desejar usar comandos de Terminal para ir se acostumando:

```
$ touch app/controllers/home_controller.rb
```

Criar um *controller* que processa uma requisição HTTP parece algo bastante complicado, certo? É preciso tratar parâmetros enviados via URL ou formulário, renderizar diferentes formatos na resposta (HTML, JSON, CSV etc), apresentar código de status da resposta… Por isso, o Rails já implementa um *controller* base e nós só precisamos criar *controllers* que herdam dele. Vamos incluir código nosso arquivo `home_controller.rb`?

```
class HomeController < ApplicationController
end
```

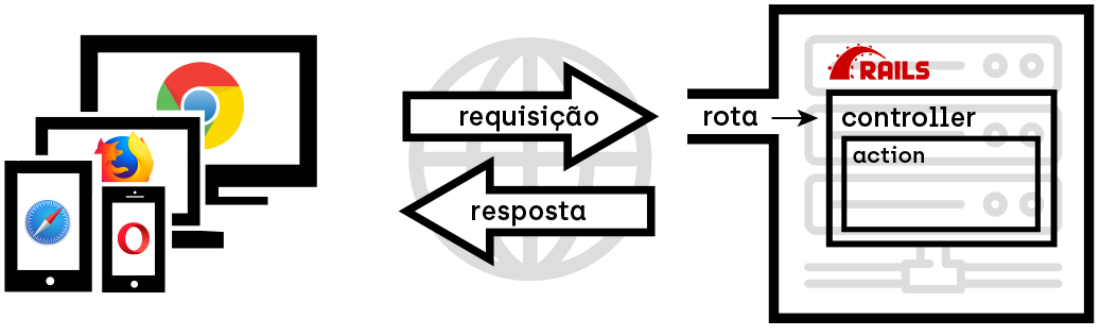
A notação `<` indica que o `HomeController` herda de `ApplicationController` os comportamentos existentes no Rails. Rode o comando `rails server` novamente. Ao recarregar a página da sua aplicação vai receber um novo erro: a *action* `welcome` não existe.



Num *controller*, a *action* é simplesmente um método que utilizamos para tratar uma requisição HTTP. Então basta criarmos o método `welcome`.

```
class HomeController < ApplicationController
  def welcome
  end
end
```

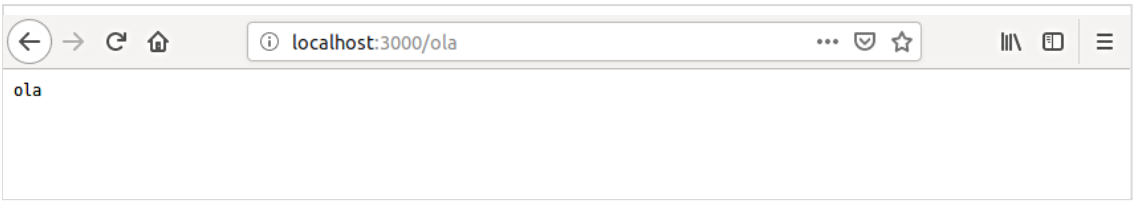
Executar a *action* `welcome` é o último passo da requisição HTTP dentro da nossa aplicação. O código que vamos colocar dentro da *action* vai determinar a resposta HTTP que será retornada para o navegador do cliente/usuário.



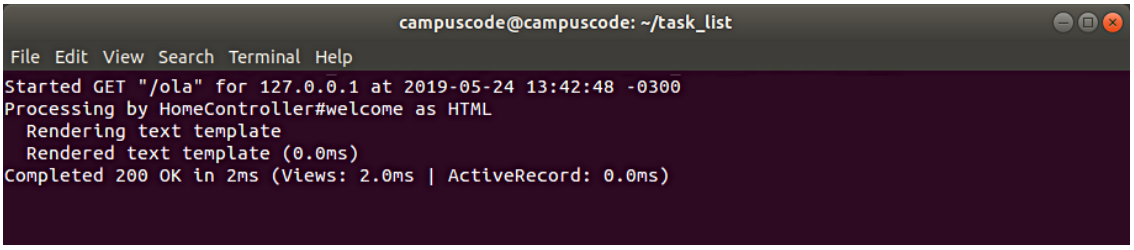
Vamos começar com a resposta mais simples possível:

```
class HomeController < ApplicationController
  def welcome
      render plain: 'ola'
  end
end
```

Ao recarregar a página no seu navegador você vai ver o texto “ola”. É isso que o `render plain` faz: ele produz uma resposta HTTP com o texto “ola”. Essa resposta, como toda resposta HTTP, possui também um código de status. Esse código indica se uma requisição foi concluída com sucesso ou erro e mais uma vez tiramos proveito do Rails: ele cuida sozinho do status.



Nesse momento já temos um ciclo completo de requisição e resposta HTTP dentro da nossa aplicação Rails. Uma forma de visualizar esse ciclo é acompanhar os logs do Rails no terminal.



Mas não queremos só mandar texto puro, certo? Queremos ter uma página HTML com vários dados da nossa tarefa. Agora vamos entrar na parte em que geramos nossas páginas HTML: as ***views*** do Rails.

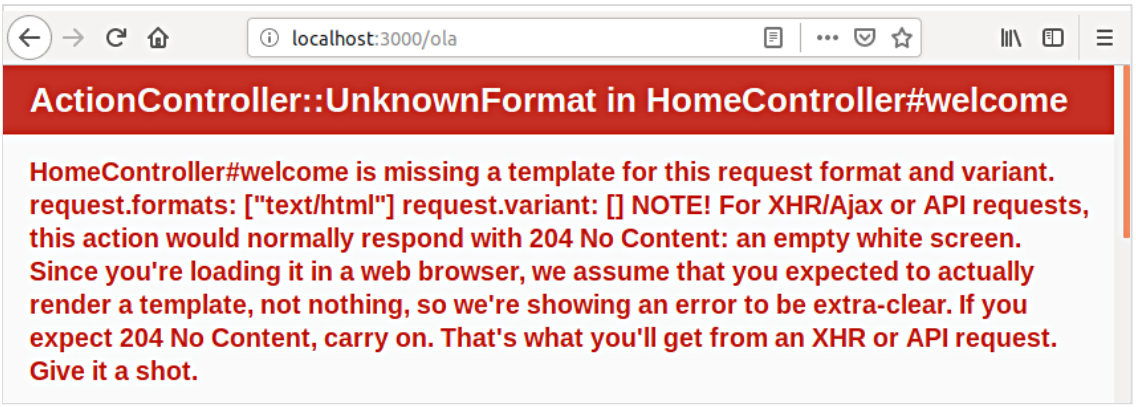
Dica

Se você quiser estudar mais *controllers* e *actions*, acesse os seguintes guias:

- Rails Guides - What Does a Controller Do?
- Rails Guides - Controller Naming Convention
- Rails Guides - Methods and Actions

Criando uma *view*

Agora que já temos a rota, o *controller* e a *action*, precisamos gerar nossa *view*. Ela é, basicamente, uma página da nossa aplicação. No exemplo anterior, nós usamos o `render plain` para mostrar um texto na página. Se você apagar essa linha e tentar recarregar a página, vai receber um erro estranho:



Isso quer dizer que o Rails não sabe o que fazer. Mas não estava tudo certo?! Está! Mas, por padrão, uma *action* do Rails procura uma página HTML com seu próprio nome, no nosso caso `welcome`. Vamos criá-la, então?

Na nossa pasta `app/views` vamos criar uma outra pasta chamada `home`, que é o nome do nosso *controller*. Dentro dela vamos criar uma página com o nome da nossa *action*, um arquivo chamado `welcome.html.erb`.

```
$ mkdir app/views/home
$ touch app/views/home/welcome.html.erb
```

Nosso arquivo tem duas extensões: HTML, para gerar páginas que os navegadores conseguem interpretar, e `.erb`, que é um formato conhecido como *Embedded Ruby*. Com ele podemos executar código Ruby no meio das páginas HTML.

Por exemplo, nesse arquivo podemos colocar o seguinte código:

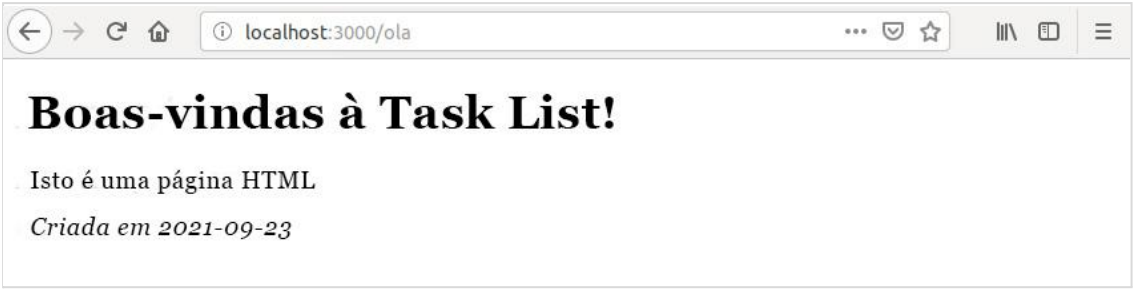
```
<%# 'Isto é um comentário' %>
<%= 1 + 1 %>
<% 2 + 2 %>
```

Recarregue sua página! Percebeu que só tem um número 2 na página? Quando usamos a notação `<%= seu_codigo_ruby_aqui %>` quer dizer que queremos mostrar na tela o retorno da execução do código Ruby. A notação `<% um_outro_codigo %>`, isto é, sem o sinal de `=`, simplesmente executa o código, mas não mostra nada na tela.

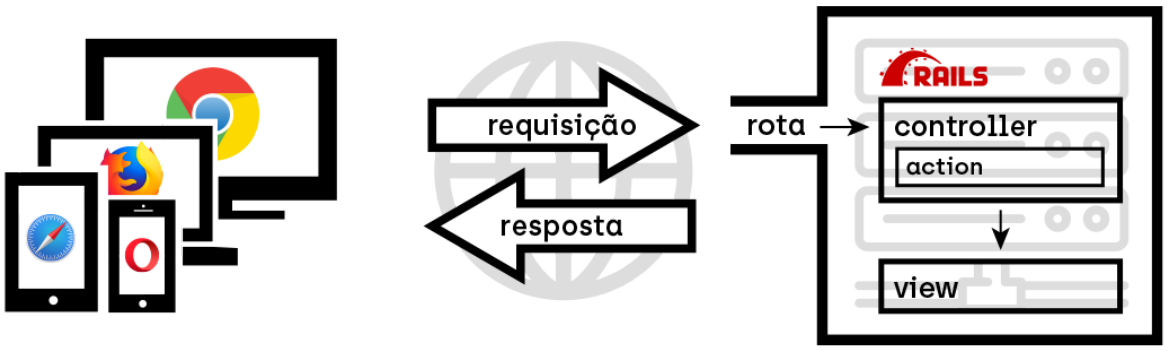
Vamos criar uma página simples, mas que demonstra esses recursos em nossa *view* `welcome.html.erb`.

```
<h1>Boas-vindas à Task List!</h1>
<p>Isto é uma página HTML</p>
<em>Criada em <%= Date.today %></em>
```

Pronto! Se você recarregar seu navegador vai ver uma página similar a esta:



É importante lembrar que os navegadores não interpretam código Ruby. Quando uma *view* é carregada, a nossa aplicação executa todo código Ruby presente no arquivo `welcome.html.erb` e produz um arquivo puramente HTML ao final. Este arquivo puramente HTML é enviado como a resposta da requisição HTTP.



A rota root

Vimos anteriormente como o Rails permite declarar rotas utilizando os verbos HTTP, mas para a página principal da nossa aplicação, podemos usar um atalho: o método `root`. Esse método define a tela inicial da nossa aplicação, ou seja, a requisição GET no caminho `'/'`. Com isso vamos mover nossa tela de boas-vindas para http://localhost:3000.

Em seu arquivo `config/routes.rb` remova a definição da rota `'ola'` e crie uma nova definição usando o método `root to`. Repare que nesse método só precisamos declarar o *controller* e a *action* uma vez que o caminho da tela inicial é sempre o mesmo.

```
Rails.application.routes.draw do
  root to: 'home#welcome' #root define a rota '/' da nossa aplicação
end
```

Model: o coração das suas regras de negócio

Criando um *model*

Independente da natureza de uma aplicação web, seja um *e-commerce*, um blog ou uma rede social, todas têm regras de negócio próprias. No Rails (e em qualquer *framework* que siga o padrão MVC), cabe aos *models* a função de definir essas regras e manipular informações junto ao banco de dados. Por exemplo: se temos uma rede social, vamos precisar armazenar dados do usuário como nome, e-mail e uma foto. Devemos ainda validar se o mesmo e-mail já está cadastrado para evitar duplicidades. Caberia a um *model* (que podemos chamar de `User`) essas responsabilidades.

Essa é a última peça do quebra-cabeça do padrão MVC de arquitetura do Rails. Então, vamos criar um *model*?

Na pasta `app/models` crie o arquivo `task.rb` como fizemos anteriormente.

```
class Task
  attr_reader :description, :status

  def initialize(description, status = false)
    @description = description
    @status = status
  end
end
```

Todos os *models* declarados na pasta `app/models` ficam automaticamente disponíveis para uso nos *controllers*. Então, no `HomeController` podemos criar alguns objetos do *model* `Task` para mostrar na nossa *view*.

```
class HomeController < ApplicationController
  def welcome
    tasks = []
    tasks << Task.new('Comprar pão')
    tasks << Task.new('Comprar leite')
  end
end
```

A *view* `welcome.html.erb` é carregada automaticamente ao final da execução da *action* acima. A *view* pode então acessar as variáveis declaradas na sua respectiva *action*, mas para isso precisamos marcar essas variáveis de uma forma especial. Atualize então o seu código no *controller* adicionando um `@` na variável `tasks`.

```
class HomeController < ApplicationController
  def welcome
    @tasks = []
    @tasks << Task.new('Comprar pão')
    @tasks << Task.new('Comprar Leite')
  end
end
```

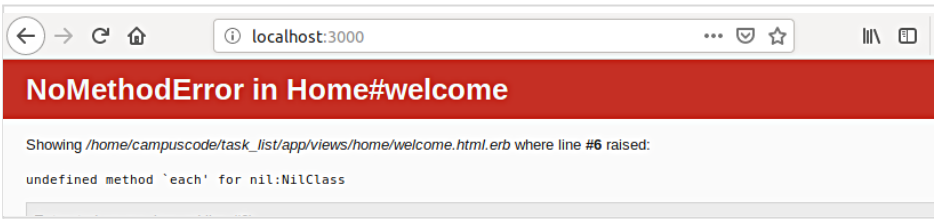
Na `welcome.html.erb` podemos então mostrar nossas tarefas, certo? Lembre-se de que todo código Ruby deve vir entre `<% %>` ou `<%= %>` se quisermos imprimí-lo na tela.

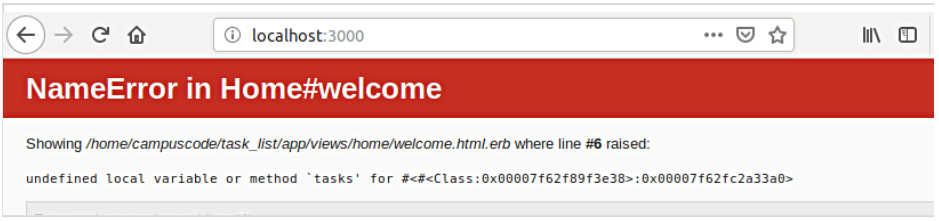
```
<h1>Boas-vindas à Task List!</h1>
<h3>Sua lista de tarefas:</h3>

<!-- Isso é um comentário: note que, no exemplo abaixo, não queremos
imprimir a iteração (each), somente a descrição das tarefas -->

<% @tasks.each do |task| %>
  <%= task.description %>
<% end %>
```

Pronto! Tudo certo! Agora temos *model*, *view* e *controller* se comunicando!

Dica

Caso você esqueça do `@` no *controller* ou na *view*, a aplicação vai lançar um erro bem estranho. Fique de olho!





O que herdamos do Rails

Nós criamos um *model* que funciona perfeitamente, mas se pensarmos em termos de aplicações web, temos que **persistir** nossos objetos para outras pessoas poderem acessar. O que isso significa?

O que temos até agora são objetos que não são persistidos na nossa aplicação e são gerados diretamente no *controller*. Eles serão sempre criados toda vez que rodarmos a *action* `welcome` da nossa aplicação e isso não é bom.

O Rails nos dá uma forma de persistir nossas tarefas no banco de dados de forma facilitada utilizando o Active Record. Ele é uma gem que está nas dependências do Rails e é responsável por operar dados, lendo e escrevendo no banco de dados.

Para começar a usar o *Active Record* temos que criar nossos *models* utilizando um comando do Rails no terminal. Isso é necessário para o Rails criar a infraestrutura no banco de dados para armazenar as informações. Então você pode apagar o arquivo `task.rb`, pois vamos criar o *model* através do Rails.

```
$ rm app/models/task.rb
```

Rode o seguinte comando no Terminal:

```
$ rails generate model task description:string status:boolean
```

O comando acima pede para o Rails gerar um *model* `Task` com os atributos `description` como texto e `status` como *boolean*. Você deve estar se perguntando: “Mas, se no Ruby podemos trocar o tipo a qualquer momento, por que estamos definindo um tipo na criação do *model*?”

A diferença aqui é que em Ruby podemos trocar os tipos dinamicamente, mas para o banco de dados é muito importante saber qual tipo de dado será salvo.

Agora que criamos um *model* do Rails, precisamos gerar a estrutura que o salva no banco de dados. No Rails, fazemos isso com *migrations*.

Migrations são a história do nosso banco: o que criamos, modificamos e apagamos de tudo que vamos salvar. Quando criamos um *model*, já foi gerada uma *migration* na pasta `db/migrations`, só temos que rodá-la. Fazemos isso com o comando de Terminal `rails db:migrate`.

No log da execução desse comando é possível ver que foi criada uma **tabela** ‘tasks’ com as **colunas** 'description' e 'status'. Já vamos falar delas novamente.

Agora, se olharmos o arquivo do *model* `Task`, ele estará assim:

```
class Task < ApplicationRecord
end
```

Não se assuste! Sumiu todo código de atributos, além do nosso método `initialize`. Isso é o que muitos chamam de “mágica do Rails”, mas, de forma simplificada, ele conecta seu *model* à tabela 'tasks' no banco de dados e recria os atributos a partir das colunas dessa tabela. Tudo isso acontece porque o código que é herdado da classe *ApplicationRecord*.

É preciso fazer uma alteração no *controller* pois nosso *model* passou a ter um novo método construtor. O novo método também foi herdado de *ApplicationRecord* e recebe um *hash* como parâmetro.

```
class HomeController < ApplicationController
  def welcome
    @tasks = []
    @tasks << Task.new(description: 'Comprar pão')
    @tasks << Task.new(description: 'Comprar leite')
  end
end
```

Dica

Não conhece *hashes*? São estruturas de chave e valor do Ruby. Você pode ver mais na documentação do Ruby ou na nossa apostila de **Lógica de Programação em Ruby**.

Se testarmos o nosso código com o `rails server`, veremos que tudo está funcionando perfeitamente e no padrão Rails. Vamos seguir nos trilhos?

Primeiros Métodos

Com o nosso *model* criado do jeito que o Rails espera, podemos começar a usar o banco de dados. Para criar objetos no banco podemos utilizar o método `create`. Para fazer isso, vamos parar o nosso servidor web com o comando de teclas `Ctrl+c` e digitar no terminal `rails console`. Aqui podemos testar códigos e, inclusive, criar elementos no banco.

No `rails console` podemos rodar os seguintes comandos para criar nossas tarefas no banco:

```
campuscode@campuscode: ~/task_list
File  Edit  View  Search  Terminal  Help
campuscode@campuscode:~/task_list$ rails console
Running via Spring preloader in process 4765
Loading development environment (Rails 5.2.3)
2.6.3 :001 > Task.create(description: 'Comprar pão', status: false)
   (0.1ms)  begin transaction
  Task Create (0.3ms)  INSERT INTO "tasks" ("description", "status", "created_at", "updated_at") VAL
UES (?, ?, ?, ?)  [["description", "Comprar pão"], ["status", 0], ["created_at", "2019-05-24 18:42:2
4.330334"], ["updated_at", "2019-05-24 18:42:24.330334"]]
   (3.0ms)  commit transaction
 => #<Task id: 1, description: "Comprar pão", status: false, created_at: "2019-05-24 18:42:24", upda
ted_at: "2019-05-24 18:42:24">
2.6.3 :002 > Task.create(description: 'Comprar leite', status: false)
   (0.1ms)  begin transaction
  Task Create (0.7ms)  INSERT INTO "tasks" ("description", "status", "created_at", "updated_at") VAL
UES (?, ?, ?, ?)  [["description", "Comprar leite"], ["status", 0], ["created_at", "2019-05-24 18:42
:58.559250"], ["updated_at", "2019-05-24 18:42:58.559250"]]
   (4.9ms)  commit transaction
 => #<Task id: 2, description: "Comprar leite", status: false, created_at: "2019-05-24 18:42:58", up
dated_at: "2019-05-24 18:42:58">
2.6.3 :003 >
```

Repare que ao confirmar o comando `create`, o Rails imprime na tela um resumo do objeto criado no banco de dados. Além da descrição e do status, você pode ver que foi criado um `id` numérico e a data e hora de criação de cada objeto.

Se tudo correu bem, você pode buscar por todas tarefas cadastradas com `Task.all`:

```
2.6.3 :003 > Task.all
  Task Load (0.7ms)  SELECT  "tasks".* FROM "tasks" LIMIT ?  [["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<Task id: 1, description: "Comprar pão", status: false, created_at: "
2019-05-24 18:42:24", updated_at: "2019-05-24 18:42:24">, #<Task id: 2, description: "Comprar leite"
, status: false, created_at: "2019-05-24 18:42:58", updated_at: "2019-05-24 18:42:58">]>
2.6.3 :004 >
```

Sucesso! Nossa aplicação agora tem um banco de dados integrado. Você pode sair e voltar para o `rails console` que todas tarefas continuam lá.

Agora vamos voltar à nossa aplicação Rails e listar todas as tarefas na *action* `welcome`. Como o `rails console` compartilha o mesmo banco de dados do `rails server`, basta você chamar o método `Task.all` dentro do *controller*.

```
class HomeController < ApplicationController
  def welcome
    @tasks = Task.all
  end
end
```

Mas o que está acontecendo? O método `all` busca todas as entradas na tabela 'tasks' do banco de dados e, para cada uma, recria um objeto da classe `Task`. Esses objetos são agrupados em um *array* que é devolvido ao final da execução do método.

Este é apenas um de centenas de métodos que o *ActiveRecord* disponibiliza nos *models*. Outro método bastante comum é o `find`, que procura somente um objeto por vez a partir do seu `id`.

```
campuscode@campuscode: ~/task_list
File  Edit  View  Search  Terminal  Help
campuscode@campuscode:~/task_list$ rails console
Running via Spring preloader in process 5095
Loading development environment (Rails 5.2.3)
2.6.3 :001 > Task.find(2) #Traz o elemento com id 2
  Task Load (0.3ms)  SELECT  "tasks".* FROM "tasks" WHERE "tasks"."id" = ? LIMIT ?  [["id", 2], ["LI
MIT", 1]]
 => #<Task id: 2, description: "Comprar leite", status: false, created_at: "2019-05-24 18:42:58", up
dated_at: "2019-05-24 18:42:58">
2.6.3 :002 >
```

Dica

Você pode ler mais sobre outros métodos no Rails Guides: Active Record Query Interface.

Agora você sabe como fazer pequenas buscas no banco, achar objetos e mostrá-los na tela! Lembre-se que você pode rodar sua aplicação e ver as tarefas no navegador.

CRUD

CRUD refere-se às operações executadas com os objetos na sua aplicação: *Create* (criar), *Read* (ler), *Update* (atualizar) e *Delete* (deletar). Vamos explorar um pouco cada uma delas.

Cadastrando uma tarefa

Já exibimos tarefas na tela e vimos como buscar tarefas específicas entre as existentes, mas, na hora de cadastrar tarefas, fizemos diretamente pelo `rails console`. Isso não é o ideal, afinal, nosso usuário não tem acesso ao console nem aos nossos servidores para cadastrar suas tarefas. Então será necessário cadastrar tarefas por meio de uma página da nossa aplicação.

Sempre que uma nova funcionalidade é adicionada à nossa aplicação, temos que passar pelos mesmos passos **rota** → ***controller*** → ***view*** e o ***model***. Então vamos começar pelas rotas.

Quando se trata de *models*, o Rails nos confere uma pequena facilidade para declarar rotas que manipulam cada *model*. Podemos declarar rotas de `resources`. Na nossa aplicação, o *model* que vamos precisar cadastrar é o `Task`. Então adicione a linha resources `:tasks` em seu arquivo de rotas.

```
Rails.application.routes.draw do
  root to: 'home#welcome' #root define a rota '/' da nossa aplicação
  resources :tasks
end
```

Dica

Ter uma rota principal (`root`) é uma boa prática, pois sempre que algo de errado acontecer sabemos que podemos mandar o usuário para o `'/'`.

Ao usarmos o `resources`, ele cria automaticamente diversas rotas. Podemos ver isso no Terminal digitando o comando `rails routes`, que mostra as rotas definidas pela aplicação. A tabela abaixo apresenta as rotas definidas no `resources` separando o seu verbo HTTP, caminho, *action* no *controller* e a explicação do seu uso para facilitar a leitura:

Verbo HTTP	Caminho	Controller#action	Uso
GET	/tasks	tasks#index	Página com todas as tarefas.
GET	/tasks/new	tasks#new	Página para o formulário de criação de tarefas.
POST	/tasks	tasks#create	Criar uma tarefa no banco.
GET	/tasks/:id	tasks#show	Mostra detalhes de uma tarefa.
GET	/tasks/:id/edit	tasks#edit	Página para o formulário de edição de tarefas.
PUT/PATCH	/tasks/:id	tasks#update	Edita uma tarefa no banco.
DELETE	/tasks/:id	tasks#destroy	Apaga uma tarefa no banco.

Dica

Outras rotas podem aparecer no seu Terminal, como o `root` e rotas complementares de funcionalidades do Rails.

Como você pode ver, o Rails não brinca em serviço e cria várias rotas prontas para consumo. Todas elas são relacionadas a verbos HTTP. O legal é que ele respeita o padrão do protocolo HTTP: todas as rotas que obtêm dados do sistema são feitas via GET, as que criam/alteram/removem dados da aplicação usam outros verbos.

Voltando à nossa tarefa, queremos cadastrar um item no sistema, mas pela tabela acima, temos duas rotas relacionadas à criação: `new` e `create`. Sempre que precisamos obter dados do usuário em uma aplicação web, precisamos exibir um formulário e depois receber os dados informados. Ou seja, cada formulário do nosso CRUD vai demandar duas requisições HTTP. A rota `new` representa o GET no formulário, que traz a página HTML com os campos para cadastro, então vamos começar por ela.

Com a aplicação rodando podemos ir até http://localhost:3000/tasks/new/ no navegador e, ao ver o erro, já descobrimos o problema: não temos um `TasksController` para criar nossas tarefas. Até agora estávamos usando o `HomeController`.

Então vamos definir nosso `TasksController` com uma *action* new. Como? Criando um arquivo `tasks_controller.rb` na pasta `app/controllers` com o seguinte:

```
class TasksController < ApplicationController
  def new
  end
end
```

Agora podemos trabalhar na *view* `/views/tasks/new.html.erb`. Lá podemos definir nosso formulário. Como estamos tratando de objetos da nossa aplicação, ao invés de usar os formulários normais do HTML, podemos usar códigos que o Rails nos fornece. Eles geram o mesmo resultado, mas facilitam nosso trabalho na hora de lidar com objetos.

Abaixo definimos um `form_for`, que, como o nome diz, é um formulário para um objeto e seus campos.

```
<%= form_for Task.new do |f| %>
  <%= f.label :description, 'Descrição' %>
  <%= f.text_field :description %>
  <%= f.label :status, 'Status' %>
  <%= f.check_box :status %>
  <%= f.submit 'Criar' %>
<% end %>
```

O `form_for` é um método que recebe qual objeto queremos criar com esse formulário, no nosso caso uma nova `Task`. Ele também usa um bloco `do |f|` para associar os campos do objeto a campos do HTML. Dessa forma, para cada atributo do nosso objeto podemos definir o jeito que queremos mostrá-lo na tela com um `label`, que é o texto mostrado junto ao campo.

A partir de agora, com o nosso formulário criado, podemos testar nossa *view* http://localhost:3000/tasks/new/ e criar uma nova tarefa.

Ao preencher os campos e clicar no botão 'Criar' note que receberemos um erro na *action* `create`. Por quê?

A *action* `new` só gera a página com o nosso formulário. Para enviar dados no HTTP nós usamos o verbo POST. Quando o usuário clica no botão 'Criar', nosso formulário já envia um POST com os dados da nossa tarefa para a *action* `create`. Agora temos que recolher esses dados e criar uma tarefa com eles.

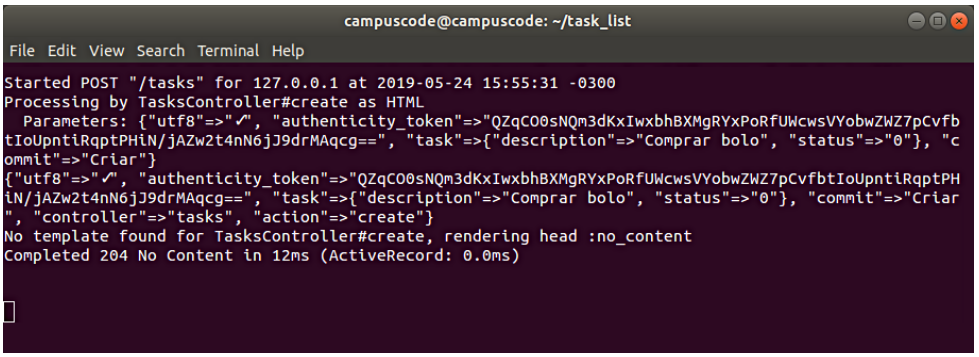
No *controller*, os dados vindos de qualquer formulário são recebidos por meio de uma variável chamada `params`. Essa variável é um *hash* e sua estrutura depende da requisição HTTP recebida. No exemplo da nossa tarefa, estamos recebendo o seguinte: `{ "task": { "description": "Comprar bolo", "status": true } }`.

Dica

Você pode ver o conteúdo do `params` adicionando o comando `puts params` dentro da *action* `create`.

```
class TasksController < ApplicationController
  def new
  end

  def create
      puts params
  end

end
```

Ao submeter o formulário, os parâmetros serão apresentados no Terminal:



Dica

Alguns tipos de bancos de dados tratam a variável *boolean* como `0` ou `1`. Por isso aqui o `status` deve aparecer como `0`, mas o Rails interpreta o valor como `true`.

Com esses dados devemos criar e salvar um objeto do tipo `Task`, de forma similar ao que fizemos via `rails console`.

```
class TasksController < ApplicationController
# Daqui em diante vamos omitir parte do código que se repete para
facilitar sua leitura.

  def create
      @task = Task.new(description: params[:task][:description],
status: params[:task][:status])
      @task.save #salva no banco a tarefa que criamos
  end
end
```

Criamos nossa tarefa! Mas o Rails nos confere uma maneira de tratar dados que garante um pouco mais de segurança e reduz nosso código, especialmente em formulários maiores.

```
params.require(:tasks).permit(:description, :status)
```

Esse formato é adotado desde a versão 4 do Rails e é chamado de *Strong Parameters*. Aqui requeremos um objeto do tipo `Task` e permitimos somente seus dois atributos `description` e `status`. Assim, nosso usuário não pode enviar qualquer outro código malicioso. :)

```
class TasksController < ApplicationController
  def create
    @task = Task.new(params.require(:task).permit(:description,
:status))
    @task.save #salva no banco a tarefa que criamos
  end
end
```

Além disso, uma outra melhoria que podemos fazer no código é, depois que a tarefa for salva com sucesso, apresentarmos a página inicial. Fazemos isso com o comando `redirect_to root_path`. O `redirect_to` gera uma nova requisição HTTP, por isso não criamos a *view* `create.html.erb`.

Dica

Em geral, somente as *actions* relacionadas ao verbo GET possuem uma *view*.

```
class TasksController < ApplicationController
  def create
    @task = Task.new(params.require(:task).permit(:description,
:status))
    @task.save #salva no banco a tarefa que criamos
    redirect_to root_path #viu como o root é importante
  end
end
```

Se você ainda não declarou um `root_path` na sua aplicação, volte no conteúdo sobre *views*. :)

Editando uma tarefa

Ao cadastrar uma tarefa pensamos num próximo problema: e se nosso usuário cadastrar uma tarefa incorretamente e quiser mudá-la?

Como já temos as rotas criadas, podemos trabalhar nos *controllers* e nas *views*. Na nossa página temos toda a lista de tarefas e podemos colocar um botão para editar cada uma delas. No Rails, podemos fazer isso com o `link_to`, que gera um link no HTML para uma rota.

Aliado a isso, precisamos explicitar que o link deve enviar para o caminho de editar uma tarefa. Se você executou o `rails routes`, deve se lembrar da coluna *prefix*, ele é um atalho que o Rails cria para aquela rota. Para cada prefixo, o Rails cria dois métodos, cada um com um sufixo diferente.

Para obter a URL com o caminho completo usamos o sufixo `_url` e para caminho relativo, o `_path`.

Ao usar o método `root_path`, o Rails retorna o caminho relativo da raiz da aplicação: `'/'`. Já o método `root_url` enviaria o usuário para http://localhost:3000/, o caminho completo da aplicação. Isso faz muita diferença quando precisamos compartilhar os links fora de nossa aplicação, como em e-mails ou relatórios, mas não se preocupe com isso agora.

Rodando `rails routes` novamente e analisando as rotas, podemos verificar que o prefixo do caminho para editar uma tarefa é `edit_task` e nessa rota temos um novo elemento, o `:id`. Para editar, exibir ou remover uma tarefa precisamos saber o seu `id`, e cada tarefa vai ter um `id` diferente. Essa notação funciona como uma variável na própria rota. Ou seja, as requisições `GET /tasks/1`, `GET /tasks/2`, `GET /tasks/3` (e assim por diante) vão todas levar a `tasks#show` e teremos os valores 1, 2 ou 3 disponíveis em `params[:id]`.

```
Prefix      Verb     URI Pattern                      Controller#Action
root        GET      /                                home#welcome
tasks       GET      /tasks(.:format)                 tasks#index
            POST     /tasks(.:format)                 tasks#create
new_task    GET      /tasks/new(.:format)             tasks#new
edit_task   GET      /tasks/:id/edit(.:format)        tasks#edit
task        GET      /tasks/:id(.:format)             tasks#show
            PATCH    /tasks/:id(.:format)             tasks#update
            DELETE   /tasks/:id(.:format)             tasks#destroy
```

Para gerar o link para edição de uma tarefa na *view* `home` utilizamos o método `link_to` com o texto do link e a rota desejada, neste caso `edit_task_path`, passando o objeto `task`. O Rails automaticamente identifica o `id` da tarefa para montar a URL correta.

```
<h1>Boas-vindas à Task List!</h1>
<h3>Sua lista de tarefas:</h3>

<% @tasks.each do |task| %>
  <%= task.description %>
  <!-- caminho para editar esse objeto especifico -->
  <%= link_to 'Editar', edit_task_path(task) %>
<% end %>
```

Quando clicarmos no botão, recebemos um erro de *action* não existente, então precisamos criá-la no *controller*:

```
class TasksController < ApplicationController
  def edit
  end
end
```

Mas editar o quê? Não é a edição da tarefa que clicamos? O link é gerado com o `id` específico da tarefa. Quando você clicou, deve ter visto no endereço do navegador algo como `'localhost:3000/tasks/1/edit'`, mas ao clicar... erro!

Isso acontece porque não temos a *view* `edit.html.erb`. Não é viável criar uma *view* para cada nova tarefa cadastrada, então temos que construir uma *view* única para editar qualquer tarefa. Para isso vamos criar uma variável `@task` que recebe o objeto vindo do banco de dados a partir do `params[:id]`.

```
class TasksController < ApplicationController
  def edit
    # Novamente utilizamos o params, dando uma olhada no routes
    # vemos o nome do parâmetro na rota de edit.
    @task = Task.find(params[:id])
  end
end
```

A *view* da edição é idêntica à *view* de `new` já que podemos editar a tarefa usando os mesmos atributos, mas vamos usar `@task` para alimentar o nosso formulário. Então crie a *view* `views/tasks/edit.html.erb`:

```
<%= form_for @task do |f| %>
  <%= f.label :description, 'Descrição' %>
  <%= f.text_field :description %>
  <%= f.label :status, 'Status' %>
  <%= f.check_box :status %>
  <%= f.submit 'Editar' %>
<% end %>
```

Rode a aplicação e tente usar o formulário. Você recebeu um erro dizendo que a *action* `update` não existe? Então vamos criá-la.

```
class TasksController < ApplicationController
  def update
  end
end
```

Agora temos que achar a tarefa que queremos editar e, através dos valores enviados pelo formulário, editar essa tarefa. Como já criamos uma tarefa, você já sabe receber parâmetros do formulário de maneira segura, só é necessário amarrar tudo.

Pense que cada *action* trabalha de forma independente, ou seja, mesmo tendo procurado nossa tarefa em `edit` temos que buscá-la novamente no `update`, pois aquela busca anterior se perdeu ao sair da *action*.

Editar é o mais simples, já que o *Active Record* nos dá um método chamado `update` para fazer a edição dos atributos. Depois podemos redirecionar para o `root_path`.

```
class TasksController < ApplicationController
  def update
    @task = Task.find(params[:id])
    @task.update(params.require(:task).permit(:description, :status))
    redirect_to root_path
  end
end
```

Se você atualizar a aplicação e tentar editar a tarefa: Taram!!! Tarefa editada com sucesso!

Apagando uma tarefa

Nós já criamos, mostramos e atualizamos tarefas, o que abrange quase todo o CRUD (*Create*, *Read*, *Update*, *Delete*), faltando apenas deletarmos uma tarefa!

Ao editar uma tarefa, vimos como criar um link na *view* para sua rota específica. Podemos fazer o mesmo para deletar, mas usando o verbo DELETE do HTTP.

Por padrão, o `link_to` é sempre um GET, mas podemos mudar esse comportamento utilizando o parâmetro `method`.

```
<h1>Boas-vindas à Task List!</h1>
<h3>Sua lista de tarefas:</h3>

<% @tasks.each do |task| %>
  <%= task.description %>
  <%= link_to 'Editar', edit_task_path(task) %>
  <%= link_to 'Apagar', task_path(task), method: :delete %>
<% end %>
```

Se você clicar no botão Apagar vai verificar que recebemos um erro por não existir a *action* `destroy` no nosso *controller*. Ela é a *action* padrão para destruir objetos no Rails. O método para deletar objetos no banco do Rails também se chama *destroy*.

```
class TasksController < ApplicationController
  def destroy
      task = Task.find(params[:id]) # não precisamos de @ pois não
temos view que usará esse objeto
    task.destroy #destrói a tarefa encontrada
    redirect_to root_path
  end
end
```

Desta forma, destruímos uma tarefa no Rails. Mas você já deve ter notado que é bem fácil clicar no botão sem querer e apagar uma tarefa por engano. Seria interessante pedir uma confirmação do usuário ao apagar uma tarefa. Podemos fazer isso no `link_to` usando o `confirm`.

```
<h1>Boas-vindas à Task List!</h1>
<h3>Sua lista de tarefas:</h3>

<% @tasks.each do |task| %>
  <%= task.description %>
  <%= link_to 'Editar', edit_task_path(task) %>
  <%= link_to 'Apagar', task_path(task), method: :delete,
      data: { confirm: 'Quer apagar esta tarefa?' } %>
<% end %>
```

Assim, apagamos nossas tarefas utilizando o Rails e terminamos todas as dinâmicas do CRUD!

Pra onde vão esses trilhos

Concluímos nosso primeiro conteúdo de Ruby on Rails. O *framework* já possui mais de 10 anos no mercado e alcançou destaque pela sua facilidade de uso e constante atualização. Em breve teremos mais conteúdos sobre outros componentes e funcionalidades do Rails. :)

Esse material está em constante evolução e sua opinião é muito importante. Se tiver sugestões ou dúvidas que gostaria de nos enviar, entre em contato pelo nosso e-mail: treinadev@campuscode.com.br.

Versão	Data de atualização
1.0.2	11/01/2022

treinadev

é um programa gratuito de
formação de devs da Campus Code

Apoio:

VINDI

RBASE

PORTAL solar

smart fit

konduto

CAMPUS CODE

Telefone/Whatsapp: [11] 2369-3476