

## CAPÍTULO 9

# Controllers e Views

*"A Inteligência é quase inútil para quem não tem mais nada"*  
— Carrel, Alexis

Nesse capítulo, você vai aprender o que são controllers e como utilizá-los para o benefício do seu projeto, além de aprender a trabalhar com a camada visual de sua aplicação.

## 9.1 – O "V" E O "C" DO MVC

O "V" de MVC representa a parte de **view** (visualização) da nossa aplicação, sendo ela quem tem contato com o usuário, recebe as entradas e mostra qualquer tipo de saída.

Há diversas maneiras de controlar as views, sendo a mais comum delas feita através dos arquivos HTML.ERB, ou eRuby (Embedded Ruby), páginas HTML que podem receber trechos de código em Ruby.

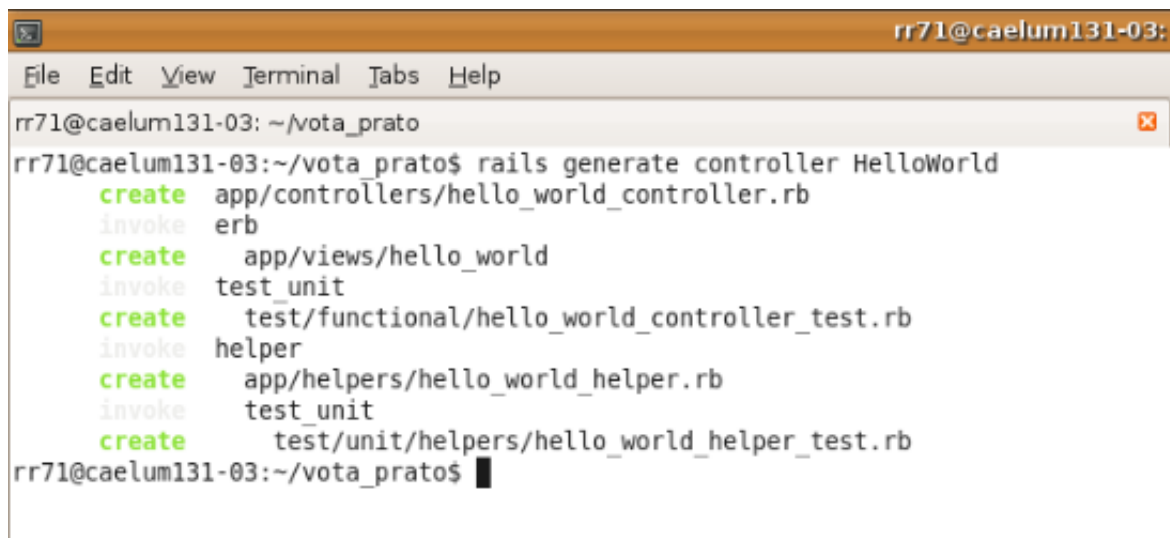
**Controllers** são classes que recebem uma ação de uma View e executam algum tipo de lógica ligada a um ou mais modelos. Em Rails esses controllers estendem a classe ApplicationController.

As urls do servidor são mapeadas da seguinte maneira: **/controller/action/id**. Onde "controller" representa uma classe controladora e "action" representa um método do mesmo. "id" é um parâmetro qualquer (opcional).

## 9.2 – HELLO WORLD

Antes de tudo criaremos um controller que mostrará um *"Hello World"* para entender melhor como funciona essa ideia do mapeamento de urls.

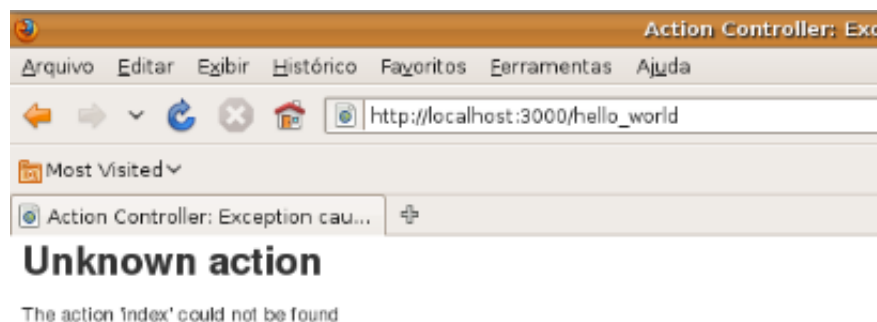
Vamos usar o generator do rails para criar um novo controller que se chamará *"HelloWorld"*. Veja que o Rails não gera apenas o Controller, mas também outros arquivos.



```
rr71@caelum131-03: ~/vota_prato
rr71@caelum131-03:~/vota_prato$ rails generate controller HelloWorld
create  app/controllers/hello_world_controller.rb
invoke  erb
create  app/views/hello_world
invoke  test_unit
create  test/functional/hello_world_controller_test.rb
invoke  helper
create  app/helpers/hello_world_helper.rb
invoke  test_unit
create  test/unit/helpers/hello_world_helper_test.rb
rr71@caelum131-03:~/vota_prato$
```

Apos habilitar o rota padrão do rails tente acessar a página [http://localhost:3000/hello\\_world](http://localhost:3000/hello_world)

Na URL acima passamos apenas o controller sem nenhuma action, por isso recebemos uma mensagem de erro.



Além de não dizer qual a action na URL, não escrevemos nenhuma action no controller.

Criaremos um método chamado `hello` no qual escreveremos na saída do cliente a frase *"Hello World!"*. **Cada método criado no controller é uma action**, que pode ser acessada através de um browser.

Para escrever na saída, o Rails oferece o comando `render`, que recebe uma opção chamada `"text"` (String). Tudo aquilo que for passado por esta chave será recebido no browser do cliente.

## Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Conheça os títulos e a nova proposta, você vai gostar.

[Casa do Código, livros para o programador.](#)

### 9.3 – EXERCÍCIOS: CRIANDO O CONTROLADOR

1. Vamos criar um controller para nossos restaurantes:

- a. Abra o Terminal.
- b. Execute **rails generate controller restaurantes**

2. Mais tarde esse controller irá mediar todas as interações relacionadas à restaurantes. Agora ele irá simplesmente renderizar um HTML com o texto "Estou no controller de restaurantes!" envolvido por um parágrafo.

- a. Crie um arquivo HTML para o index: **app/views/restaurantes/index.html.erb**
- b. Como conteúdo do arquivo digite:

```
<p>
  Estou no controller de Restaurantes!
</p>
```

3. a. Abra seu novo controller (**app/controllers/restaurantes\_controller.rb**)
- b. Inclua a action **index**:

```
def index
  render "index"
end
```

4. Vamos habilitar as rotas padrões para restaurantes:

- a. Abra o arquivo **config/routes.rb**.

b. Dentro do bloco de código adicione a linha:

```
resources :restaurantes
```

5. a. O último passo antes de testar no browser é iniciar o server.

b. Execute no Terminal: **rails server**

c. Confira o link <http://localhost:3000/restaurantes>.

## 9.4 – TRABALHANDO COM A VIEW: O ERB

### ERB

ERB é uma implementação de eRuby que já acompanha a linguagem Ruby. Seu funcionamento é similar ao dos arquivos JSP/ASP: arquivos html com injeções de código. A ideia é que o HTML serve como um template, e outros elementos são dinamicamente inseridos em tempo de renderização.

### sintaxe básica

Para uma página aceitar código Ruby, ela deve estar entre "<%" e "%>". Há uma variação deste operador, o "<%= ", que não só executa códigos Ruby, mas também imprime o resultado na página HTML.

Logo, o seguinte código ERB:

```
<html>
  <body>
    <p>Meu nome é: <%= "João" %></p>
    <p>Não vai imprimir nada ao lado: <% "Não será impresso" %></p>
  </body>
</html>
```

Irá resultar em:

```
<html>
  <body>
    <p>Meu nome é: João</p>
    <p>Não vai imprimir nada ao lado: </p>
  </body>
</html>
```

### if, else e blocos

O operador "<%" é muito útil quando precisamos que um pedaço de HTML seja adicionado com uma condição. Por exemplo:

```
<body>
  <% if nomes.empty? %>
    <div class="popup">
      Nenhum nome
    </div>
  <% else %>
    <div class="listagem">
      <%= nomes %>
    </div>
  <% end %>
</body>
```

Caso a variável nomes seja igual à [], o resultado será:

```
<body>
  <div class="popup">
    Nenhum nome
  </div>
</body>
```

Por outro lado, se nomes for igual à ["João", "Maria"] o resultado será:

```
<body>
  <div class="listagem">
    ["João", "Maria"]
  </div>
</body>
```

Podemos ainda iterar pelos nomes imprimindo-os em uma lista, basta mudar o ERB para:

```
<body>
  <% if nomes.empty? %>
    <div class="popup">
      Nenhum nome
    </div>
  <% else %>
    <ul class="listagem">
      <% nomes.each do |nome| %>
        <li><%= nome %></li>
      <% end %>
    </ul>
  <% end %>
</body>
```

Para nomes igual à ["João", "Maria"] o resultado seria:

```
<body>
  <ul class="listagem">
    <li><%= João %></li>
```

```
<li><%= Maria %></li>
</ul>
</body>
```

## Do controller para view

É importante notar que todos os atributos de instância (`@variavel`) de um controlador estão disponíveis em sua view. Além disso, ela deve ter o mesmo nome da action, e estar na pasta com o nome do controlador o que significa que a view da nossa action `index` do controlador `restaurantes_controller.rb` deve estar em: **`app/views/restaurantes/index.html.erb`**.

## 9.5 – ENTENDENDO MELHOR O CRUD

Agora, queremos ser capazes de criar, exibir, editar e remover restaurantes. Como fazer?

Primeiro, temos de criar um controller para nosso restaurante. Pela view Generators, vamos criar um controller para restaurante. Rails, por padrão, utiliza-se de sete actions "CRUD". São eles:

- `index`: exibe todos os items
- `show`: exibe um item específico
- `new`: formulário para a criação de um novo item
- `create`: cria um novo item
- `edit`: formulário para edição de um item
- `update`: atualiza um item existente
- `destroy`: remove um item existente

### Agora é a melhor hora de aprender algo novo



Se você gosta de estudar essa apostila aberta da Caelum, certamente vai gostar dos novos **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum.

[Conheça a Alura.](#)

## 9.6 – A ACTION INDEX

Desejamos listar todos os restaurantes do nosso Banco de Dados, e portanto criaremos a action index.

Assim como no console buscamos todos os restaurantes do banco de dados com o comando find, também podemos fazê-lo em controllers (que poderão ser acessados pelas nossas views, como veremos mais adiante).

Basta agora passar o resultado da busca para uma variável:

```
def index
  @restaurantes = Restaurante.order :nome
end
```

Com a action pronta, precisamos criar a view que irá utilizar a variável @restaurantes para gerar o HTML com a listagem de restaurantes:

```
<table>
  <tr>
    <th>Nome</th>
    <th>Endereço</th>
    <th>Especialidade</th>
  </tr>
  <% @restaurantes.each do |restaurante| %>
    <tr>
      <td><%= restaurante.nome %></td>
      <td><%= restaurante.endereco %></td>
      <td><%= restaurante.especialidade %></td>
    </tr>
  <% end %>
</table>
```

De acordo com a convenção, esta view deve estar no arquivo: **app/views/restaurantes/index.html.erb**.

Agora, basta acessar nossa nova página através da URL:  
<http://localhost:3000/restaurantes/>.

## 9.7 – EXERCÍCIOS: LISTAGEM DE RESTAURANTES

Vamos agora criar uma forma do usuário visualizar uma listagem com todos os restaurantes:

1. Gere um controller para o modelo restaurante:

- a. Vá ao Terminal;
- b. Execute **rails generate controller restaurantes**

2. a. Abra o seu controller de Restaurantes  
(**app/controllers/restaurantes\_controllers.rb**)

- b. Adicione a action index:

```
def index
  @restaurantes = Restaurante.order :nome
end
```

3. a. Vamos criar também o arquivo ERB  
(**app/views/restaurantes/index.html.erb**) que servirá de base para a página HTML.

```
<h1>Listagem de Restaurantes</h1>

<table>
  <tr>
    <th>ID</th>
    <th>Nome</th>
    <th>Endereço</th>
    <th>Especialidade</th>
  </tr>

  <% @restaurantes.each do |restaurante| %>
    <tr>
      <td><%= restaurante.id %></td>
      <td><%= restaurante.nome %></td>
      <td><%= restaurante.endereco %></td>
      <td><%= restaurante.especialidade %></td>
    </tr>
  <% end %>
</table>
```

4. a. Teste agora entrando em: <http://localhost:3000/restaurantes> (Não esqueça de iniciar o servidor)

b. É possível que não tenhamos nenhum restaurante em nossa base de dados, nesse caso, vamos usar um pouco do poder de ruby para rapidamente criar alguns restaurantes via console (**rails console**):

```
especialidades = %w{massas japonês vegetariano}
50.times do |i|
  Restaurante.create!(
    nome: "Restaurante #{i}",
    endereco: "Rua #{i} de setembro",
    especialidade: especialidades.sample
  )
end
```



## 9.8 – HELPER

Helpers são módulos que disponibilizam métodos para serem usados em nossas views. Eles provêm atalhos para os códigos mais usados e nos poupam de escrever muito código. O propósito de um helper é simplificar suas views.

### Helpers padrões

O Rails já nos disponibiliza uma série de helpers padrões, por exemplo, se quisermos criar um link, podemos usar o helper `link_to`:

```
<%= link_to "Restaurantes", controller: "restaurantes", action: "index" %>
```

Abaixo, uma lista com alguns helpers padrões:

- `link_to` (âncora)
- `image_tag` (img)
- `favicon_link_tag` (link para favicon)
- `stylesheet_link_tag` (link para CSS)
- `javascript_include_tag` (script)

#### Helper Method

Existe também o chamado `helper_method`, que permite que um método de seu controlador vire um Helper e esteja disponível na view para ser chamado. Exemplo:

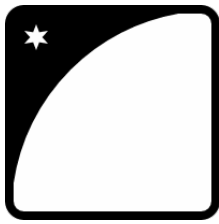
```
class TesteController < ApplicationController
  helper_method :teste

  def teste
    "algum conteudo dinâmico"
  end
end
```

E em alguma das views deste controlador:

```
<%= teste %>
```

Você pode também fazer o curso RR-71 dessa apostila na Caelum



Querendo aprender ainda mais sobre a linguagem Ruby e o framework Ruby on Rails? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?

A Caelum oferece o **curso RR-71** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas

incompany.

[Consulte as vantagens do curso \*Desenv. Ágil para Web com Ruby on Rails\*.](#)

## 9.9 – A ACTION SHOW

Para exibir um restaurante específico, precisamos saber qual restaurante buscar. Pela convenção, poderíamos visualizar o restaurante de id 1 acessando a URL: <http://localhost:3000/restaurantes/1>. Logo, o id do restaurante que buscamos deve ser passado na URL.

Ou seja, o caminho para a action show é: /restaurantes/:id. A action show receberá um parâmetro :id ao ser evocada. E esse id será utilizado na busca pelo restaurante:

```
def show
  @restaurante = Restaurante.find(params[:id])
end
```

Após criar a action, devemos criar a view que irá descrever um restaurante:

```
<h1>Exibindo Restaurante</h1>
```

```
Nome: <%= @restaurante.nome %>
```

```
Endereço: <%= @restaurante.endereco %>
```

```
Especialidade: <%= @restaurante.especialidade %>
```

Agora, basta acessar nossa nova página através da URL:

<http://localhost:3000/restaurantes/id-do-restaurante>.

## 9.10 – EXERCÍCIOS: VISUALIZANDO UM RESTAURANTE

1. a. Abra o seu controller de Restaurantes  
(app/controllers/restaurantes\_controllers.rb)

b. Adicione a action show:

```
def show
  @restaurante = Restaurante.find(params[:id])
end
```

2. Vamos criar também o arquivo ERB (**app/views/restaurantes/show.html.erb**).

```
<h1>Exibindo Restaurante</h1>

<p>
  <b>Nome: </b>
  <%= @restaurante.nome %>
</p>

<p>
  <b>Endereço: </b>
  <%= @restaurante.endereco %>
</p>

<p>
  <b>Especialidade: </b>
  <%= @restaurante.especialidade %>
</p>
```

3. Vamos agora usar o helper de âncora (**link\_to**) para criar o link que tornará possível visualizar um restaurante a partir da listagem de restaurantes.

a. Abra a view de index (**app/views/restaurantes/index.html.erb**).

b. Vamos adicionar um link **Mostrar** ao lado de cada restaurante, que irá direcionar o usuário para o show do restaurante. Para isso, vamos substituir esse código:

```
<% @restaurantes.each do |restaurante| %>
  <tr>
    <td><%= restaurante.nome %></td>
    <td><%= restaurante.endereco %></td>
    <td><%= restaurante.especialidade %></td>
  </tr>
<% end %>
```

Por esse:

```
<% @restaurantes.each do |restaurante| %>
  <tr>
    <td><%= restaurante.nome %></td>
    <td><%= restaurante.endereco %></td>
    <td><%= restaurante.especialidade %></td>
    <td>
      <%= link_to 'Mostrar', action: 'show', id: restaurante %>
    </td>
  </tr>
<% end %>
```

4. Teste clicando no link **Mostrar** de algum restaurante na listagem de restaurantes (<http://localhost:3000/restaurantes>).

## 9.11 – A ACTION DESTROY

Para remover um restaurante, o usuário enviará uma requisição à nossa action destroy passando no id da url o id do restaurante a ser excluído:

```
def destroy
  @restaurante = Restaurante.find(params[:id])
  @restaurante.destroy
end
```

Observe que a action destroy não tem intenção de mostrar nenhuma informação para o usuário. Por isso, após finalizar seu trabalho, ela irá chamar a index. Abaixo iremos entender melhor o mecanismo que iremos utilizar para implementar isso.

### Tire suas dúvidas no novo GUJ Respostas



O GUJ é um dos principais fóruns brasileiros de computação e o maior em português sobre Java. A nova versão do GUJ é baseada em uma ferramenta de *perguntas e respostas* (QA) e tem uma comunidade muito forte. São mais de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

[Faça sua pergunta.](#)

## 9.12 – REDIRECIONAMENTO DE ACTION

Dentro de uma action, podemos redirecionar a saída para uma outra action.

Por exemplo, se tivermos um controller qualquer com duas actions podemos utilizar o método `redirect_to` para que o usuário seja redirecionado para a **action1** no momento que ele acessar a **action2**:

```
class ExemploController < ApplicationController
  def action1
    render text: "ACTION 1!!!"
  end

  def action2
```

```
    redirect_to action: 'action1'  
  end  
end
```

Dessa forma, é possível que uma action exerça sua responsabilidade e depois delegue o final do processo para alguma outra action.

Imagine no nosso controller de restaurantes, por exemplo. Criamos uma action para apagar um restaurante, porém o que acontecerá após o restaurante ser deletado?

Faz sentido que o usuário volte para a página de index, para obtermos esse comportamento utilizaremos do `redirect_to`:

```
def index  
  @restaurantes = Restaurante.all  
end  
  
def destroy  
  # código que deleta o restaurante  
  
  redirect_to action: 'index'  
end
```

### Redirecionamento no servidor e no cliente

O redirecionamento no servidor é conhecido como *forward* e a requisição é apenas repassada a um outro recurso (página, controlador) que fica responsável em tratar a requisição.

Há uma outra forma que é o **redirecionamento no cliente** (*redirect*). Nesta modalidade, o servidor responde a requisição original com um **pedido de redirecionamento**, fazendo com que o navegador dispare uma nova requisição para o novo endereço. Neste caso, a barra de endereços do navegador muda.

## 9.13 – EXERCÍCIOS: DELETANDO UM RESTAURANTE

1. a. Abra o seu controller de Restaurantes  
(`app/controllers/restaurantes_controllers.rb`)
- b. Adicione a action destroy:

```
def destroy
  @restaurante = Restaurante.find(params[:id])
  @restaurante.destroy

  redirect_to(action: "index")
end
```

2. a. Abra a view de index (**app/views/restaurantes/index.html.erb**).

b. Vamos adicionar um link **Deletar** ao lado de cada restaurante, que irá invocar a action destroy:

```
<!-- Deve ficar abaixo do td que criamos para o link "Mostrar" -->
<td>
  <%= link_to 'Deletar', {action: 'destroy', id: restaurante},
                                {method: "delete"} %>
</td>
```

3. Teste clicando no link **Deletar** de algum restaurante na listagem de restaurantes (<http://localhost:3000/restaurantes>).

## 9.14 – HELPERS PARA FORMULÁRIOS

Quando trabalhamos com formulários, usamos os chamados **FormHelpers**, que são módulos especialmente projetados para nos ajudar nessa tarefa. Todo **FormHelper** está associado a um ActiveRecord. Existem também os **FormTagHelpers**, que contém um *\_tag* em seu nome. **FormTagHelpers**, não estão necessariamente associados a ActiveRecord algum.

Abaixo, uma lista dos FormHelpers disponíveis:

- check\_box
- fields\_for
- file\_field
- form\_for
- hidden\_field
- label
- password\_field
- radio\_button

- text\_area
- text\_field

E uma lista dos FormTagHelpers:

- check\_box\_tag
- field\_set\_tag
- file\_field\_tag
- form\_tag
- hidden\_field\_tag
- image\_submit\_tag
- password\_field\_tag
- radio\_button\_tag
- select\_tag
- submit\_tag
- text\_area\_tag
- text\_field\_tag

Agora, podemos escrever nossa view:

```
<%= form_tag :action => 'create' do %>
  Nome: <%= text_field :restaurante, :nome %>
  Endereço: <%= text_field :restaurante, :endereco %>
  Especialidade: <%= text_field :restaurante, :especialidade %>
  <%= submit_tag 'Criar' %>
<% end %>
```

Este ERB irá renderizar um HTML parecido com:

```
<form action="/restaurantes" method="POST">
  Nome: <input type="text" name="restaurante[nome]">
  Endereço: <input type="text" name="restaurante[endereco]">
  Especialidade: <input type="text" name="restaurante[especialidade]">
  <input type="submit">
<% end %>
```

Repare que como utilizamos o form\_tag, que não está associado a nenhum ActiveRecord, nosso outro Helper **text\_field** não sabe qual o ActiveRecord que

estamos trabalhando, sendo necessário passar para cada um deles o parâmetro :restaurante, informando-o.

Podemos reescrever mais uma vez utilizando o FormHelper form\_for, que está associado a um ActiveRecord:

```
<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %>
  Endereço: <%= f.text_field :endereco %>
  Especialidade: <%= f.text_field :especialidade %>
  <%= f.submit %>
<% end %>
```

Repare agora que não foi preciso declarar o nome do nosso modelo para cada text\_field, uma vez que nosso Helper form\_for já está associado a ele.

### Nova editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não conhecem programação para revisar os livros tecnicamente a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

## 9.15 – A ACTION NEW

Para incluir um novo restaurante, precisamos primeiro retornar ao browser um restaurante novo, sem informação alguma. Vamos criar nossa action new

```
def new
  @restaurante = Restaurante.new
end
```

## 9.16 – EXERCÍCIOS: PÁGINA PARA CRIAÇÃO DE UM NOVO RESTAURANTE

1. a. Abra o seu controller de Restaurantes  
(app/controllers/restaurantes\_controllers.rb)



b. Adicione a action new:

```
def new
  @restaurante = Restaurante.new
end
```

2. a. Vamos criar também o arquivo ERB  
(**app/views/restaurantes/new.html.erb**).

```
<h1>Adicionando Restaurante</h1>

<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= f.submit %>
<% end %>
```

3. a. Abra a view de index (**app/views/restaurantes/index.html.erb**).

b. Vamos adicionar um link **Novo** para direcionar o usuário para a action new. Diferente dos outros links que criamos, esse deverá ser inserido abaixo da lista de restaurantes:

```
<!-- lista de restaurantes -->

<br/>
<%= link_to 'Novo', action: 'new' %>
```

4. Teste clicando no link **Novo** na listagem de restaurantes  
(<http://localhost:3000/restaurantes>).

## 9.17 – RECEBENDO UM PARÂMETRO POR UM FORMULÁRIO

Não são raros os momentos onde precisamos que o usuário digite informações que utilizaremos no lado do servidor. Para isso utilizamos formulários como:

```
<form action='/restaurantes'>
  Nome: <input type='text' name='nome' />
  <input type='submit' value='Create' />
</form>
```

Porém, como teremos acesso ao texto digitado pelo usuário? Toda informação introduzida em um formulário, é passada para o controller como um parâmetro. Logo, para receber este valor nome no controlador, basta usar o hash params. (Repare que agora usamos outra action, create, para buscar os dados do formulário apresentado anteriormente):

```
class RestaurantesController < ApplicationController
  def create
    nome = params[:nome]
  end
end
```

Porém essa não é a action create dos restaurantes, precisamos de algo mais completo.

## Recebendo parâmetros ao utilizar o helper form\_for

Diferente do exemplo anterior, nossa view `new.html.erb` utiliza o helper `form_for` para representar o formulário de um restaurante, da seguinte maneira:

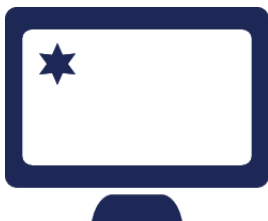
```
<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= f.submit %>
<% end %>
```

Ao utilizar o helper `form_for` o Rails irá agrupar os parâmetros desse formulário na chave `:restaurante`. Logo, é possível imprimir cada um dos parâmetros da seguinte maneira:

```
def create
  puts params[:restaurante][:nome]
  puts params[:restaurante][:endereco]
  puts params[:restaurante][:especialidade]
end
```

Porém, imprimir os parâmetros não é suficiente para persistir um restaurante.

### Já conhece os cursos online Alura?



A **Alura** oferece dezenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Java, Ruby, Web, Mobile, .NET e outros, com uma **assinatura** que dá acesso a todos os cursos.

[Conheça os cursos online Alura.](#)

Para criarmos um restaurante, podemos fazer a atribuição dos parâmetros na forma de um *hash* conforme mostrado anteriormente. Porém, para que isto funcione no Controller, precisamos especificar exatamente quais são as propriedades que podem ter seu valor atribuído a partir do *hash*.

É preciso dizer isso explicitamente através da invocação dos métodos `require` (para solicitar o objeto) e `permit` (para informar as propriedades), conforme abaixo:

```
Restaurante.new params.require(:restaurante).permit(:nome, :endereco, :especialidade)
```

Não faz sentido que criemos um novo restaurante sem persisti-lo. Vamos criar uma variável e utilizá-la em seguida para salvá-lo:

```
def create
  @restaurante = Restaurante.new params.require(:restaurante).permit(:nome, :endereco, :especialidade)
  @restaurante.save
end
```

Como estes parâmetros precisarão ser liberados em outras actions, é uma boa prática criarmos um método para agrupar esta funcionalidade. Se algum dia precisarmos incluir algum novo parâmetro, não precisaremos modificar todas as actions e sim apenas este método.

```
def restaurante_params
  params.require(:restaurante).permit(:nome, :endereco, :especialidade)
end
```

Com a criação do método `restaurante_params` nossa action `create` ficará da seguinte forma:

```
def create
  @restaurante = Restaurante.new restaurante_params
  @restaurante.save
end
```

### mass assignment

Essa ideia de precisar especificar quais campos podem ser atribuídos através de um *hash* é referida por muitos como "mass assignment whitelist". A ideia é que o programador deve decidir quais propriedades de um objeto podem ser atribuídas diretamente.

Até a versão do Rails 3.2, esta liberação de parâmetros era efetuada diretamente na classe de modelo através da invocação ao método `attr_accessible`.

Nossas classes de modelo ficariam da seguinte forma, caso estivéssemos utilizando o Rails 3.2:

```
class Restaurante < ActiveRecord::Base
  attr_accessible :nome, :endereco, :especialidade
end
```

O problema disso é que, se o programador liberasse um propriedade para usar em somente uma action do controller, ele deveria liberar para todas e fazer o controle manualmente. Se por algum descuido ele esquecesse de verificar quais propriedades estavam abertas a atribuição, algumas falhas de segurança podem ocorrer.

Nenhuma propriedade pode ser atribuída diretamente a partir dos valores de um *hash*, caso queira isso, é preciso especificar através da invocação dos métodos `require` e `permit`

Você pode ler mais a respeito nesse post:

<http://blog.caelum.com.br/seguranca-de-sua-aplicacao-e-os-frameworks-ataque-ao-github/>

## 9.19 – EXERCÍCIOS: PERSISTINDO UM RESTAURANTE

1. a. Abra o seu controller de Restaurantes

(`app/controllers/restaurantes_controllers.rb`)

b. Vamos criar o método que efetuará a solicitação para atribuição das propriedades de um restaurante:

```
def restaurante_params
  params.require(:restaurante).permit(:nome, :endereco, :especialidade)
end
```

c. Adicione a action create:

```
def create
  @restaurante = Restaurante.new(restaurante_params)
  @restaurante.save
  redirect_to(action: "show", id: @restaurante)
```

end

## 9.20 – A ACTION EDIT

Para editar um restaurante, devemos retornar ao browser o restaurante que se quer editar, para só depois salvar as alterações feitas:

```
def edit
  @restaurante = Restaurante.find(params[:id])
end
```

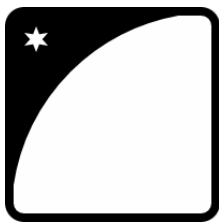
A view de **edit.html.erb** terá o mesmo formulário utilizado na view **new.html.erb**:

```
<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= f.submit %>
<% end %>
```

Perceba que ao editar, é usual que o usuário possa ver as informações atuais do restaurante. Para isso, não precisaremos fazer nada pois o helper `form_for` irá se encarregar de restaurar os dados do restaurante no formulário.

Após implementar a action e a view, podemos acessar o formulário de edição acessando a URL: <http://localhost:3000/restaurantes/id-do-restaurante/edit>.

**Você não está nessa página a toa**



Você chegou aqui porque a Caelum é referência nacional em cursos de Java, Ruby, Agile, Mobile, Web e .NET.

Faça curso com **quem escreveu essa apostila**.

[Consulte as vantagens do curso \*Desenv. Ágil para Web com Ruby on Rails\*.](#)

## 9.21 – A ACTION UPDATE

Uma vez que o usuário tenha atualizado as informações do restaurante e deseje salvá-las, enviará uma requisição à nossa action update passando o id do

restaurante a ser editado na url, bem como os novos dados do restaurante.

A hash **params** terá uma estrutura parecida com:

```
{
  id: 1,
  restaurante: {
    nome: "Comidubom",
    endereco: "Rua da boa",
    especialidade: "massas"
  }
}
```

Como já criamos o método para liberar o acesso às propriedades anteriormente, podemos implementar nosso **update** da seguinte maneira:

```
def update
  @restaurante = Restaurante.find params[:id]
  @restaurante.update_attributes restaurante_params
end
```

## 9.22 - ATUALIZANDO UM RESTAURANTE

Vamos implementar as actions e views envolvidas na atualização de um restaurante. Ao final, note como o processo de atualização tem várias semelhanças com o de criação.

1. a. Abra o seu controller de Restaurantes  
(**app/controllers/restaurantes\_controllers.rb**)

b. Adicione a action edit:

```
def edit
  @restaurante = Restaurante.find params[:id]
end
```

2. a. Vamos criar também o arquivo ERB  
(**app/views/restaurantes/edit.html.erb**).

```
<h1>Editando Restaurante</h1>

<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= f.submit %>
<% end %>
```

3. a. Abra o seu controller de Restaurantes

(app/controllers/restaurantes\_controllers.rb)

b. Adicione a action update:

```
def update
  @restaurante = Restaurante.find(params[:id])
  @restaurante.update_attributes(restaurante_params)

  redirect_to action: "show", id: @restaurante
end
```

4. a. Abra a view de index (app/views/restaurantes/index.html.erb).

b. Vamos adicionar um link **Editar** ao lado de cada restaurante, que irá direcionar o usuário para a action edit:

```
<!-- Deve ficar abaixo do td que criamos para o link "Deletar" -->

<td>
  <%= link_to 'Editar', action: 'edit', id: restaurante %>
</td>
```

5. Teste clicando no link **Editar** na listagem de restaurantes

(<http://localhost:3000/restaurantes>).

## 9.23 – EXERCÍCIOS OPCIONAIS: LINKANDO MELHOR AS VIEWS

Até agora só criamos links na página que lista os restaurantes, como se fosse uma espécie de menu principal. Porém, na grande maioria dos casos, criamos links nas outras páginas também. Por exemplo, se o usuário estiver visualizando um restaurante, seria conveniente ele ter um link para poder voltar à listagem. Vamos criar esses links agora:

1. a. Abra o arquivo ERB da action new (app/views/restaurantes/new.html.erb).

b. Adicione no final do ERB o seguinte link:

```
<%= link_to 'Voltar', action: 'index' %>
```

2. a. Abra o arquivo ERB da action edit (app/views/restaurantes/edit.html.erb).

b. Adicione no final do ERB o seguinte link:

```
<%= link_to 'Voltar', action: 'index' %>
```

3. a. Abra o arquivo ERB da action show (app/views/restaurantes/show.html.erb).

b. Adicione no final do ERB os seguintes links:

```
<%= link_to 'Editar', action: 'edit', id: @restaurante %>  
<%= link_to 'Voltar', action: 'index' %>
```

### Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Conheça os títulos e a nova proposta, você vai gostar.

[Casa do Código, livros para o programador.](#)

## 9.24 – MOSTRANDO ERROS AO USUÁRIO

Desenvolvemos todas as actions e views necessárias para um CRUD de restaurante, porém não nos preocupamos de impedir que o usuário crie um restaurante inválido.

Felizmente, as validações que fizemos no modelo já impedem que restaurantes inválidos sejam salvos. Porém, se testarmos, veremos que ao tentarmos criar um restaurante inválido veremos uma página de erro.

Apesar de já estarmos protegidos contra o mau uso do sistema, nosso sistema responde de uma forma ofensiva, através de uma mensagem de erro.

### Usuário ter a oportunidade de tentar novamente

Para resolver esse problema, mudaremos nossa action **create** para renderizar novamente o formulário caso um dado inválido esteja errado, dessa forma o usuário terá a oportunidade de corrigir os dados e enviá-los novamente.

Inicialmente, poderíamos pensar em utilizar o método **redirect\_to**, dessa forma:

```
def create  
  @restaurante = Restaurante.new restaurante_params
```



```

if @restaurante.save
  redirect_to action: "show", id: @restaurante
else
  redirect_to action: "new"
end
end

```

Utilizando o **redirect\_to** o usuário iria ver novamente o formulário caso digitasse dados inválidos, porém o formulário iria aparecer vazio e o usuário seria obrigado à digitar todas as informações novamente, ao invés de corrigir o erro. (Faça o teste)

Para resolvermos isso, iremos utilizar o método **render** para fazer com que a action **create** reaproveite a view **app/views/restaurantes/new.html.erb**. Dessa forma:

```

def create
  @restaurante = Restaurante.new restaurante_params

  if @restaurante.save
    redirect_to action: "show", id: @restaurante
  else
    render action: "new"
  end
end

```

A utilização do método **render** irá resultar em um formulário com os dados do restaurante, por que a view **new.html.erb** irá utilizar a variável **@restaurante** que foi criada na action **create**.

## Avisar ao usuário os erros que foram cometidos

Apesar do usuário poder corrigir o erro, será difícil para ele enxergar sozinho que campos estão com erros.

Por isso, iremos mudar a view **new.html.erb** para mostrar os erros ao usuário. É importante lembrarmos o método **errors** que retorna um objeto que representa os erros do restaurante em questão. Iremos utilizar também o método **full\_messages** que retorna uma array com as mensagens de erro.

Na nossa view **new.html.erb** iremos criar uma **<ul>** e iterar pela array de mensagens encaixando elas criando uma **<li>** para cada mensagem:

```

<ul>
<% @restaurante.errors.full_messages.each do |msg| %>
  <li><%= msg %></li>
<% end %>
</ul>

```

Feito isso, ficará fácil para o usuário tomar conhecimentos dos erros cometidos e consertá-los caso necessário.

## 9.25 – EXERCÍCIOS: TRATAMENTO DE ERROS NA CRIAÇÃO

1. O usuário não pode salvar um restaurante inválido, porém não há nada que avise o usuário. Vamos alterar nossa página de forma que sejam mostrados os erros do restaurante logo acima do formulário.

a. Abra a view do formulário de criação de um restaurante (**app/views/restaurantes/new.html.erb**).

b. Adicione a lista de erros logo acima do formulário:

```
<ul>
<% @restaurante.errors.full_messages.each do |msg| %>
  <li><%= msg %></li>
<% end %>
</ul>

<%= form_for @restaurante do |f| %>
<!-- resto do HTML -->
```

2. Vamos preparar o controller de restaurantes para quando a criação de um restaurante falhar nas validações. Nesse caso, ele deve renderizar o formulário de criação (**new.html.erb**) a partir do restaurante com erros:

a. Abra o seu controller de Restaurantes (**app/controllers/restaurantes\_controllers.rb**)

b. Altere a action create para:

```
def create
  @restaurante = Restaurante.new(restaurante_params)
  if @restaurante.save
    redirect_to action: "show", id: @restaurante
  else
    render action: "new"
  end
end
```

## 9.26 – MENSAGENS DE ERRO NA ATUALIZAÇÃO

Um problema parecido com o que experimentamos no momento da criação de um novo restaurante ocorrerá no processo de atualização de um restaurante. Para

resolvê-lo, iremos alterar a action **update** para renderizar a view **edit.html.erb** caso o usuário tenha entrado com dados inválidos. Em seguida adicionaremos a listagem de erros na view **edit.html.erb** assim como fizemos na **new.html.erb**.

## As alterações necessárias

Assim como na action **create**, também utilizaremos o método **render** na action **update**:

```
def update
  @restaurante = Restaurante.find params[:id]

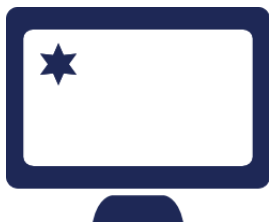
  if @restaurante.update_attributes(restaurante_params)
    redirect_to action: "show", id: @restaurante
  else
    render action: "edit"
  end
end
```

Para mostrar os erros ocorridos, assim como tivemos que implementar na view **new.html.erb**, precisaremos inserir a lista de erros na view **edit.html.erb**:

```
<ul>
<% @restaurante.errors.full_messages.each do |msg| %>
  <li><%= msg %></li>
<% end %>
</ul>

<%= form_for @restaurante do |f| %>
<!-- resto do HTML -->
```

### Agora é a melhor hora de aprender algo novo



Se você gosta de estudar essa apostila aberta da Caelum, certamente vai gostar dos novos **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum.

[Conheça a Alura.](#)

## 9.27 – EXERCÍCIOS: TRATAMENTO DE ERROS NA ATUALIZAÇÃO

Assim como no formulário e action de criação, temos que preparar nossa aplicação para o caso de uma atualização de restaurante falhar em uma validação:

1. a. Abra a view do formulário de edição de um restaurante (**app/views/restaurantes/edit.html.erb**).

b. Adicione a lista de erros logo acima do formulário:

```
<ul>
<% @restaurante.errors.full_messages.each do |msg| %>
  <li><%= msg %></li>
<% end %>
</ul>

<%= form_for @restaurante do |f| %>
<!-- resto do HTML -->
```

2. a. Abra o seu controller de Restaurantes (**app/controllers/restaurantes\_controllers.rb**)

b. Altere a action update para:

```
def update
  @restaurante = Restaurante.find(params[:id])

  if @restaurante.update_attributes(restaurante_params)
    redirect_to action: "show", id: @restaurante
  else
    render action: "edit"
  end
end
```

3. a. (Opcional) Quando o restaurante não tem erros, uma ul vazia é renderizada. Para evitar esse tipo de poluição, podemos envolver a **<ul>** com um if:

```
<% if @restaurante.errors.any? %>
  <!-- ul com erros aqui -->
<% end %>
```

b. (Opcional) Faça a mesma alteração do exercício acima na página **app/views/restaurantes/new.html.erb**.

## 9.28 – PARTIAL

Agora, suponha que eu queira exibir em cada página do restaurante um texto, por exemplo: "Controle de Restaurantes".

Poderíamos escrever esse texto manualmente, mas vamos aproveitar essa necessidade para conhecer um pouco sobre Partials.

Partials são fragmentos de *html.erb* que podem ser incluídas em uma view. Eles

permitem que você reutilize sua lógica de visualização.

Para criar um Partial, basta incluir um arquivo no seu diretório de views (**app/views/restaurantes**) com o seguinte nome: `_meupartial`. Repare que Partials devem obrigatoriamente começar com `_`.

Para utilizar um Partial em uma view, basta acrescentar a seguinte linha no ponto que deseja fazer a inclusão:

```
render partial: "meupartial"
```

## 9.29 – EXERCÍCIOS: REAPROVEITANDO FRAGMENTOS DE ERB

1. Vamos criar um partial para reaproveitar o formulário que se repete nos templates **new.html.erb** e **edit.html.erb**:

a. Crie o arquivo: **app/views/restaurantes/\_form.html.erb**

b. Coloque o seguinte conteúdo:

```
<ul>
<% @restaurante.errors.full_messages.each do |msg| %>
  <li><%= msg %></li>
<% end %>
</ul>

<%= form_for @restaurante do |f| %>
  Nome: <%= f.text_field :nome %><br/>
  Endereço: <%= f.text_field :endereco %><br/>
  Especialidade: <%= f.text_field :especialidade %><br/>
  <%= f.submit %>
<% end %>
```

2. a. Abra a view **new.html.erb**. (**app/views/restaurantes/new.html.erb**)

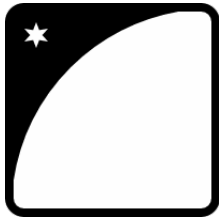
b. Substitua a lista de erros e o formulário por:

```
<%= render partial: "form" %>
```

3. Repita o processo acima com a view **edit.html.erb**.

4. Teste novamente as actions **new** e **edit** acessando <http://localhost:3000/restaurantes>.

Você pode também fazer o curso RR-71 dessa apostila na Caelum



Querendo aprender ainda mais sobre a linguagem Ruby e o framework Ruby on Rails? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?

A Caelum oferece o **curso RR-71** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas

incompany.

[Consulte as vantagens do curso \*Desenv. Ágil para Web com Ruby on Rails\*.](#)

## 9.30 – RESPONDENDO EM OUTROS FORMATOS COMO XML OU JSON

Um controlador pode ter diversos resultados. Em outras palavras, controladores podem responder de diversas maneiras, através do método `respond_to`:

```
class RestaurantesController < ApplicationController
  def index
    @restaurantes = Restaurante.all
    respond_to do |format|
      format.html {render "index"}
      format.xml {render xml: @restaurantes}
    end
  end
end
```

Dessa forma, conseguimos definir uma lógica de resposta diferente para cada formato.

Também é possível omitir o bloco de código:

```
class RestaurantesController < ApplicationController
  def index
    @restaurantes = Restaurante.all
    respond_to do |format|
      format.html
      format.xml {render xml: @restaurantes}
    end
  end
end
```

Por convenção, quando o bloco de código é omitido o Rails vai uma view que atenda ao seguinte padrão:

```
app/views/:controller/:action.:format.:handler
```

No caso de omitirmos para o formato HTML, o Rails irá renderizar a view: `app/views/restaurantes/index.html.erb`

## 9.31 – PARA SABER MAIS: OUTROS HANDLERS

O Rails já vem com suporte a outros *handlers* para geração de views. Além do ERB, podemos também usar o **Builder**.

O **Builder** é adequado quando a view a ser gerada é um arquivo XML, já que permite a criação de um xml usando sintaxe Ruby. Veja um exemplo:

```
# app/views/authors/show.xml.builder
xml.author do
  xml.name('Alexander Pope')
end
```

O xml resultante é:

```
<author>
  <name>Alexander Pope</name>
</author>
```

Outra alternativa muito popular para a geração das views é o **HAML**:

<http://haml.hamptoncatlin.com>

```
#content
  .left.column
    %h2 Welcome to our site!
    %p= print_information
  .right.column= render partial: "sidebar"
```

E o equivalente com ERB:

```
<div id='content'>
  <div class='left column'>
    <h2>Welcome to our site!</h2>
    <p>
      <%= print_information %>
    </p>
  </div>
  <div class="right column">
    <%= render partial: "sidebar" %>
  </div>
</div>
```

## 9.32 – EXERCÍCIOS: DISPONIBILIZANDO RESTAURANTES COMO XML E

# JSON

1. Vamos deixar explícito que a action **index** do controller de restaurantes responder com html:

a. Abra o controller de restaurantes.

(**app/controllers/restaurantes\_controller.rb**)

b. Altere a action **index** para:

```
def index
  @restaurantes = Restaurante.order :nome

  respond_to do |format|
    format.html
  end
end
```

2. Agora que temos a forma explícita de declarar formatos de resposta, será simples capacitar a action **index** para responder com **XML** e **JSON**:

a. Ainda no controller de restaurantes.

(**app/controllers/restaurantes\_controller.rb**)

b. Altere a action **index** para:

```
def index
  @restaurantes = Restaurante.order :nome

  respond_to do |format|
    format.html
    format.xml {render xml: @restaurantes}
    format.json {render json: @restaurantes}
  end
end
```

3. Teste os novos formatos acessando as urls:

a. <http://localhost:3000/restaurantes.xml>

b. <http://localhost:3000/restaurantes.json>

## Tire suas dúvidas no novo GUJ Respostas

O GUJ é um dos principais fóruns brasileiros de computação e o maior em português sobre Java. A nova versão do GUJ é baseada em uma ferramenta de *perguntas e respostas* (QA) e tem uma comunidade muito forte. São mais





de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

[Faça sua pergunta.](#)

## 9.33 – EXERCÍCIOS OPCIONAIS: OUTRAS ACTIONS RESPONDENDO XML E JSON

Já ensinamos o controller de restaurantes a responder um conjunto de restaurantes. Vamos implementar para que ele conseguir responder com um JSON ou XML representando apenas um restaurante.

1. a. Abra o controller de restaurantes.  
(`app/controllers/restaurantes_controller.rb`)

b. Altere a action **show** para:

```
def show
  @restaurante = Restaurante.find params[:id]

  respond_to do |format|
    format.html
    format.json {render json: @restaurante}
    format.xml {render xml: @restaurante}
  end
end
```

2. Para testar, procure pelo id de algum restaurante e acesse:  
<http://localhost:3000/restaurantes/<id>.json>.

## 9.34 – FILTROS

O módulo ActionController::Filters define formas de executar código antes e depois de todas as actions.

Para executar código antes das actions:

```
class ClientesController < ApplicationController
  before_action :verifica_login

  private
  def verifica_login
    redirect_to controller: 'login' unless usuario_logado?
  end
end
```

De forma análoga, podemos executar código no fim do tratamento da requisição:

```
class ClientesController < ApplicationController
  after_action :avisa_termino

  private
  def avisa_termino
    logger.info "Action #{params[:action]} terminada"
  end
end
```

Por fim, o mais poderoso de todos, que permite execução de código tanto antes, quanto depois da action a ser executada:

```
class ClientesController < ApplicationController
  around_action :envolvendo_actions

  private
  def envolvendo_actions
    logger.info "Antes de #{params[:action]}: #{Time.now}"
    yield
    logger.info "Depois de #{params[:action]}: #{Time.now}"
  end
end
```

Os filtros podem também ser definidos diretamente na declaração, através de blocos:

```
class ClientesController < ApplicationController
  around_action do |controller, action|
    logger.info "#{controller} antes: #{Time.now}"
    action.call
    logger.info "#{controller} depois: #{Time.now}"
  end
end
```

Caso não seja necessário aplicar os filtros a todas as actions, é possível usar as opções `:except` e `:only`:

```
class ClientesController < ApplicationController
  before_action :verifica_login, only: [:create, :update]

  # ...
end
```

## Logger

As configurações do log podem ser feitas através do arquivo `config/environment.rb`, ou especificamente para cada environment nos

arquivos da pasta config/environments. Entre as configurações que podem ser customizadas, estão qual nível de log deve ser exibido e para onde vai o log (stdout, arquivos, email, ...).

```
Rails::Initializer.run do |config|  
  # ...  
  config.log_level = :debug  
  config.log_path = 'log/debug.log'  
  # ...  
end
```

Mais detalhes sobre a customização do log podem ser encontrados no wiki oficial do Rails:

<http://wiki.rubyonrails.org/rails/show/HowtoConfigureLogging>

CAPÍTULO ANTERIOR:

[Rotas](#)

PRÓXIMO CAPÍTULO:

[Completando o Sistema](#)

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter