

Ruby On Rails

Seminários da Computação



PUC Minas
Poços de Caldas

Luiz Alberto Ferreira Gomes

Curso de Ciência da Computação

2 de setembro de 2019

Agenda

1 Ruby on Rails

2 Modelo

3 Controlador

4 Visão

Rails

Rails é um **framework** para construção de **aplicações web** baseado na **linguagem Ruby**.

Rails (1)

- Rails é fornecido em uma **gem** Ruby (gem é um pacote Ruby)
- Rails fornece uma extenso conjunto de geradores de código e scripts de automação de testes
- Um conjunto de ferramentas adicionais são fornecidos como parte do ecossistema Rails:
 - **Rake** - utilitário similar ao **make do Unix** para criar e migrar bancos de dados, limpar sessões de uma Web app
 - **Puma** - servidor web de desenvolvimento para execução de aplicações Rails
 - **SQLite** - um servidor de banco de dados simples pré-instalado como o Rails

Rails (2)

- Rack Middleware - interface padronizado para interação entre um servidor web e uma Web App

Histórico do Rails (1)

- Rails é um *framework* para construção de aplicações web
- David Heinemeier Hanson **derivou** o Rails a partir do BaseCamp – uma ferramenta de gestão de projetos da empresa 37Signals.
 - a primeira versão de código aberto (em inglês: *open source*) foi liberada em julho de 2004.
 - mas direitos para que outros desenvolvedores **colaborassem** com o projeto foram liberados em fevereiro de 2005.
- Em agosto de 2006, o Ruby on Rails atingiu um **marco importante** quando a Apple decidiu distribuído juntamente com a versão do seu sistema operacional Mac OS X v10.5 "Leopard"

Histórico do Rails (2)

- nesse mesmo ano o Rails começou a ganhar muita atenção da comunidade de desenvolvimento web.
- Rails é utilizado por diversas companhias, como por exemplo:
 - Airbnb, BaseCamp, Disney, GitHub, Hulu, Kickstarter, Shopify e Twitter.

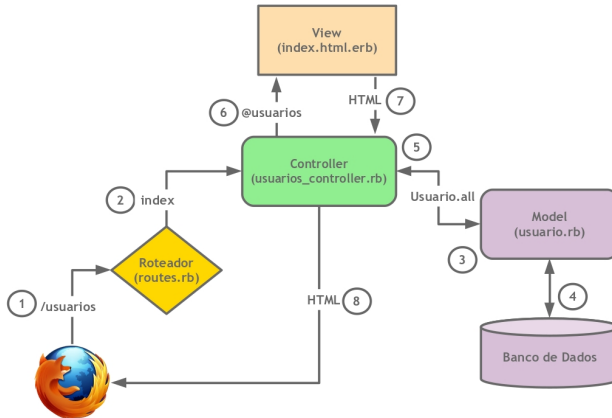
Histórico do Rails (3)

Version history			
Version	Date	Notes	
1.0 ^[22]	December 13, 2005		
1.2 ^[23]	January 19, 2007		
2.0 ^[24]	December 7, 2007		
2.1 ^[25]	June 1, 2008		
2.2 ^[26]	November 21, 2008		
2.3 ^[27]	March 16, 2009		
3.0 ^[28]	August 29, 2010		
3.1 ^[29]	August 31, 2011		
3.2 ^[30]	January 20, 2012		
4.0 ^[31]	June 25, 2013		
4.1 ^[16]	April 8, 2014		
4.2 ^[17]	December 19, 2014		
5.0 ^[18]	June 30, 2016		
5.1 ^[19]	May 10, 2017		
5.2 ^[32]	April 9, 2018		
6.0 ^[33]	August 16, 2019		
<div><div>Old version</div><div>Older version, still supported</div><div>Latest version</div><div>Future release</div></div>			

Filosofia do Rails (2)

- "Don't Repeat Yourself" (DRY) sugere que escrever que o mesmo código várias vezes é uma coisa ruim
- O *Representational State Transfer* (REST) é o melhor padrão para desenvolvimento de aplicações web
 - organiza a sua aplicação em torno de recursos e padrões HTTP (verbs)

Model-View-Controller



Hora de Colocar a Mão na Massa

- Conecte-se na máquina com o seu usuário e sua senha

1. Inicie uma janela de terminal e digite no prompt:

```
$ rails new blog
```

2. Mude para o diretório da aplicação (RAILS.root)

```
$ cd blog
```

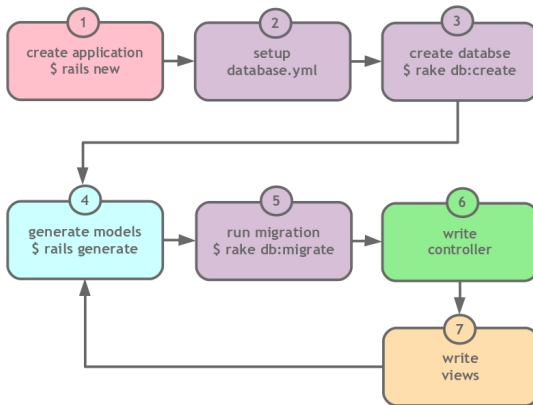
3. Execute o servidor web embutido:

```
$ rails server
```

4. Abra uma janela do navegador e digite:

```
$ http://localhost:3000
```

Metodologia de Trabalho



Passos Iniciais do Blog App (1)

1. Inicie uma janela de terminal e digite no prompt:

```
$ cd  
$ rails new blog
```

2. Inicie o servidor web embutido:

```
$ rails s
```

3. Abra uma janela do navegador e digite:

```
$ http://localhost:3000
```

Estrutura de uma Aplicação Rails (1)

Arquivo/Pasta	Descrição
app	Arquivos contendo os principais códigos da aplicação, incluindo modelos, visões, controladores e auxiliares(<i>helpers</i>)
app/assets	Arquivos contendo folhas de estilos (CSS), códigos Javascript e imagens da aplicação
bin	Arquivos ou scripts executáveis
config	Configurações da aplicação
db	Migrações, esquema e outros arquivos relacionados ao banco de dados
doc	Documentação do sistema
lib	Bibliotecas auxiliares
lib/assets	Arquivos contendo folhas de estilos (CSS), códigos Javascript e imagens das bibliotecas

Estrutura de uma Aplicação Rails (2)

Arquivo/Pasta	Descrição
log	Informações de log
public	Páginas que podem ser acessadas publicamente via navegador, tais como páginas de erros
test	Testes da nossa aplicação
tmp	Arquivos temporários como cache e informações de sessões
vendedor	Dependências e bibliotecas de terceiros
vendedor/assets	Arquivos contendo folhas de estilos (CSS), códigos Javascript e imagens de terceiros
README.rdoc	Uma breve descrição da aplicação
Rakefile	Tarefas que podem ser executadas pelo comando rake
Gemfile	Pacotes(gems) necessários para a aplicação
Gemfile.lock	Uma lista de gems utilizadas para garantir que todas as cópias da aplicação utilizam as mesmas versões de gems
config.ru	Um arquivo de configuração para o Rack Middleware
.gitignore	Define de arquivos ou padrões de arquivos que deverão ser ignorados pelo Git

Agenda

1 Ruby on Rails

2 Modelo

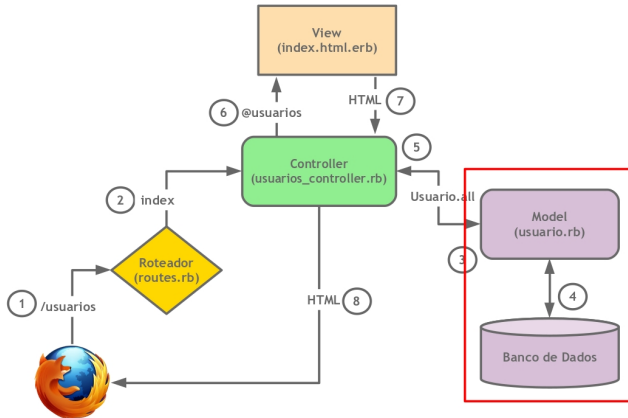
3 Controlador

4 Visão

Modelo

O modelo gerencia os dados, a lógica e as regras de negócios da aplicação.

Modelo



Banco de Dados Relacionais (1)

- Um aspecto importante da programação web é a habilidade de coletar, armazenar e recuperar diferentes formas de dados
 - uma das formas mais populares são os **bancos de dados relacionais**
- Um banco de dados relacional é baseado entidades, denominadas **tabelas**, no relacionamento, **associações**, entre elas
- O contêiner fundamental em um banco de dados relacional é denominado de **database** ou **schema**
 - podem incluir estruturas de dados, os dados propriamente ditos e permissões de acesso

Banco de Dados Relacionais (2)

- Os dados são armazenados em **tabelas** e as tabelas são divididas em **linhas** e **colunas**. Por exemplo:

Tabela: comment

id	post_id	body
10	1	Ruby realmente...
11	2	Rails facilita...
13	2	Concordo, ...

Banco de Dados Relacionais (3)

- Relacionamentos são estabelecidos entre tabelas para que a consistência dos dados seja mantida em qualquer situação e podem ser:
 - 1:1, 1:N ou N:M

Tabela: comment

id	post_id	body
10	1	Ruby realmente...
11	2	Rails facilita...
13	2	Concordo, ...

Tabela: post

id	title	body
1	A Linguagem Ruby	Ruby é legal.
2	O Framework Rais	O Rais facilita...

I1 - Hora de Colocar as Mãos na Massa (1)

1. Gerar o modelo para os *posts*

```
$ rails generate model Post title:string body:text
```

2. Gerar o modelo para os *comentários*

```
$ rails generate model Comment post_id:integer body:text
```

3. Gere as tabelas post e comment no banco de dados

```
$ rake db:create  
$ rake db:migrate
```

SQLite (1)

- O banco de dados que o Rails utiliza em diversos ambientes (desenvolvimento, teste e produção) é especificado em:
`config/database.yml`

```
1 default: &default
2   adapter: sqlite3
3   pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
4   timeout: 5000
5
6 development:
7   <<: *default
8   database: db/development.sqlite3
9
10 # Warning: The database defined as "test" will be erased and
11 # re-generated from your development database when you run "rake".
12 # Do not set this db to the same as development or production.
```


SQLite (2)

```
13 test:
14   <<: *default
15   database: db/test.sqlite3
16
17 production:
18   <<: *default
19   database: db/production.sqlite3
```

- Rails usa por padrão o SQLite como gerenciador padrão
 - relacional, embutido, sem servidor, configuração zero, transacional, suporta SQL

ATENÇÃO: SQLite não um banco de dados para produção !

SQLite (3)

- Banco de dados de produção populares: MySQL e PostgreSQL

Database Console

- O comando **rails db** fornece uma console para acesso aos bancos dados MySQL, PostgreSQL e SQLite.

```
$ rails db
SQLite version 3.8.7.1 2014-10-29 13:59:56
Enter ".help" for usage hints.
sqlite> .headers on
sqlite> .mode columns
sqlite> select * from posts;
```

id	title	body	created_at	updated_at
5	A Linguagem Ruby	Ruby e legal.	2016-04-30 22:45:20.636363	2016-04-30 22:45:20.636363

```
sqlite>
```

- Dica: utilize **headers on** e **mode coluns**

Hora de Colocar a Mão na Massa (1)

- Inicialize **na pasta da aplicação** a console do banco de dados e configure a sua exibição:

```
$ rails db  
sqlite> .headers on  
sqlite> .mode columns
```

- Exiba os colunas da tabela posts:

```
sqlite> .schema posts
```

Hora de Colocar a Mão na Massa (2)

- Crie um novo post e salve no banco de dados:

```
sqlite> INSERT INTO posts  
(title, body, created_at, updated_at)  
VALUES ("Seminarios da Computacao", "Tem varios cursos legais!",  
"2017-10-16 19:50:00", "2017-16-03 19:50:00");
```

- Crie outro post e salve no banco de dados:

```
sqlite> INSERT INTO posts  
(title, body, created_at, updated_at)  
VALUES ("Calor em Pocos", "Como esta quente!!!",  
"2017-10-18 19:50:00", "2017-10-18 19:50:00");
```

- Exiba todos os posts:

```
sqlite> SELECT * FROM posts;
```

Hora de Colocar a Mão na Massa (3)

- Exiba todos os posts ordenados pelo título (title):

```
sqlite> SELECT * FROM posts ORDER BY title;
```

- Exiba um post:

```
sqlite> SELECT * FROM posts LIMIT 1
```

- Exiba o post cujo id é 2:

```
sqlite> SELECT * FROM posts WHERE id=2;
```

- Atualize o título post cujo o id é 2:

```
sqlite> UPDATE posts SET title="O tempo esta louco"  
WHERE id=2;
```

Hora de Colocar a Mão na Massa (4)

- Remova post cujo o id é 2:

```
sqlite> DELETE FROM posts WHERE id=2;
```

Migrations (1)

- Como podemos rastrear e desfazer alterações em um banco de dados?
- Não existe uma maneira fácil - manualmente é confuso e propenso a erros.
- Tipicamente, comandos SQL são dados para criar e modificar tabelas em um banco de dados
- Mas se houver a necessidade de trocar o banco de dados "durante o voo"?
 - por exemplo, desenvolve-se em SQLite e implanta-se em MySQL.

Migrations (2)

SOLUÇÃO: Migrations

Migrations (3)

- A cada vez que o **generate model** é executado na aplicação, o Rails cria um arquivo de **migration** de banco de dados. Este arquivo é armazenado em **db/migrate**
- Por exemplo: o arquivo `20160430140114_create_posts.rb`

```
1 class CreatePosts < ActiveRecord::Migration
2   def change
3     create_table :posts do |t|
4       t.string :title
5       t.text :body
6
7       t.timestamps null: false
8     end
9   end
10 end
```

Migrations (4)

- Rails utiliza o comando **rake** para executar os **migrations** e fazer as alterações no banco de dados.

```
$ rake db:migrate
```

Object-Relational Mapping (1)

- Um ORM **preenche a lacuna** entre banco de dados relacionais e as linguagens de programação orientadas a objetos
- **Simplifica** bastante a escrita de códigos para acessar o banco de dados.
- Tipicamente, comandos SQL são dados para criar e modificar tabelas em um banco de dados
- No Rails, o Model do MVC utiliza algum framework de ORM

Active Record (1)

- ActiveRecord é o nome do **ORM padrão** do Rails?

Listing 1: app/models/post.rb

```
1 class Post < ApplicationRecord
2 end
```

Onde está código ?
R: Metaprogramação +
Convenção

- Para que "**mágica**" ocorra:
 - o ActiveRecord tem que saber como encontrar o banco de dados (ocorre via **config/database.yml**)
 - (**Convenção**) existe uma **tabela** com o **nome no plural** da subclasse ApplicationRecord

Active Record (2)

- **(Convenção)** espera-se que a tabela tenha uma chave primário denominada **id**

Object-Relational Mapping (1)

- Um ORM **preenche a lacuna** entre banco de dados relacionais e as linguagens de programação orientadas a objetos
- **Simplifica** bastante a escrita de códigos para acessar o banco de dados.
- Tipicamente, comandos SQL são dados para criar e modificar tabelas em um banco de dados
- No Rails, o Model do MVC utiliza algum framework de ORM

Hora de Colocar a Mão na Massa (1)

- Inicialize **na pasta da aplicação** a console do Rails (não a do banco de dados):

```
$ rails c
```

- Exiba os atributos da classe Post:

```
irb(main):004:0> Post.column_names
```

- Crie um novo post e salve no banco de dados:

```
irb(main):005:0> p1 = Post.new  
irb(main):006:0> p1.title="Temperatura em Pocos"  
irb(main):007:0> p1.body="Esta muito frio..."  
irb(main):008:0> p1.save
```


Hora de Colocar a Mão na Massa (2)

- Exiba todos os posts:

```
irb(main):007:0> Post.all
```

- Exiba todos os posts ordenados pelo título (title):

```
irb(main):007:0> Post.all.order(title: :asc)
```

- Exiba um post:

```
irb(main):007:0> Post.first
```

- Exiba o post cujo id é 2:

```
irb(main):007:0> Post.find_by(id: 2)
```

Hora de Colocar a Mão na Massa (3)

- Atualize o título do primeiro post:

```
irb(main):007:0> p1=Post.first  
irb(main):008:0> p1.update(title: "Pensando...")
```

- Remova do primeiro post:

```
irb(main):007:0> p1=Post.first  
irb(main):008:0> p1.destroy
```

Validação em Aplicações Web

- **Validação de Dados** é o processo para **garantir** que a aplicação web operem **corretamente**. Exemplo:
 - garantir a validação do e-mail, número do telefone e etc
 - garantir que as "regras de negócios" sejam validadas
- A **vulnerabilidade** mais comum em aplicação web é a **injeção SQL**

Client Side

- Envolve a verificação de que os formulários HTML sejam preechidos corretamente
 - **JavaScript** tem sido tradicionalmente utilizado.
 - **HTML5** possui "input type" específicos para checagem.
 - Funciona melhor quando combinada com validações do lado do servidor.

Server Side

- A validação é feita após a submissão do formulário HTML
 - **banco de dados**(stored procedure) - dependente do banco de dados
 - **no controlador** - veremos mais tarde que não se pode colocar muita lógica no controlador (controladores magros)
 - **no modelo** - boa maneira de garantir que dados válidos sejam armazenados no banco de dados (database agnostic)
 - Funciona melhor quando combinada com validações do lado do servidor.

Validação em Rails (1)

- **Objetos** em um sistema OO como tendo um **ciclo de vida**
 - eles são criados, atualizados mais tarde e também destruídos.
- Objetos ActiveRecord têm **métodos** que podem ser chamados, a fim de assegurar a sua **integridade** nas várias fases do seu ciclo de vida.
 - garantir que todos os atributos são **válidos** antes de salvá-lo no banco de dados
- **Callbacks** são métodos que são invocados em um ponto do ciclo de vida dos objetos ActiveRecord
 - eles são "ganchos" para gatilhos para acionar uma lógica quando houver alterações de seus objetos

Validação em Rails

- **Validations** são tipo de **callbacks** que podem ser utilizados para garantir a validade do dado em um banco de dados
- Validação são definidos nos **modelos**. Exemplo:

```
1 class Person < ApplicationRecord
2   validates_presence_of :name
3   validates_numeracality_of :age, :only_integer => true
4   validates_confirmation_of :email
5   validates_length_of :password, :in => 8..20
6 end
```

I2 - Hora de Colocar a Mão na Massa (1)

- Modifique o arquivo `app/models/post.rb` para exigir que o usuário digite o título e o texto do blog:

```
1      class Post < ApplicationRecord
2          validates_presence_of :title, :body
3      end
```

- Modifique o arquivo `app/models/comment.rb` para exigir que o usuário digite texto do comentário blog:

```
1      class Post < ApplicationRecord
2          validates_presence_of :body
3      end
```

- **Reinicie** a console do Rails tente criar um Post e um Comment

I2 - Hora de Colocar a Mão na Massa (2)

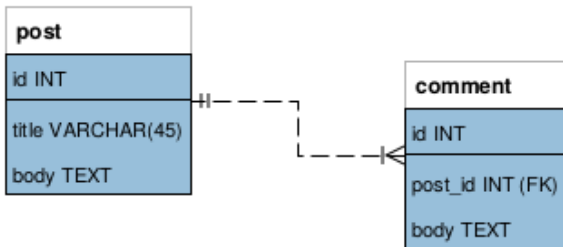
```
irb(main):005:0> p1 = Post.new
irb(main):006:0> p1.body="Tem algo errado..."
irb(main):007:0> p1.save
irb(main):008:0> Post.all
irb(main):009:0> c1 = Comment.new
irb(main):010:0> c1.save
irb(main):011:0> Comment.all
```

Associações em Rails (1)

- O gerador de modelos utiliza por padrão o ActiveRecord. Isto significa:
 - Tabelas para postagens e comentários foram criadas quando executamos as migrações
 - Um conexão com o banco de dados é estabelecida
 - O ORM é configurado para as postagens e comentários foi criado - o "M" do MVC.
- No entanto, uma coisa está faltando:
 - **tem-se que assegurar que qualquer comentários sejam associados às suas postagens**
- Para tornar os modelos em Rails totalmente funcionais precisamos adicionar **associações**:

Associações em Rails (2)

- cada postagem precisa saber os comentários associado a ele
- cada comentário precisa saber qual é a postagem ele pertence
- Há uma relação **muitos-para-um** entre comentários e postagens uma:



Associações em Rails (3)

- O ActiveRecord contém um conjunto de métodos de classe para **vinculação** de objetos por meio de **chaves estrangeiras**
- Para habilitar isto, deve-se declarar as **associações** dentro dos modelos usando:

Associação	Modelo Pai	Modelo Filho
Um-para-um	has_one	belongs_to
Muitos-para-um	has_many	belongs_to
Muitos-para-muitos	has_and_belongs_to_many	*na tabela junção

13 - Hora de Colocar a Mão na Massa (1)

- Modifique o arquivo `app/models/post.rb` para associar o post aos seus comentário:

```
1      class Post < ApplicationRecord
2          validates_presence_of :title, :body
3          has_many :comments
4      end
```

- Modifique o arquivo `app/models/comment.rb` para associar o comentário ao seu post:

```
1      class Comment < ApplicationRecord
2          validates_presence_of :body
3          belongs_to :post
4      end
```

13 - Hora de Colocar a Mão na Massa (2)

- Crie um novo post e salve no banco de dados (**Reinicie a console do Rails**):

```
irb(main):005:0> p1 = Post.new
irb(main):006:0> p1.title="Associacao"
irb(main):007:0> p1.body="Eu tenho comentarios!"
irb(main):008:0> p1.save
```

- Crie um novo comment e o vincule a um post:

```
irb(main):005:0> c1 = Comment.new
irb(main):006:0> c1.body="Eu sou de um post!"
irb(main):007:0> c1.post = p1
irb(main):008:0> c1.save
```

13 - Hora de Colocar a Mão na Massa (3)

- Consulte os comentários do post p1:

```
irb(main):005:0> p1.comments.all
```

- Consulte os comentários 2 do post p1:

```
irb(main):005:0> p1.comments.where(id: 2)
```

- Consulte o post do comentário c1:

```
irb(main):005:0> c1.post
```

Agenda

1 Ruby on Rails

2 Modelo

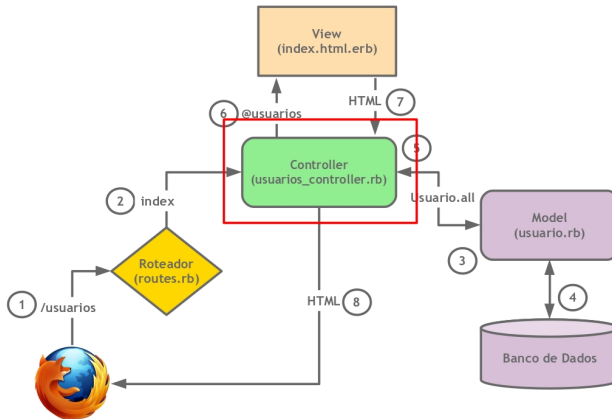
3 Controlador

4 Visão

Controlador (1)

- Um **Action Controller** é classe Ruby contendo uma ou mais ações
- Cada **ação** é responsável pela resposta a uma requisição
- Quando uma ação é concluída a **visão** de mesmo nome é **renderizada**
- Uma ação deve estar **mapeada** no arquivo **routes.rb**:

Controlador (2)



Representational State Transfer

- Rails utiliza Representational State Transfer (REST) para mapear os recursos (resources) de uma aplicação:
 - **List** todos os recursos disponíveis
 - **Show** um recurso específico
 - **Destroy** um recurso existente
 - **Provide a way to create** um novo recurso
 - **Create** um novo recurso
 - **Provide a way to update** um recurso existente
 - **Update** um recurso existente

14 - Hora de Colocar a Mão na Massa (1)

- Inicie o servidor web

```
$ rails s
```

- Acesse a url `<http:\localhost:3000/posts>`. Veja o erro que ocorreu.
- Gere o controlador para os posts:

```
$ rails generate controller posts
```

- Reinicie o servidor web e acesse a url `<http:\localhost:3000/posts>`. Veja o erro que ocorreu.

14 - Hora de Colocar a Mão na Massa (2)

- Modifique o arquivo `config/routes.rb` para acrescentar a rota para os posts:

```
1 Rails.application.routes.draw do
2   resource :posts
3 end
```

- Execute o comando `rake` para visualizar as rotas para os posts:

```
$ rake routes
```

- Reinicie o servidor web e acesse a url `<http://localhost:3000/posts>`. Veja o erro que ocorreu.

Ação: Index (1)

- Ação que recupera **todas as postagens** do blog
- (Implicitamente) procura pelo template **index.html.erb** para renderizar a resposta

Listing 2: controllers/posts_controller.rb

```
1 Class PostsController < ApplicationController
2   def index
3     @posts = Post.all
4   end
5 end
```

Ação: Index (2)

■ index.html.erb:

Listing 3: views/posts/index.html.erb

```
1 <p id="notice"><%= notice %></p>
2 <h1>Posts</h1>
3 <table>
4   <thead>
5     <tr>
6       <th>Titulo</th>
7       <th>Conteudo</th>
8       <th colspan="3"></th>
9     </tr>
10  </thead>
11
12  <tbody>
13    <% @posts.each do |post| %>
14      <tr>
```

Ação: Index (3)

```
15      <td><%= post.title %></td>
16      <td><%= post.body %></td>
17      <td><%= link_to 'Show', post %></td>
18      <td><%= link_to 'Edit',
19          edit_post_path(post) %></td>
20      <td><%= link_to 'Delete', post, method: :delete,
21          data: { confirm: 'Tem certeza ?' } %></td>
22  </tr>
23  <% end %>
24  </tbody>
25 </table>
26 <%= link_to 'New Post', new_post_path %>
```


15 - Hora de Colocar a Mão na Massa

- Implemente a ação **index** no controlador de posts e a visão correspondente conforme os slides anteriores
- Reinicie o servidor web e acesse a url `<http:\localhost:3000/posts>`.

Ação: Show (1)

- Recupera **uma** postagem específica no parâmetro **id** passado como parte da URL
- (Implicitamente) procura pelo **show.html.erb** para renderizar a resposta

Ação: Show (2)

Listing 4: controllers/posts_controller.rb

```
1  Class PostsController < ApplicationController
2    before_action :set_post, only: [:show, :edit,
3      :update, :destroy]
4
5    def show
6    end
7
8    private
9      def set_post
10        @post = Post.find(params[:id])
11      end
```

Ação: Show (3)

■ `show.html.erb`:

Listing 5: `views/posts/show.html.erb`

```
1 <p>
2   <strong>Title:</strong>
3   <%= @post.title %>
4 </p>
5 <p>
6   <strong>Body:</strong>
7   <%= @post.body %>
8 </p>
9 <%= link_to 'Edit', edit_post_path(@post) %> |
10 <%= link_to 'Back', posts_path %>
```

I6 - Hora de Colocar a Mão na Massa

- Implemente a ação **show** no controlador de posts e a visão correspondente conforme os slides anteriores
- Reinicie o servidor web e acesse a url `<http:\localhost:3000/posts>` e clique em "**Show**".

Ação: Destroy (1)

- Remove uma postagem específica pelo parâmetro **id** passado como parte da URL

Listing 6: posts_controller.rb

```
1  class PostsController < ApplicationController
2    before_action :set_post, only: [:show, :edit,
3      :update, :destroy]
4
5    def destroy
6      @post.destroy
7      redirect_to posts_url,
8        notice: 'Post foi removido com sucesso!'}
9  end
```

17 - Hora de Colocar a Mão na Massa

- Implemente a ação **destroy** no controlador de posts e a visão correspondente conforme os slides anteriores
- Reinicie o servidor web e acesse a url `<http:\localhost:3000/posts>` e clique em "**Remove**".

Partials (1)

- Rails encoraja o princípio **DRY**
- O layout da aplicação é mantida em um único local no arquivo **application.html.erb**
- O código comum dos templates ser reutilizado em **múltiplos templates**
- Por exemplo, os formulários do **edit** e do **new** - são realmente muito diferentes ?
- Partials são similares aos templates regulares, mas ele possuem capacidades mais **refinadas**
- Nomes de partials começam com **underscore** (**_**)

Partials (2)

- Partials são renderizados com `render 'partialname'` (sem underscore)
- `render` também aceita um segundo argumento, um hash com as variáveis locais utilizadas no partial
- Similar a passagem de variáveis locais, o `render` pode receber um objeto
- `<%= render @post %>` renderizara `app/views/posts/_posts.html.erb` com o conteúdo da variável `@post`

Partials (3)

- `<%= render @posts %>` renderiza uma coleção e é equivalente a:

Listing 7: controllers/posts_controller.rb

```
1      <% @posts.each do |post| %>
2          <%= render post %>
3      <% end %>
```

Partials (4)

■ `_form.html.erb`

Listing 8: views/posts/_form.html.erb

```
1 <%= form_for(@post) do |f| %>
2   <% if @post.errors.any? %>
3     <div id="error_explanation">
4       <h2><%= pluralize(@post.errors.count, "error") %>
5         impedem que os post seja salvo:</h2>
6       <ul>
7         <% @post.errors.full_messages.each do |message| %>
8           <li><%= message %></li>
9         <% end %>
10      </ul>
11    </div>
12    <% end %>
13
14    <div class="field">
```

Partials (5)

```
15     <%= f.label :title %><br>
16     <%= f.text_field :title %>
17 </div>
18 <div class="field">
19     <%= f.label :body %><br>
20     <%= f.text_area :body %>
21 </div>
22 <div class="actions">
23     <%= f.submit %>
24 </div>
25 <% end %>
```

18 - Hora de Colocar a Mão na Massa

- Implemente o partials `_form.html.erb` na pasta `view/posts`

Ação: New (1)

- Cria um novo objeto `post`(vazio)
- (Implicitamente) procura pelo `new.html.erb` para renderizar a resposta

Listing 9: posts_controller.rb

```
1 Class PostsController < ApplicationController
2
3   def new
4     @post = Post.new
5   end
```

Ação: New (2)

■ new.html.erb:

Listing 10: views/posts/new.html.erb

```
1 <h1>Novo Post</h1>
2 <%= render 'form' %>
3 <%= link_to 'Back', posts_path %>
```

l11 - Hora de Colocar a Mão na Massa

- Implemente a ação **new** no controlador de posts e a visão correspondente conforme os slides anteriores
- Reinicie o servidor web e acesse a url `<http:\localhost:3000/posts>` e clique em "**New Post**".

Ação: Create (1)

- Cria um novo objeto **post** como os parâmetros que foram passados pelo formulário **new**
- Tenta **salvar** o objeto no **banco de dados**
- Se sucesso, redireciona para o template **show**
- Se insucesso, renderiza o template **new** novamente

Ação: Create (2)

Listing 11: controllers/posts_controller.rb

```
1  class PostsController < ApplicationController
2    def create
3      @post = Post.new(post_params)
4
5      if @post.save
6        redirect_to @post, notice: 'Post foi criado com sucesso!'
7      else
8        render :new
9      end
10   end
11
12   private
13   def post_params
14     params.require(:post).permit(:title, :body)
15   end
16 end
```

Ação: Create (3)

- a linha 20 implementa **strong parameters** para aumentar a segurança da aplicação

I10 - Hora de Colocar a Mão na Massa

- Implemente a ação **create** no controlador de posts
- Reinicie o servidor web e acesse a url `<http:\localhost:3000/posts>`, clique em "**New Post**" e em "**Create Post**".

Flash (1)

- **Problema:** Queremos **redirecionar** um usuário para uma página diferente do nosso site, mas ao mesmo tempo **fornecer** a ele algum tipo de mensagem. Exemplo: "Postagem criada !"
- **Solução:** flash - uma **hash** onde a dado persiste por exatamente **UMA requisição APÓS** a requisição corrente
- Um conteúdo pode ser colocado em um flash assim:

Listing 12: controllers/posts_controller.rb

```
1 flash[:attribute] = value
```

- Dois atributos **comuns** são **:notice(good)** e **:alert (bad)**

Flash (2)

- Estes dois atributos (:notice ou :alert) podem ser colocados no `redirect_to`
- `show.html.erb`:

Listing 13: `views/posts/show.html.erb`

```
1      <p id="notice"><%= notice %></p>
2      <p>
3        <strong>Title:</strong>
4        <%= @post.title %>
5      </p>
6      <p>
7        <strong>Body:</strong>
8        <%= @post.body %>
9      </p>
10     <%= link_to 'Edit', edit_post_path(@post) %> |
11     <%= link_to 'Back', posts_path %>
```

Ação: Edit (1)

- Recupera uma postagem específica no parâmetro `id` passado como parte da URL
- (Implicitamente) procura pelo `edit.html.erb` para renderizar a resposta

Listing 14: controllers/posts_controller.rb

```
1 Class PostsController < ApplicationController
2   before_action :set_post, only: [:show, :edit
3     , :update, :destroy]
4
5   def edit
6   end
```

Ação: Edit (2)

■ edit.html.erb:

Listing 15: view/posts/edit.html.erb

```
1      <h1>Edit Post</h1>
2      <%= render 'form' %>
3      <%= link_to 'Show', @post %> |
4      <%= link_to 'Back', posts_path %>
```


l11 - Hora de Colocar a Mão na Massa

- Implemente a ação **edit** no controlador de posts e a visão correspondente conforme os slides anteriores
- Reinicie o servidor web e acesse a url `<http:\localhost:3000/posts>` e clique em "**Edit**".

Ação: Update (1)

- Recupera um objeto **post** utilizando o parâmetro **id**
- Atualiza o objeto **post** com os parâmetros que foram passados pelo formulário **edit**
- Tenta **atualizar** o objeto no **banco de dados**
- Se sucesso, redireciona para o template **show**
- Se insucesso, renderiza o template **edit** novamente

Ação: Update (2)

Listing 16: posts_controller.rb

```
1  Class PostsController < ApplicationController
2    def update
3      if @post.update(post_params)
4        redirect_to @post, notice: 'Post modificado com sucesso!' }
5      else
6        render :edit
7      end
8    end
```

I12 - Hora de Colocar a Mão na Massa

- Implemente a ação **update** no controlador de posts
- Reinicie o servidor web e acesse a url `<http:\localhost:3000/posts>`, clique em "**Edit**" e em "**Submit**".

113 - Hora de Colocar a Mão na Massa (1)

- Modifique o arquivo de rotas para aninhar os comentários às postagens e reinicie o servidor:

Listing 17: config/routes.rb

```
1 Rails.application.routes.draw do
2   resources :posts do
3     resources :comments
4   end
5 end
```

- Modifique o código do template `views/posts/show.html.erb`. Insira o código abaixo do parágrafo do body.

113 - Hora de Colocar a Mão na Massa (2)

```
1 <h2>Comments</h2>
2 <div id="comments">
3   <% @post.comments.each do |comment| %>
4     <p>
5       <strong>Posted <%= time_ago_in_words(comment.created_at) %></st
6       <%= h(comment.body) %>
7     </p>
8     <% end %>
9 </div>
```

- Agora no navegador visualize uma postagem que tenha comentários.
- Acrescente o código a seguir logo abaixo do código anterior no arquivo `views/posts/show.html.erb`:

113 - Hora de Colocar a Mão na Massa (3)

```
1  <%= form_for([@post, Comment.new]) do |f| %>
2  <p>
3  <%= f.label :body, "New Comment" %><br>
4  <%= f.text_area :body %>
5  </p>
6  <p>
7  <%= f.submit "Add Comments" %>
8  </p>
9  <% end %>
```

- Gere o controlador para os comments:

```
$ rails generate controller comments
```

- Modifique a ação create do controlador `controllers/comments_controller.rb`:

113 - Hora de Colocar a Mão na Massa (4)

```
1  before_action :set_comment, only: [:show, :edit, :update, :destroy]
2
3  def create
4    @post = Post.find(params[:post_id])
5    @comment = @post.comments.create(comment_params)
6
7    if @comment.save
8      redirect_to @post, notice: 'Comment foi criado com sucesso!'
9    else
10      redirect_to @post
11    end
12  end
13
14  private
15    def set_comment
16      @comment = Comment.find(params[:id])
17    end
18
```


I13 - Hora de Colocar a Mão na Massa (5)

```
19  def comment_params
20      params.require(:comment).permit(:body)
21  end
```

- Escolha uma postagem qualquer e escreva alguns comentários.

Agenda

1 Ruby on Rails

2 Modelo

3 Controlador

4 Visão

Action View

- Arquivo HTML com a extensão **.erb**
 - ERb é uma **biblioteca** que permite a colocação de código Ruby no HTML
- Dois padrões a aprender:
 - `<% ...código ruby..%>` avalia o código Ruby
 - `<%= ...código ruby..%>` retorna o resultado do código avaliado

Form Helpers (1)

- `form_for` gere a tag form para o objeto passado como parâmetro
- Rails utiliza a método `POST` por padrão
- Isto faz sentido:
 - uma password não é passada como parâmetro na URL
 - qualquer modificação deverá ser feita via POST e não GET

Listing 18: views/posts/_form.html.erb

```
1 <%= form_for(@post) do |f| %>
2   ...
3 <% end %>
```

f.label

- Gera a tag HTML **label**
- A descrição pode ser **personalizada** passando um segundo parâmetro

```
1 <div class="field">
2   <%= f.label :title, "Titulo" %><br>
3   <%= f.text_field :title %>
4 </div>
```

f.text_field

- Gera o campo `input type="text"`
- Utilize `:placeholder` para mostrar um valor dentro do campo

```
1 <div class="field">
2   <%= f.label :title, "Titulo" %><br>
3   <%= f.text_field :title, placeholder: "Escreva o titulo aqui." %>
4 </div>
```

f.text_area

- Similar ao `f.text_field`, mas gera um text area de tamanho (40 cols x 20 rows)
- O tamanho pode ser modificado através do atributo `size`:

```
1 <div class="field">
2   <%= f.label :body, "Conteúdo" %><br>
3   <%= f.text_area :body, size: "10x3" %>
4 </div>
```

Outros Form Helpers

- `date_select`
- `search_field`
- `telephone_field`
- `url_field`
- `email_field`
- `number_field`
- `range_field`

f.submit

- Renderiza o botão **submit**
- Aceita o **nome** do botão submit como primeiro argumento
- Se o nome não for fornecido - gera um baseado no modelo e na ação. Por exemplo: "Create Post" ou "Update Post"

```
1 <div class="actions">
2   <%= f.submit "Postar"%>
3 </div>
```

- Mais form helpers:
(http://guides.rubyonrails.org/form_helpers.html)